ABSTRACT

Title of dissertation: EFFICIENT GEOMETRY AND ILLUMINATION REPRESENTATIONS FOR INTERACTIVE PROTEIN VISUALIZATION

Xuejun Hao, Doctor of Philosophy, 2004

Dissertation directed by: Professor Amitabh Varshney
Department of Computer Science

This dissertation explores techniques for interactive simulation and visualization for large protein datasets. My thesis is that using efficient representations for geometric and illumination data can help in developing algorithms that achieve better interactivity for visual and computational proteomics. I show this by developing new algorithms for computation and visualization for proteins. I also show that the same insights that resulted in better algorithms for visual proteomics can also be turned around and used for more efficient graphics rendering.

Molecular electrostatics is important for studying the structures and interactions of proteins, and is vital in many computational biology applications, such as protein folding and rational drug design. We have developed a system to efficiently solve the non-linear Poisson-Boltzmann equation governing molecular electrostatics. Our system simultaneously improves the accuracy and the efficiency of the solution by adaptively refining the computational grid near the solute-solvent interface. In addition, we have explored the possibility of mapping the PBE solution onto GPUs.

We use pre-computed accumulation of transparency with spherical-harmonics-based compression to accelerate volume rendering of molecular electrostatics.

We have also designed a time- and memory-efficient algorithm for interactive visualization of large dynamic molecules. With view-dependent precision control and memory-bandwidth reduction, we have achieved real-time visualization of dynamic molecular datasets with tens of thousands of atoms. Our algorithm is linearly scalable in the size of the molecular datasets.

In addition, we present a compact mathematical model to efficiently represent the six-dimensional integrals of bidirectional surface scattering reflectance distribution functions (BSSRDFs) to render scattering effects in translucent materials interactively. Our analysis first reduces the complexity and dimensionality of the problem by decomposing the reflectance field into non-scattered and subsurface-scattered reflectance fields. While the non-scattered reflectance field can be described by 4D bidirectional reflectance distribution functions (BRDFs), we show that the scattered reflectance field can also be represented by a 4D field through pre-processing the neighborhood scattering radiance transfer integrals. We use a novel reference-points scheme to compactly represent the pre-computed integrals using a hierarchical and progressive spherical harmonics representation. Our algorithm scales linearly with the number of mesh vertices.

Efficient Geometry and Illumination Representations for Interactive Protein Visualization

by

Xuejun Hao

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2004

Advisory Committee:

Professor Amitabh Varshney, Chair/Advisor
Professor Leila De Floriani
Professor David Jacobs
Professor David Mount
Professor Hanan Samet
Professor Sergei Sukharev

ACKNOWLEDGMENTS

I owe my gratitude to all the people who have made this thesis possible.

First, I'd like to thank my advisor, Dr. Amitabh Varshney for his excellent advice and for giving me the opportunity to work on challenging and extremely interesting projects. He has not only taught me how to do good fundamental research, but also given me a lifetime philosophy so as to be more successful as a human. His constant cheering has made my graduate experience such an enjoyable one.

I would like to thank my other committee members for their insightful comments and cheerful inspiration. In particular I would like to thank Dr. Leila De Floriani for her wonderful advice on research topics, as well as on career opportunities; Dr. David Jacobs for his inspirational work on spherical harmonic representation for reverse rendering; Dr. David Mount for teaching me computational geometry, and sharing his bright ideas on research problems in illumination and geometry; Dr. Hanan Samet for sharing his key insights on various research topics and giving numerous advices on research and life related issues; Dr. Sergei Sukharev for his tutorial on computational biology, especially on ion-channel studies.

Dr. Yiannis Aloimonos and Dr. Cornelia Fermller have given me numerous wonderful advice and encouragement. I would like to them for all their help.

I would like to thank Dr. Dianne P. O'Leary for sharing her deep insights on scientific computations, and Dr. Marc Olano at University of Maryland at Baltimore

County for sharing his wonderful insights on GPU programming.

I would also like to thank Bill Baxter at University of North Carolina at Chapel Hill, and Won-Ki Jeong at University of Utah for sharing their insights on GPU programming, and all members of GVIL Lab at Maryland for their support, including Thomas Baby, Indrajit Bhattacharya, Aravind Kalaiah, Chang Ha Lee, Ziyun Li, and Youngmin Kim.

I owe my deepest thanks to my family - my parents, my wife, and my sister, for their continuous support and encouragement.

I have been fortunate to be with the people mentioned and because of them my graduate experience has been one that I will cherish forever.

Table of Contents

List of Tables

List of Figures

Chapter 1

Overview and Results

1.1 Motivation

Interactive simulation and visualization of dynamic proteins are vital for many important applications, such as protein folding and rational drug design.

Globular proteins are well packed and adopt ordered three-dimensional structures. More importantly, they possess a variety of motions such as bond vibrations, side-chain rotations, segmental motions, and domain movements. It is motion that is required for proteins to work, and it is our inability to fully understand protein dynamics and their role in protein function that restricts our understanding of the mechanisms of protein folding, recognition, allostery, and catalysis. Protein dynamics are extremely complex and difficult to analyze, because a variety of motions take place in the same molecule and at the same time. Being able to simulate and visualize protein motions on a computer is therefore of utmost importance for the understanding of the very complex picture of protein dynamics and for the development of proper theoretical models for analysis.

Virtual environments offer a powerful interaction medium for exploring such datasets in real time, enabling superior insights into the underlying biochemical processes. I consider Visual Proteomics to be the field that studies the relationship

between structure and function of proteins through visualization of various protein properties, such as their 3D structure, electron density, and electrostatic potential. Visual Proteomics can enhance the accessibility of protein modeling methods and assist the analysis and interpretation of voluminous data to distill the essential findings. Interactive visualization can not only provide a high-bandwidth human-computer interface to convey the rich multi-dimensional information space, but also make computation more insightful and efficient through computational steering.

Computational steering is the interactive control over a computational process during execution. By closely coupling simulation and visualization, it becomes possible to control the execution of the simulation through visualization of its output.

In an interactive computational process, a sequence of specification, computation, and analysis is performed. For each adaptation that is to be made to the computational model, this process has to be repeated. Computational steering closes the loop such that scientists can respond to results as they occur by interactively manipulating the input parameters for model exploration, algorithm experimentation, and performance optimization.

Computational steering enhances productivity by greatly reducing the time between changes to parameters and the viewing of the results, and makes the cause-effect relationships self-evident [101]. Computational steering has become a powerful paradigm for scientific discovery and has been applied successfully to molecular dynamics visualization and protein visualization [13, 17, 88].

A crucial aspect of visual steering and molecular dynamics visualization is interactivity. We believe that the design of efficient data representations with better

precision and resolution control will improve the interactivity of simulation and visualization of protein dynamics. So I have developed a set of techniques for better representation of geometry and illumination.

In this thesis, I will detail these representations, as well as the resulting algorithms for visual proteomics.

Let me first give an overview of the geometry representations that I have developed. I have devised a distance-field-guided tetrahedral-space decomposition to improve the efficiency of protein electrostatics computations [55] since molecular electrostatics is one of the most important aspects of non-bonded protein interactions. The distance field is computed from the solvent-accessible surface. To improve the interactivity of protein dynamics visualization, I have designed a variable-precision representation that leverages the multiple-precision operations in modern processors [54]. In addition, I have generated a real-time triangle-strip representation to render from connectivity-compressed data. I have also used a real-time occlusion map to cull occluded atoms and reduce the geometry bandwidth as well as the pixel-fill rate bottlenecks [57]. My algorithm for displaying dynamic proteins is four times faster than the popular VMD system [69].

Let me next give an overview of the illumination representations that I have developed. I have used spherical-harmonic-based accumulated transparency to develop an order-independent splatting algorithm for electrostatic field visualization. I have taken advantage of spatial coherence in the subsurface scattered reflectance field and developed a reference-point-based representation for light scattering integrals. This enables an interactive rendering of translucent materials with a factor

of 60 speedup over the previous best result [53, 56]. Similarly, my work on variable-precision lighting uses spatial-coherence of light, view, and normal vectors to speed up the rendering [54].

This chapter is an extended abstract of the dissertation, outlining the problem, our approaches, and the results.

1.2 Simulation of Molecular Electrostatics

Electrostatic interactions play a central role in biological processes. Electrostatics influence nearly all biochemical reactions, such as macromolecular folding and conformational stability. Electrostatics also determine the structural and functional properties of biological samples, such as their shapes, binding energies, and association rates. The successful modelling of electrostatics has great practical, as well as, theoretical importance.

1.2.1 Problem Definition

Modern electrostatic models are based on the non-linear Poisson-Boltzmann equation (PBE). Development of fast computational methods to solve PBE is vital for biomolecular modeling and simulation packages.

In most cases, an analytical solution to PBE does not exist and numeric methods have been developed. Among them, the finite difference method is the most widely used. The accuracy of the results is highly dependent on grid spacing, while the computational cost increases steeply with the number of grid points.

The *multi-grid method* has been used successfully to reduce the cost. But it might not converge when applied to the non-linear PBE. Since the computational cost of using a regular grid is proportional to the cube of the grid size, the adaptive space-subdivision approach has been introduced to address the problem. It works by increasing the accuracy of the solution by explicitly giving a higher spatial resolution to the solvent-solute boundary region. However, it tends to over-subdivide around the solvent-solute boundary region and results in slow convergence.

To resolve the tradeoffs between a more accurate solution and faster convergence, we believe that the previous 3D grid data structures are not the best. We want to design an irregular 3D grid structure such that the local resolution of each grid region is determined by its importance to the final solution.

Our method is based on the observation that the accuracy and stability of the solution to PBE is quite sensitive to the boundary layer between the solvent and the solute. Therefore an accurate construction of this boundary with adaptively controlled grid density should improve the accuracy and convergence rate of the solution.

1.2.2 Algorithm Overview

Our approach first analytically constructs the solvent-accessible surface of the molecule. This surface is the interface between the solvent and molecule where the physical properties such as dielectric constant, and therefore the electrical potential, changes dramatically. We then build a tetrahedral decomposition of the 3D space around the surface and construct a distance field from the surface. Next, we build iso-surfaces

by the marching-tetrahedra method on the distance field with progressively greater distances. This results in nested isosurfaces at varying distances from the solvent-accessible surface. After that, we apply an edge-collapse-based volume-simplification algorithm to simplify the tetrahedral grid to adaptively adjust the grid density according to its influence on the solution. We maintain a higher resolution in the vicinity of the solvent-accessible surface that determines the accuracy and convergence rate of the solution. Finally, we solve PBE on this irregular grid with a generalized finite difference method based on Taylor's series expansion. Details of this algorithm and its implementation are given in Chapter 2.

1.2.3 Results

In this and the following chapters, except otherwise specified, results have been obtained on a 2GHz Pentium 4 PC running Windows 2000 with a nVIDIA Geforce3 graphics card.

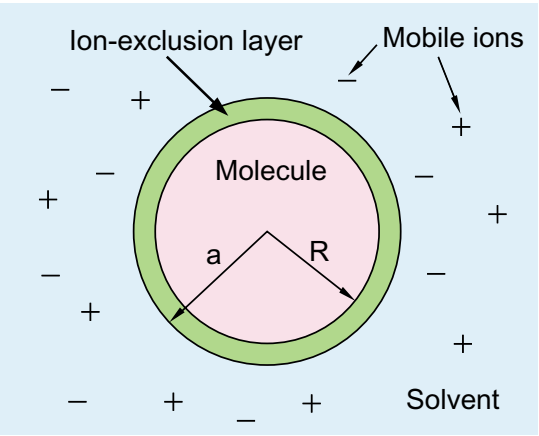


Figure 1.1: An analytically solvable case of PBE

We have tested our algorithm on an analytical solvable case (see Figure 1.1)

and compared the results with the popular DelPhi (V.4) program. The results are summarized in Table 1.1. Here we have used a spherical surface charge with a diameter of 27 Å and a positive charge of $20e$ (where e is the charge of an electron). The sphere is immersed in a cubic solvent volume whose each side is 66 $\mathring{A}$ long.

	DelPhi			Our Method
Grid size	67^3	133^3	199^3	N/A
Number of pts	300,763	2,352,637	7,880,599	26,987
PSNR	8.17	19.1	25.1	27.7
Average error	30.88%	17.91%	13.27%	15.98%
PBE Time	0.31 sec	4.50 sec	20.09 sec	0.25 sec

Table 1.1: Comparison of our method with DelPhi

The average error in Table 1.1 is defined as the average of the relative error over all grid points. Peak-signal-to-noise-ratio (PSNR) is defined as $20\log_{10}(\frac{signal\ energy}{noise\ energy})$. The *signal energy* is defined as the sum of the squares of the potential values over all grid points. The *noise energy* is defined as the sum of the squares of the errors over all grid points. PBE time is the time for solving linear PBE on the grid. One can see the advantages of our method from Table 1.1. To get the same accuracy, our method needs only $27K$ points instead of several million needed by DelPhi, and takes only 0.25 seconds to converge, compared with several seconds by DelPhi. For about the same amount of time, our method is much more accurate than DelPhi, e.g., 15.98% instead of 30.88% error. Our algorithm with $27K$ points has even higher PSNR than DelPhi with about $8M$ points.

1.3 Visualization of Molecular Electrostatics and Dynamics

Protein electrostatics and dynamics data are large and contain complex structures. Interactive display of these data provides (a) a high-bandwidth human-computer interface to provide a better understanding of the relation between the structure and function of proteins, and (b) help in computational steering of large protein-folding and molecular docking simulations.

1.3.1 Order-independent Splatting for Efficient Electrostatics Visualization

The 3D electrostatic potential field is a scalar field defined over a volume. Volume visualization methods have been developed to display volumetric data. These methods are categorized into indirect and direct volume rendering methods. Indirect volume rendering first converts the volume data into a polygonal iso-surface representation and then displays the surface. The Marching Cubes algorithm [92] is an example of the indirect volume rendering. Direct volume rendering renders volume data directly using methods such as ray casting, splatting, shear-warp, and 3D texture mapping. Here we choose splatting, one of the direct volume rendering methods for displaying the electrostatic field. We will detail reasons for this in Chapter 3.

Algorithm Overview

The basic element of direct volume rendering methods is the low-albedo volume rendering integral. It simulates the scattering of the radiance along the ray to the viewer. This integral requires a strict order of evaluation, either from back to front, or from front to back. This ordering requirement might lead to inefficient memory access patterns and thus inefficient rendering.

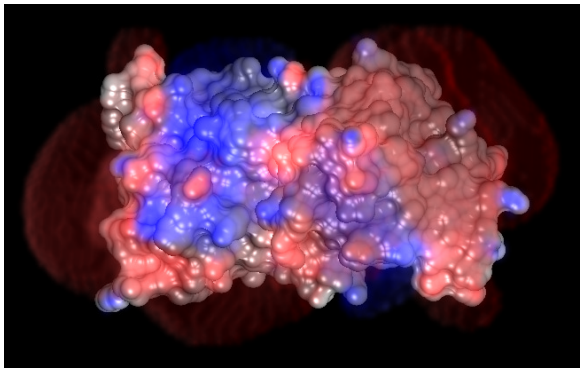
Since 3D electrostatic potential around molecules is generally a smoothly varying scalar field, we exploit this high coherence in the electrostatic field to improve the rendering efficiency of the traditional splatting algorithm by designing an order-independent splatting algorithm.

Our algorithm pre-computes the accumulated volume shadowing and transparency factors and stores them using spherical harmonics. At run time, we synthesize images through order-independent traversal of the 3D field data using pre-computed data and achieve better rendering speed. Details of this algorithm and its implementation are given in Chapter 3.

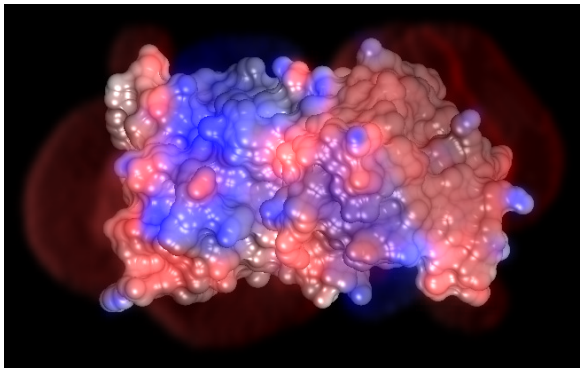
Results

We show our results for two real molecules. The first dataset is superoxide dismutase (SOD) enzyme with 2196 atoms. Our second dataset is a 10585 atom ion-channel on the outer membrane of the Escherichia coli (Ecoli) bacterium molecule.

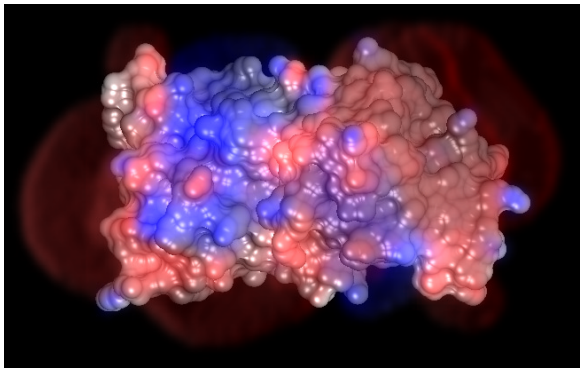
Figure 1.2 compares the images generated using raycasting, our order-independent splatting, and axis-aligned splatting [145]. The three images are largely similar in



(a) Raycasting (6.297 seconds)



(b) Axis-aligned splatting (3.156 seconds)



(c) Our splatting (1.397 seconds)

Figure 1.2: Comparison of raycasting, our splatting, and axis-aligned splatting of electrostatics on SOD dataset (red is for negative potential, and blue is for positive potential)

Dataset	Volume size	Image size	Rendering time (seconds)		
			Ray casting	Regular splatting	Our splatting
SOD	128^3	512×512	6.297	3.156	1.397
Ecoli	256^3	512×512	14.593	8.329	3.967

Table 1.2: Results on SOD and Ecoli membrane channel

visual quality while our method is significantly faster. It takes us 1.397 seconds to generate a 512×512 image for a 128^3 grid data, compared to 3.156 seconds required by regular splatting, and 6.297 seconds used by raycasting. A similar conclusion also holds for Ecoli dataset, for which we use a 256^3 grid data. The results are summarized in Table 1.2.

1.3.2 Variable Precision Representation

As shown in previous sections, compact data representation is essential for fast access and manipulation. Protein structures have limited dynamic range and accuracy. They are determined by using X-ray crystallography, NMR experiments, or gel electrophoresis. All of these methods have their accuracy limitations. Similarly, picking the right precision for data representation at the modeling stage will also help to interactively visualize large proteins.

Problem Definition

As the complexity of visualization datasets such as large proteins and their properties such as electrostatics and hydrophobicity have increased beyond the interactive rendering capabilities of the graphics hardware, new techniques have to be developed to reconcile the conflicting goals of visual accuracy and interactivity.

In addition, increase in the geometric complexity of the graphics datasets has far outpaced the increase in the display complexity. This has resulted in a bottleneck in transferring 3D vertex data from the CPU processor to the graphics processor.

The geometry operations for graphics primitives are currently carried out at full floating-point precision only to be converted to a fixed-point representation during the rasterization phase. Such high accuracy during geometry transformation and lighting stages sometimes exceeds even the display accuracy and thus causes several bits worth of unnecessary precision computation.

Our idea is to relate the minimum (or optimum) number of bits of accuracy required in the input data to achieve a desired accuracy in the display output.

Algorithm Overview

To find the minimum number of bits of precision for input data, we first carry out a careful analysis of different kinds of errors in geometric transformation and lighting stages, assuming fixed-point data representation. In addition to representation error, error also arises from pipeline operations of two fixed-point numbers, such as addition, multiplication, division, vector dot product, square root operation, and

exponentiation evaluation.

We build an octree bounding volume hierarchy for efficient estimation of the projected size of different parts of an object. The idea is to find the minimum and maximum number of bits required for accurately rendering each bounding box. If the two numbers are equal, then all vertices within this box will need the same number of bits. Otherwise, we recurse lower in the octree hierarchy.

We use the reduced precision to optimize the pipeline operations by using Single Instruction Multiple Data (SIMD) parallelism in modern processors and reduce the precision even further by spatial-temporal coherence in frame-to-frame transformations and lighting.

Details of this algorithm and its implementation are given in Chapter 4.

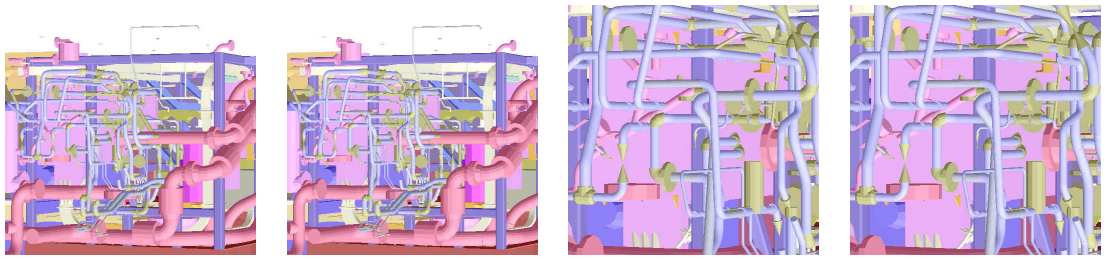
Results

We show here our results on polygonal datasets from several application domains including molecular, laser-scanned, mechanical CAD, and procedurally generated datasets. They are summarized in Table 1.3 and appear in Figure 1.3.

The results are obtained by using a Pentium II 400MHz PC with 128MB RAM and a Voodoo3 3500 graphics card. We achieve more than a factor of four speedup in all the datasets tested. From Table 1.3, we can see that under the pixel-level accuracy constraint required for the output, the maximum difference between the two methods is less than 0.00033 of the size of the bounding box for all the six datasets tested. Since we perform the worst-case analysis, our method actually delivers 2 to 3 sub-pixel bits of accuracy.

Model		Bunny	DHFR	Dragon	Venus	AMR	Buddha
Size (triangles)		69K	145K	202K	268K	376K	1087K
Floating Point	Rendering(seconds)	0.586	1.28	1.726	2.278	3.131	9.351
	Precision(bits/vert coord)	32	32	32	32	32	32
Variable Precision	Rendering (seconds)	0.138	0.275	0.385	0.499	0.632	1.733
	Precision(bits/vert coord)	7.9	7.9	7.6	7.1	4.2	5.6
Speedup		4.25	4.65	4.48	4.57	4.96	5.40
e_{rms} (object space)		1.3e-4	1.3e-4	1.2e-4	1.2e-4	1.1e-4	1.2e-4
Max error (obj. space)		3.0e-4	3.1e-4	3.0e-4	2.9e-4	2.6e-4	3.1e-4
e_{rms} (image space)		8.5e-3	8.8e-3	8.7e-3	6.0e-3	8.4e-3	7.0e-3

Table 1.3: Results from rendering at varying precisions



(a) Floating Point (32 bits/vert. coord.)

(b) Variable Precision (4.2 bits/vert. coord.)

(c) Floating Point Closeup

(d) Variable Precision Closeup

Figure 1.3: Auxiliary Machine Room (376K triangles) rendered in Variable Precision

1.3.3 Efficient Display of Dynamic Proteins

In this section, we develop and extend various rendering techniques for efficient display of time-varying molecular data.

Problem Definition

The existing acceleration techniques work by either reducing the number of graphics primitives to be rendered, such as multi-resolution rendering and visibility-based

culling, or by improving the memory bandwidth efficiency by better organization of the data (for example triangle strips and triangle fans). These techniques work by a pre-analysis of data with the design of clever data structures for efficient run-time access. Although they have achieved impressive results on static data, it is non-trivial to adapt the above techniques to time-varying datasets. There is little prior art for accelerating the rendering of time-varying datasets.

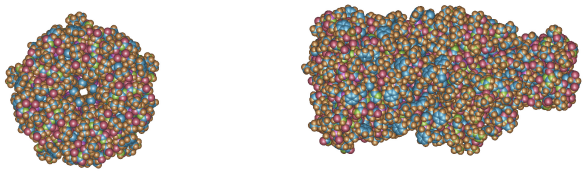
We try to address the problem of interactive rendering and visualization of large time-varying protein datasets. Our goal is to use no pre-processing and little memory overhead.

Algorithm Overview

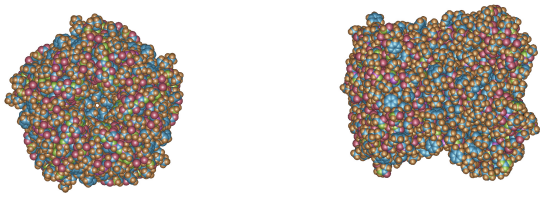
The main idea is to only display the visible parts of the visible atoms of the protein data at the proper resolution and precision.

We start by loading the list of atoms with their 3D positions for current time frame, and sort them according to their distance from the viewer using a quick-sort algorithm. Next we determine the visibility of each atom by using our conservative visibility-based culling algorithm. We use multi-resolution techniques to decide the appropriate number of triangles to represent the spherical atoms. We also decide the necessary precision for vertex data from display resolution specification. For the triangles that survive the back-face culling phase we generate triangle strips and compute illuminated color. Finally, we send the triangle strips and triangle fans with appropriate precision to the graphics card for rasterization and display.

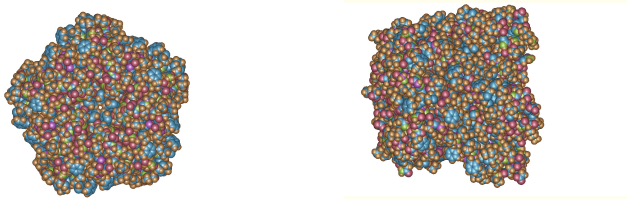
Details of this algorithm and its implementation are given in Chapter 5.



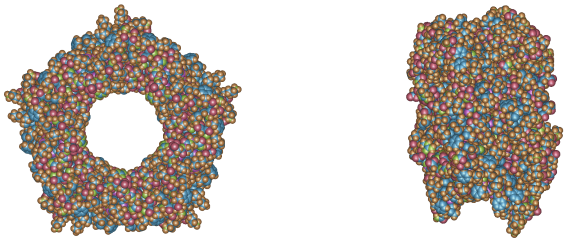
(a) Top-view of frame 0 (b) Side-view of frame 0



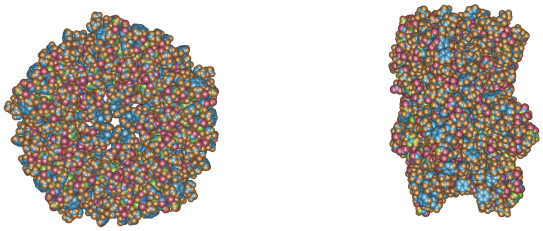
(c) Top-view of frame 50 (d) Side-view of frame 50



(e) Top-view of frame 100 (f) Side-view of frame 100



(g) Top-view of frame 150 (h) Side-view of frame 150



(i) Top-view of frame 200 (j) Side-view of frame 200

Figure 1.4: Top and side views of five stages of Escherichia coli mechanosensitive channel showing its opening and closing

Results

We have applied our approach to ion-channel studies, an area of biophysics with applications in neurobiology, pharmaceutical research, and many other branches of biomedical science.

EcoMscL complex shown in Figure 1.4 consists of 10585 atoms. We display the ion-channel transition process as a two hundred frame animation of the large-conductance mechanosensitive channel MscL as it transitions from the closed to the open state. We have achieved more than 32 frames per second (fps) rendering speed on this time-varying dataset, each frame of which consists of 1.3 million triangles. Our approaches are about four times faster than VMD (version 1.8.2) with the same image quality.

The techniques we develop for visualization of protein properties, such as efficient data representations and rendering from compressed data can also be applied to a broader range of graphics rendering problems. As an example, we next show how to adapt them to render the translucent materials interactively.

1.4 Interactive Rendering of Subsurface Scattered Reflectance Field

Interactive photorealistic rendering remains one of the primary goals of Computer Graphics. To achieve this, it is necessary to correctly and efficiently simulate the interaction of light with matter. As an example, accurate modeling of the scattering of light inside objects is crucial for rendering translucent materials such as skin,

milk, marble, clouds, and snow. Previous methods for subsurface scattering were memory-intensive and computationally expensive to render.

1.4.1 Problem Definition

Modeling the interaction of light with objects is an exciting, but difficult task. The widely used 4-dimensional (4D) bi-directional reflectance distribution functions (BRDFs) are inadequate to simulate the appearance of translucent materials. More general 8D bi-directional scattering surface reflectance distribution functions (BSSRDFs), special cases of surface reflectance field, are necessary. It is both memory and computation expensive to render the 8D BSSRDFs for translucent materials.

Our goal is to develop a $O(N)$ (N is the number of vertices) run-time algorithm with minimal storage requirements. We have achieved this goal with better mathematical representation of the scattered 8D reflectance field. We reduce the complexity and dimension of the problem by decomposing the reflectance field into non-scattered and scattered reflectance fields. While the non-scattered reflectance field can be described by general 4D BRDFs, we show that the scattered reflectance field can also be represented by a 4D field through pre-processing the neighborhood scattering radiance transfer integrals.

1.4.2 Algorithm Overview

Our algorithm consists of pre-processing and real-time rendering stages.

In the preprocessing stage we build subsurface scattering neighborhood information, which includes all the vertices within effective scattering range from each

vertex. We then modify the traditional local illumination model into a run-time two-stage process. The first stage involves computation of reflection and transmission of light on surface vertices. The second stage bleeds in scattering effects from a vertex's neighborhood to generate the final result. We then merge the run-time two-stage process into a run-time single-stage process using pre-computed integrals, and reduce the complexity of our run-time algorithm to $O(N)$, where N is the number of vertices. The selection of the optimum set size for pre-computed integrals is guided by a standard image-space error-metric. Furthermore, we compress the pre-computed integrals using spherical harmonics. We compensate for the inadequacy of spherical harmonics for storing high frequency components by a reference points scheme to store high frequency components of the pre-computed integrals explicitly.

Details of this algorithm and its implementation are given in Chapter 6.

1.4.3 Results

The results of using our approach are summarized in Table 1.4 and in Figure 1.5.

As one can see from Table 1.4, our scattering model can generate subsurface scattered images within a few tenths of a second for a model with over one million triangles, and achieve interactive frame rates for objects with less than $300K$ triangles. We compute the BSSRDF for all the model vertices as in Table 1.4. Our algorithm has an effective $O(N)$ complexity (N is the number of vertices), with small constant factors. The extra storage for pre-computed integrals is less than 28 bytes per vertex. Figure 1.5 shows increasing subsurface scattering effects on a horse model from left to right.

(a) Without Scattering (181 fps) (b)(c)(d) With scattering (79.1 fps)

Figure 1.5: Rendering the Horse model with subsurface scattering increasing from left to right (14,521 vertices with 10% vertices in $N(x_o)$ at (b), 20% vertices in $N(x_o)$ at (c), and 30% vertices in $N(x_o)$ at (d))

Model Name	No. of Vertices	No. of Triangles	No. of ref pts	Extra storage (Bytes/vert)	Compression ratio by using ref pts	Frame rate (fps)
Horse	14,521	29,054	1,034	27	7.4	79.1
Venus	42,656	90,044	2,827	26	7.7	27.3
Santa	75,781	151,558	3,458	22	9.1	14.6
Teapot	150,510	292,168	5,176	20	10.0	8.6
Dragon	437,645	871,414	10,285	18	11.1	2.7
Buddha	543,652	1,087,716	12,330	18	11.1	2.4

Table 1.4: Total rendering times for our approach

1.5 A Guide to Chapters

The rest of this dissertation is organized as follows.

In Chapter 2, we give an overview of molecular electrostatics and the resulting Poisson-Boltzmann Equation (PBE). We then describe our approach for constructing and solving PBE on a irregular grid whose resolution is based on the importance to the solution. We present an overview of a few implementation details and conclude with our results.

We describe in Chapter 3 our order-independent splatting algorithm for visualization of the generated electrostatic field. We give details of pre-computation and spherical-harmonic compression of the transparency functions.

Chapter 4 gives the details of view-dependent variable precision data representation, includes error analysis for different kinds of pipeline operations, and view-dependent precision control that takes advantage of spatio-temporal coherence. The results show the effectiveness of our algorithm.

Chapter 5 explains our time- and memory-efficient algorithms for the display of dynamic proteins, including the development of new techniques as well as the extension of current techniques.

In Chapter 6, we discuss dimension reduction and compression of subsurface scattered reflectance field. We describe algorithm for interactive rendering of the compressed illumination data, followed by results.

We conclude in Chapter 7 with directions for further work.

Chapter 2

Efficient Solution of Poisson-Boltzmann Equations for Molecular Electrostatics

Electrostatic interactions are of central importance for many biological processes [86, 133]. Experiments have shown that electrostatics influences nearly all biochemical reactions, such as macromolecular folding and conformational stability. Electrostatics also determines the structural and functional properties of biological samples, such as their shapes, binding energies, and association rates.

Molecular modeling packages [69] have invested significant effort in correctly and efficiently modeling the electrostatics to simulate the static structure and binding energy, in addition to modeling user-defined conformations [84] or trajectories [86]. The successful modelling of electrostatics has great practical, as well as, theoretical importance, for structure-based drug design and protein folding.

There are two ways to model the electrostatic properties of biological samples – quantum mechanical methods and classical electrostatics. Quantum mechanical methods are more accurate, but due to their immense computational demands, can only be applied to small molecules. Thus the application of quantum mechanical methods to large molecules, such as the ones we consider here, is currently not possible for real-time systems.

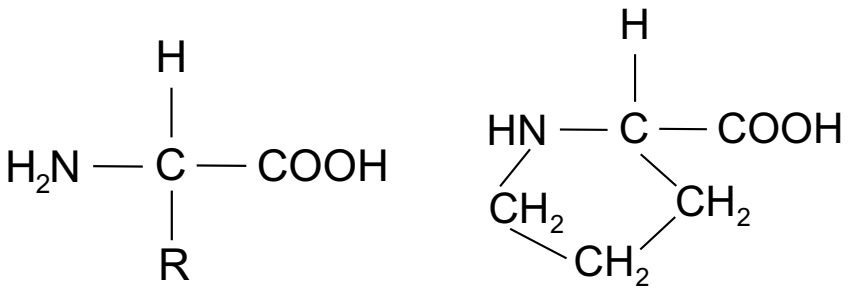
Classical electrostatic interactions are modeled as the interactions between partial atomic charges (also called net atomic charges). Partial atomic charges arise since electronegative elements, such as Oxygen, attract electrons more readily than elements such as Hydrogen. This give rise to an unequal distribution of charges in a molecule. The electrostatics of molecules depends not only on their 3D structures and charge distributions, but also on their environment. Biological processes occur in aqueous solution, so solvent plays an important role in determining the electrostatics of the solute molecules. Solvent properties are usually described in terms of average values. Thus, instead of treating each solvent atom explicitly, we treat them as a continuum with average properties. Only the most important solute molecules are treated explicitly [65].

In this chapter, I will first give a brief overview of proteins. I will also briefly review the theoretical foundation of molecular electrostatics, and work related to its solutions. Then I will describe our new approach [55] for efficiently solving the problem based on the observation that the accuracy and stability of the solution is quite sensitive to the boundary layer between the solvent and the solute. I will show the results of using our approach.

2.1 Proteins

Proteins are long chains of linked amino acids. There are 20 naturally-occurring amino acids. Each amino acid consists of a central carbon atom to which are attached a hydrogen atom, an amino group (NH_2), a carboxyl group ($COOH$), and a

distinguishing *sidechain* R (Figure 2.1(a)). R can be as simple as a single hydrogen atom, or it can be a long chain consisting of carbon, nitrogen, sulfur, oxygen, and hydrogen atoms. One of the 20 amino acids, proline, is special in that it has a bond between the sidechain R and the nitrogen atom (Figure 2.1(b)). Two amino acids adjacent in a protein chain are bonded by a covalent linkage, which is called a *peptide bond*. The chain of amino acids is also known as a *polypeptide*.



(a) A typical amino acid (b) Proline

Figure 2.1: Amino acids

The various amino acids can be characterized as hydrophilic or hydrophobic based on the interactions of their sidechains with water. Thus all amino acids that have aliphatic hydrocarbon sidechains are hydrophobic, and all amino acids that have polar atoms such as oxygen are hydrophilic. The book by Brande and Tooze [16] gives more complete information on protein structures.

In addition to protein structures, the protein properties can be broadly divided into bonded and non-bonded properties. The bonded properties include bond length, bond angle, dihedral angle, and proton donor/acceptor distributions (ability to form hydrogen bonds). Non-bonded interaction properties include van der Waals potential, electrostatic force, and shape-based interactions. Amongst these proper-

ties, protein electrostatics is one of the most challenging one as we discuss in detail next.

2.2 Fundamentals of Molecular Electrostatics

The current trends in real-time molecular electrostatics follow the principles of classical electrostatics, explicitly treating each atom in the protein molecule and each ion in the surrounding solution. The solvent is treated as a continuum.

2.2.1 Electrostatics in Uniform Dielectric Medium – Poisson Equation and Coulomb's Law

Electrostatics has a simple form when all the charges and the field considered are in a uniform dielectric medium, including vacuum. The electrical potential then satisfies the Poisson equation [70]:

$$\nabla^2\phi(\overrightarrow{r}) + \frac{4\pi\rho(\overrightarrow{r})}{\epsilon} = 0$$

where $\phi(\overrightarrow{r})$ is the electrostatic potential, $\rho(\overrightarrow{r})$ is the charge density, and both $\phi(\overrightarrow{r})$ and $\rho(\overrightarrow{r})$ are functions of position. Here the dielectric constant, ϵ, is independent of the position in a uniform media.

As an example, the electric potential field generated by a point charge is given by Coulomb's Law [70]:

$$\phi(r) = \frac{q}{\epsilon r}$$

where the point charge q is assumed to be at the origin and r is the distance from the origin.

Here the linear superposition rule holds and the electric potential field generated by a set of point charges is the summation of the fields generated by each point charge [70]:

$$\phi(\overrightarrow{r}) = \sum_{i=1}^{n} \phi_i(\overrightarrow{r}) = \sum_{i=1}^{n} \frac{q_i}{\epsilon \left| \overrightarrow{r} - \overrightarrow{r_i} \right|}$$

where n is the number of point charges, q_i is the charge and $\overrightarrow{r_i}$ is the position vector of point charge i.

If instead of assuming point charges, one assumes continuously distributed charges the superposition rule still holds with the summation changing to an integral:

$$\phi(\overrightarrow{r}) = \iiint \frac{\rho(\overrightarrow{r'})}{\epsilon \left| \overrightarrow{r} - \overrightarrow{r'} \right|} d\overrightarrow{r'}$$

where $\rho(\overrightarrow{r'})$ is the charge density.

2.2.2 Electrostatics in Nonuniform Medium with Environmental Response – Poisson-Boltzmann Equation

The Poisson equation given in the previous subsection assumes a uniform medium without the environmental response. If the dielectric ϵ varies through space, then we arrive at a general form of the Poisson equation:

$$\bigtriangledown[\epsilon(\overrightarrow{r})\nabla(\phi(\overrightarrow{r})] + 4\pi\rho(\overrightarrow{r}) = 0$$

where $\epsilon(\overrightarrow{r})$ is a function of position. Normally the solute is treated as a uniform medium with a low relative dielectric of about $2 \sim 4$. The solvent is also treated as a uniform medium with a relative dielectric of about 80 [42].

The environmental response consists of three physical processes that screen the effects of charge: (a) electronic polarization, (b) reorientation of permanent

dipole in polar materials, and (c) redistribution of charges, such as mobile ions. Combining these factors, we get a general form for the molecular electrostatics – the Possion-Boltzmann Equation (PBE):

$$\bigtriangledown[\epsilon(\overrightarrow{r})\nabla(\phi(\overrightarrow{r})] - \kappa'^2(\overrightarrow{r})\sinh[\phi(\overrightarrow{r})] + 4\pi\rho(\overrightarrow{r}) = 0$$

where κ' is the modified Debye–Hückel parameter defined as:

$$\kappa'^2 = \frac{8\pi N_a e^2 I}{1000kT}$$

where N_a is Avogadro's number, e is the electron charge, k is the Boltzmann constant, T is the absolute temperature, and I is the ionic strength of the bulk solution. The variables ϕ, ϵ, κ', and ρ are all functions of the position vector $\overrightarrow{r}$. The general form of the PBE above incorporates electronic and dipole polarization through ϵ and ion-screening through κ'.

If there are no highly-charged molecules and ionic strengths are low, we can make an approximation to linearize the sinh term:

$$\sinh[\phi(\overrightarrow{r})] \approx \phi(\overrightarrow{r})$$

and then the general PBE simplifies to the linear PBE:

$$\bigtriangledown[\epsilon(\overrightarrow{r})\nabla(\phi(\overrightarrow{r})] - \kappa'^2(\overrightarrow{r})\phi(\overrightarrow{r}) + 4\pi\rho(\overrightarrow{r}) = 0$$

If there are no mobile ions present in the system, the modified Debye–Hückel parameter κ' will be equal to zero, and PBE reduces to the general Poisson equation.

2.3 Previous and Related Work

Analytic solution for linear PBE is only possible for simple cases [134]. In most cases, however, an analytical solution does not exist, and numeric methods have been developed to solve linear [143] or nonlinear PBE. Among them the finite difference method (FDM) [143] is the most widely used. In finite difference methods the molecule is mapped onto a 3D grid. Partial atomic charges are assigned to grid points and the electrostatic potential at each grid point is calculated using the finite difference approximation of the PBE. The accuracy of the results is highly dependent on grid spacing, while the computational cost increases steeply with the number of grid points. One approach to reduce the cost is called *focusing* [46], in which the mesh of the grid is reduced only in the vicinity of ionizable groups of particular interest with potentials from coarser grids used as initial guesses. A more powerful approach is the *multi-grid method* [103], in which the solution on a given grid (generally the finest grid), is obtained by iterating over a hierarchy of coarser grids. The key advantage is that the accuracy of the solution is iteratively improved by solving the problem on the coarser grids where the computational cost is low with infrequent visits to the finer grids where the computational cost is high. One drawback of the multi-grid method is that, while it works gracefully for linear PBE, the solution might not converge when applied to non-linear PBE.

The finite difference algorithms using a regular grid, though quite successful, have several shortcomings. First, their computational cost is proportional to the cube of the grid size, which makes it very hard to increase the resolution. Sec-

ond, they do not scale well. For fixed grid size, the resolution will decrease as the dataset becomes bigger. Third, the low resolution approach, even with multi-grid refinements, easily introduces visual artifacts at the visualization stage.

The adaptive space-subdivision approach [10,63] has been used to address the high cost of using a regular grid. This approach increases the accuracy of the solution by explicitly giving a higher spatial resolution to the solvent-solute boundary region. However, since the adaptive space-subdivision approach does not start from an analytical definition of the solvent-solute boundary (the solvent-accessible surface), it tends to over-subdivide around the boundary region. This over-subdivision results in two drawbacks. First, it increases the number of grid points and therefore the time for each iteration of the PBE solver. Second, it increases the number of iterations to reach a desired convergence threshold since a greater number of closely-spaced points near the boundary increases the time to propagate the solvent-solute boundary effects.

Our new approach solves the linear and non-linear PBE on an irregular grid. It has the advantage that the PBE-solution-sensitive boundary layer is constructed analytically. Before we present details of our approach, we briefly give an overview of the finite difference method(FDM).

2.4 Finite Difference Method for PBE

In FDM the molecule and a region of the surrounding solvent are mapped onto a 3D grid. Each grid point represents a small region of either the molecule or the solvent.

Values are assigned at each point for the charge density, dielectric constant, and ionic strength parameters in the PBE. With a sufficiently fine grid scale, variation in the dielectric response can be represented at the atomic resolution. The electrostatic potential at each grid point is calculated using the finite difference approximation of the PBE:

$$\phi_i^{new} = \frac{\sum \epsilon_{ij}\phi_j + 4\pi q_i/h}{\sum \epsilon_{ij} + \kappa_i'^2 h^2[1 + \phi_i^2/3! + \ldots + \phi_i^{2n}/(2n+1)! + \ldots]}$$

where the non-linear term is represented as an infinite series, which equals 1 for linear PBE, h is the grid spacing in Å, ϕ_i is the electrostatic potential at the central grid point, q_i is the charge at this grid point, and the summations are over the six neighboring grid points (j = 1..6) [143].

To assign charge density, dielectric constant, and ionic strength parameters to the grid points, we first define the molecule-solvent boundary, which is the smooth solvent-accessible surface [87], using a probe radius for the water molecule (assumed 1.4 Å).

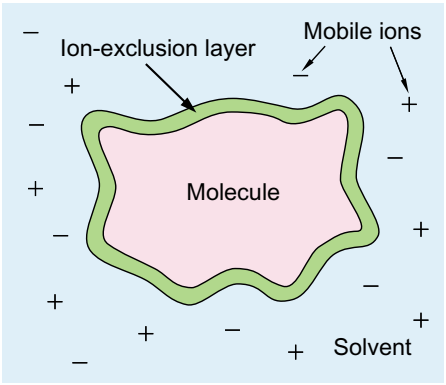


Figure 2.2: 2D view of the electrostatic model (based on [64])

Figure 2.2 shows the two-dimensional view of the electrostatic model. The

grid points within the molecule are normally assigned a uniform dielectric of $\epsilon_1 = 2$ as an approximation to the high-frequency dielectric constant of organic liquids. All grid points within the solvent region and the ion-exclusion layer are assigned a dielectric constant $\epsilon_2 = 80$. The modified Debye–Hückel parameter κ' is zero inside the molecule and at the ion-exclusion layer, where there are no mobile ions; κ' is non-zero in the solvent region. With these considerations we get following forms of the PBE:

$$\begin{cases} \epsilon_1 \nabla^2 \phi(\overrightarrow{r}) + 4\pi\rho(\overrightarrow{r}) = 0 & \text{inside molecule} \\ \epsilon_2 \nabla^2 \phi(\overrightarrow{r}) + 4\pi\rho(\overrightarrow{r}) = 0 & \text{at ion-exclusion layer} \\ \epsilon_2 \nabla^2 \phi(\overrightarrow{r}) - \kappa'^2(\overrightarrow{r}) \sinh[\phi(\overrightarrow{r})] + 4\pi\rho(\overrightarrow{r}) = 0 & \text{within solvent} \end{cases}$$

The numerical solvers for partial differential equations using FDM initialize the grid boundary values using various methods. The boundary values, once estimated, do not change. In our case, a trivial possibility is to set the potential at the grid boundary nodes to be zero. We use the analytical approximation obtained using Debye-Hückel potentials [46] that are accurate if the solution grid is large enough relative to the size of the molecule.

2.5 Our Approach

From the discussion above, we find that the solvent-accessible surface boundary layer is critical to the accuracy of the FDM solution of the PBE. This is because that all the atomic charges lie within the molecule, and also because there are large

differences in the dielectric constants between the two regions separated by the molecular surface boundary layer. As several biological processes occur at or near the molecular surface, a high accuracy for the solution to PBE close to this boundary layer is critical. Not coincidentally, the stability and accuracy of numerical methods also depend largely on the discretization of the grid in this region. To the best of our knowledge, no previous algorithm for solving PBE for molecules exists that builds the tetrahedral grid based on the solvent-accessible surface at the solvent-solute boundary. Our algorithm builds an adaptive tetrahedral space-decomposition about the solvent-accessible molecular surface and gives higher priority and resolution to the boundary region.

2.5.1 Analytical Solvent-accessible Surface

Previous methods to solve PBE approximate the solvent-accessible surface after building a 3D grid around the molecule. For each grid point, a binary marker indicates whether it is inside the molecule or inside the solvent. The solvent-accessible surface is then defined as passing between those grid points that have dissimilar markers. With such methods the accuracy of the solvent-accessible surface is limited to the grid resolution; the actual solvent-accessible surface points do not in general coincide with the grid points.

Several analytical solvent-accessible surface algorithms have been published [2, 7, 26, 77, 122, 141]. We analytically generate the solvent-accessible surface using the approach in [141] and then incorporate it in the 3D grid used for the solution of the PBE. Guaranteeing that the grid points at the solvent-solute boundary are

actually on the exact surface improves the accuracy and speed of the algorithm.

2.5.2 Distance-Field-based Tetrahedralization

The accuracy of the FDM solution to PBE depends on the ionic strength assignment, which is zero in the 2Å ion-exclusion layer from the solvent-accessible molecular surface, and constant outside. Therefore we need an accurate ionic-screening surface that is offset $2\mathring{A}$ outwards from the molecular surface. Our tetrahedralization algorithm is based on the distance field from the solvent-accessible molecular surface and can generate the ionic-screening surface as well as provide an adaptive space decomposition.

We use an odd/even scheme for splitting rectilinear and curvilinear grids into tetrahedra as done in [97]. We use a method similar to the one described in [45] to build a signed-distance map of the space that measures the distance of each grid point to the solvent-accessible surface.

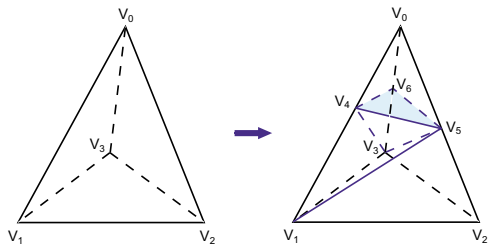


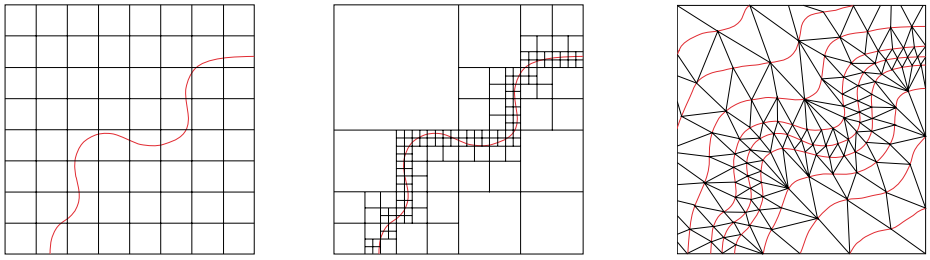
Figure 2.3: Marching-tetrahedra-based Iso-surfaces and Tetrahedral Grid Refinement

Next, we generate a sequence of iso-surfaces from the distance map using a tetrahedral variant of the Marching Cubes algorithm [92]. We use tetrahedra instead of cubes for simplicity and stability. We insert new grid points into the

3D grid such that they form surfaces at a fixed distance away from the real solvent-accessible surface. One case of the marching tetrahedra is shown in Figure 2.3. Here the processing of tetrahedron $V_1V_2V_3V_0$ generates triangle $V_4V_5V_6$, which splits the original tetrahedron into four new tetrahedra: $V_4V_5V_6V_0$, $V_4V_5V_3V_6$, $V_1V_5V_3V_4$, and $V_1V_2V_3V_5$.

2.5.3 Adaptive Tetrahedralization

Hierarchical space decompositions have a rich and long history of research [120,121]. In our case, adaptive tetrahedral space decomposition is driven by the twin goals of accuracy and efficiency of the PBE solution. We desire a finer grid near the solvent-solute boundary for accuracy and a sparser grid elsewhere for efficiency.



(a) Regular (b) Adaptive Subdivision (c) Interface-focused

Figure 2.4: Different Space Decompositions for PBE solvers

We depict a 2D version of our adaptive tetrahedral grid in Figure 2.4(c). Here the thick red curve represents the molecular surface, while the thin red curves are the iso-distance layers from the surface. Figure 2.4 clarifies the conceptual difference between our approach and the approaches using regular grids(Figure 2.4(a)) or octree-based space-subdivision scheme(Figure 2.4(b)). The advantage of our approach is that we refine the grid directly on the molecular surface and in regions

close to it. Regular grids are not adaptive and hence suffer from low accuracy, or high computational costs, or both. The adaptive octree-based subdivision scheme in most cases generates an excessively fine grid around the molecular surface to approximate it well. Our approach can adjust the resolution progressively and seamlessly based on the distance from the molecular surface.

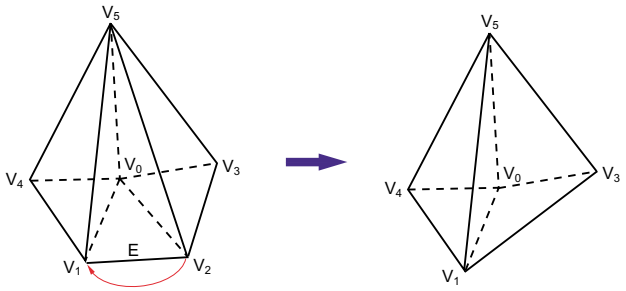


Figure 2.5: Edge Collapse for Tetrahedral Decimation

We achieve adaptive tetrahedral decomposition by using edge collapses. Multiresolution tetrahedral grid hierarchies have been built using bottom-up edge-collapses [22] or top-down longest edge bisections [50]. An edge collapse will decrease the triangle count on the iso-surfaces, as well as the tetrahedra count. We use a half-edge collapse scheme so that an edge will collapse to one of its vertices and no new vertices need to be generated. Each edge collapse decreases the vertex count by one, and decreases the triangle and tetrahedron count based on its local connectivity. As an example, Figure 2.5 shows the collapse of edge E that results in decimation of tetrahedron $V_0V_1V_2V_5$ and vertex V_2, while the tetrahedron $V_0V_2V_3V_5$ is changed to $V_0V_1V_3V_5$.

The finite difference method (FDM) computes the value at a grid point from

the values at its neighboring points (Section 2.4). During volume simplification we have to be careful not to simplify the volume into a state in which some grid points lose some of their neighbors necessary for FDM.

We also carefully avoid generating tetrahedra with negative volume (i.e., tetrahedra with a wrong orientation) and flipped triangles with reversed normals during the simplification process. In Figure 2.6, we show a 2D projection of one of these cases. Here the collapse of edge E results in grid point V_0 losing one of the six neighbors necessary for FDM. To avoid this we check and invalidate edge collapses that result in a new tetrahedron with three collinear vertices. In Figure 2.6, we get a new tetrahedron with collinear vertices V_0, V_3, and V_1 and therefore the edge E's collapse should not be allowed.

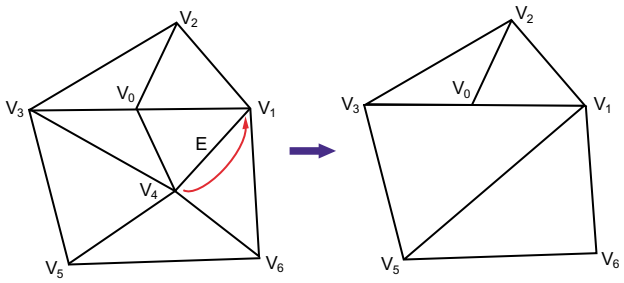


Figure 2.6: Invalid edge collapse causes V_0 to lose one of its neighbors necessary for FDM solver

Another constraint for edge collapses is to preserve the spatial grid's outer (volumetric) boundary. We do this to ensure that the total volume of all the tetrahedra in the grid does not change as a result of the edge-collapse-based simplification. If a candidate edge for collapse has one of its vertices on the grid boundary we collapse the edge to the boundary vertex. If the candidate edge for collapse has both of its

vertices on the grid boundary we carry out the collapse only if it will not result in a change in the total volume of all the affected tetrahedra.

2.5.4 Derivative Computation for Irregular Grids

The FDM solver for the PBE has to compute first and second derivatives of the 3D potential field at each grid point. The derivatives can be computed for regular grids by taking the finite differences between the potential value at each grid point with values at their six axis-aligned neighboring grid points. The regular structure of the regular grids makes this procedure straight forward. For irregular grids the situation is complicated by the fact that not only do the distances between grid points vary, but also the neighboring points are rarely axis-aligned.

We compute the derivatives of the potential at a grid point i by using the values at the vertices of the tetrahedra that are adjacent to i and include the principal axes from point i. As an example, consider the derivatives along x-axis for point V_0. First, we find the two tetrahedra that share the vertex V_0 extend towards positive and negative x-axis from point V_0 as shown in Figure 2.7. This can be done by a simple orientation test.

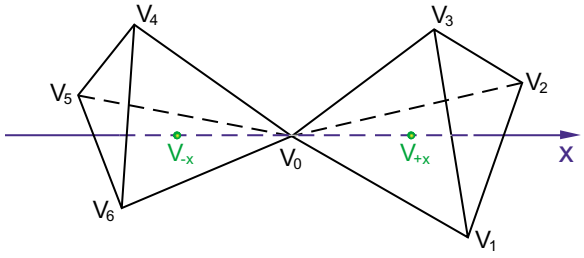


Figure 2.7: Neighboring tetrahedra of V_0 along X-axis

After we have identified the two tetrahedra we locate points V_{-x} and V_{+x}, that are equidistant from V_0 and along the x-axis. Let the distances between V_{-x} and V_0 and between V_{+x} and V_0 be h, then the second-order derivative of potential at V_0 can be approximated by:

$$\frac{\partial}{\partial x}[\epsilon(\overrightarrow{r})\frac{\partial}{\partial x}\phi(\overrightarrow{r})]\Big|_{V_0} \approx \frac{\epsilon(V_{+x})\phi'_x(V_{+x}) - \epsilon(V_{-x})\phi'_x(V_{-x})}{2h}$$

The first partial derivative of potential along x at V_{-x} and V_{+x} can be estimated by using a Taylor series expansion. We express potential values at vertices of each tetrahedron in terms of the value and derivatives at point V_{-x} and V_{+x}. As an example, $\phi'_x(V_{+x})$ can be estimated by solving the following system of four linear equations in four unknowns $(\phi(V_{+x}), \phi'_x(V_{+x}), \phi'_y(V_{+x}), \phi'_z(V_{+x})$:

$$\begin{cases} \phi(V_0) = \phi(V_{+x}) + \phi'_x(V_{+x})(-h) \\ \phi(V_1) = \phi(V_{+x}) + \phi'_x(V_{+x})\triangle x_1 + \phi'_y(V_{+x})\triangle y_1 + \phi'_z(V_{+x})\triangle z_1 \\ \phi(V_2) = \phi(V_{+x}) + \phi'_x(V_{+x})\triangle x_2 + \phi'_y(V_{+x})\triangle y_2 + \phi'_z(V_{+x})\triangle z_2 \\ \phi(V_3) = \phi(V_{+x}) + \phi'_x(V_{+x})\triangle x_3 + \phi'_y(V_{+x})\triangle y_3 + \phi'_z(V_{+x})\triangle z_3 \end{cases}$$

where $[\triangle x_i, \triangle y_i, \triangle z_i], i = 1, 2, 3$, is the vector difference between 3D positional vector of point V_i and V_{+x}:

$$[\triangle x_i, \triangle y_i, \triangle z_i] = \overrightarrow{r}(V_i) - \overrightarrow{r}(V_{+x})$$

The above equations can be solved analytically. With such a solution we can express the second-derivative of the potential at V_0 along x-axis as the linear-

weighted sum of the potentials at V_0 to V_6. We similarly compute the derivatives along y and z axes.

This method of computing the derivatives for irregular meshes will have the same degree of accuracy as the method used for regular grids because both use the first-order Taylor series expansion to connect the values at neighboring points.

2.5.5 GPU Solver for PBE on Regular Grids

The techniques described above are for efficient solution of PBE on CPU. In this section, we explore the possibility of mapping the solution of PBE to modern Graphics Processing Units (GPUs). The goal is to study, understand, and harness the GPU power for high-performance computing and scientific applications.

The recent rise in the capabilities and programmability of the GPUs such as the ATI Radeon and Nvidia GeForce FX (among others) has enabled them to rise above the threshold where they have now become powerful enough to be serious contenders to the CPUs as high performance computational engines for floating point intensive applications. The major advancements in recent GPU development includes:

Computing Power: The steady advances in semiconductor VLSI coupled with a deeply pipelined and SIMD architectures in modern GPUs have enabled a faster-than-Moore's law improvement in GPU performance. Since 1993 the performance of the GPUs has achieved an annual growth rate of 2.8 (versus the Moore's law's annual rate factor of 1.7). This growth rate is expected to be maintained for at least another five years. As an example, Nvidia GeForce FX 5900's performance

peaks at 20 gigaflops, the equivalent of a 10 GHz Pentium. GeForce FX 5900 has a four-way pixel pipeline architecture with two texture units per pipe (4×2) resulting in a fill-rate of 3.6 Gigatexels/second. Even better, the current GeForce 6800 has a 16-way parallelism in pixel pipelines and a six-way parallelism in vertex pipelines and delivers more than twice the floating point performance of GeForce FX 5900. GeForce 6800 has been released less than ten months after the release of GeForce 5900 and seems to be maintaining the annual growth rate factor of 2.8.

Programmability: Initial GPU programming models had to be programmed at the assembly-language level with no function calls, no branches, and no loops. Recent development of high-level programming languages (Cg from Nvidia and HLSL from Microsoft) have made the current-generation GPUs significantly easier to program. In addition, current GPUs allow the floating-point computation through the entire rendering pipeline. Together, the ease of programmability and floating-point representation have made GPUs attractive for a much wider set of computing applications.

Modern GPUs possess vertex and fragment (also called pixel) shaders. The fragment shader can be viewed as a stream processor. The processor executes the same kernel (fragment program) to produce each element (rasterized pixel) of an output stream (group of rasterized primitives). The output stream can be saved using texture memory and used as input (via texture fetches) for downstream kernels. Our design of a PBE solver on a GPU maps the appropriate data structures and algorithms into streams (textures) and kernels (shaders) respectively. The GPU has

two SIMD characteristics: (1) the same kernel is executed over all elements of a stream, and (2) processor instructions can operate on wide data types, i.e., 4-tuples of floating point values.

Mapping 3D grid data to 2D textures

The first step of our GPU solver is to map the 3D grid data into streams, which are textures for GPU fragment shader. We have used a regular tiled mapping from 3D to 2D for data values on the grid as shown in Figure 2.8. Each slice of the 3D grid is mapped onto a square region of the texture. Since both the length and the width of the texture area have to be a power of 2, there can be texels that have no grid points mapped on them.

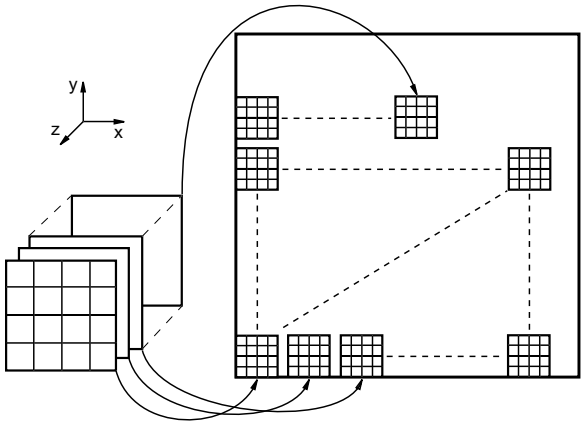


Figure 2.8: Mapping of a 3D grid onto a 2D texture

From Section 2.4, we know that the potential value at a grid point in a new iteration is computed from the weighted potential values of its six axis-aligned neighboring points from the previous iteration, together with the charge density at the

point as follows:

$$\phi_i^{new} = \frac{\sum \epsilon_{ij}\phi_j + 4\pi q_i/h}{\sum \epsilon_{ij} + \kappa_i'^2 h^2[1 + \phi_i^2/3! + \ldots + \phi_i^{2n}/(2n+1)! + \ldots]} \tag{2.1}$$

We store the potential values ϕ_i in a texture map. Each grid point i needs to use the weights for each of its six neighbors $\epsilon_{ij}, j = 1..6$, its charge density q_i and the modified Debye–Hückel parameter κ_i' from 3D grid to 2D textures. Unlike the potential values that change between iterations, these eight parameters are fixed for the whole solution process and thus can be stored in regular 2D textures. Since each texel on the texture map has 4 tuples for the four color channels, we only need two texture maps to store these eight parameter maps.

Fragment program and Render-to-texture

The highly pipelined and parallelized fragment shader applies the same kernel operations to each pixel on the processed fragment. For the iterative solution of the PBE, we render the potential values ϕ_i of one iteration to a texture object, and then use the texture as the input to the next iteration. This can be realized by creating two texture objects using front pBuffer (on Windows) and switch between them after each iteration. During each iteration, the fragment program will read the potential texture from the previous iteration, together with the two constant parameter textures, and generate the output potential texture object. Modern GPUs provide random access memory fetch into saved streams (textures in our case) and hide the latency of the random memory access so long as the bandwidth is not limited. So the access to the potential values of each grid point's neighbors is easily realized as

an offset from the grid point's current location in the potential texture map. The kernel function for each grid point is the one used in Equation 2.1. After the iterative process, we use glGetTexImage to read the contents of the potential texture object for the final results. The results of this approach appear in Section 2.6.3.

2.6 Results and Discussion

We present results on an analytical solvable case and compare them with the results by the well-known DelPhi system for computing molecular electrostatics. Our results show clear advantages of our algorithm over the standard DelPhi algorithm. We achieve better accuracy with less computation time. We then show our results on some real molecular datasets. We display the results by color coding smooth solvent-accessible molecular surfaces.

2.6.1 Results and Comparisons on an Analytical Solvable Case

Normally it is difficult (or impossible) to obtain analytical solutions to the PBE. In some special cases we may have analytical solutions to the linearized PBE. One example is that of a spherical molecule, with total charge q uniformly distributed on the surface, immersed in a solvent containing mobile ions, as shown in Figure 2.9.

The analytical solution to this special case is [64]:

$$\begin{cases} \phi(r) = \frac{q}{\epsilon_2 R}\left(1 - \frac{R\kappa}{1+\kappa a}\right) & \text{inside molecule} \\ \phi(r) = \frac{q}{\epsilon_2 r}\left(1 - \frac{r\kappa}{1+\kappa a}\right) & \text{ion-exclusion layer} \\ \phi(r) = \frac{qe^{\kappa a}}{\epsilon_2(1+\kappa a)} \cdot \frac{e^{-\kappa r}}{r} & \text{inside solvent} \end{cases}$$

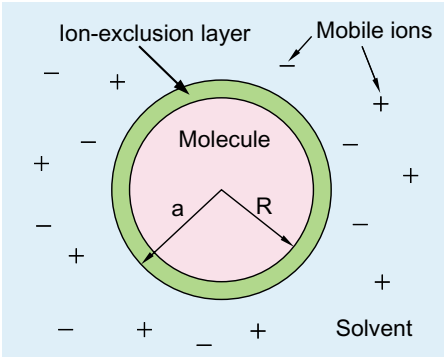


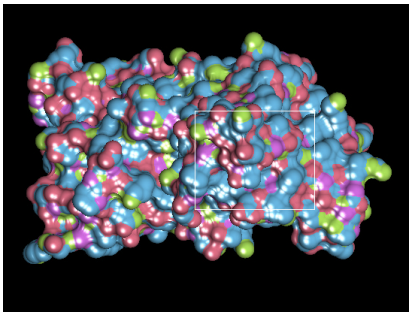
Figure 2.9: An analytically solvable case of PBE

The comparison of results with DelPhi (V.4) program for the analytically solvable case are shown in Table 2.1. The test consists of a spherical surface charge with a diameter of 27 Å and a positive charge of $20e$ (where e is the charge of an electron) which is immersed in a cubic solvent volume with each side 66 Ålong.

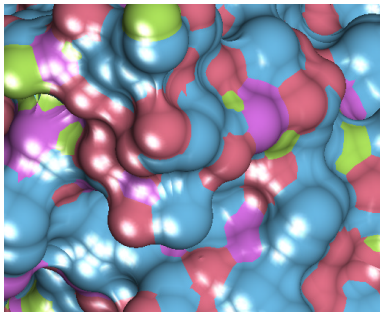
	DelPhi			Our Method
Grid size	67^3	133^3	199^3	N/A
# of pts	300,763	2,352,637	7,880,599	26,987
PSNR	8.17	19.1	25.1	27.7
Avg. error	30.88%	17.91%	13.27%	15.98%
PBE Time	0.31 sec	4.50 sec	20.09 sec	0.25 sec

Table 2.1: Comparison of our method with DelPhi

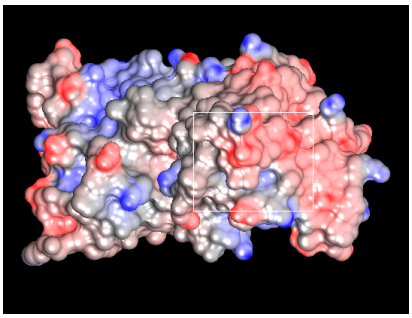
Our method needs many fewer points and less time to get the same accuracy. For example our method requires $27K$ points with 0.25 seconds to converge, instead of several million needed by DelPhi with several seconds to converge. For the same amount of time, our method is much more accurate. For instance, our approach has



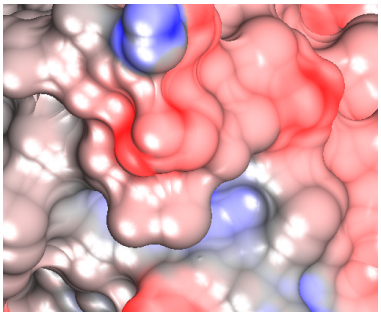
(a) Solvent-accessible surface



(b) Closeup of (a)

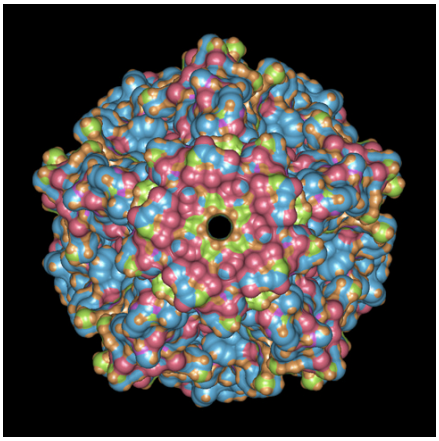


(c) Lighted on-surface potential

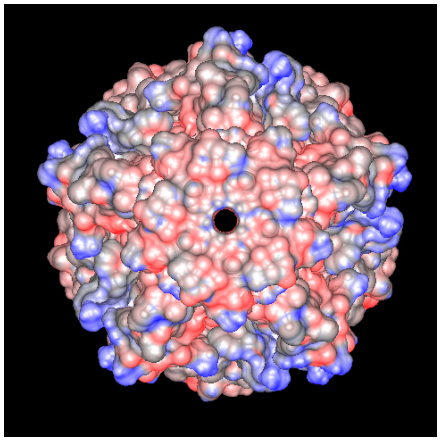


(d) Closeup of (c)

Figure 2.10: Electrostatics on SuperOxide Dismutase (SOD) dataset (red is for negative potential, and blue is for positive potential)



(a) Solvent-accessible surface



(b) Lighted on-surface potential

Figure 2.11: Electrostatics on Ecoli membrane channel (red is for negative potential, and blue is for positive potential)

a 15.98% error compared to an error 30.88% for DelPhi in 0.31 seconds.

2.6.2 Results on Real Molecules

We now show our results for some real molecules. The first dataset is superoxide dismutase (SOD) enzyme, which consists of 2196 atoms. Our second dataset is a channel on the outer membrane of the Escherichia coli (Ecoli) bacterium molecule [131], which consists of 10585 atoms. The results are shown in Figures 2.10 and 2.11.

Figures 2.10(a) and 2.11(a) display the smooth solvent-accessible surfaces of SOD and Ecoli membrane channel using the SURF algorithm [141]. Electrostatic potential is traditionally displayed on molecular surfaces [117]. Figures 2.10(c) and Figure 2.11(b) display the electrostatic potential on the surfaces, with red for negative and blue for positive potential; both use the potential information to modulate lighting color with grey denoting neutral potential. Figures 2.10(b) and (d) are closeups of Figures 2.10(a) and (c), respectively.

2.6.3 Results of PBE solver on the GPU

We have tested our GPU electrostatics solver on Nvidia GeForce FX5900 Ultra graphics card with 256MB on-chip memory. The results are compared with those obtained on a 2GHz Pentium 4 PC running Windows 2000 with 2GB RAM.

Table 2.2 compares the CPU and GPU performance in solving the PBE on SOD dataset. One can see that GPU has a big advantage over CPU in solving the PBE. For this application, GPU is about twice as fast as CPU for a grid size of 64^3, but is an order of magnitude faster than CPU for a grid size of 128^3. Both CPU

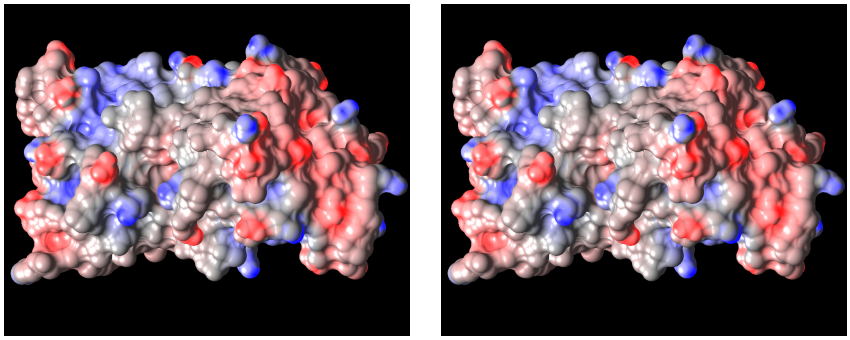
Grid size	CPU time/iteration (msec)	GPU time/iteration (msec)
64^3	60.5	26.9
128^3	472	39.1

Table 2.2: Comparison of CPU and GPU solution of PBE

and GPU take the same number of iterations to converge, which is 155 for a grid size of 64^3 under the convergence threshold of 0.1% of the largest potential value. Similar to the observation by Hillesland and Lastra [60], we find that the floating point operations on GPU are less precise than those on CPU. We have observed that for our datasets, the maximum error thus introduced can be as high as 5.8% for the grid size of 64^3. We have not gone beyond grid size of 128^3 due to the 256MB GPU memory limitation. With the rapid advances in GPUs, we anticipate that the advantage of using GPU will be even higher for larger grid sizes when the support for larger on-chip memory becomes available in the near future. Figure 2.12 visually compares the CPU and GPU solutions of PBE on SOD dataset.

2.7 Conclusions

Our new algorithm uses compact and efficient 3D data representation to achieve faster convergence and better accuracy for computing electrostatic potentials of large molecular datasets. Our method gives higher priority and resolution to the solution-sensitive region to improve the accuracy and accelerate convergence rates. The advantage of our algorithm in solving partial differential equations directly from



(a) CPU solution (b) GPU solution

Figure 2.12: Electrostatics on SuperOxide Dismutase (SOD) dataset solved by CPU and GPU respectively (red is for negative potential, and blue is for positive potential)

the geometrical point of view gives it a broad range of possible applications in other scientific domains as well.

With the advances presented here our electrostatic computation methods are now almost fast enough to be used in interactive molecular docking experiments with interleaved computation and visualization of large molecules.

Chapter 3

Order-independent Splatting for Visualization of Electrostatics

The 3D electrostatic potential field computed in Chapter 2 is a scalar field defined over a volume. Real-time visualization of the electrostatic potential is important for better understanding of the relation between the structure and function of proteins and thus their applications in protein folding and molecular docking.

Traditional electrostatics visualization methods either display only the subset of the electrostatic field that lies on the solvent-accessible molecular surface [117] or visualize the iso-value surfaces using marching cubes algorithm [92]. We believe that it will be more informative to visualize electrostatics using direct volume rendering algorithms, such as raycasting [33, 90], splatting [100, 145], shear-warp [85], or 3D texture-mapping hardware-based approaches [18, 144].

Most volume visualization datasets are characterized by an underlying structure, usually with meaningful boundaries. For instance, medical imaging datasets embody connected tissues and bones. However, 3D electrostatic potential around molecules is generally a smooth-varying scalar field. We exploit this high coherence in the electrostatic field to improve the rendering efficiency of the traditional splatting algorithm. We have developed an order-independent splatting algorithm to

speedup the rendering by pre-computing the accumulated transparency information and storing them at each grid point using spherical harmonics.

3.1 Previous and Related Work

Volume visualization methods can be categorized into indirect and direct volume rendering methods. Indirect volume rendering first converts the volume data into a polygonal iso-surface representation and then displays the surface. The Marching Cubes algorithm [92] is a representative of indirect volume rendering. Direct volume rendering displays volume data directly using methods such as ray casting, splatting, shear-warp, and 3D texture mapping. *Ray casting* methods [5, 33, 90] cast rays from viewer into data volume, composite along the ray, and at each step sample the volume with filtering (typically tri-linear) from neighboring voxels. *Splatting* algorithms [100, 145] treat data volume as an array of 3D reconstruction kernels. Each kernel generates a splat (or footprint) on the screen and the final image is a composite of all splats. The *shear-warp* method [85] traverses both image and object space at the same time. The shear-warp method renders the run-length-encoded volume using a ray-casting-like scheme by shearing the volume such that the rays are perpendicular to the volume slices. Three-dimensional *texture-based* volume rendering loads the volume into 3D texture memory and rasterizes and composites the slices parallel to the image plane in a back-to-front order.

We choose direct volume rendering for displaying the 3D electrostatic fields for a couple of reasons. First, the potential field of molecular datasets has positive

and negative values mingled together in several disconnected components. This gives the iso-surface rendering a fragmented appearance and makes direct volume rendering quite attractive. Second, the potential field normally has no sharp features and thus anti-aliasing in direct volume rendering does not lead to artifacts. In the following sections, we present an order-independent splatting algorithm to render the 3D potential field by pre-computing the accumulated shadowing and transparency information for each voxel.

3.2 The Volume Rendering Integral

The basic element of direct volume rendering methods is the low-albedo volume rendering integral [15, 96]. It computes the amount of light $I(\overrightarrow{x}, \overrightarrow{r})$ of wavelength λ coming from ray direction $\overrightarrow{r}$, received by point $\overrightarrow{x}$ as:

$$I_\lambda(\overrightarrow{x}, \overrightarrow{r}) = \int_0^L C_\lambda(s(u))\tau(s(u)) \exp\left(-\int_0^u \tau(s(v))\mathrm{d}v\right)\mathrm{d}u$$

Here s is the scalar value of the volume data, C_λ is the light of wavelength λ reflected at location u in the direction $\overrightarrow{r}$, L is the length of ray along $\overrightarrow{r}$, and τ is the light extinction coefficient [96]. Light and extinction coefficients are specified by transfer functions [140], usually defined as one-dimensional functions over s as $C(s)$ and $\tau(s)$. The transfer functions can also be defined as a multi-dimensional function [80]. The light scattered at u is attenuated by the particles between u and the eye according to an exponential attenuation function. This integral cannot be computed analytically in general. We can approximate the integral by discretizing

it into a series of sequential intervals i of width d:

$$I_\lambda(\overrightarrow{x}, \overrightarrow{r}) = \sum_{i=0}^{L/d-1} C_\lambda(s(id))\tau(s(id))d \prod_{j=0}^{i-1} \exp\left(-\tau(s(jd))d\right)$$

We can further approximate the summation by keeping only the first two terms of Taylor expansion of the exponential term and get the compositing equation:

$$I_\lambda(\overrightarrow{x}, \overrightarrow{r}) = \sum_{i=0}^{L/d-1} C_\lambda(s(id))\alpha(id) \prod_{j=0}^{i-1} (1 - \alpha(jd)) \tag{3.1}$$

where α is opacity, and $(1 - \alpha)$ is defined as transparency. The definition of α is:

$$\begin{aligned} \alpha(id) &\equiv 1 - \exp\left(-\int_{id}^{(i+1)d} \tau(s(u))\ \mathrm{d}u\right) \\ &\approx 1 - \exp\left(-\tau(s(id))d\right) \\ &\approx \tau(s(id))d \end{aligned}$$

The above assumes a constant opacity across each interval. A more accurate approximation is to compute a linearly-interpolated opacity using pre-integrated transfer functions, as introduced by Max *et al.* [97] in 1990 and recently used by Engel *et al.* [37], Knittel [82], and Schulze *et al.* [124]:

$$\alpha(id) \approx 1 - \exp\left(-\int_0^1 \tau((1-\omega)s_f + \omega s_b)d\ \mathrm{d}\omega\right)$$

where $s_f \equiv s(id)$ is the scalar value at the start (front) of the segment, and $s_b \equiv s((i+1)d)$ is the scalar value at the end (back) of the segment. So $\alpha(id)$ is a function of s_f, s_b, and d. Since d is assumed constant and scalar values s_f, and s_b are generally 8-bits each, this function can be pre-computed and stored in a table. At run-time the value can be fetched by a table lookup.

The above discussion is directly applicable to a general raycasting approach. Splatting reorders the volume rendering integral so that each voxel's contribution to the integral can be computed separately from other voxels and the volume can be treated as a field of overlapping interpolation kernels. For radially symmetric kernels, we can pre-integrate the kernel into a 2D lookup table called a *footprint*. We then map voxel footprints scaled by voxel values on the image plane and accumulate them to form the final image. The accumulation requires either back-to-front or front-to-back compositing order.

Some of the recent work in direct volume rendering is based on simulating the light-scattering properties of real-world media such as atmospheric particles, clouds, and human skin. Riley *et al.* [116] have developed a multi-field visualization algorithm of weather data. Their approach is based on the light scattering properties of atmospheric particles. The individual extinction and scattering of these particles form the basis of the transfer function and enable them to achieve realistic weather visualization. Kniss *et al.* [81] have developed an interactive volume shading model to incorporate volumetric shadows, phase functions, forward scattering and chromatic attenuation, and provide subtle appearance of translucency. The light transport is computed in image space and achieves interactive volume rendering with non-static transfer function, light direction, or volume data. The visualization of the electrostatic potential is different from the above since it does not correspond to visually observable phenomenon. We have decided to implement a simple lighting model for direct volume rendering and we discuss it below. However, should the need arise for more sophisticated lighting models as in [81, 116] we can incorporate

them in our approach at the pre-processing stage.

3.3 Pre-computation of Accumulated Transparency

From Equation(3.1) we can see that the contribution to the final image from voxel at id is the product of $C_\lambda(s(id))\alpha(id)$ with the accumulated transparency along ray $\overrightarrow{r}$ from viewer to this voxel, i.e., $\prod_{j=0}^{i-1}(1-\alpha(jd))$. This means, if we pre-compute the accumulated transparency and store it at the voxel, then at run time, we can get the contribution from this voxel by a simple product with $C_\lambda(s(id))\alpha(id)$. This pre-computation already takes the ordering information between voxels into consideration by only accumulating those voxels in front of this voxel. So at run-time, voxels can be splatted in any order onto the image plane and we would get the correct result. As an example in Figure 3.1, the accumulated transparency along ray $\overrightarrow{r}$ for voxel W can be pre-computed as a product of $(1-\alpha_a)(1-\alpha_b)(1-\alpha_c)(1-\alpha_d)(1-\alpha_e)$, where the α values can be computed as usual or looked up from a pre-integrated transfer functions table.

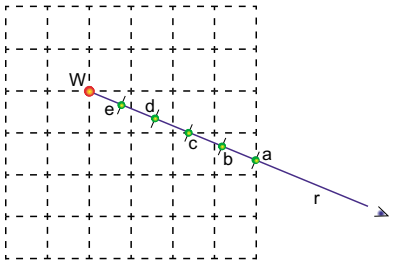


Figure 3.1: Accumulation of transparency along ray r for voxel A

Our approach is similar to the pre-integrated transfer function approach [37, 82, 97, 124] in that we too assume fixed (or static) transfer functions. If the transfer

functions were to be changed, our approach like the pre-integrated transfer function approach will require re-computation. However, we will like to note here that our approach and the pre-integrated transfer function approach are addressing two fundamentally different aspects of volume rendering. The goal of our accumulated transparency approach is to improve the *efficiency* of rendering. The pre-integrated transfer function approaches improve the *accuracy* of assigned opacity for a small interval along the ray by linearly interpolating the values across the interval. Thus, they effectively replace the zeroth-order approximation of the opacity by a first-order approximation. Although our approach does not require the linear-interpolation assumption, if the pre-integrated transfer function tables are available, we can benefit by getting more accurate opacity information for each interval along the accumulation ray.

In practice we sample a few thousand discrete set of viewer directions and compute the accumulated transparency in those directions for every voxel. Storage and retrieval of such vast amount of additional information per voxel is a challenge for real-time volume rendering applications. Fortunately for displaying highly-coherent and gradually varying volume datasets such as the 3D electrostatic potential, we can greatly compress the storage requirement using spherical harmonics.

Nulkar and Mueller [102] have used pre-computation of accumulated shadowing factors for fixed light sources by storing them on a 3D grid to render volume shadows. Our method can deal with dynamic lighting with unlimited number of light sources. We also use the accumulated opacity information for speeding up the rendering with changes in viewing direction.

3.4 Compression by Spherical Harmonics

The pre-computed accumulated shadowing and transparency information for each voxel requires a large amount of storage. We use spherical harmonic functions to compress this information. Spherical harmonic functions form an efficient basis to represent functions defined over the directional space, such as incident radiance and BRDFs [12,114,125,127]. Using the spherical harmonic functions for expansion and storing the coefficients up to a given order is similar to a filtering process of the angularly distributed signal [12,114].

Spherical harmonic basis functions y_l^m are solutions of the following eigenvalue function [35]:

$$\left[\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}(\sin\theta\frac{\partial}{\partial\theta}) + \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\varphi^2}\right] y_l^m(\theta,\varphi) = l(l+1)y_l^m(\theta,\varphi)$$

y_l^m are expressed in terms of associated Legendre functions $P_l^m(x)$, which satisfy the Legendre's differential equation ($-1 \le x \le +1$, $|m| \le l$):

$$(1-x^2)\frac{d^2P_l^m(x)}{dx^2} - 2x\frac{dP_l^m(x)}{dx} + \left[l(l+1) - \frac{m^2}{(1-x^2)}\right] P_l^m(x) = 0$$

If $m = 0$ the solutions are Legendre polynomials $P_l(x)$:

$$\begin{aligned} P_l(x) &= \frac{1}{2^l l!}\frac{d^l}{dx^l}(x^2-1)^l \\ &= \sum_{r=0}^{\nu}(-1)^r\frac{(2l-2r)!x^{l-2r}}{2^l r!(l-r)!(l-2r)!} \end{aligned}$$

For $m \neq 0$, solutions are associated Legendre functions:

$$P_l^m(x) = (1-x^2)^{m/2}\frac{d^m P_l(x)}{dx^m}$$

Based on associated Legendre functions, real-valued spherical harmonic basis functions [127] can be expressed as:

$$y_l^m = \begin{cases} \sqrt{2} K_l^m \cos(m\varphi) P_l^m(\cos\theta), & m > 0 \\ \sqrt{2} K_l^m \sin(-m\varphi) P_l^{-m}(\cos\theta), & m < 0 \\ K_l^0 P_l^0(\cos\theta), & m = 0 \end{cases}$$

where K_l^m are the normalization constants:

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi} \frac{(l-|m|)!}{(l+|m|)!}}$$

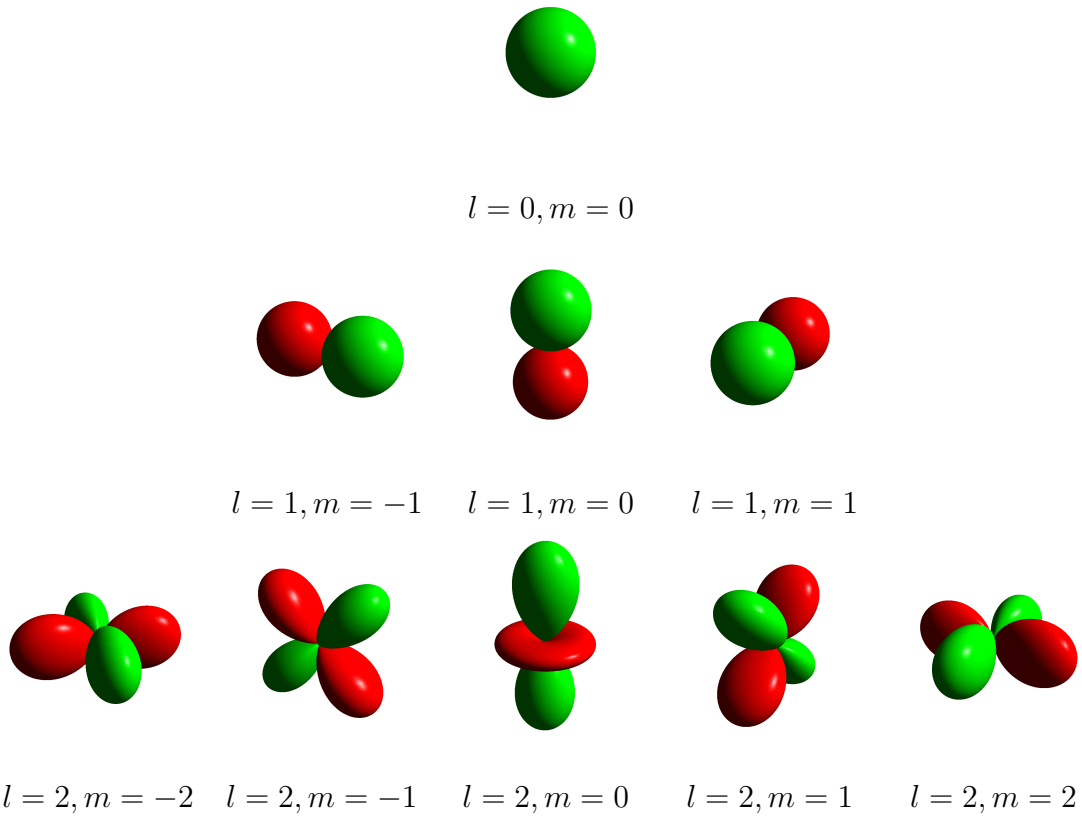


Figure 3.2: The first three SH bands plotted as unsigned spherical functions by distance from the origin (green is for positive values and red is for negative values)

Figure 3.2 shows the first three bands (each band has the same l value) of real spherical harmonic basis functions. One can see that the angular frequency goes up as l increases.

The projection of the pre-computed accumulated transparency $T(W)$ of voxel W onto the spherical harmonic basis is given by:

$$T_l^m(W) = \int T(W, \overrightarrow{r}) y_l^m(\overrightarrow{r}) d\overrightarrow{r}$$

The reconstructed function up to the n-th order is:

$$\widetilde{T}(W, \overrightarrow{r}) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} T_l^m(W) y_l^m(\overrightarrow{r})$$

where

$$\overrightarrow{r} = (\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta)$$

One can use the standard recurrence on l to compute the associated Legendre polynomials [113]:

$$(l-m)P_l^m = x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m$$

Double-precision numbers can store numbers ranging from 2.22507×10^{-308} to 1.79769×10^{308}. A factorial of 171 will overflow in this representation. This will happen during the factorial computation inside the normalization factor K_l^m with $l = m = 86$ and thus limits the order of spherical harmonics functions to $l = 85$. To overcome the range limitations in using double-precision values and to improve the efficiency of computing spherical harmonics, we have built lookup tables to store the K_l^m values. To avoid the overflow in computing factorial values

in the evaluation of K_l^m, we place the square root operation inside the factorial computation. To address the underflow possibility of K_l^m for large l and m, we have used several tables to store them by splitting each underflowed K_l^m into a sequence of products such that each product element is within the effective range of the double-precision representation. The whole product sequence representing K_l^m is multiplied by the associated Legendre polynomial P_l^m to get y_l^m. During the evaluation of P_l^m, we continually check the value being evaluated with the overflow limit of double precision value. If the two numbers are close, we apply the product elements from the pre-evaluation of K_l^m. This allows us to multiply very small values of K_l^m with very large values of P_l^m and still get answers that are within the double-precision range.

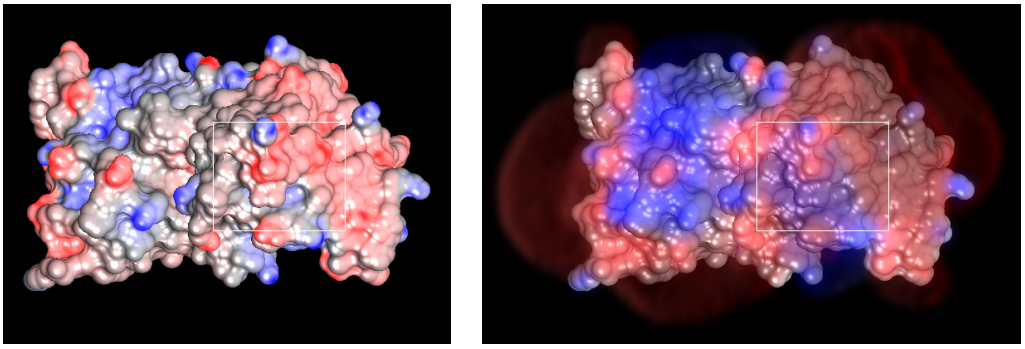
We store the spherical harmonic coefficients as short integers. If we use 16 or 25 basis functions, then we will need $32 - 50$ bytes extra storage for each voxel.

Molecular electrostatic potentials have a dynamic range that exceeds the displayable ranges of our 8-bits per color channel monitors. A good solution here will be to use graphics algorithms such as tone-mapping, that display high-dynamic range images on regular monitors. We have, for now, chosen a simpler alternative. We currently clamp the absolute electrostatic potential values from above and below. At the lower limit, we only consider those voxels whose electrostatic potential is above a certain threshold. For example, if we assume 8-bit color channels then we clamp values less than 1/256 of the highest distinguishable potential to zero. This reduces the number of voxels (and their associated spherical harmonic coefficients' memory) that need to be processed for the final display.

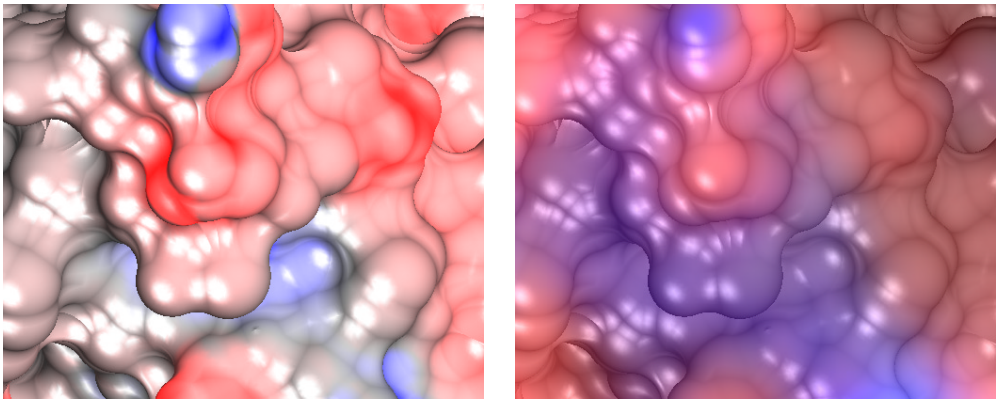
3.5 Results

As in Chapter 2, we show our visualization results on superoxide dismutase (SOD) enzyme (with 2196 atoms) and an ion-channel on the outer membrane of the Escherichia coli (Ecoli) bacterium molecule [131] (with 10585 atoms). The results are shown in Table 3.1, and Figures 3.3, 3.5, and 3.4. We have used about 2000 directions to pre-compute the accumulated transparencies for each contributing voxel.

Figure 3.3(a) and Figure 3.4(a) display the electrostatic potential on the sur-



(a) Lighted on-surface potential (b) Our splatting (up to surface)



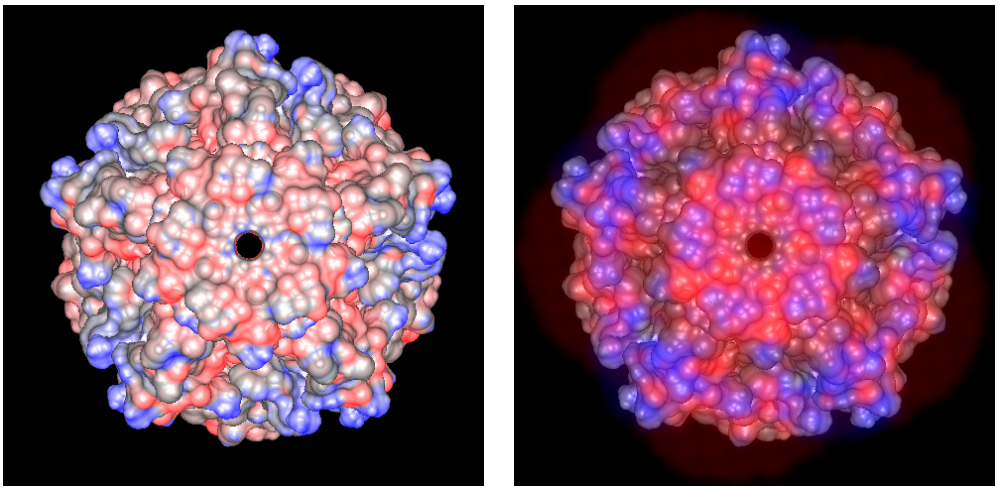
(c) Closeup of (a) (d) Closeup of (b)

Figure 3.3: Electrostatics on SuperOxide Dismutase (SOD) dataset (red is for negative potential, and blue is for positive potential)

Dataset	Volume size	Image size	Rendering time (seconds)		
			Ray casting	Regular splatting	Our splatting
SOD	128^3	512×512	6.297	3.156	1.397
Ecoli	256^3	512×512	14.593	8.329	3.967

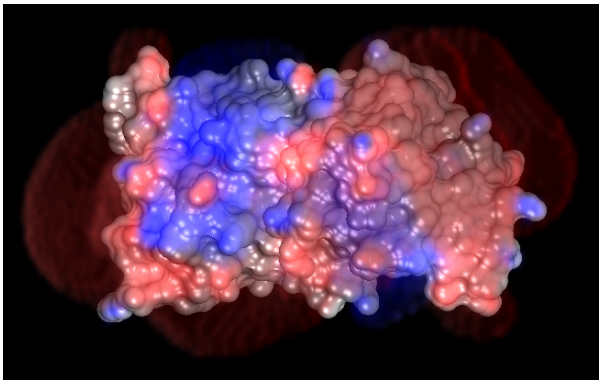
Table 3.1: Results on SOD and Ecoli membrane channel

faces, with red for negative and blue for positive potential; both use the potential information to modulate lighting color with grey for neutral potential. Figures 3.3(b) and 3.4(b) show the volume rendered 3D potential field, from the viewer up to the molecular surface, using our splatting algorithm. Figures 3.3(c) and (d) are close-ups of Figures 3.3(a) and (b) respectively. Electrostatic potential is traditionally displayed on molecular surfaces [117]. We find it is more informative to use direct

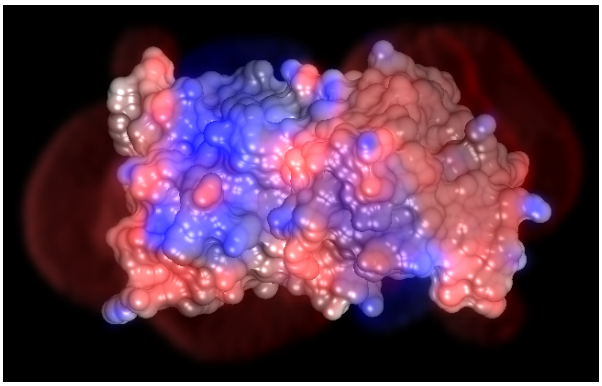


(a) Lighted on-surface potential (b) Our splatting

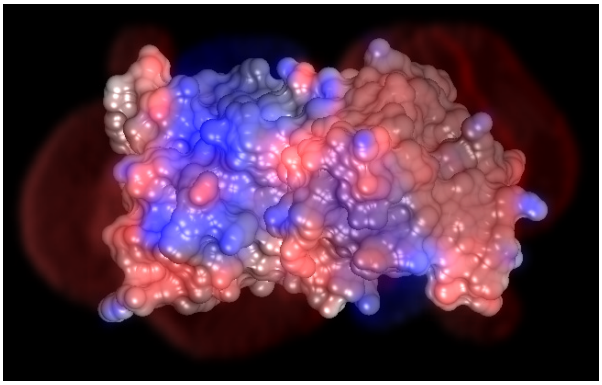
Figure 3.4: Electrostatics on Ecoli membrane channel (red is for negative potential, and blue is for positive potential)



(a) Raycasting (6.297 seconds)



(b) Axis-aligned splatting (3.156 seconds)



(c) Our splatting (1.397 seconds)

Figure 3.5: Comparison of raycasting, our splatting, and axis-aligned splatting of electrostatics on SOD dataset (red is for negative potential, and blue is for positive potential)

volume rendering to show the potential field between the viewer and the molecular surface. Comparing Figure 3.3(c) with 3.3(d), one can clearly see that in front of the negative on-surface potential in the central region, there is a sizable positive potential region. It would have been hard to use the traditional electrostatic potential display methods such as color-coded molecular surfaces or electrostatic iso-potential surfaces to convey the same amount of visual information. Figure 3.5 compares the images generated using raycasting, our order-independent splatting, and axis-aligned splatting [145]. It clearly shows the advantage of our method.

3.6 Conclusions

Our order-independent splatting algorithm gives us more efficient display of the computed 3D electrostatic potential field. By pre-computing and storing the accumulated shadowing and transparency information, we are able to achieve faster rendering speed. This algorithm can also be used to display other volume data that are without clear underlying structures.

Chapter 4

View-dependent Variable Precision Data Representation

Protein structures have limited dynamic range. They are determined by using X-ray crystallography, NMR experiments, or gel electrophoresis. All of these methods have their accuracy limitations. In this chapter, I will develop the idea of using variable-precision data representation to accelerate the rendering of protein structure data and other 3D datasets. Our approach complements the multiresolution techniques as it reduces the *precision* of each graphics primitive in addition to the *number* of graphics primitives.

Interactive visualization of protein data enables investigators to have interactive control over a computational steering process and thus to gain more insight from the computation with the instant display of the intermediate results.

4.1 Introduction

Several novel techniques have been developed to reconcile the conflicting goals of scene realism and interactivity. These techniques can be broadly classified into two lines of research. The first line of research includes techniques such as multiresolution rendering and visibility-based culling. Such techniques operate by reducing

the number of graphics primitives to be rendered based on viewing and illumination parameters, such that there are minimal visually discernible differences between viewing higher complexity and lower complexity scenes. Orthogonal to these advances, we have been witnessing another line of research whose goal is to reduce the precision of each graphics primitive being rendered. Recently, reduction in precision of the object properties such as colors [59, 148], normals [31, 151], and vertex coordinates [76, 91] has been successfully attempted. The contribution of our approach lies in merging these two lines of research for variable-precision, view-dependent rendering.

Most transformations and lighting for graphics primitives are currently carried out at full floating-point precision only to be later converted to fixed-point representation during the rasterization phase. An argument can be made that such high accuracy during geometry transformation and lighting stage sometimes exceeds even the display accuracy and thus causes several bits worth of unnecessary precision computation. We are currently witnessing these important trends in 3D graphics that have increased the need for variable-precision rendering:

1. View-dependent Rendering: View-dependent rendering has already introduced the concept of rendering different regions of a scene at varying geometric, illumination, and texture detail [67, 93, 147] based on their perceptual significance. A natural extension of this approach is to render each object at the precision appropriate for it. Under a perspective projection, objects that are close to the observer need more bits of precision than objects that are far.

2. Processor-Level Support: With rapid growth in the size of the 3D datasets, geometry processing (transformation and lighting) has become a significant computational component of the 3D graphics pipeline. To partly alleviate such computations in graphics and image processing, a variety of matrix math extensions to the CPU instruction sets have emerged: Intel's Pentium II with MMX and Pentium III with SSE, AMD's K6/Athlon with 3DNow!, and the Motorola PowerPC G4 with AltiVec. All of these instruction sets take advantage of SIMD (single-instruction multiple-data) parallel execution of instructions [61]. For instance, the Intel MMX [106] allows variable precision integer arithmetic to be implemented in SIMD parallelism where either two 32-bit, four 16-bit or eight 8-bit integer values are operated on in parallel. Such processor-level support for variable-precision arithmetic has enabled efficient implementation for variable-precision rendering.

3. Geometry Bandwidth Bottleneck: Increase in the geometric complexity of the graphics datasets has far outpaced the increase in the display complexity. This has resulted in a bottleneck in transferring 3D vertex data from the geometry processor to the graphics processor. If the variable-precision rendering techniques discussed in this chapter are adopted in a graphics API and/or implemented on the chip itself (in a manner similar to the MMX technology), this could significantly reduce the bus traffic to the graphics chip and accelerate the transformation and lighting stages on the graphics chip beyond the results reported here.

In the following sections, I will lay down the mathematical groundwork for performing variable-precision geometry transformations and lighting for 3D graph-

ics. In particular, we explore the relationship between the distance of a given sample from the viewpoint, its location in the view-frustum, to the required accuracy with which it needs to be transformed and lighted to yield a given screen-space error bound.

4.2 Related Work

In computational geometry and solid modeling, research has been done on performing robust geometric operations. Exact rational arithmetic (i.e. in homogeneous coordinates) has been found to address several shortcomings of the conventional floating-point arithmetic [62]. However, successive geometric operations can result in an unbounded growth in the precision required to accurately compute the result. One way to limit the growth of the required precision is to intersperse rounding between arithmetic operations. Rounding-off vertex coordinates (or even line and plane coefficients [62, 130]) is reasonably well-understood now. However, such rounding is much more difficult if it must preserve some combinatorial or topological structure amongst the primitives (in/out, above/below, clockwise/counterclockwise orientation etc.). Several sophisticated approaches have been proposed that perform rounding and preserve some of these relationships by adding some extra points [40] or re-adjusting the rounded-off numbers to approximately maintain the relationships [99]. In a number of cases, such results are used only to establish topological relationships amongst primitives. This can be efficiently done by using sufficiently accurate (as opposed to exact) arithmetic [11, 41, 75].

Most of the research in graphics dealing with limiting the precision of vertex coordinates has focused on rounding-off the vertex coordinates (perhaps with attributes) independently of the topological structure defined by the vertices. Thus, with such approaches it is possible that the lower-precision models suffer from artifacts such as self-intersection and false incidences, even if the original higher-precision models did not. In practice, such artifacts have not been observed frequently enough yet, to convince most graphics practitioners to adopt the more time-consuming algorithms to preserve the topological structures. We continue this line of thinking and quantize the vertex coordinates independently of the underlying topological constraints. Deering [31] has demonstrated that quantizing the normals down to 12 bits (i.e. only $4K$ unique normals) and vertex coordinates to 24 bits results in only minimal degradation in the rendered image quality. Reducing the precision of the vertex coordinates is implicit in the work of Rossignac and Borrel [119] and more recently, Luebke and Erikson [93]. The focus there is on reducing the geometric complexity of the high detail models. Consequently, even though the resulting vertex coordinates are effectively quantized on a grid and octree respectively, the reduced precision has not been taken advantage of during transformation and rendering.

Within the area of compression of 3D models, a lot of attention has been given to reducing the number of bits to represent vertex coordinates. Most approaches have used multi-stage quantization with Huffman encoding of delta-differences between successive vertices [9, 21, 31, 91, 135, 137]. Recently, progressive compression and transmission has been actively exploited [8, 25, 104, 105, 135]. Using the tech-

niques of geometry prediction and progressive mesh encoding [8], combined with batch processing [25, 104] and entropy encoding [105], compression ratios for progressive compression have started approaching those for single resolution compression. King and Rossignac [76] have further balanced the reduction of the number of vertices and the reduction of bits per vertex coordinate using a shape complexity measure. For a nice survey of 3D geometry compression the interested reader may refer [9, 118].

4.3 Our Approach

4.3.1 Precision and Complexity

Let us first note the difference between multi-resolution and variable-precision rendering for 3D graphics models. Multi-resolution hierarchies have traditionally involved modeling each object at multiple levels of detail, where the detail is usually measured in the number of geometric primitives required for representation. Thus, a high-detail triangle-mesh object will require a higher number of vertices, edges, and triangles for representation. This complexity is largely independent of the precision at which each vertex is being represented. As can be seen in Figure 4.1(b), a multiresolution technique can be used to identify how *many* primitives are necessary for a faithful representation of a given object with a given set of viewing and lighting parameters. A variable-precision technique provides bounds on the bits of *accuracy* per primitive that are required for high-fidelity rendering. This can be seen in Figure 4.1(c) where the points selected to represent the circle all fall on

the quantization grid. Thus, the two techniques are orthogonal to each other and depending on the application requirements for accuracy and speed can be used in a complementary manner.

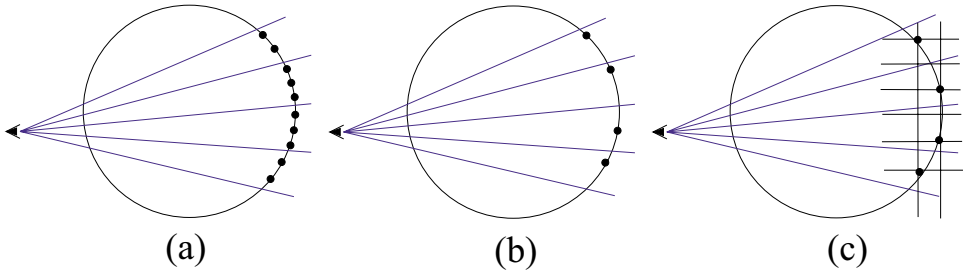


Figure 4.1: Varying complexity versus varying precision

In the following analysis of different kinds of errors in geometric transformations, we shall assume that a minimum-sized cube has been constructed to cover the whole object using an algorithm similar to [47] and each axis has been normalized to the range of $[-1.0, 1.0]$. Thus the operands a and b are n-bit fixed-point representations of floating-point quantities within $[-1.0, 1.0]$. Additionally, we assume that we computed the n-bit fixed-point representation from such normalized floating-point representation by multiplying by 2^{n-1} and rounding to the nearest integer. Also, we would like to point out that we perform a worst-case analysis to guide the selection of appropriate precision. A good reference for sources and propagation of numerical errors is [111].

Representation Error

Often input data has uncertainty. A recent standards report from NIST outlines several types of uncertainty [138]. These include statistical (e.g., confidence intervals

with mean and variance) and error (differences among estimates of the data from multiple sources and/or multiple time instants) uncertainties. Such uncertainties often limit the data acquisition precision. Other sources of error in the input data include approximations in the abstractions from which data is derived, numerical errors in computing the data using limited precision arithmetic, as well as instabilities in the mathematical models (as in ill-conditioned systems). Such errors often limit the number of bits of precision in the input dataset. For a n-bit fixed-point representation derived by rounding from a normalized floating-point representation, the representation error is at most half bit: $\varepsilon^{rep} \leq \frac{1}{2}$

Addition Error

For adding two n-bit integers, the error arises from the propagated error from the representation. $\varepsilon = \varepsilon^{gen} + \varepsilon^{prop} = \varepsilon^{prop} \leq \frac{1}{2} + \frac{1}{2} = 1$. So we will lose at most one bit of accuracy due to each addition.

Multiplication Error

We shall use $2n$ bits to store the intermediate result of multiplication of two n-bit integers. Since each normalized floating-point operand was magnified by a factor of 2^{n-1} during conversion to fixed-point before multiplication, we need to take out that extra 2^{n-1} factor by right shifting the intermediate result $n-1$ bits. The n-bit final result thus obtained has the largest error when both multiplier and multiplicand are close to 2^{n-1} and the absolute representation error is $\frac{1}{2}$: $\varepsilon \leq \frac{\frac{1}{2} \times 2^{n-1} + \frac{1}{2} \times 2^{n-1}}{2^{n-1}} = 1$. Thus, we lose one bit of accuracy due to each multiplication.

Division Error

Per-vertex division happens during the transformation from homogeneous coordinate to 3D image-space coordinates. The propagated error due to the division is:

$$\varepsilon^{prop} = \varepsilon_{\frac{a}{b}}{}^{prop} = \left|\frac{\partial(\frac{a}{b})}{\partial a}\right|\varepsilon_a + \left|\frac{\partial(\frac{a}{b})}{\partial b}\right|\varepsilon_b = \frac{\varepsilon_a}{b} + \frac{a}{b^2}\varepsilon_b$$

Here, ε_a and ε_b are the representation errors in a and b and are each at most $\frac{1}{2}$. For vertex within the view volume, we have $a \leq b$. Also, the generated error due to truncation is 1. Thus:

$$\varepsilon = \varepsilon^{gen} + \varepsilon^{prop} = 1 + \frac{\frac{1}{2}}{b} + \frac{a \times \frac{1}{2}}{(b)^2} \leq 1 + \frac{1}{b}$$

Since in viewing transformations, the divisor b is the distance of a scene vertex to eye in normalized view-volume representation (where the distance of the farthest point is 1.0),

$$\varepsilon \leq 1 + \frac{\text{distance of far plane in view-volume from eye}}{\text{distance of scene vertex from eye}}$$

So the loss of number of bits accuracy is

$$\left\lceil \log_2(1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}) \right\rceil$$

Putting it all together

For a 1024×1024 window, with pixel level accuracy, we need 10 bits in each x and y to represent the position of a vertex on the screen. Transformation of a vertex in homogeneous coordinates with a 4×4 matrix requires four multiplications and three additions for each coordinate. The height of this operation tree is three

(leaves at level 3 have four multiplies, level 2 has two additions, and the root at level 1 has the final addition). Thus, we will lose 3 bits of accuracy in this matrix-vector multiplication. To get n bits of accuracy after transformation and homogeneous division, we need m bits to represent the input data:

$$m = n + 3 + \left\lceil \log_2(1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}) \right\rceil$$

Thus, if the display window is 1024×1024, $n = 10$ for pixel level accuracy; and if the distance of the point being rendered is half way across the view-volume, we shall need 15 bits to represent the vertex data: $m = 10 + 3 + \lceil \log_2(1 + 2) \rceil = 15$. This can be used to compute the requisite number of bits of precision required for each vertex based on its distance from the eye and forms the basis of view-dependent precision-based rendering.

For applications which require sub-pixel accuracy, we can increase the window resolution in the above formula. For example, if the application needs four bits of sub-pixel accuracy along each dimension, then we add four more bits to the requirements, which in the above example will result in a requirement of 19 bits of accuracy per input vertex coordinate for a 1024×1024 window.

4.3.2 View-dependent Transformation

The formula from the last section gives the upper bound on the number of bits needed to transform the vertices in order to get n bits of accuracy. In reality, if the object projects to the screen in an area that is small compared to the screen size, we may need less than n bits to get window-resolution-level accuracy. For a

view-dependent transformation, we have to find out the number of bits needed for vertices at different locations.

Octree-based Bounding Volume Hierarchy

To take advantage of the view-dependent information, we need an efficient way to estimate the projected size of different parts of an object. An octree bounding volume hierarchy is easy to build and very efficient to get the bounding volume of the projected vertices.

The idea is to find the minimum and maximum number of bits required for each bounding box using equations in the following subsections. If the two numbers are equal, then all vertices within this box will need the same number of bits during the transformation. Otherwise, vertices in this bounding box need different number of bits, and we should recurse to the lower levels of the octree hierarchy. In our implementation, we have used the normalized object coordinates, i.e., all x, y, and z coordinates lie within $[-1.0, +1.0]$.

Projected Size of the Dataset

For each view point, we calculate the projection of the eight corner points of the root level bounding box. From these projected points, we can determine the size of the object on the screen. The corner points are transformed into canonical viewing volume. The whole viewport will map into $[-1.0, +1.0]$ in both x and y direction of this viewing volume, so the relative size of the projected object to screen is just half the range of these projected corner points. During the calculation, we store

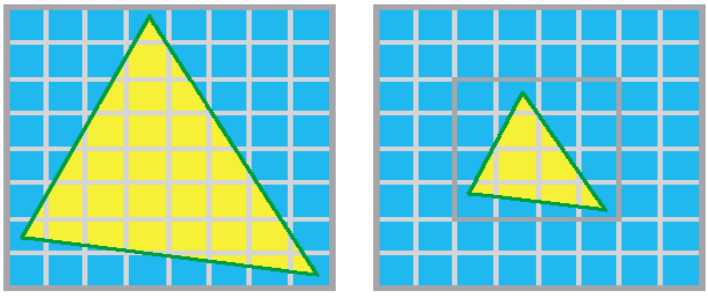
the transformed minimum and maximum W value (i.e., minimum and maximum depth, W_{min} and W_{max}) of these eight points for later usage. The distance from the nearest visible scene vertex to view point is just the bigger one of W_{min} and the near clipping plane.

Nearest Visible Vertex Accuracy

Given the width and height of the screen in number of pixels, the number of bits for pixel-level accuracy is:

$$n = screen\_bits = \max(\lceil \log_2 width \rceil, \lceil \log_2 height \rceil)$$

If sub-pixel accuracy of s extra bits is desired, just add s to the number of bits computed above. From last subsection, we know the ratio of the projected object size to the screen size is the projected range divided by 2, so the bits needed to represent the object will be $\lfloor |\log_2(\frac{\text{projected range}}{2})| \rfloor$ bits less than $screen\_bits$.



(a) Need 3 bits in x and y (b) Need 2 bits (and 1 bit of offset)

Figure 4.2: Objects of smaller projected size needs less precision

Taking into account the computation error due to the multiplication, addition, and division as mentioned in the last section, the number of bits needed for the

nearest visible scene vertex is:

$$near\_bits = n + 3 - \left\lfloor \left| \log_2(\frac{\text{projected range}}{2}) \right| \right\rfloor + \left\lceil \log_2(1 + \frac{\text{distance of far plane from eye}}{\text{distance of nearest vertex to eye}}) \right\rceil$$

As shown in Figure 4.2, the smaller object in (b) only occupies less than half of the screen in each dimension, so it will need one bit less than the bigger object in (a). The extra screen offset will be added in the final viewport transformation step.

Accuracy to Represent Each Vertex

Due to the perspective foreshortening, an object appears smaller as its distance to the viewpoint increases. As an example, an object at twice the distance will have half the size on the screen, and thus needs one bit less to represent.

Generally, given the *near_bits* as defined before, we try to find the number of bits for any other vertex. After the transformed W value (i.e., depth) is known, we calculate the *vertex_bits* as:

$$vertex\_bits = near\_bits - \left\lfloor \log_2(\frac{\text{transformed } W \text{ of this vertex}}{\text{distance of nearest vertex to eye}}) \right\rfloor$$

It will be expensive if we need to do this calculation for each vertex. Fortunately, with the bounding box hierarchy, very few calculations need to be done.

Starting from the top of the hierarchy, we calculate the minimum and maximum transformed W value for each node. First we calculate the transformed W

value of the center of the node (denoted as W'_{center}), then we check the eight corner points of the node to figure out the minimum and the maximum. As we already have the W_{min} and W_{max} of the corner points at the root level, for the subtree at level k:

$$W'_{\min} = W'_{center} + \frac{W_{\min}}{2^k} \quad \text{and} \quad W'_{\max} = W'_{center} + \frac{W_{\max}}{2^k}$$

where the denominator is due to the fact that the node size is reduced by a factor of two when we go down one level in the octree.

Using the above two equations, we can find out the minimum and the maximum number of bits needed for vertices within the box:

$$V_{min} = near\_bits - \left\lfloor \log_2\left(\frac{W'_{\max}}{\text{distance of nearest vertex to eye}}\right) \right\rfloor$$

$$V_{max} = near\_bits - \left\lfloor \log_2\left(\frac{W'_{\min}}{\text{distance of nearest vertex to eye}}\right) \right\rfloor$$

If these two numbers are equal, then we know that all the vertices within the box will need these number of bits to represent. Otherwise, vertices in the box require different numbers of bits and we need to recurse down one more level of the octree.

4.3.3 Spatio-Temporal Coherence

In the last two sections we have seen the relationship between the input bits of accuracy and the bits of accuracy required for the output. For the same number of bits of accuracy for the output, we can further reduce the bits of accuracy required in the input by taking advantage of spatial and temporal coherence. This can result

in further savings in processing time as well as in the bandwidth to the graphics processor.

Spatial Coherence

The basic idea that we use to take advantage of the spatial coherence is that the difference in spatially close vertices can usually be represented in far fewer bits than those required to represent each vertex coordinate in its entirety. This idea has been used with great success in the research on 3D compression of geometry as discussed in Section 4.2. If a vertex coordinate x' can be represented by a delta-difference with respect to another coordinate x as $x' = x + \Delta x$ then one can decompose the transformation for coordinate x' as: $Mx' = M(x + \Delta x) = Mx + M\Delta x$

Since the number of bits of accuracy required to transform Δx is much smaller, one can perform several of them in parallel. To exploit this idea, we can partition the dataset by any spatial subdivision scheme, such as an octree, over the vertices of the model. In our implementation we have used an octree that subdivides by the volume centroid at each level. In this scheme, since each level reduces the range by half, the vertices in each lower level require one bit less than their parents. The accuracy of the transformation of a vertex coordinate with a matrix is represented by the lower of the two accuracies. Thus if the vertex coordinates can be quantized in less bits, the transformation matrix values can also be quantized with fewer bits.

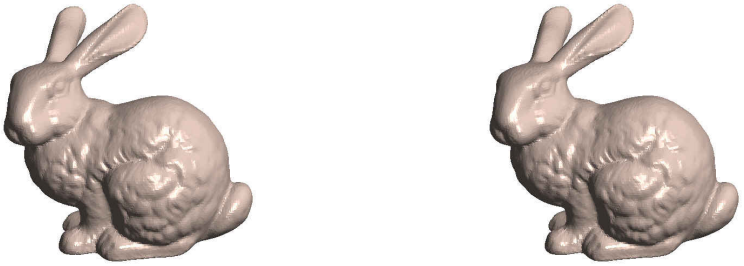
In this approach we independently transform the delta difference in the vertex coordinate position between the current level of the octree and its parent. Then we can get the final transformed results for each vertex by a top-down tree traversal as

shown in Figure 4.3 ($LIMIT$ is the lowest level of tree below which the difference between the transformed parent and children is negligible).

```
Top-Down-Tree-Traversal(x)
if x ≠ NULL
    if x.level ≤ LIMIT
        x.value = x.parent.value + x.transform
        for i from 1 to 8
            Top-Down-Tree-Traversal(x.child(i))
    else
        x.value = x.parent.value
        for i from 1 to 8
            Top-Down-Tree-Traversal(x.child(i))
```

Figure 4.3: Pseudo code for top-down tree traversal

As an example, if we could operate on byte- and short-precision operands and we required 16 bits of accuracy, then we could transform the top eight levels of the octree in short-precision and the lower levels could be transformed in byte-precision (or even lesser, if available). By using such hierarchical schemes, one can get a better precision efficiency without losing accuracy. Figure 4.4 compares the results on the Stanford Bunny model using floating point and variable-precision transformations.



(a) Floating Point Transform (32 bits/vertex coordinate)

(b) Variable Precision Transform (Average 7.9 bits/vertex coordinate)

Figure 4.4: Variable-Precision transformation of the Stanford Bunny model (69K triangles; lighting for both images has been calculated in floating point)

Temporal Coherence

Similar to the idea of spatial coherence, we can take advantage of temporal coherence by noting that the difference in the transformed vertex positions does not differ significantly from one frame to the next. Thus if we calculate the difference in the transformation matrix from one frame to the next and use the difference matrix ΔM to transform a vertex, we can then add it to the previously transformed vertex position in fewer bits: $M'x = (M + \Delta M)x = Mx + \Delta Mx$. Extending this idea further, we note that one can combine the spatial and temporal coherences: $M'x' = (M + \Delta M)(x + \Delta x) = Mx + \Delta Mx + M\Delta x + \Delta M\Delta x$ As we show in Tables 4.2 and 4.3 for the Auxiliary Machine Room dataset, the average number of bits that are operated upon for each vertex as well as the equivalent number of operations can both be greatly reduced by taking advantage of both spatial and temporal coherences.

4.3.4 Variable-Precision Lighting

Color is usually represented by 8-bits of precision in red, green, and blue components. Also, if depth cueing is turned on and the far objects are displayed at lower intensities, their color can be represented using fewer bits.

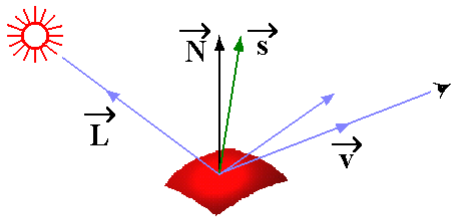


Figure 4.5: Lighting Calculation

Before we go to the detailed treatment of the variable-precision lighting, let us review the formula for the lighting calculation we have used. Although there are good psychophysically-based light reflection models [107], we decided to implement the OpenGL illumination model due to its widespread use. As in OpenGL, we assumed diffuse and Phong illumination with Gouraud shading without per-pixel normal evaluation:

$$\begin{aligned} Color \;=\; & \text{emission}_{\text{material}} + \text{ambient}_{\text{light\_model}} * \text{ambient}_{\text{material}} + \\ & + \sum_{i=0}^{m-1} \left(\frac{1}{k_c + k_l d + k_q d^2}\right)_i * (\text{spotlight effect})_i * \\ & (C_{\text{ambient}} + C_{\text{diffuse}} + C_{\text{specular}})_i \end{aligned}$$

where m is the number of light sources, $(\frac{1}{k_c+k_l d+k_q d^2})$ gives the attenuation factor in which d is the distance between the vertex and the local light source. $C_{\text{ambient}} = \text{ambient}_{\text{light}} * \text{ambient}_{\text{material}}$, $C_{\text{diffuse}} = (\max\{\overrightarrow{L} \cdot \overrightarrow{N}, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}}$ and

$C_{\text{specular}} = (\max\{\overrightarrow{s} \cdot \overrightarrow{N}, 0\})^{\text{shin}} * \text{specular}_{\text{light}} * \text{specular}_{\text{material}}$. $\overrightarrow{L}$ is the unit vector that points from the vertex to the light position, $\overrightarrow{N}$ is the unit normal vector at the vertex, $\overrightarrow{s}$ is the normalized half way vector between the directions of the light source and the viewer, and *shin* is the shininess, i.e., the specular exponent. Our goal here is to find the necessary number of bits to represent the input illumination data in order to get the required accuracy in output color.

Sources of Error in Local Illumination

There are several additional sources of error in local illumination computation beyond the sources of error we have already discussed in the transformation stage (representation error, addition error, multiplication error, and division error). In lighting computations we have to deal with addition and multiplication errors for operands with different bits of accuracy, the square root operation error which results from vector normalization, and the error induced by exponentiation in specular illumination. Also, the special case of dot product of two unit vectors is worthy of separate analysis.

To reduce the error propagation, we can multiply the light coefficient with the object material property coefficient in floating-point form before converting to the n bit fixed-point representation. For example, instead of converting $ambient_{\text{light\_model}}$ and $ambient_{\text{material}}$ to n bits of integer, we multiply them in floating-point representation and then convert the result to a n bit integer. This way, we can save one bit of accuracy which would have been lost due to the multiplication of two n-bit integers. We next consider the other sources of error.

Error for Operands with Different Accuracy

Let us consider two operands with different bits of accuracy, say n and n' where $n' < n$. This means that if the maximum possible value is 1, then the representation errors are $2^{-(n+1)}$ and $2^{-(n'+1)}$, respectively. For addition, the error ε can be computed as:

$$\varepsilon \leq 2^{-(n+1)} + 2^{-(n'+1)} = 2^{-(n'+1)}(1 + 2^{-(n-n')})$$

As an example, if $n - n' = 2$, then: $\varepsilon \leq 2^{-(n'+1)}(1 + \frac{1}{4})$. The error will stay at $(n'+1)^{th}$ bit, and the result will get n' bits of accuracy, i.e., the same accuracy as the less accurate operand.

Similarly, for multiplication of operands with n and n' ($n' < n$) accuracy, the maximum possible error happens when the operands are close to the maximum possible value which we treat as 1, as discussed in previous section:

$$\varepsilon \leq 2^{-(n+1)} * 1 + 2^{-(n'+1)} * 1 = 2^{-(n'+1)}(1 + 2^{-(n-n')})$$

Again, the result has the same accuracy as the less accurate operand.

Error in the Dot Product of Unit Vectors

Let us consider two unit vectors, say $\overrightarrow{\alpha}$ and $\overrightarrow{\beta}$, with n bits of accuracy in each of their three components:

$$\overrightarrow{\alpha} = (\alpha_1, \alpha_2, \alpha_3) \text{ and } \overrightarrow{\beta} = (\beta_1, \beta_2, \beta_3)$$

Since the error in the three components ε_{α_i} and ε_{β_i} $(i = 1, 2, 3)$ is in the $(n+1)^{th}$ bit, i.e., $2^{-(n+1)}$, their dot product error is:

$$\varepsilon_{(\overrightarrow{\alpha}.\overrightarrow{\beta})} = \sum_{i=1}^{3}(\beta_i \varepsilon_{\alpha_i} + \alpha_i \varepsilon_{\beta_i}) \leq 2^{-(n+1)}(\sum_{i=1}^{3}\alpha_i + \sum_{i=1}^{3}\beta_i)$$

For unit vector $\overrightarrow{\alpha}$, we have: $\alpha_1^2 + \alpha_2^2 + \alpha_3^2 = 1$.

From the inequality: $a^2 + b^2 \geq 2ab$, we get:

$$(\alpha_1 + \alpha_2 + \alpha_3)^2 \leq 3(\alpha_1^2 + \alpha_2^2 + \alpha_3^2) = 3$$

So we have $(\alpha_1 + \alpha_2 + \alpha_3) \leq \sqrt{3}$. Similarly, we have $(\beta_1 + \beta_2 + \beta_3) \leq \sqrt{3}$. Then:

$$\varepsilon_{(\overrightarrow{\alpha} \cdot \overrightarrow{\beta})} \leq 2^{-(n+1)}(\sum_{i=1}^{3} \alpha_i + \sum_{i=1}^{3} \beta_i) \leq \sqrt{3} * 2^{-n}$$

That means, we will lose one to two bits of accuracy for dot product of two unit vectors.

Error in the Square Root Operation

For lighting calculations we need to normalize the vectors to unit length before we compute the dot product. Normalization involves division by the magnitude of the vector which requires a square root operation. In order to perform all the operations in the fixed-point arithmetic, we use a table lookup to get the square root of an unsigned integer.

For an unsigned integer X with $2n$ bits of accuracy we take the most significant n bits (say X') as the lookup index into the square-root table to find the square root.

$$X' = (X >> n) << n$$

The maximum possible error of X' relative to X is 2^n (because the information in the lower n bits is lost). We can reduce this error by half, though. If the value

of the n^{th} bit of X is one, we can add one to $X >> n$, so that it becomes a kind of rounding error instead of truncation error.

Next, we use the square-root table to find the square-root of X'. Let this be a' in integer representation: $X' = a'^2$. Suppose the square root of X in integer representation is $a : X = a^2$. Let $a' = a + \varepsilon_a$ (ε_a is the error), then:

$$a'^2 = (a + \varepsilon_a)^2 = a^2 + 2a\varepsilon_a + (\varepsilon_a)^2 = X'$$

That is, $2a\varepsilon_a + (\varepsilon_a)^2 = X' - X \leq 2^{n-1}$.

If $X > 2^{2n-2}$, then $a > 2^{n-1}$, thus:

$$2a\varepsilon_a < 2a\varepsilon_a + (\varepsilon_a)^2 \leq 2^{n-1} \text{ and } \varepsilon_a < \frac{2^{n-1}}{2a} < \frac{2^{n-1}}{2*2^{n-1}} < \frac{1}{2}$$

Which means that if we use the most significant n bits of the unsigned integer as index into the square root table then as long as the integer is bigger than 2^{2n-2}, the result has n bits of accuracy.

Error in the Evaluation of Specular Exponentiation

To calculate the specular component of illumination, we have to compute the exponent of the dot product of the half-way vector (computed as the average of view vector and light vector) with the normal vector. Due to the fact that the dot product of two unit vectors is always smaller or equal to 1 and that we are only dealing with positive values of the dot product, we use 2^m to represent the largest value 1. The maximum possible representation error will be $\frac{1}{2}$, i.e., $2^{-(m+1)}$ relative to 1.

If ε_a is the error in the value a of the dot product, then:

$$(a + \varepsilon_a)^n \cong a^n + na\varepsilon_a \text{ (if } \varepsilon_a << a)$$

The maximum absolute error happens when $a = 1$, $\varepsilon_a = 2^{-(m+1)}$, and n is the maximum value of 128: (as implemented by OpenGL)

$$na\varepsilon_a < 128 * 2^{-(m+1)} < 2^{-(m-6)}$$

So we will have $m - 6$ bits accuracy in the result, i.e., we will lose 6 bits accuracy due to this exponentiation.

Putting it all together

From the above analysis, we can get an equation which relates the input data accuracy with the output color accuracy. Assume the output color needs n bits accuracy per R, G, and B, which requires m bits of accuracy in the input data. We next relate n and m.

First, the normalization of each vector will lose one bit. As shown before, the square root will have nearly the same accuracy as the input data. To avoid the loss of accuracy due to division, instead of storing the square root, we store the reciprocal of the square root in the lookup table. This reciprocal is calculated in the floating-point representation before converting it to the n-bit fixed-point representation. Thus the only error induced in the normalization will be in the final multiplication which is a loss of one bit of accuracy.

The dot product of two unit vectors will lose one to two bits of accuracy. Since the exponentiation will lose six bits, the term $(\max\{\overrightarrow{s} \cdot \overrightarrow{N}, 0\})^{\text{shin}}$ will lose $1 + (1 \text{ to } 2) + 6 = 8$ to 9 bits of accuracy. So the above term have between $m - 8$ and $m - 9$ bits of accuracy. Further, the term C_{specular} will have the same accuracy because

$\text{specular}_{\text{light}} * \text{specular}_{\text{material}}$ will have m bits of accuracy, which is much higher than the accuracy of $(\max\{\overrightarrow{s} \cdot \overrightarrow{N}, 0\})^{\text{shin}}$.

Similarly, the term C_{diffuse} will get between $m-2$ and $m-3$ bits of accuracy. C_{ambient} will have m bits of accuracy as explained in the overview.

Overall, $(C_{\text{ambient}}+C_{\text{diffuse}}+C_{\text{specular}})$ will have the accuracy of the least accurate term C_{specular}, i.e., $m-8$ or $m-9$ bits of accuracy.

Since the attenuation and the spotlight terms can all be evaluated with more than $m-8$ bits of accuracy, the required color accuracy bits for the entire illumination equation can be expressed as:

$$n = m - 8 \text{ or } m - 9$$

For example, if $n = 8$, i.e., eight bits per R, G, and B, then the required accuracy for the input data will be $n+8$ or $n+9$, i.e., we will need 16 or 17 bits to represent the input data to get the desired accuracy of 8 bits per color component.

View-dependent Variable-Precision Lighting

Similar to the case of transformation in Section 4.3.3, we can take advantage of the spatial coherence of the adjacent vertices in lighting calculations.The basic idea is that the viewing and lighting directions do not vary much for the spatially close vertices. Once we find those directions for one vertex, we can calculate the directions for the nearby vertices incrementally, i.e., calculate the difference in far fewer number of bits. The direction difference between the nearby vertices depends not only on the absolute spatial difference of the vertices, but also on their distances from the

viewer and light source to the vertices. Once the viewer moves closer to the vertex and below a threshold (which we will describe below) we will switch back to the original case, i.e., treat that particular vertex independently of its adjacent vertices.

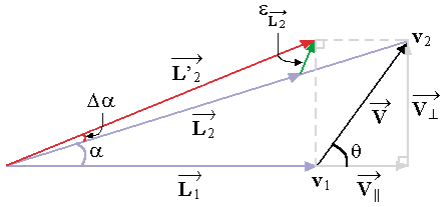


Figure 4.6: Incremental Lighting Calculation

In Figure 4.6 we show how to compute the lighting incrementally. Let $\overrightarrow{L_1}$ be the light vector for vertex v_1 for which we have already calculated the illumination. Now suppose we would like to find out the light vector $\overrightarrow{L_2}$ for its adjacent vertex v_2. The displacement vector $\overrightarrow{V}$ between v_1 and v_2 is normalized by the distance between the vertex v_1 and the light source, i.e., its length is equal to the real distance between v_1 and v_2 divided by the distance between the v_1 and the light source.[1] Both $\overrightarrow{L_1}$ and $\overrightarrow{L_2}$ are unit vectors.

One way to accurately compute $\overrightarrow{L_2}$ is to normalize the sum of $\overrightarrow{L_1}$ and the vector between v_1 and v_2. This approach requires roughly the same amount of computation as computing $\overrightarrow{L_2}$ directly from the vector between v_2 and the light source. To reduce the computation, we instead use $\overrightarrow{L'_2}$ (equal to $(\overrightarrow{L_1} + \overrightarrow{V_\perp})$) as the approximation of $\overrightarrow{L_2}$ if it satisfies our accuracy requirements. $\overrightarrow{V_\perp}$ is the component vector of $\overrightarrow{V}$ on the perpendicular direction of $\overrightarrow{L_1}$, which can be easily computed: $\overrightarrow{V_\perp} = \overrightarrow{V} - \overrightarrow{L_1}(\overrightarrow{V} \cdot \overrightarrow{L_1})$.

[1]Note that this assumption is not shown in Figure 4.6, where $\overrightarrow{V}$ is shown to have its length as the distance between v_1 and v_2.

Using this approach, the induced error $\varepsilon_{\overrightarrow{L_2}}$ is equal to $\overrightarrow{L'_2} - \overrightarrow{L_2}$.

If the length of $\overrightarrow{V}$, $\left\|\overrightarrow{V}\right\|$, is much smaller than 1 (the length of $\overrightarrow{L_1}$ and $\overrightarrow{L_2}$), then we have:

$$\begin{aligned}\left\|\overrightarrow{L'_2}\right\| &= \sqrt{\left\|\overrightarrow{L_1}\right\|^2 + \left\|\overrightarrow{V_\perp}\right\|^2} = \sqrt{1 + \left\|\overrightarrow{V_\perp}\right\|^2} \\ &\approx 1 + \frac{\left\|\overrightarrow{V_\perp}\right\|^2}{2} \leq 1 + \frac{\left\|\overrightarrow{V}\right\|^2}{2}\end{aligned}$$

Let the angle between $\overrightarrow{L_1}$ and $\overrightarrow{L_2}$ be α, between $\overrightarrow{L_2}$ and $\overrightarrow{L'_2}$ be $\Delta\alpha$, and $\overrightarrow{V_\parallel}$ be the component vector of $\overrightarrow{V}$ along the direction of $\overrightarrow{L_1}$:

$$\alpha = \arctan\left(\frac{\left\|\overrightarrow{V_\perp}\right\|}{\left\|\overrightarrow{L_1}\right\| + \left\|\overrightarrow{V_\parallel}\right\|}\right) = \arctan\left(\frac{\left\|\overrightarrow{V_\perp}\right\|}{1 + \left\|\overrightarrow{V_\parallel}\right\|}\right)$$

$$(\alpha + \Delta\alpha) = \arctan\left(\frac{\left\|\overrightarrow{V_\perp}\right\|}{\left\|\overrightarrow{L_1}\right\|}\right) = \arctan\left(\left\|\overrightarrow{V_\perp}\right\|\right)$$

$$\begin{aligned}\text{So} \qquad \Delta\alpha &= \arctan\left(\left\|\overrightarrow{V_\perp}\right\|\right) - \arctan\left(\frac{\left\|\overrightarrow{V_\perp}\right\|}{1 + \left\|\overrightarrow{V_\parallel}\right\|}\right) \\ &\approx \left\|\overrightarrow{V_\perp}\right\| \left(1 - \frac{1}{1 + \left\|\overrightarrow{V_\parallel}\right\|}\right) = \frac{\left\|\overrightarrow{V_\perp}\right\| \left\|\overrightarrow{V_\parallel}\right\|}{1 + \left\|\overrightarrow{V_\parallel}\right\|} \\ &< \left\|\overrightarrow{V_\perp}\right\| \left\|\overrightarrow{V_\parallel}\right\| \leq \frac{1}{2}\left\|\overrightarrow{V}\right\|^2\end{aligned}$$

The last inequality is because $\left\|\overrightarrow{V_\perp}\right\| = \left\|\overrightarrow{V}\right\| \sin\theta$ and $\left\|\overrightarrow{V_\parallel}\right\| = \left\|\overrightarrow{V}\right\| \cos\theta$, and $\sin\theta\cos\theta = \frac{1}{2}\sin(2\theta) \leq \frac{1}{2}$. Thus if the distance between v_1 and v_2 is much less than the distance between v_1 and the light source, then $\left\|\overrightarrow{L'_2}\right\| \approx 1 = \left\|\overrightarrow{L_2}\right\|$ and $\Delta\alpha << 1$, therefore

$$\left\|\varepsilon_{\overrightarrow{L_2}}\right\| \approx 2\left\|\overrightarrow{L_2}\right\| \tan\left(\frac{\Delta\alpha}{2}\right) \approx \Delta\alpha \leq \frac{1}{2}\left\|\overrightarrow{V}\right\|^2$$

This means, the error of using $\overrightarrow{L'_2}$ as an approximation of $\overrightarrow{L_2}$ is less than $\frac{1}{2}\left\|\overrightarrow{V}\right\|^2$. If we want 15 bits of accuracy in $\overrightarrow{L_2}$, we only need $\left\|\overrightarrow{V}\right\| \leq 2^{-7}$, i.e., the distance between v_1 to the light source should be $2^7 = 128$ times larger than the distance between v_1 and its adjacent vertex v_2. This way we only use the local spatial differences in calculating the new direction and avoid an expensive vector normalization operation.

(a) Floating Point Lighting (b) Variable-Precision Lighting (Speedup: 2.99)

Figure 4.7: Variable-Precision lighting of Bunny model

(Transformations for both cases have been calculated in floating-point)

4.3.5 Some Implementation Details

In addition to what we have already described in the previous sections, there are some other implementation details which are worth mentioning.

Batched Transformation and Lighting

Most graphics APIs (OpenGL, Direct3D, Glide) allow the user to transform and light the triangles one at a time and send the transformed and lighted triangles in floating-point screen coordinates to the rasterizer. Since these APIs do not accept screen-

space triangles in the fixed-point representation, we had to convert our fixed-point results to floating-point representation before asking the graphics API to rasterize the triangles. In MMX technology, this means that we need to reset the register flag back and forth when we switch from the integer operation to floating point because these two share the same registers. The frequent resetting costs time, so the intuitive solution is to minimize the number of resets, e.g., transform and light the whole dataset first in object space, then do the viewport transform and then send to the rasterizer. There are two problems with this approach. First we lose some opportunities of pipelining which the hardware is very smart at. Second, there are lots of extra memory accesses due to the write-back, so this does not work well.

To solve this problem, we make a tradeoff. Instead of transforming and lighting the triangle one by one or all at the same time, we do them batch by batch. The resetting of the flag only happens between batches and we avoid the extra memory accesses. In practice, we find batch size of several hundred triangles works gracefully. If the graphics APIs accepted screen-space fixed-point representation triangles, we would not have to deal with this and our results would have been better than reported here since switching from fixed-point to floating-point is expensive even when we do them in batches.

Full-precision Matrix Calculation

At each view point we first calculate the transformation matrix and then apply it to all the vertices in the dataset. The initial matrix calculation is a negligible fraction of the overall computation which includes transformation of hundreds of thousands

of vertices. So we compute it in full-precision floating point before converting it into the fixed-point representation. This way, we save the precision of the matrix elements, and avoid the possibility of error build up when we take advantage of the temporal coherence of the frames in transformation because the matrix is computed in full precision separately for each frame.

4.4 Results

We have tested our approach on polygonal datasets from several application domains including molecular, laser-scanned, mechanical CAD, and procedurally generated datasets. The results of our approach are summarized in Tables 4.1, 4.2, 4.3 and appear in Figures 1.3 and 4.4– 4.13.

Model		Bunny	DHFR	Dragon	Venus	AMR	Buddha
Size (triangles)		69K	145K	202K	268K	376K	1087K
Floating Point	Transform	61	130	185	230	330	968
	Lighting	469	1042	1374	1830	2503	7481
	Other	56	108	167	218	298	902
	Total	586	1280	1726	2278	3131	9351
Variable Precision	Transform	17	33	46	59	83	235
	Lighting	79	155	212	280	337	882
	Other	42	87	127	160	212	616
	Total	138	275	385	499	632	1733
Speedup		4.25	4.65	4.48	4.57	4.96	5.40
e_{rms} (object space)		1.3e-4	1.3e-4	1.2e-4	1.2e-4	1.1e-4	1.2e-4
Max error (obj. space)		3.0e-4	3.1e-4	3.0e-4	2.9e-4	2.6e-4	3.1e-4
e_{rms} (image space)		8.5e-3	8.8e-3	8.7e-3	6.0e-3	8.4e-3	7.0e-3

Table 4.1: Results from rendering at varying precisions

Table 4.1 compares the results using variable precision with the one using

traditional single-precision floating point and times are reported in milliseconds. The variable precision rendering shown here is under the requirements of guaranteed pixel-level position accuracy and eight bits per R, G, and B color. The object space root-mean-square error and maximum error are measured in transformed object space as the distance between the single-precision floating-point transformed vertices and variable-precision transformed ones, while the image space root-mean-square error is measured in the final image space as the difference between the R, G, B color components. The formula for image space root-mean-square error as follows:

$$e_{rms} = \left[\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[\widehat{f}(x,y) - f(x,y) \right]^2 \right]^{1/2}$$

Here $f(x,y)$ represents the original image, $\widehat{f}(x,y)$ denotes an estimate of the image, and $M \times N$ is the image size.

From Table 4.1, we can see that under the pixel-level accuracy, the maximum transformed distance between the two methods is less than 0.00033 of the size of the bounding box for all the six datasets tested . We know the normalized transformed object space is in the range [-1.0, +1.0], so the difference is less than six-thousandth of the total range. This shows robustness of our method. Further, instead of getting pixel-level accuracy, our method actually gave us 2 to 3 sub-pixel bits of accuracy. This is because our error analysis gives the upper bounds of the error; the real error is usually much less. To roughly compare how variable precision rendering stacks up against multiresolution rendering, we compared the object-space Hausdorff error in a 16K triangle model of the Bunny using Metro [23] against a 69K triangle model of the Bunny using 7.9 bits/vertex coordinate. Although both give a factor of 4

speedup, the variable precision method has an order of magnitude smaller object space Hausdorff error (0.012% of the bounding box diagonal) compared with 16K triangle full precision model (0.12% of the bounding box diagonal).

We can see more than a factor of four speedup in all the datasets tested. One aspect of our algorithm is that it scales well. The speedup factor goes up with the increase in scene complexity (which means more data will be rendered in less precision) and the number of light sources. See Figure 4.8 and Table 4.1.

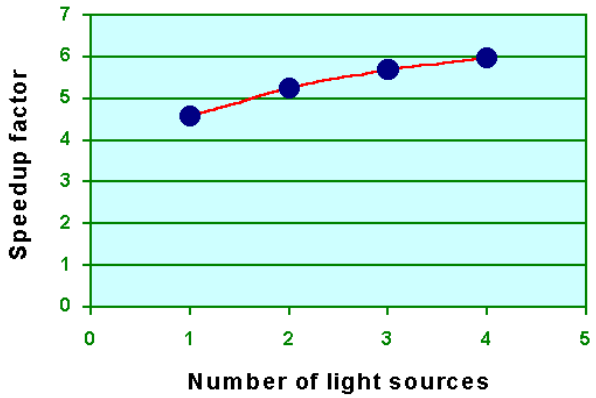


Figure 4.8: Speedup factor as a function of number of light sources (Venus model)

Output	Conventional		Spatial		Spatio-Temporal	
bits	Add	Mult	Add	Mult	Add	Mult
32 bits	32	32	42.21	31.55	54.08	29.23
16 bits	16	16	12.95	7.77	13.77	4.02
8 bits	8	8	1.35	0.66	0.77	0.08
4 bits	4	4	0.03	0.02	0.01	0.001

Table 4.2: Average number of bits per vertex coordinate operated upon for appropriate output precision

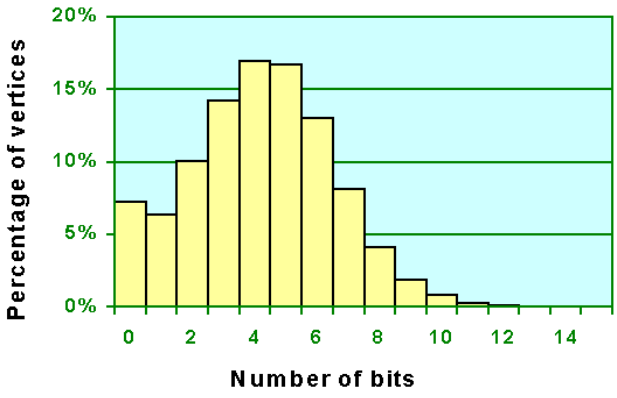


Figure 4.9: Histogram of vertices transformed in different number of bits using Variable Precision (AMR model)

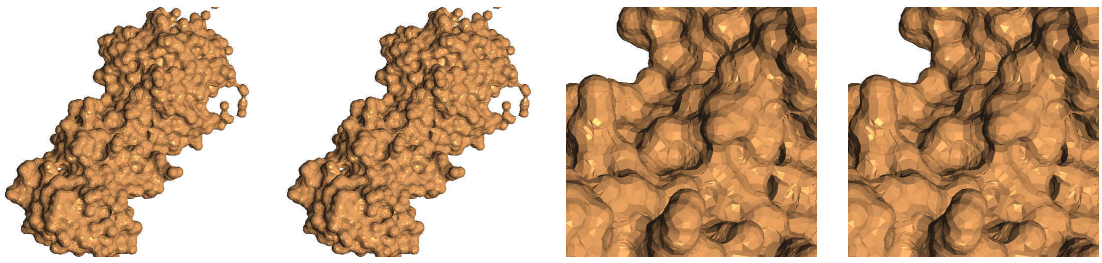
Figure 4.9 shows the histogram of percentage of vertices transformed in different number of bits for AMR dataset, which has a very low average 4.18 bits/vertex coordinate for variable-precision transformation, instead of 32 bits/vertex coordinate as in the single-precision floating point case. Figures 4.10–4.13(FP abbreviated for floating point, VP abbreviated for variable precision) show the images rendered by variable-precision rendering and compare them with the single-precision floating point rendering. Even with zoomed-in views, there are hardly any visually distinguishable differences.

Table 4.2 shows the average number of bits that have to be manipulated per vertex during the transformations while exploiting spatial and temporal coherences. Since the vertices that are at the lower levels of the octree require less number of bits for transformation, the overall average number of bits turns out to be much less. The leftmost column indicates the number of bits that are required in the output display.

Output bits	Conventional		Spatial		Spatio-Temporal	
	Add	Mult	Add	Mult	Add	Mult
32 bits	6	8	7.92	7.89	10.14	7.31
16 bits	3	4	2.43	1.94	2.58	1.01
8 bits	1.5	2	0.25	0.17	0.14	0.02
4 bits	0.75	1	0.01	0.004	0.002	0.0002

Table 4.3: Average number of equivalent 32-bit operations per vertex coordinate for appropriate output precision

Table 4.3 shows the average number of equivalent 32-bit operations per vertex during the transformations while exploiting spatial and temporal coherences. Central to this idea is that one 32-bit operation is equivalent to two 16-bit, four 8-bit, and eight 4-bit operations. Even though SIMD parallelism at the level of 4-bits is not yet available in the current generation of processors, the table shows the effectiveness of our scheme if such parallelisms were to become available in future. As



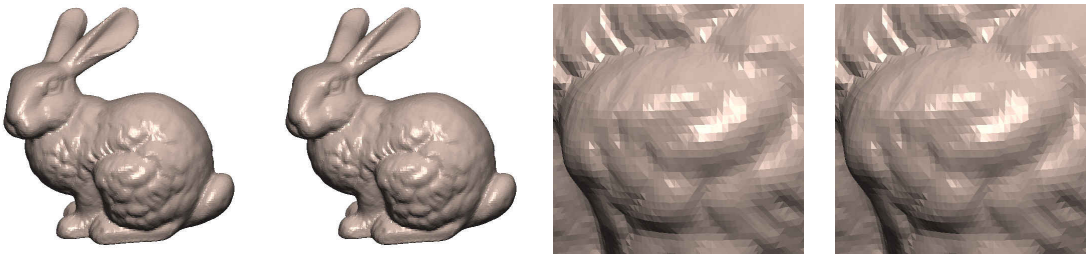
(a) Floating Point (32 bits/Vert. Coord.)

(b) Variable Precision (7.9 bits/Vert. Coord.)

(c) Floating Point Closeup

(d) Variable Precision Closeup

Figure 4.10: Dihydrofolate Reductase Molecular Surface (145K triangles) rendered in variable precision



(a) Floating Point (32 bits/Vert. Coord.) (b) Variable Precision (7.9 bits/Vert. Coord.) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 4.11: Stanford Bunny (69K triangles) rendered in variable precision

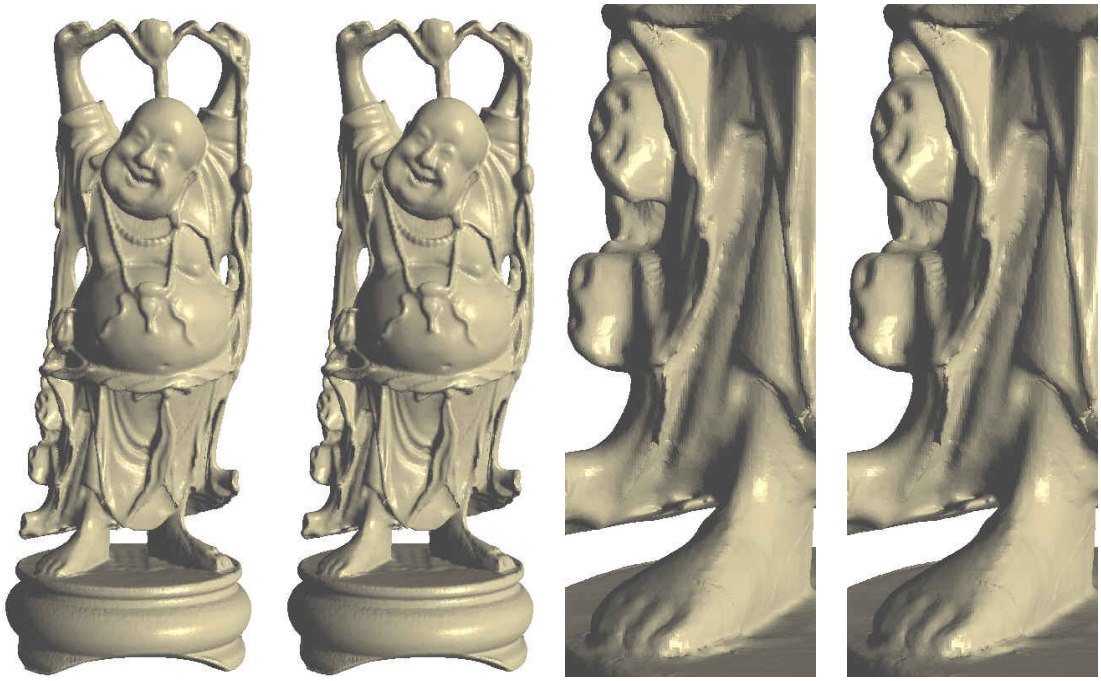
(a) Floating Point (32 bits/Vert. Coord.) (b) Variable Precision (7.9 bits/Vert. Coord.) (c) Floating Point Closeup (d) Variable Precision Closeup

Figure 4.12: Cyberware Venus (268K triangles) rendered in variable precision

in Table 4.2, the leftmost column indicates the number of bits that are required in the output display.

4.5 Conclusions

Our variable-precision approach takes advantage of SIMD parallelism in modern processors to speedup the transformation and lighting stages of the graphics pipeline. It can successfully trade-off precision for speed without significantly affecting the



(a) Floating Point (32 bits/Vert. Coord.)

(b) Variable Precision (7.9 bits/Vert. Coord.)

(c) Floating Point Closeup

(d) Variable Precision Closeup

Figure 4.13: Buddha Model(1087K triangles) rendered in variable precision

visual quality of the rendered images. In addition, our method is complementary to the conventional multiresolution approaches.

Chapter 5

Interactive Visualization of Large Time-Varying Molecules

Interactive visualization of molecular datasets is an important tool to better understand the structural and functional properties of biological samples. It remains a challenge to interactively display large molecular datasets, especially time-varying ones. In this chapter, we develop a time- and memory-efficient algorithm to solve this problem [57]. Our approach speeds up the graphics rendering pipeline at several stages by developing and extending various rendering techniques for efficient display of time-varying molecular data, such as view-dependent precision control as discussed in Chapter 4, run-time triangle strip and triangle fan generation, visibility-based culling, and memory-bandwidth reduction. More importantly, our algorithm requires no pre-processing and little memory overhead. It is linearly scalable in the sizes of the molecular datasets. Our algorithm is flexible and scalable and our ideas for this problem can also be applied to visualization of other large time-varying datasets.

5.1 Previous and Related Work

Interactive visualization of large datasets has remained one of the major challenges for computer graphics and scientific visualization researchers. Many techniques, such as level-of-detail hierarchies, triangle-strip generation, and occlusion-based culling, have been developed for speeding up the visualization of large datasets, especially static scenes.

Multi-resolution hierarchies for level-of-detail-based rendering have traditionally involved modelling each object at multiple levels of detail. The detail is usually measured in the number of geometric primitives required for representation. A high-detail triangle-mesh object will require a higher number of vertices, edges, and triangles. At run-time, an appropriate level of detail is selected based on viewing parameters for a faithful representation. Even better, level of detail can be applied in a view-dependent manner to take advantage of temporal coherence and adaptively refine or simplify the geometry between adjacent frames [94]. Normally the multi-resolution hierarchies of the geometry are built as a pre-process.

Triangle strips provide a compact representation of triangular meshes and are supported by popular graphics APIs such as OpenGL. The use of triangle strips results in fast rendering and transmission. A triangle strip with n triangles can be rendered with only $n + 2$ instead of $3n$ vertices. Thus substantial savings for memory bandwidth and computation of per-vertex operations such as transformations, lighting, and clipping can be achieved. Triangle strips can be generated as a preprocess and stored with the geometry for later usage [38], or can be pre-generated

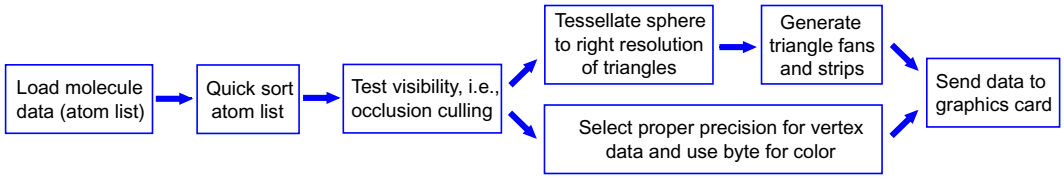


Figure 5.1: Pipeline of our run-time algorithm

and later updated view-dependently [36]. It can be costly to generate triangle strips from scratch at run-time [36]. Triangle strips can also be used for polygonal mesh compression [51, 136].

Occlusion culling works by culling away portions of the scene that are not visible from the viewer. Culling can be done in object space through the use of spatial partitions or bounding volume hierarchies [28, 68]. Object-space algorithms have been developed for several specialized environments such as architectural or urban datasets [1, 68, 146]. However such techniques are not very effective on scenes with several small occluders. Culling can also be done in image space using hierarchical Z-buffer [49] or hierarchical occlusion maps [152]. Image-space occlusion culling usually achieves better occluder fusion. Normally occlusion culling is done conservatively [79, 149]. Non-conservative culling [78] can lead to popping artifacts when objects appear and disappear between adjacent frames.

Most of the above techniques rely on a certain level of pre-processing of the scenes and build the appropriate data structures before the rendering phase. Hence they are well-suited for scenes with static geometry. For time-varying scenes, especially molecular dynamics simulations with significant changes, these techniques can not be readily applied.

5.2 Our Approach

The space-filling display of time-varying molecules involves rendering each atom of the molecule as a sphere with a van der Waals radius for every time frame. Different atoms are represented by different-sized spheres, with possibly different resolution tessellations. Since viewing individual atoms is not a normal real-life experience, parallel projections are often considered more informative in molecular graphics than foreshortened perspective projection. Here we assume parallel projection.

Figure 5.1 shows the pipeline of our algorithm for displaying each frame of a time-varying molecular dynamics data. We start by loading the list of atoms with their 3D positions for current time frame, and we sort them according to their distance from the viewer using a quick-sort algorithm. Next we determine the visibility of each atom, by using our visibility-based culling algorithm (detailed in Section 5.2.1). We use multi-resolution techniques to decide the appropriate number of triangles with which to represent the spherical atoms. We also decide the necessary precision for vertex data from display resolution specification. For the triangles that survive the back-face culling phase we generate triangle strips and compute illuminated color. Finally, we send the triangle strips and triangle fans with appropriate precision to the graphics card for rasterization and display.

5.2.1 Determination of the Visible Set of Atoms

Previous approaches for occlusion culling deal with general environments. They achieve efficiency of the occlusion test by pre-processing the scene and building

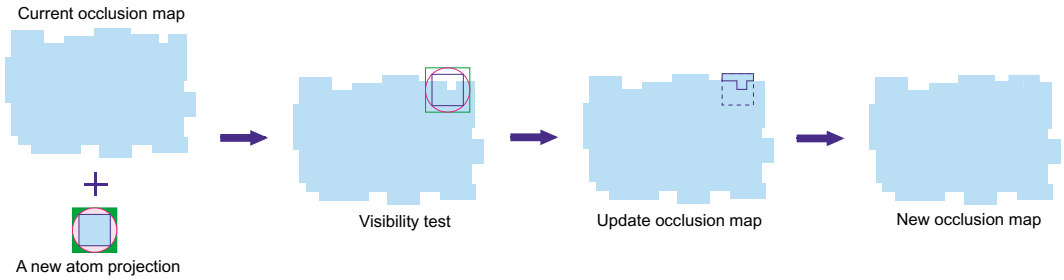


Figure 5.2: Visibility test of an atom

a good data structure such as a cluster hierarchy [149] to store the relationships between objects. Run-time efficiency is achieved through temporal coherence, since the viewing parameters seldom change significantly from one frame to the next. Occlusion culling for time-varying molecules is different from previous situations. First, molecules go through large structure changes, so the occlusion information between adjacent frames may change significantly. This makes it difficult and less efficient to use temporal coherence by pre-processing the scene. Second, there are no large occluders in time-varying molecules. Each molecule consists of thousands of atoms whose sizes are of the same order of magnitude. The relationships among this large number of small potential occluders vary significantly over time. Thus, instead of trying to use pre-processing with temporal coherence, we decided to build per-frame occlusion map on-the-fly to achieve better efficiency. We use the culling

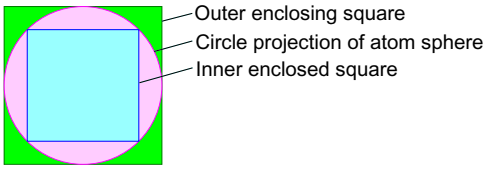


Figure 5.3: Over- and under-estimation for Occlusion Culling

algorithm described below to build per-frame occlusion map and estimate the visible set of atoms.

The list of atoms for each frame is sorted using the quick sort algorithm. We use a conservative culling scheme to determine the potentially visible atoms. Since our algorithm is conservative, it is possible for a few non-visible atoms to be sent to the graphics card for rendering. Figure 5.3 shows the conservative nature of our culling. Each spherical atom is projected to a circle on the image plane under parallel projection. We project the atoms in a front-to-back order. If all the screen-space pixels of an atom are already occupied by the nearer atoms, then the current atom will not be visible. Since the projection and checking for overlap of circles is hard to implement efficiently, we instead use two different-sized nested squares. As shown in Figure 5.3, the blue square covered by the projected circle is used to represent the definite occlusion by this atom for the atoms behind it. We use the inner square of each atom to build the occlusion map and the outer green square (in Figure 5.3) to check if the atom has been blocked by previously rendered atoms. An example is shown in Figure 5.2. Here the atom is visible since its outer square has not been totally blocked. The occlusion map is then updated using the atom's inner square.

For memory and time efficiency, we use a bit pattern to store the occlusion map and check for atom visibility. Initially, each bit is set to zero to indicate non-occupancy. The bit is set to one when the pixel gets covered by the inner square of an atom's projection for the first time. The pixel will from then on act as an occluder for the atoms that project on it later. The bit pattern storage using

integers improves the memory usage. For a 1024×1024 image, we only need $128K$ bytes for storing the occlusion map. More importantly, storing the bit pattern as packed integers improves efficiency. As an example, if a new atom is visible and we want to adjust the occlusion map according to its inner-projected square, then we just set the occlusion map pixels covered by the square to one. So we can use a bitwise-OR operation of the values of the covered pixels with an all-one pattern to simultaneously cover several pixels in a single operation. Similarly if we want to check for visibility of a new atom, we need only to use a bitwise exclusive-OR operation of the values of the covered pixels with an all-one pattern and check if the result is zero.

5.2.2 Generation of Appropriate Triangle Tesselations of Spheres

Recent multi-resolution literature [39, 94] has discussed the interactivity and visual realism tradeoffs in selecting an appropriate resolution for geometry. The screen-space size of an atom is an important determinant for picking the number of triangles for representing a sphere. We pick the tesselation resolution such that each triangle

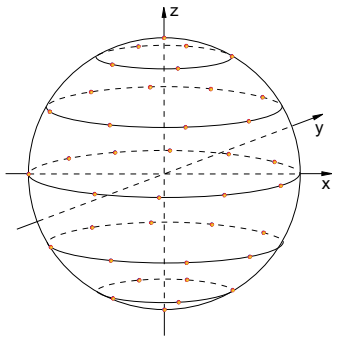
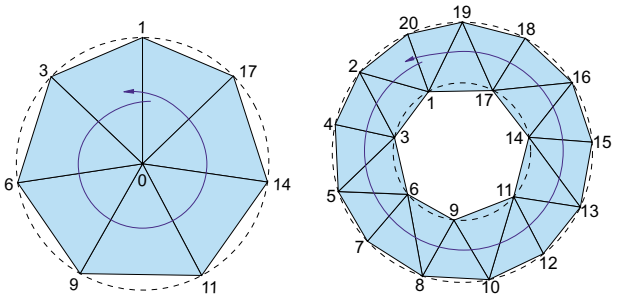


Figure 5.4: Generating points on a sphere



(a) Triangle fan (b) Triangle strip

Figure 5.5: A complete triangle fan and triangle strip as seen from above the North pole of a sphere

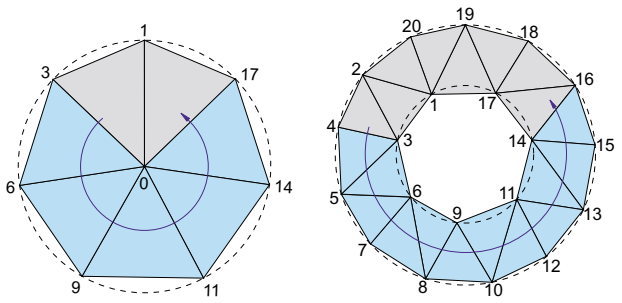
will have about ten pixels on the image plane.

For flexibility in adjusting the tesselation resolution, we generate points on spheres along circles with same latitude (i.e. same angle to the z–axis) and symmetric over the x–y plane as shown in Figure 5.4. We then connect the points to form triangles. Points connected to the North or South pole will form a complete triangle fan. One such triangle fan $(0, 1, 3, 6, 9, 11, 14, 17)$ is shown in 5.5(a), where the i^{th} triangle is described by the 0^{th}, i^{th}, and $(i+1)^{st}$ vertices in the sequence. The points between adjacent circles form a complete triangle strip. One such strip $(20, 1, 2, 3, 4, \mathbf{3}, 5, 6, 7, \mathbf{6}, 8, 9, 10, 11, 12, \mathbf{11}, 13, 14, 15, \mathbf{14}, 16, 17, 18, \mathbf{17}, 19, 1, 20)$ is shown in 5.5(b). Note that this is generalized triangle strip [38] where the repeated vertices are shown in bold.

5.2.3 Run-time Triangle Strip and Triangle Fan Generation

After we decide the appropriate tesselation of an atom, we know that sending triangle strips or fans to the graphics card is more efficient than sending separate triangles.

We can generate the triangle strips and fans easily from our tesselation scheme in Section 5.2.2 using traditional methods. However, that will not always be the best solution. A pre-generated triangle strip is fixed and will include both visible and non-visible triangles for each viewing direction. Even if an atom is visible, its back-facing triangles will not be visible. Pre-generated triangle strips can be updated at run-time for complex geometry [36]. However, here we observe that we can take advantage of the spherical atoms to generate the proper triangle strips and fans for their front-facing triangles. Thus we can benefit from the efficiency of triangle strips without the need to send back-facing triangles or to update the triangle strips for every frame.



(a) Front-facing fan (b) Front-facing strip

Figure 5.6: Triangle fan and triangle strip of front-facing triangles (in blue) as seen from above the North pole of a sphere

We first decide which triangles are front-facing. The front-facing triangle is defined as one with at least one front-facing vertex. The front-facing vertex can be easily determined by a dot product of its normal with viewing direction. Then we generate the triangle fan for triangles consisting of front-facing vertices connecting to the pole of the sphere, and triangle strips for triangles consisting of front-facing

vertices between adjacent circles on the sphere. As an example, Figure 5.6(a) shows a run-time triangle fan $(0, 3, 6, 9, 11, 14, 17)$ of Figure 5.5(a), while Figure 5.6(b) shows a run-time triangle strip $(4, 3, 5, 6, 7, \mathbf{6}, 8, 9, 10, 11, 12, \mathbf{11}, 13, 14, 15, 16)$ of Figure 5.5(b). The grey triangles in the figure are back-facing triangles.

An alternative to run-time generation of triangle strips and triangle fans is to generate a fixed set of triangle strips and triangle fan for the visible hemisphere, and rotate spheres at run-time to keep their orientation relative to the viewer fixed.

5.2.4 Memory Bandwidth Reduction

To further improve the memory and run-time efficiency, we adapt the variable-precision approach [54] to reduce the precision of the vertex data. As shown in [54], we need no more than 16 bits of accuracy to represent vertex data for pixel-level positional accuracy in up to $2^{13} \times 2^{13}$ rendering window under parallel and $2^{11} \times 2^{11}$ window under perspective projection. In this work we use 16 bits instead of 32 bits (floating-point representation) to reduce the memory bandwidth by half.

We also save some bandwidth by computing the color of vertices on the CPU and sending only the unsigned byte color (three bytes total for red, green, and blue) to the graphics card, instead of sending a vector vertex normal.

We have also used display list mechanism provided by OpenGL to get better memory coherence to display atoms. At each frame, we generate a new display list for each *type* of atom (carbon, oxygen, nitrogen, hydrogen, etc). Every visible atom is just a differently translated instantiation of the display list containing the triangles for its canonical representation.

5.3 Results and Discussion

We have applied the approach described above to ion-channel studies, specifically, a two hundred-frame animation of the structure of the large-conductance mechanosensitive channel MscL as it transitions from the closed to the open state. The animation is based on five models representing the closed, open and three intermediate conformations. The smooth transition between these modeled states has been implemented using a linear interpolation algorithm.

MscL is a ubiquitous part of the osmoregulation system residing in the cytoplasmic membrane of most bacteria, both free-living and pathogenic. *Escherichia coli* MscL (EcoMscL) is the best understood model mechanosensitive channel gated directly by membrane tension. The atomic-scale model of EcoMscL based on the crystal structure of its homolog from *Mycobacterium tuberculosis* was built in order to relate the structural information to the wealth of experimental data available specifically for the *E. coli* channel. In the closed state the channel core is represented as a tightly packed bundle of ten transmembrane alpha helices (Fig. 1.4a, b). The opening transition driven by external tension has been modeled as an iris-like expansion of the transmembrane bundle accompanied by tilting of alpha-helices. The open conformation is characterized by a large (3 nm) pore capable of passing a large current or a flux of small omolytes (Fig. 1.4g, h). The last row of images (Fig. 1.4i, j) represent the intermediate semi-closed conformation.

5.4 Conclusions

In this chapter we show that large time-varying molecular datasets can be displayed interactively using our time- and memory-efficient algorithm. Various techniques have been developed or extended to accelerate the graphics rendering pipeline. Our algorithm has several properties which makes it very attractive. It has no memory overhead, requires no pre-processing, and is linearly scalable. Though the techniques have been designed for time-varying molecular datasets, the concepts are general enough to benefit interactive display of large time-varying datasets in other application domains as well.

Chapter 6

Real-Time Rendering of Translucent Materials

Interactivity is an important goal not only for scientific simulation and visualization, but also for graphics rendering. Another long-standing goal of computer graphics is creating high-quality images which are indistinguishable from photographs. However, these two goals have historically been at odds with each other. Photorealism has resulted in beautiful pictures, but at the cost of slow algorithms taking hours to days. Interactivity is normally achieved at the cost of sacrificing some degree of realism. It is desirable to combine the two goals and achieve real-time photorealistic rendering.

Among various of interactions between light and matter, subsurface scattering is one of the most complicated to simulate. The complexity comes from the fact that the incident light gets partly absorbed and is then re-emitted. This process may occur many times before the light finally gets out of the object. Scattering effects are important to accurately simulate the appearance of translucent materials, such as human skin, clouds, marble, and milk.

In previous chapters, I have shown that efficient geometry representations, such as adaptively controlled irregular grids, variable-precision representations, and spherical harmonic representations, can be used to achieve the goal of interactive simulation and visualization of protein properties. I will show in this chapter that

similar data representations used for illumination data can help us achieve real-time rendering of translucent materials with optimal linear complexity in the number of surface points with minimal memory overhead.

6.1 Introduction and Related Work

Illumination models are important for photo-realistic image synthesis. Correctly modelling the physical interaction of light with objects is an exciting, but difficult task. Over the years, many illumination models have been developed for image synthesis. They can be classified as either empirically-based or physically-based. For example, the Phong illumination model [110] is an empirically-based model. Physically-based models are derived from principles of light-object interaction, using either geometrical optics or wave optics. Most of them model the bidirectional reflectance distribution function (BRDF).

One example of physically-based models using geometrical optics is the Cook-Torrance [27] model, which can compute directional distribution of light and color shift with incident angles and materials. Other geometrical-optics-based models include microfacet-based approaches [3, 14]. Inverse rendering methods can produce high-quality illumination models from images [12, 19, 30, 114, 123, 150]. Significant efforts have been devoted to determining the BRDF of an object. Researchers have also developed methods to directly measure the BRDF [48, 95, 142].

Compared with geometrical-optics-based models, wave-optics-based models are more complicated, but have the advantage of being able to model phenom-

ena which cannot be directly modelled using geometrical optics, such as interference and diffraction patterns. Kajiya [74] has used scalar-form Kirchhoff approximation to compute the BRDF of surfaces with anisotropy. He *et al.* [58] have presented a general local reflection model based on vector-formed Kirchhoff wave diffraction theory and have given an analytical formula to compute the BRDF for surfaces with roughness, including polarization and directional Fresnel effects. Bahar and Chakrabarti [6] have computed the differential scattering cross-section of a wave from rough metallic surfaces using electromagnetic theory. Stam [128] and Sun *et al.* [132] have extended the He-Torrance model [58] to handle anisotropic reflections and demonstrated diffraction effects on a compact disk.

A good BRDF model, either derived or measured, can give highly realistic visual effects. The basic assumption of BRDF models is that light enters and exits an object on the same surface point. In most cases this assumption is valid and the resulting BRDF models provide convincing visual appearance for simulating many visual effects. But for some cases, the assumption is not valid. For example, BRDF models alone are inadequate to simulate the appearance with subsurface scattering, where light enters an object at one point and exits at another. This effect is very important for simulating the appearance of translucent materials, such as marble, skin, and milk. To simulate these materials, we have to go back to the more general bidirectional surface scattering reflectance distribution function (BSSRDF) models. While BRDF models are just approximations of BSSRDF models.

Many researchers have successfully simulated subsurface scattering effects. Hanrahan and Krueger [52] have modelled subsurface scattering in layered surfaces

in terms of one-dimensional linear transport theory, and derived analytical expressions for single scattering events. They have incorporated their results into a BRDF model. The model is fast but also has the shortcoming of the BRDF assumption. More recently, Dorsey *et al.* [32] have simulated subsurface transfer by solving the radiative transfer equation using photon maps. Koenderink and van Doorn [83] model light scattering in translucent objects as a diffusion process. Stam [129] used a discrete-ordinate solution of the radiative transfer equation to model multiple anisotropic scattering for human skin layer bounded by two rough surfaces. Another contribution of [129] is derivation of a bidirectional transmittance distribution function (BTDF) to complement BRDF models. Pharr and Hanrahan [109] have taken a different approach. Instead of simulating energy transport, they have focused on scattering behavior and solve a non-linear integral scattering equation using Monte Carlo evaluation. Jensen *et al.* [72] have used path tracing to simulate subsurface scattering in wet materials.

The approaches above are able to simulate all the effects of subsurface scattering and generate impressive images, but are slow. Jensen *et al.* [73] have suggested a more efficient approach to simulate scattering media by using a dipole diffusion approximation for multiple scattering events, with an exact solution for single scattering events. With this simple approximation, they achieve more than two orders of magnitude speedup compared with the approach of using full Monte Carlo simulation. As an example, for one scene they have reduced the rendering time from 1250 minutes to 5 minutes with nearly indistinguishable visual difference. Jensen and Buhler [71] have taken this one step further. They decouple the computation of

the incident illumination from the evaluation of the BSSRDF with a two-pass approach. The first pass samples the irradiance at selected points on the surface. The second pass evaluates the diffusion approximation using a fast hierarchical scheme. They achieve up to 7 seconds per frame using ray-tracing for a teapot dataset with $150K$ vertices, using a dual 800MHz Pentium III PC. Lensch *et al.* [89] have used a preprocessing step to compute the impulse response for each surface point under subsurface scattering. They separate the response into a local and a global effect. While the local effect is modelled as a filter kernel and stored in a texture map, the global response is stored as vertex-to-vertex throughput factors. The local and global responses are combined during run-time to form the final image. They achieve 5 frames per second on a dataset with about $9K$ vertices, using a dual 1.7GHz Xeon computer. In addition, they can accommodate non-homogenous material properties. All these make practical simulation of subsurface scattering phenomena feasible. The next step is to enable subsurface scattering effects for interactive rendering of larger datasets.

Recently, Sloan *et al.* [126] have incorporated surface scattering effects into their pre-computed radiance transfer scheme and have achieved 27 frames per second on a Buddha dataset with $50K$ vertices, using a 2.2GHz Pentium 4 machine. They represent pre-computed view-independent subsurface-scattered radiance using low-order spherical harmonics. In addition to subsurface scattering effects, their scheme has successfully simulated many of the global illumination effects, such as soft shadows, inter-reflections, and caustics. If their approach is used only for simulating subsurface scattering effects, they can achieve significantly faster frame rates. They

have assumed low-frequency lighting environments. We instead, focus on subsurface scattering effects, but for high-frequency lighting environments (for example, a single directional point light source). Carr *et al.* [20] have modelled multiple-scattering subsurface light transport to resemble a single radiosity gathering step. By using their GPU algorithm for radiosity with a hierarchy of precomputed subsurface links, they have achieved about 30 frames per second for a dataset with $70K$ triangles, using GeForce FX graphics card. Mertens *et al.* [98] use a hierarchical boundary element method to solve the integral describing subsurface scattering and achieve more than 5 frames per second on a dataset with $132K$ triangles, using a dual 2.4GHz Xeon computer. Their algorithm allows users to change object geometry, subsurface scattering properties, lighting, as well as viewpoint at run time. Dachsbacher and Stamminger [29] extend shadow maps to store depth and incident light information, and compute subsurface scattering effects by filtering the shadow map neighborhood using a hierarchical approach. They have implemented their algorithm on graphics hardware and achieved 5.7 frames per second on a dataset with $100K$ vertices, using a 2.4GHz Pentium 4 machine with ATI Radeon 9700 graphics card.

We have built a simpler, approximate model [53, 56] for subsurface scattering and incorporated it into a local illumination model to make the effects more widely accessible for different applications. Our approach is based on the observation that subsurface scattering, although a global effect, is largely a local one due to its exponential falloff, which limits the volume it can affect. Therefore even though the light does not necessarily exit an object at the same point where it enters, as required by a BRDF model, it will for all practical purposes exit within a short distance of

its entry point. This enables us to make modifications to existing local illumination models to accommodate subsurface scattering effects. We approximate the BSSRDF for subsurface scattering based on both, the underlying physical processes and visual appearance. Jensen and Buhler [71] have shown that the visual appearance for translucent materials can be almost entirely simulated by only considering multiple scattering. We have used this fact and developed a macroscopic appearance-driven approach to capture the most important features of subsurface scattering: multiple scattered reflection and transmission. We modify local illumination process into a run-time two-stage process: a traditional local lighting stage and a scatter-bleeding stage. We then merge the run-time two-stage process into a run-time single-stage process by using pre-computed integrals and improve the complexity of our run-time algorithm from $O(N^2)$ to $O(N)$. The local illumination characteristics and the preprocessed scattering neighborhood information make our approach very efficient. In addition, we greatly reduce memory storage requirements for our pre-computed integrals by using reference points with spherical harmonics. We demonstrate that using only low-order spherical harmonics for representing pre-computed integrals produces somewhat unsatisfactory image quality for high frequency lighting (e.g., single directional light source). To address this we have designed a reference points scheme. In our scheme we select a subset of the input mesh vertices and store the pre-computed integrals at these reference points. We use spherical harmonics for efficiently representing low frequency integral differences between the reference points and the rest of the mesh vertices. This results in little extra storage for pre-computed integrals (less than 28 bytes per vertex) without loss of image quality

and a further improvement in the efficiency of our algorithm.

6.2 Subsurface Scattering Model and Our Simplifications

To describe subsurface scattering effects for translucent (i.e., highly-scattering) materials, we need general BSSRDF models instead of BRDF models. A BSSRDF model relates the illumination of one surface point with light distribution at other surface points by the following formula [73]:

$$dL_o(x_o, \overrightarrow{\omega_o}) = S(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o}) d\Phi_i(x_i, \overrightarrow{\omega_i})$$

where $L_o(x_o, \overrightarrow{\omega_o})$ is the outgoing radiance at point x_o in direction $\overrightarrow{\omega_o}$, $\Phi_i(x_i, \overrightarrow{\omega_i})$ is the incident flux at point x_i in direction $\overrightarrow{\omega_i}$, and $S(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o})$ is the BSSRDF. Thus the total outgoing radiance is computed by an integral over incoming directions and area A [73]:

$$L_o(x_o, \overrightarrow{\omega_o}) = \int_A \int_{2\pi} S(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o}) L_i(x_i, \overrightarrow{\omega_i}) (\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) d\omega_i dA(x_i)$$

where $\overrightarrow{n_i}$ is the surface normal at x_i. As can be seen in Figure 6.1, the effect of BSSRDF results in a scatter-bleeding of the illumination for a surface point from its neighborhood.

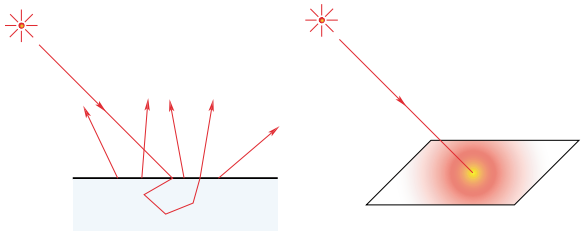


Figure 6.1: Scattering of light in BSSRDF models (based on [Jensen *et al.*,2001])

In the following sections, we will assume static geometry with homogeneous translucency and given scattering properties where multiple scattering dominates, and each vertex of the mesh represents a small area on the surface.

6.2.1 Locality of Subsurface Scattering Effects

To improve efficiency and achieve interactive frame rates for simulating translucent material properties, we incorporate subsurface scattering into local illumination. The main rationale in combining local illumination with subsurface scattering is based on the key observation that the subsurface scattering effects are well localized. First, scattering within one object will have very little effect on the appearance of another object; the influence between different objects can be well described by the reflectance values on their surfaces only. So unlike the situation addressed by radiosity methods where every patch has an effect on every other patch in the same scene, subsurface scattering only has prominent effect within an object. Second, even within the same object, the subsurface scattering due to light entering from one surface point will have little effect on another surface point on the same object if the distance between the two points is large. This property is a result of the exponential falloff of light intensity due to absorption and scattering within the material. Therefore, subsurface scattering, although a global illumination property in the sense that the illumination on one surface point is affected by the illumination on other surface points, is still largely a local effect. Although, the local-effect property of subsurface scattering is useful for efficiency reasons at the preprocessing stage, it will not affect the run-time efficiency of our algorithm.

We therefore conclude that to model the appearance of a surface point with subsurface scattering to a first approximation, we only need to know its scattering neighborhood and associated material properties.

6.2.2 Multiple Scattering Approximation

As mentioned earlier, a BSSRDF model is needed to describe subsurface scattering effects. The complete BSSRDF model S for subsurface scattering is a sum of a single scattering term $S^{(1)}$ and a multiple scattering term S_d [73]:

$$S(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o}) = S^{(1)}(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o}) + S_d(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o})$$

Jensen and Buhler [71] have shown that multiple scattering alone is sufficient to simulate the visual appearance of highly-scattering translucent materials. We follow their results and focus here on modelling multiple scattering effects only. Jensen *et al.* [73] have also shown that the dipole diffusion method is a good approximation for volumetric effects due to subsurface multiple scattering. The dipole approximation of the diffusion equations is expressed by the following formula:

$$S_d(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o}) = \frac{1}{\pi} F_t(\eta, \overrightarrow{\omega_i}) R_d(\|x_i - x_o\|) F_t(\eta, \overrightarrow{\omega_o})$$

where F_t is the Fresnel transmission term and R_d is the single dipole approximation for multiple scattering [71]:

$$\begin{aligned} R_d(r) &= -D\frac{(\overrightarrow{n} \cdot \overrightarrow{\nabla}\phi(x_s))}{d\Phi_i} \\ &= \frac{\alpha'}{4\pi}[z_r(\sigma_{tr} + \frac{1}{d_r})\frac{e^{-\sigma_{tr}d_r}}{d_r^2} + z_v(\sigma_{tr} + \frac{1}{d_v})\frac{e^{-\sigma_{tr}d_v}}{d_v^2}] \end{aligned}$$

where D is the diffusion constant, ϕ is the radiant fluence, Φ_i is the incident flux, α' is the reduced albedo, σ_{tr} is the effective transport coefficient, z_r and z_v are the

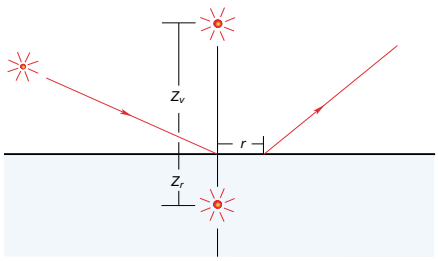


Figure 6.2: Dipole approximation of multiple scattering (based on [Jensen *et al.*,2001])

distance from the dipole lights to the surface, d_r is the distance from x to the real source, and d_v is the distance from x to the virtual source. The configuration is shown in Figure 6.2. From this equation, we can see that if the scattering property of a material is homogeneous, i.e., the scattering cross-sections are constant, then the formula relates reflectance at one surface point to incident flux at other surface points. Since subsurface scattering has a limited effective range, we can obtain the reflectance of a surface point due to multiple scattering by integrating flux incident at points within a certain distance.

The multiple scattering term, $S_d(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o})$, depends on the transmission terms at the entering and exiting surface points, and the dipole factor $R_d(r)$. We note that the dipole factor, $R_d(r)$, only depends on the distance between two points and decays exponentially with the distance. We define the *scattering neighborhood* $N(x_o)$ of a vertex x_o, to include all vertices x_i of that object that lie within the effective scattering range from x_o. We then compute multiple scattering contribution from the scattering neighborhood of each vertex during the pre-processing stage. Every such neighboring vertex x_i is assumed to represent a small surface area whose

size can be approximately defined. We assign the integral of $R_d(\|x_i - x_o\|)$ over this small surface area as the contribution to the multiple scattering at x_o due to x_i and append this information to x_o's list of multiple scattering contributors. Then at rendering time, once we have $F_t(\eta, \overrightarrow{\omega_i})$ and $F_t(\eta, \overrightarrow{\omega_o})$ from the local illumination computation, the contribution of point x_i to x_o due to subsurface scattering is just the multiplication of $F_t(\eta, \overrightarrow{\omega_i})$ with $F_t(\eta, \overrightarrow{\omega_o})$ and the pre-computed $R_d(\|x_i - x_o\|)$ factor of x_i from x_o's neighborhood list. The values of Fresnel terms and their associated relative indices of refraction that we used in our work can be found in [73]. The pre-computation and storing of the dipole factors is similar to the approach taken by Lensch *et al.* [89]. In their algorithm, instead of storing vertex-to-vertex dipole factors for every vertex in the scattering neighborhood, they distinguish between local responses and global responses. They store the global responses as vertex-to-vertex throughput factors, and the local ones as texture atlas. We have not made that distinction here, and store all of them as vertex-to-vertex factors.

6.2.3 Run-Time Two-Pass Local Illumination Model

We incorporate subsurface scattering effects into a local illumination model by extending the model into a run-time two-pass one. The traditional local illumination model computes the outgoing radiance from a surface point according to lighting direction, surface normal, and viewing direction in a single pass, using the particular light and material properties.

In our run-time two-pass approach, the first pass generates reflection and transmission radiance at each surface point as if there is no subsurface scattering, using

the Fresnel terms for reflection or transmission. After we compute the illumination at all surface points, we come to the second pass, i.e., the bleeding pass. During this pass, we combine on-surface reflection with subsurface scattering to get the total radiance at the exterior surface points according to the multiple scattering factors given in Section 6.2.2, using each point's weighted contributions from its neighbors. This bleeding pass adds subsurface reflection and transmission effects on the surface.

6.3 Improving Efficiency

Our run-time two-pass process is somewhat similar, but still quite different from the approach proposed by Jensen and Buhler [71]. The main difference is when the scattering neighborhood factors are computed. We pre-compute the factors at the preprocessing stage, so bleeding the neighboring effects due to scattering in the second pass is quite efficient, instead of traversing a hierarchical N-body data structure for each frame as in [71].

The run-time complexity of this version of our algorithm is $O(N^2)$, where N is the number of surface points, assuming the size of the object and the scattering properties remain constant. This is due to the fact that the number of vertices at which we have to perform the bleeding step is N, and the scattering neighborhood size is proportional to surface point density, which in turn is proportional to the number of surface points N. While Jensen and Buhler [71] build a hierarchical $O(N \log N)$ data structure to solve the inherent $O(N^2)$ complexity problem, we propose a quantized light source scheme to merge the two stages of our run-time

lighting process into a single stage process to further improve the efficiency of our algorithm. We can thus reduce the complexity of our run-time algorithm to $O(N)$ with quite small constant factors. It enables us to achieve interactive frame rates for simulating subsurface scattering effects on larger datasets. However, the preprocessing also means that any change of the material subsurface scattering properties will require a new pre-computation, which is a limitation not incurred by Jensen and Buhler [71].

6.3.1 Quantized Light Sources for Pre-computed Neighborhood Factor

We make further simplifications to reduce the complexity of our algorithm based on the fact that each surface point in the neighborhood of another surface point represents a small area on the surface and that real surfaces are usually rough. The subsurface scattering contribution to the appearance of a surface point from a directional light source with fixed direction ω_i can be pre-processed as follows:

$$\begin{aligned}
L_o(x_o, \overrightarrow{\omega_o}) &= \int_A S(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o}) L_i(x_i, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) \mu_i dA \\
&\approx \int_A S_d(x_i, \overrightarrow{\omega_i}; x_o, \overrightarrow{\omega_o}) L_i(x_i, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) \mu_i dA \\
&= \int_A F_t(\eta, \overrightarrow{\omega_i})[\frac{1}{\pi} R_d(\|x_i - x_o\|)] F_t(\eta, \overrightarrow{\omega_o}) \cdot L_i(x_i, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) \mu_i dA \\
&= \left\{ \int_A F_t(\eta, \overrightarrow{\omega_i})(\frac{1}{\pi} R_d) L_i(x_i, \overrightarrow{\omega_i}) \overrightarrow{n_i} \mu_i dA \right\} \cdot \overrightarrow{\omega_i} \cdot F_t(\eta, \overrightarrow{\omega_o}) \\
&\equiv \overrightarrow{Q}(\eta, x_o, \overrightarrow{\omega_i}) \cdot \overrightarrow{\omega_i} \cdot F_t(\eta, \overrightarrow{\omega_o})
\end{aligned}$$

where μ_i is defined as:

$$\mu_i = \begin{cases} 1 & (\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) \geq 0 \\ 0 & otherwise \end{cases}$$

This means we can pre-compute the vector integral $\overrightarrow{Q}(\eta, x_o, \overrightarrow{\omega_i})$ for the scattering factor during the preprocessing stage, and at run time perform the dot-product and multiplication operations. Due to the discrete nature of input mesh geometry, the vector integral above will be expressed as a vector summation in a real implementation:

$$\overrightarrow{Q}(\eta, x_o, \overrightarrow{\omega_i}) = \sum_{x_i \in N(x_o)} F_t(\eta, \overrightarrow{\omega_i})(\frac{1}{\pi} R_d) L_i(x_i, \overrightarrow{\omega_i}) \overrightarrow{n_i} \mu_i \Delta A(x_i)$$

where the summation is over all the vertices in the scattering neighborhood $N(x_o)$ of x_o. $\Delta A(x_i)$ is the area represented by vertex x_i, which is a constant if vertices are distributed uniformly as in [71]. For non-uniformly distributed vertices, we can either resample the geometry, or use one third of the total area of the triangles sharing the vertex as an approximation to ΔA at the vertex. So we actually pre-compute the summation $\overrightarrow{Q}(\eta, x_o, \overrightarrow{\omega_i})$ for each vertex. Note, if a vertex at x_i in the scattering neighborhood of x_o is in shadow, then it will not contribute to $\overrightarrow{Q}(\eta, x_o, \overrightarrow{\omega_i})$, because x_i receives no direct irradiance from light source. The summation will not be affected by presence of shadow on x_o, though. We use a technique similar to shadow maps to determine if a vertex is in shadow. We first generate a depth image of the scene as seen by the light source. Then for each vertex, we transform it into light space and compare its depth value against the value on the depth image. If the depth value of the vertex is bigger, the vertex is in shadow.

It will be impossible to compute the vector integral $\overrightarrow{Q}$ for each possible light source direction, of which the number is infinite. Instead, we quantize the directional space and pre-compute $\overrightarrow{Q}$ for a set of uniformly distributed light source directions.

For each light source j within the set, we compute the scattering neighborhood integral $\overrightarrow{Q_j}$ at each vertex during the preprocessing stage. An alternative to pre-computing and storing a vector integral $\overrightarrow{Q}(\eta, x_o, \overrightarrow{\omega_i})$ is to pre-compute a scalar dot-product value $q(\eta, x_o, \overrightarrow{\omega_i})$ instead:

$$\begin{aligned} L_o(x_o, \overrightarrow{\omega_o}) &= \left\{ \sum_{x_i \in N(x_o)} F_t(\eta, \overrightarrow{\omega_i})(\frac{1}{\pi} R_d) L_i(x_i, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) \mu_i \Delta A(x_i) \right\} \cdot F_t(\eta, \overrightarrow{\omega_o}) \\ &\equiv q(\eta, x_o, \overrightarrow{\omega_i}) \cdot F_t(\eta, \overrightarrow{\omega_o}) \end{aligned}$$

The advantage of using q instead of $\overrightarrow{Q}$ is the reduction of memory usage. The pseudo-code for pre-computing $q(\eta, x_o, \overrightarrow{\omega_i})$ for vertex x_o is shown below (assume the area $\Delta A(x_i)$ and incoming flux $L_i(x_i, \overrightarrow{\omega_i})$ associated with each vertex x_i has been computed, and the effective scattering range is represented by $RANGE$):

```
Find–Scalar–Integral (η, x_o, ω_i)
    q = 0
    for i from 1 to N
        if (x_i == x_o OR x_i in shadow)
            skip
        else
            r = ||x_i − x_o||
            if (r > RANGE)
                skip
            else
                q += F_t(η, ω_i)((1/π) R_d(r)) L_i(x_i, ω_i)(n_i · ω_i) μ_i ΔA(x_i)
    return q
```

It is clear that the pre-processing stage shown above has complexity of $O(N^2)$. If we use an octree-based data structure as in [71], then the complexity will go down to $O(N \log N)$.

6.3.2 Rendering from Quantized Light Sources

After we pre-compute either the vector integral $\overrightarrow{Q}$ or scalar integral q for a set of directional light sources, we use interpolation at run time to find the scattering integral $\overrightarrow{Q}$ or q for a specific light source direction. We use quaternion-based vector interpolation [112] to compute $\overrightarrow{Q}$ from its four closest $\overrightarrow{Q_j}$'s in the set (as in Figure 6.3). Then we compute dot-product of the interpolated scattering integral $\overrightarrow{Q}$ with real light source direction. This kind of interpolation is similar to the normal interpolation scheme used in Phong shading, though quaternion interpolation gives a more accurate result and avoids a vector re-normalization step. To compute q, we simply use a linear scalar interpolation scheme, which is similar to the interpolation used in the Gouraud shading algorithm.

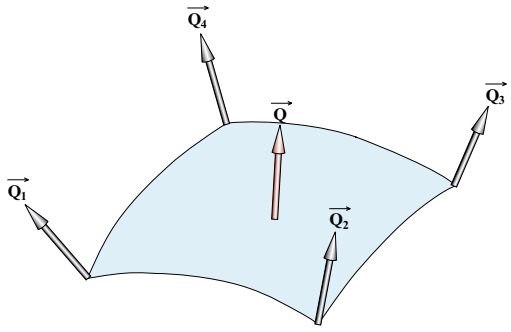


Figure 6.3: Interpolation of the vector integral for a new light source direction from its four nearest neighbors in the pre-computed set

During the rendering of the scene we combine scattering effects with direct on-surface reflected light (including shadows), to give the final appearance of each vertex. As an example, for a light source in direction $\overrightarrow{\omega_i}$, the scattering amount for vertex x_o along viewing direction $\overrightarrow{\omega_o}$ will be $F_t(\eta, \overrightarrow{\omega_o})$ multiplied with the pre-computed factor $q(\eta, x_o, \overrightarrow{\omega_i})$, and scaled by this light source's actual intensity. We compute direct on-surface reflected light by a local illumination model.

The pseudo-code for computing the outgoing radiance $L(x_o, \overrightarrow{\omega_o})$ for vertex x_o in direction $\overrightarrow{\omega_o}$ appears below. Here we assume the use of scalar integral q.

Find–Outgoing–Radiance $(\eta, \overrightarrow{\omega_i}, x_o, \overrightarrow{\omega_o})$

$L(x_o, \overrightarrow{\omega_o}) = 0$

Find 4 nearest matches q_j from the pre-computed set of $\{q\}$ for x_o

Interpolate the 4 matched values based on $\overrightarrow{\omega_i}$ to get $q(\overrightarrow{\omega_i})$

$L_{scattered} = q(\overrightarrow{\omega_i}) \cdot F_t(\eta, \overrightarrow{\omega_o})$

Compute reflected radiance $L_{reflected}(x_o, \overrightarrow{\omega_o})$ using a local illumination model

$L(x_o, \overrightarrow{\omega_o}) = L_{scattered} + L_{reflected}(x_o, \overrightarrow{\omega_o})$

return $L(x_o, \overrightarrow{\omega_o})$

The visual difference between using $\overrightarrow{Q}$ and q for the models we have tested is insignificant. This can be attributed to the diffuse nature of subsurface scattering. Hence we are currently using the pre-computed scalar dot-products. Figure 6.4(a) shows a image generated using $\overrightarrow{Q}$ on a horse model, and Figure 6.4(b) shows the image generated using q on the same model. The difference image is shown in

Figure 6.4(c). The image space root-mean-square error between Figure 6.4(a) and 6.4(b) is 5.26×10^{-3}.

(a) Scattering using $\overrightarrow{Q}$ (b) Scattering using q (c) Difference of (a) and (b)

Figure 6.4: Comparison of subsurface scattering using pre-computed vector integral and scalar integral on the Horse model (14,521 vertices)

What we have shown is that the light flux at a vertex on the surface due to direct reflection and subsurface scattering can now be computed at the same time under a local illumination model. Thus with pre-computed integral, the run-time two-pass algorithm we suggested before now becomes a run-time single-pass algorithm. Furthermore, this pre-computed integral scheme also indicates that the run-time computation of the scattering effect on a vertex is just an interpolation of the four nearest neighbors in the set of the pre-computed integrals which have the same size as the light source set we have selected. So the complexity of computing the

scattering component at run-time is constant, and not related to surface point density. The total complexity of our run-time algorithm becomes $O(N)$, instead of $O(N^2)$, where N is the number of vertices. Subsurface scattering increases if the translucency of the material increases or the physical size of the object decreases.

This increases the scattering neighborhood size that needs to be considered. However, since the scattering neighborhood size only affects the pre-computation of integrals $\overrightarrow{Q}$ or q, the rendering-time complexity of our display algorithm stays $O(N)$.

6.3.3 Determining the Size of Light Source Set

The above sections show that if we use a quantized light source scheme for pre-computation of scattering integrals and do interpolation at run-time, then we will have a linear complexity single-pass run-time algorithm for rendering translucent materials. We have not yet mentioned how to pick the size of light source set.

As we know, the scattering integral $\overrightarrow{Q}(\eta, x_o, \overrightarrow{\omega_i})$ or $q(\eta, x_o, \overrightarrow{\omega_i})$ is a continuous function of the directional space variable $\overrightarrow{\omega_i}$. Quantization of light source directions is a sampling process and interpolation is a reconstruction process. Similar to other sampling processes, there is a tradeoff between sampling rate and time and storage. Using a lower sampling rate is time and memory efficient, but gives us less accurate results. Even worse, a low sampling rate may introduce aliasing problems when the sampling frequency is lower than the Nyquist rate. A general frequency-space analysis for the scattering integrals is difficult because the scattering integrals depend on geometry and scattering properties of the object and different vertices will have different frequency distributions.

We instead experiment with different sizes of the light source set. We measure the image-space root-mean-square error for our test datasets as follows:

$$e_{rms} = \left[\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[\widehat{f}(x,y) - f(x,y) \right]^2 \right]^{1/2}$$

Here $f(x, y)$ represents the image generated without the specified approximation, $\widehat{f}(x, y)$ denotes an estimate of the image, either interpolated or approximated, $M \times N$ is the image size, and the range for $f(x, y)$ is $[0, 1]$.

The results of the experiment are summarized in Figure 6.5. The root-mean-square error is measured by comparing the results obtained by interpolation using pre-computed scalar integrals q with the exact results. We compute this error for about 100 randomly generated view directions and take the maximum RMS error as the representative. We can see from Figure 6.5 that with a set of about 200 light sources, the root-mean-square error is 3×10^{-3} for all the four datasets. So we use 200 light source directions to pre-compute the scalar integrals q.

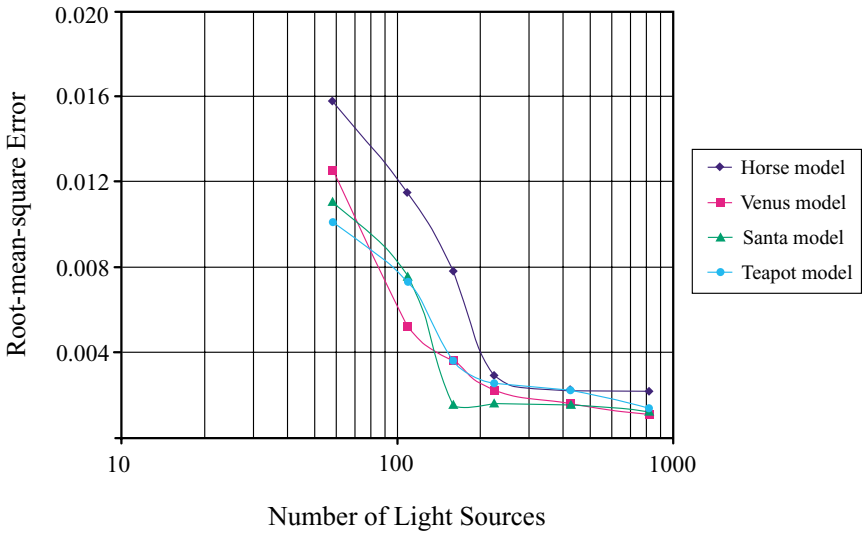


Figure 6.5: Root-mean-square error as a function of the number of light sources

This directional quantization scheme can also be extended to include point light sources. We can add one more dimension to the interpolation, i.e., we quantize the distance from light source to the object along with quantization of its direction.

Then we can trilinearly interpolate 8 nearest neighbors to get an $O(N)$ complexity algorithm for directional and point light sources.

Here we limit ourselves to local illumination, so we ignore on-surface inter-reflections between vertices during computation of the pre-computed integral $q(\eta, x_o, \overrightarrow{\omega_i})$. If we use efficient ray-tracing [4] or Monte Carlo simulation in the preprocessing stage, we can incorporate it in our algorithm and get more accurate $q(\eta, x_o, \overrightarrow{\omega_i})$.

6.4 Controlling the Memory Usage

For a set of 200 lights, we need to store 200 integrals per vertex. Instead of storing a floating-point value per integral, we store a normalized unsigned byte value to serve as an index into a lookup table. Thus we need 200 bytes of extra storage per vertex. We have used the Lloyd quantizer algorithm [44] to design the lookup table. As an example of the quantized result, the signal-to-noise ratio, SNR [44] is 50.99dB for pre-computed integrals of the teapot dataset by using this quantization, with a resulting image-space root-mean-square error of 8.83×10^{-4}. Normally for each vertex we need to store three numbers each for position and normal direction, as well as for texture coordinates and color, if any. If we assume floating-point numbers to store these values, we will need 24 bytes to store the position and normal direction alone. Even with this uncompressed number, the extra storage needed for pre-computed integrals will increase it by a factor of 8, which is significant a disadvantage for our algorithm. This factor can be reduced though. In the following subsections we show how to dramatically reduce this factor so that the extra storage

is comparable with the original storage required for the vertex data.

6.4.1 Decomposition by Spherical Harmonic Basis Functions

Due to the diffuse-like nature of subsurface scattering effects, we apply spherical harmonic functions to compress the pre-computed integrals. This is similar to the approach used in Chapter 3.4.

The projection of the pre-computed scalar integral of $q(\eta, x_o, \overrightarrow{\omega_i})$ onto the spherical harmonic basis is given by:

$$q_l^m(\eta, x_o) = \int q(\eta, x_o, \overrightarrow{\omega_i}) y_l^m(\overrightarrow{\omega_i}) d\overrightarrow{\omega_i}$$

The reconstructed function up to the n-th order is:

$$\widetilde{q}(\eta, x_o, \overrightarrow{\omega_i}) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} q_l^m(\eta, x_o) y_l^m(\overrightarrow{\omega_i})$$

where

$$\overrightarrow{\omega_i} = (x, y, z) = (\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta)$$

As an example, we apply the above projection and reconstruction scheme to the subsurface scattered teapot and results are shown in Figure 6.6 as the order n varies from 1 to 7. Closeup versions are shown in Figure 6.7. The number of the basis functions is equal to n^2, which results in 1 to 49 basis functions being used. For each basis function, we store a normalized short integer value (2 bytes) for $q_l^m(\eta, x_o)$. We therefore need 98 bytes per vertex for $n = 7$. We have not gone to higher n because then the storage required becomes comparable to the method that does not use spherical harmonics. From Figure 6.6 and Figure 6.7 we can

see that the image quality increases with the number of spherical basis functions. With 49 basis functions, the visual quality is close to the one without compression. However, if one notices carefully, some differences near the shadow boundaries are still visible (Figures 6.7(a) and (h)). The reason is that the spatial frequency of the pre-computed integral q is beyond the spatial frequency that 49 spherical harmonic basis functions can completely cover. So with spherical harmonics, we can achieve a factor of two compression ratio with small loss of image quality. For low frequency lighting environments an interesting alternative is to use clustered principal component analysis (CPCA) based compression of spherical harmonic coefficients to achieve faster rendering [126]. For general lighting environments, we have to find some way to suppress the spatial frequency of q.

6.4.2 Reference Points with Spherical Harmonic Basis Functions

We observe that the scattering from light entering one particular vertex and exiting at two other points will not differ much if those two points are close to each other. This is due to the diffuse nature of multiple scattering. The fact that each point receives contribution from all its scattering neighborhood will smooth out the difference even further. This means, the difference of the scattering integrals Δq between nearby points will have much lower spatial frequency. So we pick some reference vertices across the surface and store their scattering integrals q explicitly. For every non-reference vertex, we compute the differences of the integrals by subtracting its original value from a weighted average of the values from its closest neighboring reference vertices. We can then expect that the spherical harmonic functions can

(a) Scattering by q

(b) By q_l^m $(n = 1, e_{rms} = 0.18)$

(c) By q_l^m $(n = 2, e_{rms} = 0.072)$

(d) By q_l^m $(n = 3, e_{rms} = 0.027)$

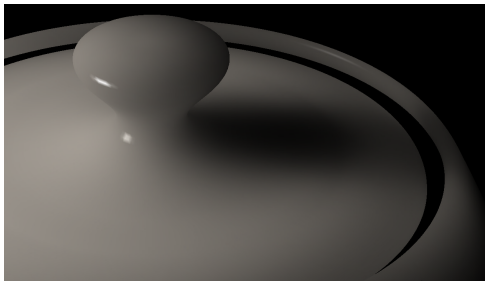
(e) By q_l^m $(n = 4, e_{rms} = 0.020)$

(f) By q_l^m $(n = 5, e_{rms} = 0.012)$

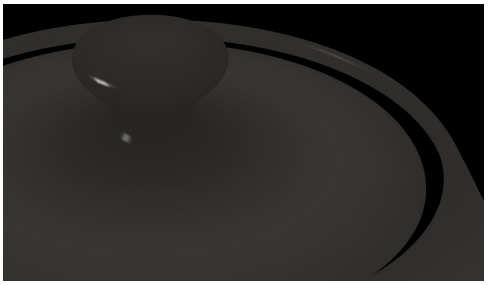
(g) By q_l^m $(n = 6, e_{rms} = 0.0096)$

(h) By q_l^m $(n = 7, e_{rms} = 0.0056)$

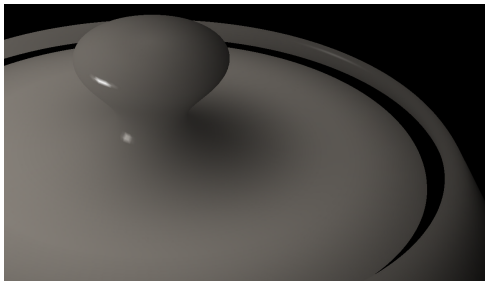
Figure 6.6: Comparison of subsurface scattered teapot using q and q_l^m (150,510 vertices)



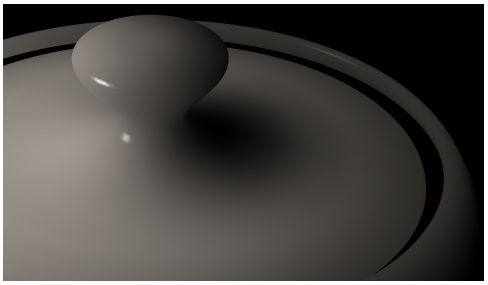
(a) Closeup of scattering by q



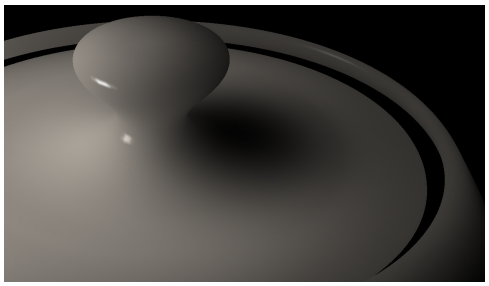
(b) Closeup of q_l^m $(n = 1)$



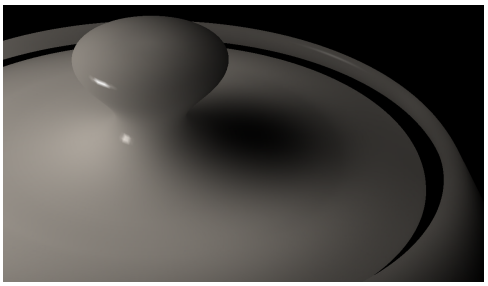
(c) Closeup of q_l^m $(n = 2)$



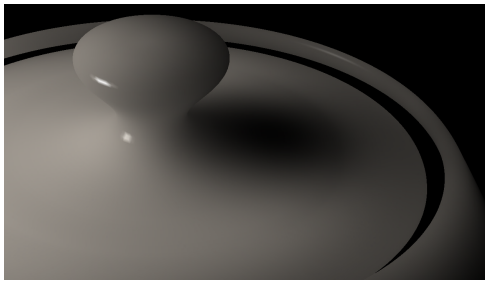
(d) Closeup of q_l^m $(n = 3)$



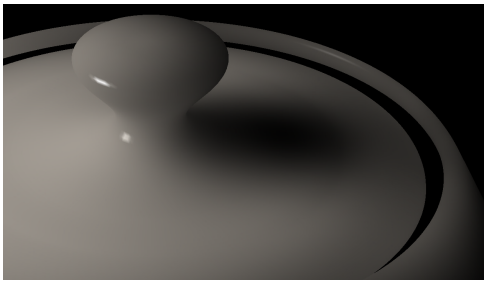
(e) Closeup of q_l^m $(n = 4)$



(f) Closeup of q_l^m $(n = 5)$



(g) Closeup of q_l^m $(n = 6)$



(h) Closeup of q_l^m $(n = 7)$

Figure 6.7: Closeup of Figure 6.6

be applied readily to those frequency-suppressed Δq. Ramamoorthi and Hanrahan [115] have treated a similar problem from a different perspective. Instead of trying to compress the frequency before applying the spherical decomposition, they determine the necessary number of basis functions for a faithful representation of the original signal using a signal-processing framework.

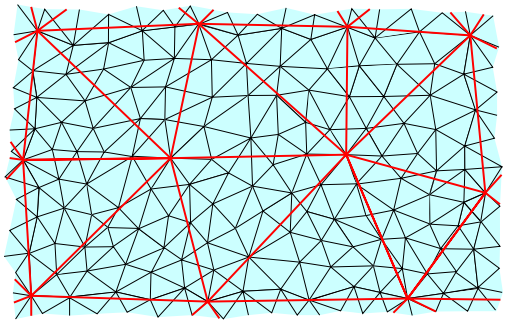


Figure 6.8: Construction of Reference Points

We can build reference points using a mesh simplification algorithm similar to [24, 43, 66] or a re-tiling scheme of Turk [139]. We prefer to generate the reference points as a subset of the original vertices to reduce the storage overhead (as shown in Figure 6.8). After we find the reference points, we generate the differences of pre-computed integrals for each vertex with respect to its reference points as discussed next.

We first determine the three reference points for each vertex. Re-tiling schemes such as by Turk [139] keep track of which triangle each vertex has been flattened to. For other mesh simplification algorithms we know one reference point for the vertex, which is its parent in the simplification hierarchy. The vertex will lie in one of the simplified triangles sharing this reference point. To find out the triangle the

vertex lies in, we project the vertex onto planes defined by those triangles, then do a simple orientation test of the projected vertex relative to the three edges of each triangle. Once we find the triangle the vertex V lies in, we compute the barycentric coordinates of the projection V' of V onto the triangle.

Assume the reference points are V_1, V_2, and V_3 with barycentric coordinates w_1, w_2, and w_3. Let $\{q_{1j}\}, \{q_{2j}\}$, and $\{q_{3j}\}$ (j is the quantized light source index) be the pre-computed integrals for V_1, V_2, and V_3, respectively. The $\{\Delta q_j\}$ set for vertex V can then be computed as:

$$\Delta q_j = q_j - \sum_{k=1}^{3} \omega_k q_{kj}$$

Finally, we decompose the integral differences $\{\Delta q_j\}$ by spherical harmonic basis functions as before.

Figure 6.9 shows the root-mean-square error of using different number of reference points and with 9 ($n = 3$) and 36 ($n = 6$) spherical harmonic basis functions on the teapot dataset.

Figure 6.10 shows the scattered teapot images generated using different number of reference points with 9 ($n = 3$) spherical harmonic basis functions. Closeup versions are shown in Figure 6.11. From Figure 6.11, we can see that even with about 1200 reference vertices and 9 ($n = 3$) basis functions (Figure 6.11(d)), the result is better (with smaller root-mean-square error) than the one using 49 ($n = 7$) basis functions and no reference points (Figure 6.7(h)). For $5K$ reference vertices (about 3% of the total) and 9 ($n = 3$) basis functions, the image is almost indistinguishable from the original one (Figure 6.11(a)), even for the closeup version.

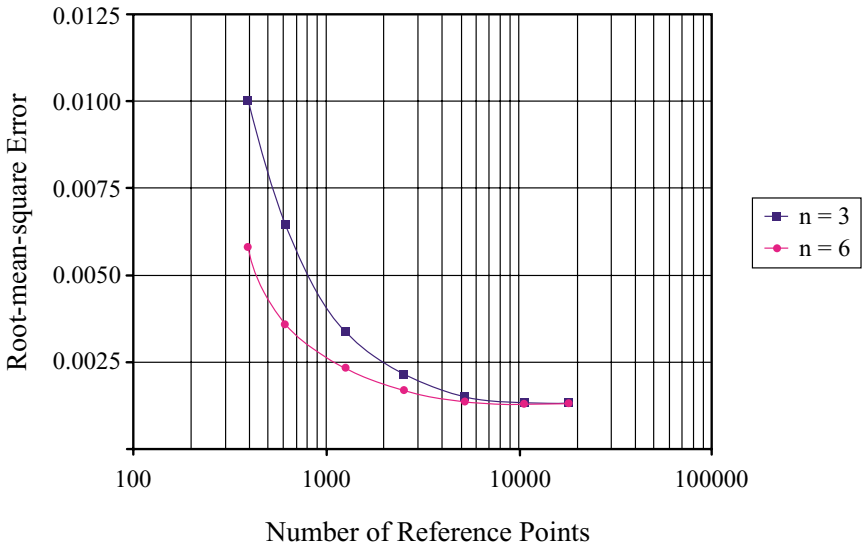
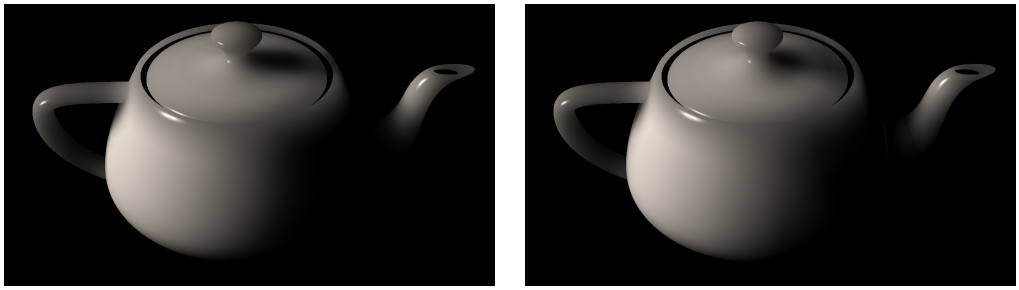


Figure 6.9: Root-mean-square error as a function of the number of reference points for teapot dataset with 9 (n = 3) and 36 (n = 6) spherical harmonic basis functions

Now let us consider the storage requirements for the above case. We need 200 bytes for each vertex in the $5K$ reference set to store their original pre-computed integrals, 4 bytes for each vertex to store its weight to its three nearest neighbors (the first two values stored as normalized short integers, while the third value can be computed at run-time by one minus the first two values), and 9 bytes for each vertex to store the spherical harmonic basis functions' coefficients (each is stored as a normalized byte because the range for the coefficients has also been significantly reduced). Overall, on average we need the following number of bytes per vertex to store the pre-computed integrals:

$$\frac{200 \times 5176 + (4+9) \times 150510}{150510} \approx 20$$

So we only need 20 bytes per vertex. We know that each vertex needs a position



(a) Scattering by q

(b) 390 Reference Points ($e_{rms} = 0.010$)

(c) 614 Reference Points ($e_{rms} = 0.0065$)

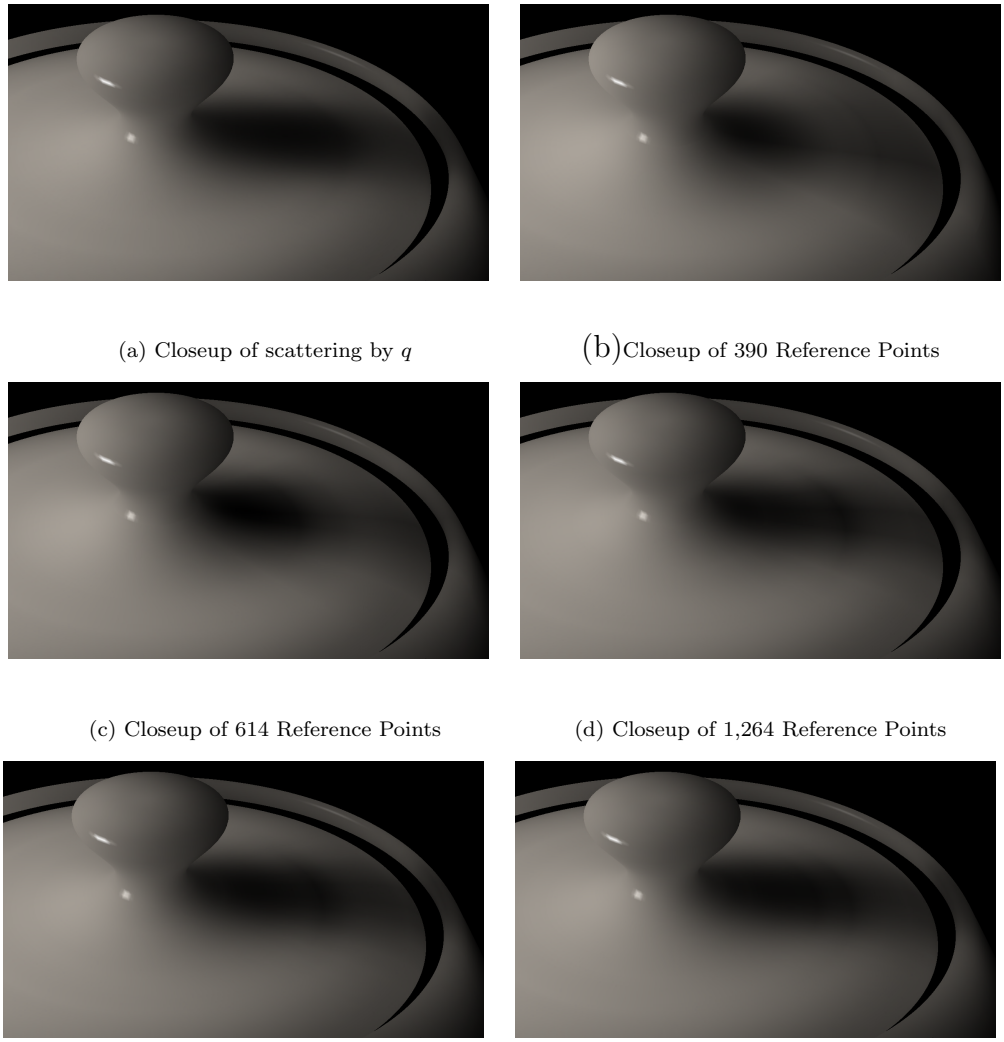
(d) 1,264 Reference Points ($e_{rms} = 0.0034$)

(e) 2,526 Reference Points ($e_{rms} = 0.0022$)

(f) 5,176 Reference Points ($e_{rms} = 0.0017$)

Figure 6.10: Comparison of subsurface scattered teapot using q and different number of reference points with 9 ($n = 3$) spherical harmonic basis functions (150,510 vertices)

vector and a normal vector. If we assume floating-point numbers to store them we will need $(3 + 3) \times 4 = 24$ bytes for each vertex. Then the storage required by the pre-computed integrals is less than the storage required by the position and normal alone. Of course, one can compress positions and normals for vertices,



(a) Closeup of scattering by q

(b)Closeup of 390 Reference Points

(c) Closeup of 614 Reference Points

(d) Closeup of 1,264 Reference Points

(e) Closeup of 2,526 Reference Points

(f) Closeup of 5,176 Reference Points

Figure 6.11: Closeup of Figure 6.10

too. Nevertheless, this storage overhead seems quite reasonable for the interactive simulation of translucent materials. Even better, the rendering speed increases from 7.5 frames/second to 8.6 frames/second, which is about 15% speedup. We achieve similar results on other datasets we have tested. The extra storage will be no more than 27 bytes per vertex, and it decreases as the complexity of the object increases (Table 6.1).

6.5 Results and Discussions

In this section, we show the results obtained by our algorithm on polygonal datasets. The results are summarized in Table 6.1 and in Figures 1.5, 6.6, 6.7, and 6.10–6.14. The images usually have about 1024 pixels in each dimension, though their sizes have nearly no effect on the total rendering time, because we use the graphics hardware mainly to do rasterization.

Model Name	No. of Vertices	No. of Triangles	No. of ref pts	Extra storage (Bytes/vert)	Compression ratio by using ref pts	Frame rate (fps)
Horse	14,521	29,054	1,034	27	7.4	79.1
Venus	42,656	90,044	2,827	26	7.7	27.3
Santa	75,781	151,558	3,458	22	9.1	14.6
Teapot	150,510	292,168	5,176	20	10.0	8.6
Dragon	437,645	871,414	10,285	18	11.1	2.7
Buddha	543,652	1,087,716	12,330	18	11.1	2.4

Table 6.1: Total rendering times for our approach

From Table 6.1 one can see that our scattering model can simulate the homogeneous scattering effects interactively and requires no more than 28 bytes storage per vertex. This small overhead should give most applications the opportunity to include the subsurface scattering effects for more photo-realistic rendering without sacrificing interactive frame rates.

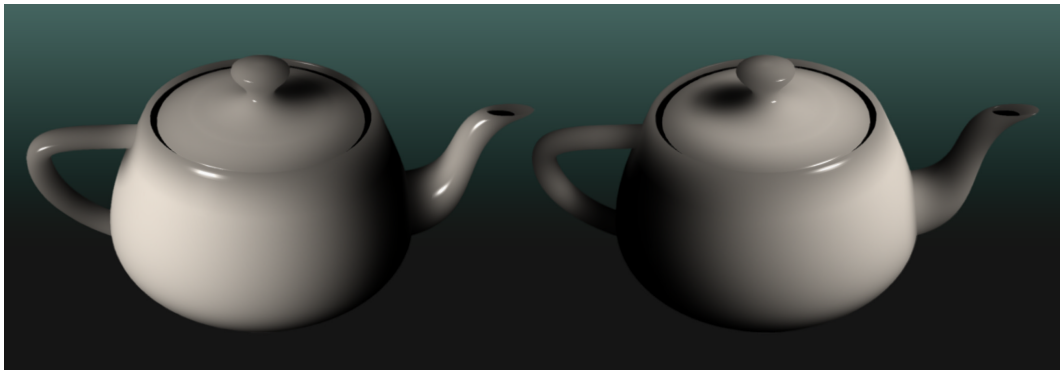
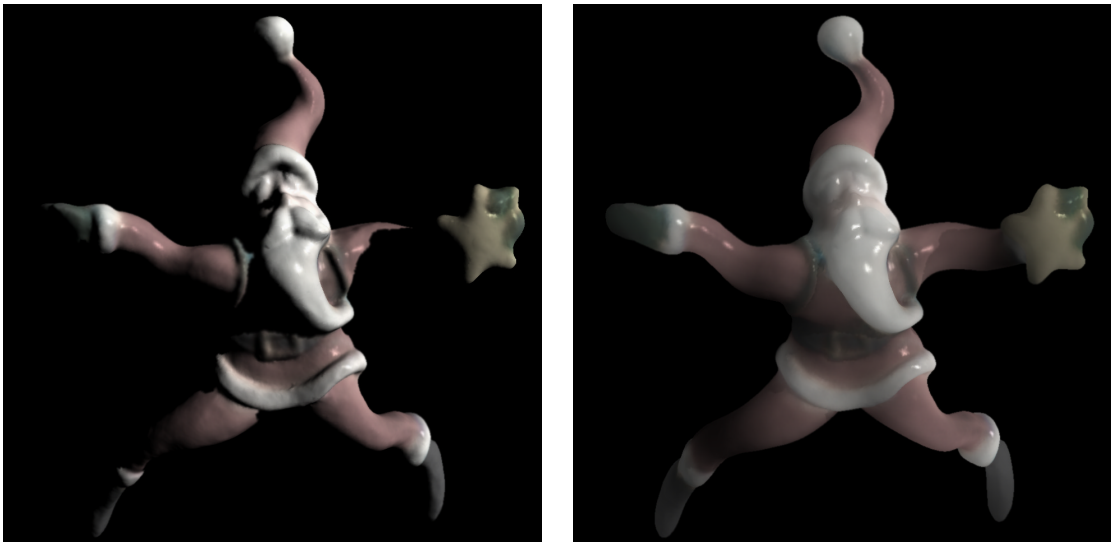


Figure 6.12: Rendering the subsurface scattered teapot model with varying light source direction (150,510 vertices, 8.6 fps)

Figure 6.12 shows the effects of varying light source direction with fixed viewer position on subsurface scattered teapot. Figure 6.13 compares the appearance of Santa model without and with subsurface scattering. We have used the Perlin noise function [34, 108] to generate the marble texture on the Venus model. Here we have made the assumption that the marble texture is on surface, and will affect both x_i and x_o. Figures 1.5 and 6.14 show how the object will appear if either its size shrinks or its material property changes to allow greater subsurface scattering.

Our algorithm can also use a full Monte Carlo simulation in the preprocessing stage. This will allow us to not only have an accurate subsurface scattering term, but also include the single scattering term, treat inhomogeneity, and relieve the algorithm from the limitation of the dipole diffusion approximations for multiple scattering. Subsurface scattering is also characterized by color-shift effects. The correct way to simulate color shifts is to do a full spectral rendering. However, the three channel RGB approximation can also give visually appealing results. If one would like to use the three-channel approximation of the subsurface scattered color



(a) Without Scattering (31.7 fps) (b) With Scattering (14.6 fps)

Figure 6.13: Santa model without and with subsurface scattering (75,781 vertices)

shifts in our algorithm, we can compute three different sets of integrals, one for each channel. The storage requirements will then be a little less than three times as before, since we only need to store the barycentric coordinates once, instead of three times. That means we will need about 46 bytes extra per vertex for large datasets.

6.6 Conclusions

In this chapter we have integrated subsurface scattering effects into a run-time single-pass local illumination model with an efficient $O(N)$ run-time complexity using pre-computed scattering integrals for a set of quantized light directions. We show that a reference points scheme, together with spherical harmonics can be applied to greatly reduce the storage requirements of pre-computed integrals and improve the run-time efficiency of our algorithm even further. The results capture the most important effects of subsurface scattering, such as neighborhood bleeding and smooth

(a) Without Scattering (62.5 fps) (b)(c)(d) With scattering (27.3 fps)

Figure 6.14: Rendering the Venus model with subsurface scattering increasing from left to right (42,656 vertices with 10% vertices in $N(x_o)$ at (b), 20% vertices in $N(x_o)$ at (c), and 30% vertices in $N(x_o)$ at (d))

illumination transitions between regions separated by sharp edges. Our method provides an approximation of subsurface scattering for applications that need to maintain the interactivity with a small memory overhead while preserving the realistic appearance for translucent materials. Our approach, by a little modification, can also be incorporated into shadow algorithms to generate soft shadow effects.

Chapter 7

Future Work

In this chapter I will outline the possible research topics that are related to this dissertation. These topics are direct or implied extensions of the areas that are addressed in this dissertation.

7.1 Use of Temporal Information in Electrostatics Computation

We have seen in Chapter 2 how to efficiently solve PBE by adaptive adjustment of the irregular grids based on their importance to the solution. There the 3D irregular grid structure is built for a static molecular geometry. To efficiently solve PBE for a continuous and dynamic motion of proteins, we can use incremental temporal information to adjust the irregular grid for a new geometric conformation, instead of reconstructing the grid from the beginning. Assuming continuous trajectories of atoms' movements, it should be possible to solve PBE incrementally and efficiently by using the information from previous time steps. Efficient algorithms that exploit the temporal coherence would benefit molecular dynamics simulation and interactive docking applications.

7.2 Modeling and Rendering of Non-homogeneous Scattering Effects

We have used dipole diffusion approximation to interactively simulate the homogeneous scattering effects in Chapter 6. Non-homogeneous scattering is more complicated, but is important for describing the appearance of many translucent materials, such as human skin and marble with impurities. Correct modeling and efficient rendering of non-homogeneous scattering effects also have great impact in many scientific applications, such as medical imaging. To achieve this, we will also need to design more efficient data representations to simulate such effects by solving the underlying radiance transfer equations directly or indirectly, together with compact representations to store the solutions.

7.3 Simulation and Rendering of Dynamic Scenes

Graphics rendering has come to a stage where static scenes alone cannot satisfy our quest for visual realism. Simulation, modeling, and rendering of dynamic scenes consisting of large collections of moving objects with real-time collision detection, response, and deformation will become more important and ubiquitous in the next generation graphics applications. Solutions to these problems will depend on efficient representation and manipulation of high-complexity dynamic geometry, modeling and representation of realistic illumination, and physics-based dynamics simulation. This should greatly benefit several graphics application areas, such as architecture and urban planning, lighting design, interactive walkthroughs, entertainment indus-

try, and next-generation user interfaces.

7.3.1 Geometry Representation

With the increasing complexity of geometry data due to advances in 3D data acquisition, simulation, and design technologies, compact data representation for efficient manipulation becomes more important. Parametric representation of shapes is one of the possible solutions. The best candidate representations should possess the properties of compactness, progressiveness, and suitability for kinetic data structures.

7.3.2 Reflectance Function Measurements and Representations

Realistic image generation requires correct simulation of the interaction of light with objects. Efficient and accurate global illumination for dynamic environments remains a challenge. Dynamic environments have either non-static geometry, or non-static lighting, or both. This requires measurement and compact representation of reflectance functions of real-world objects under varying lighting environments, more efficient and accurate global illumination solutions for dynamic environments, and the use of each object's real reflectance functions instead of using the Lambertian assumption.

7.3.3 Dynamics Simulation

In addition, rendering of dynamic scenes requires us to simulate the dynamics of moving objects in a physically correct manner. Currently the simulation is limited

to a small collection of moving objects for interactive display. Development of new algorithms to simulate dynamic scenes consisting of large collections of moving objects will greatly enhance the visual realism.

Bibliography

[1] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, NC 27599-3175, 1990.

[2] N. Akkiraju and H. Edelsbrunner. Triangulating the surface of a molecule. *Discrete Appl. Math.*, 71:5–22, 1996.

[3] M. Ashikhmin, S. Premoze, and P. Shirley. A microfacet-based BRDF generator. In Kurt Akeley, editor, *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 65–74, July 2000.

[4] F. B. Atalay and D. M. Mount. Ray interpolants for fast ray-tracing reflections and refractions. In V. Skala, editor, *Journal of WSCG 2002*, volume 10(3), pages 1–8, 2002.

[5] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: A diversified volume visualization system. In R. D. Bergeron and A. E. Kaufman, editors, *IEEE Visualization '94*, pages 31–39, October 1994.

[6] E. Bahar and S. Chakrabarti. Full-wave theory applied to computer-aided graphics for 3D objects. *IEEE Computer Graphics and Applications*, 7(7):46–60, July 1987.

[7] C. Bajaj, V. Pascucci, A. Shamir, R. Holt, and A. Netravali. Dynamic maintenance and visualization of molecular surfaces. *Discrete Appl. Math.*, 127:23–51, 2003.

[8] C. Bajaj, V. Pascucci, and G. Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *Proceedings Visualization 99*, pages 307 – 316, Los Alamitos, California, 1999. IEEE, Computer Society Press.

[9] C. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitrary triangular meshes with properties. *Computational Geometry: Theory and Applications*, vol 14:167–186, 2000.

[10] N. Baker, M. Holst, and F. Wang. Adaptive multilevel finite element solution of the Poisson-Boltzmann equation II: refinement at solvent accessible surfaces in biomolecular systems. *J. Comput. Chem.*, 21:1343–1352, 2000.

[11] R. Banerjee and J. Rossignac. Topologically exact evaluation of polyhedra defined in CSG with loose primitives. *Computer Graphics Forum*, 15, No. 4:205–217, 1996.

[12] R. Basri and D. Jacobs. Lambertian reflectance and linear subspaces. In *Proc. International Conference On Computer Vision (ICCV-01)*, pages 383 – 390, 2001.

[13] L. D. Bergman, J. S. Richardson, R. Richardson, and F. P. Brooks, Jr. VIEW – an exploratory molecular visualization system with user-definable interaction

sequences. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 117–126, August 1993.

[14] J. F. Blinn. Models of light reflection for computer graphics. *Computer Graphics*, 11(2):192–198, 1977.

[15] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics*, 16(3):21–29, July 1982.

[16] C. Brande and J. Tooze. *Introduction to Protein Structure*. Garland Publishing, second edition, 1999.

[17] F. P. Brooks, Jr. Grasping reality through illusion – interactive graphics serving science. In *Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems*, pages 1–11, 1988.

[18] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In A. Kaufman and W. Krueger, editors, *1994 Symposium on Volume Visualization*, pages 91–98, October 1994.

[19] B. Cabral, N. Max, and R. Springmeyer. Bidirectional reflection functions from surface bump maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):273–281, July 1987.

[20] N. A. Carr, J. D. Hall, and J. C. Hart. GPU algorithms for radiosity and subsurface scattering. In M. Doggett, W. Heidrich, W. Mark, and A. Schilling, editors, *Graphics Hardware 2003*, pages 51–59, July 2003.

[21] M. Chow. Optimized geomerty compression for real-time rendering. In *IEEE Visualization '97 Proceedings*, pages 403 – 410. ACM Press, October 1997.

[22] P. Cignoni, L. De Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):29–45, 2004.

[23] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17, No. 2:167–174, June 1998.

[24] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. Wright. Simplification envelopes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 119–128, 1996.

[25] D. Cohen-Or, D. Levin, and O. Remez. Progressive compression of arbitrary triangular meshes. In *Proceedings Visualization 99*, pages 67–72, Los Alamitos, California, 1999. IEEE, Computer Society Press.

[26] M. L. Connolly. Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221:709–713, 1983.

[27] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, volume 15(3), pages 307–316, August 1981.

[28] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 83–90, 1997.

[29] C. Dachsbacher and M. Stamminger. Translucent shadow maps. In Per Christensen and Daniel Cohen-Or, editors, *Proceedings of the 14th Eurographics Symposium on Rendering*, pages 197–201, 2003.

[30] P. Debevec, T. Hawkins, C. Tchou, H.-P. Duiker, W. Sarokin, and M. Sagar. Acquiring the reflectance field of a human face. In Kurt Akeley, editor, *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 145–156, July 2000.

[31] M. F. Deering. Geometry compression. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 13–20. ACM SIGGRAPH, Addison Wesley. Los Angeles, California, August 1995.

[32] J. Dorsey, A. Edelman, J. Legakis, H. W. Jensen, and H. K. Pedersen. Modeling and rendering of weathered stone. In Alyn Rockwood, editor, *SIGGRAPH 99, Computer Graphics Proceedings*, Annual Conference Series, pages 225–234, August 1999.

[33] R. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):65–74, August 1988.

[34] D. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling.* Morgan Kaufmann Publishers, third edition, 2003.

[35] A. R. Edmonds. *Angular Momentum in Quantum Mechanics*. Princeton University Press, Princeton, New Jersey, second edition, 1960.

[36] J. El-Sana, E. Azanli, and A. Varshney. Skip strips: Maintaining triangle strips for view dependent rendering. In *IEEE Visualization '99 Proceedings*, pages 131 – 138. ACM/SIGGRAPH Press, October 1999.

[37] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, pages 9–16, 2001.

[38] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96 Proceedings*, pages 319 – 326. ACM/SIGGRAPH Press, October 1996.

[39] L. De Floriani, L. Kobbelt, and E. Puppo. A survey on data structures for level-of-detail models. In N.Dodgson, M.Floater, and M.Sabin, editors, *Proceedings MINGLE Workshop 2004*, page to appear, 2004.

[40] S. Fortune. Vertex rounding a three-dimensional polyhedral subdivision. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 116–125. ACM Press, June 1998.

[41] S. Fortune and C. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, 1996.

[42] G. Allen (editor). *Protein: A Comprehensive Treatise, vol 2, 61 – 97.* JAI Press, 1999.

[43] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In Turner Whitted, editor, *SIGGRAPH 97, Computer Graphics Proceedings*, Annual Conference Series, pages 209 – 216, August 1997.

[44] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic, 1991.

[45] S. F. Gibson. Using distance maps for accurate surface representation in sampled volumes. In *IEEE Symposium on Volume Visualization*, pages 23–30, 1998.

[46] M. K. Gilson, K. A. Sharp, and B. Honig. Calculating electrostatic interactions in biomolecules: method and error assessment. *J. Comp. Chem.*, 9:327 – 335, 1988.

[47] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 171–180. Addison Wesley, August 1996.

[48] D. P. Greenberg, K. E. Torrance, P. Shirley, J. Arvo, J. A. Ferwerda, S. Pattanaik, E. P. F. Lafortune, B. Walter, S.-C. Foo, and B. Trumbore. A framework for realistic image synthesis. In Turner Whitted, editor, *SIGGRAPH 97, Computer Graphics Proceedings*, Annual Conference Series, pages 477–494, August 1997.

[49] N. Greene and M. Kass. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[50] B. Gregorski, M. Duchaineau, P. Lindstrom, V. Pascucci, and K. I. Joy. Interactive view-dependent rendering of large isosurfaces. In *IEEE Visualization '02*, pages 475–482, 2002.

[51] S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. In *SIGGRAPH 98 Conference proceedings*, Annual Conference Series, pages 133–140. ACM SIGGRAPH, 1998.

[52] P. Hanrahan and W. Krueger. Reflection from layered surfaces due to subsurface scattering. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 165–174, August 1993.

[53] X. Hao, T. Baby, and A. Varshney. Interactive subsurface scattering for translucent meshes. In *ACM Symposium on Interactive 3D Graphics*, pages 75 – 82, Monterey, CA, 2003.

[54] X. Hao and A. Varshney. Variable-Precision rendering. In *2001 ACM Symposiom on Interactive 3D Graphics*, pages 149–158, Research Triangle Park, NC, 2001.

[55] X. Hao and A. Varshney. Efficient solution of Poisson-Boltzmann equation for electrostatics of large molecules. In *High Performance Computing Symposium*, pages 71–76, Arlington, VA, 2004.

[56] X. Hao and A. Varshney. Real-time rendering of translucent meshes. *ACM Transactions on Graphics*, 23(2):120–142, 2004.

[57] X. Hao, A. Varshney, and S. Sukharev. Real-time visualization of large time-varying molecules. In *High Performance Computing Symposium*, pages 109–114, Arlington, VA, 2004.

[58] X. D. He, K. E. Torrance, F. X. Sillion, and D. P. Greenberg. A comprehensive physical model for light reflection. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25(4), pages 175–186, July 1991.

[59] P. Heckbert. Color image quantization for frame buffer display. In *Computer Graphics (SIGGRAPH 82 Procs)*, volume 16(3), pages 297–307, July 1982.

[60] K. Hillesland and A. Lastra. Egpu floating-point paranoia. In *GP2*, pages C–8, 2004.

[61] J. Hirshon. A guide to CPU 3D instruction sets, August 1999. http://www.3d-design.com /newsletter /1999 /0899/horizon0899.html.

[62] C. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, California, 1989.

[63] M. Holst, N. Baker, and F. Wang. Adaptive multilevel finite element solution of the Poisson-Boltzmann equation I: algorithms and examples. *J. Comput. Chem.*, 21:1319–1342, 2000.

[64] M. J. Holst. Multilevel methods for the Poisson-Boltzmann equation. *Ph.D. thesis, Numerical Computing Group, University of Illinois at Urbana-Champaign*, 1993.

[65] B. Honig and A. Nicholls. Classical electrostatics in biology and chemistry. *Science*, 268:1144 – 1149, 1995.

[66] H. Hoppe. Progressive meshes. In *SIGGRAPH 96, Computer Graphics Proceedings*, Annual Conference Series, pages 99 – 108, August 1996.

[67] H. Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH 97 (Los Angeles, CA)*, Computer Graphics Proceedings, Annual Conference Series, pages 189 – 197. ACM Press, August 1997.

[68] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frustra. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 1–10, 1997.

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

[70] J. D. Jackson. *Classical Electrodynamics*. John Wiley and Sons, second edition, 1975.

[71] H. W. Jensen and J. Buhler. A rapid hierarchical rendering technique for translucent materials. In John F. Hughes, editor, *SIGGRAPH 2002, Computer Graphics Proceedings*, Annual Conference Series, pages 576–581, July 2002.

[72] H. W. Jensen, J. Legakis, and Julie Dorsey. Rendering of wet materials. In D. Lischinski and G. W. Larson, editors, *Rendering Techniques '99*, pages 273–282. Springer Verlag, 1999.

[73] H. W. Jensen, S. Marschner, M. Levoy, and P. Hanrahan. A practical model for subsurface light transport. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, Annual Conference Series, pages 511–518, August 2001.

[74] J. T. Kajiya. Anisotropic reflection models. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 15(3), pages 15–21, July 1985.

[75] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *15th ACM Symp. on Computational Geometry*, pages 351–359, 1999.

[76] D. King and J. Rossignac. Optimal bit allocation in compressed 3D models. *Journal of Computational Geometry, Theory and Applications*, 14:91–118, November 1999.

[77] T. E. Klein, C. C. Huang, E. F. Pettersen, G. S. Couch, T. E. Ferrin, and R. Langridge. A real-time malleable molecular surface. *J. Mol. Graphics*, 8(1):16–24 and 26–27, 1990.

[78] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6 (2):108–123, 2000.

[79] J. T. Klosowski and C. T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379, 2001.

[80] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In T. Ertl, K. Joy, and A. Varshney, editors, *IEEE Visualization '01*, pages 255–262, 2001.

[81] J. Kniss, S. Premože, C. Hansen, P. Shirley, and A. McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, 2003.

[82] G. Knittel. Using pre-integrated transfer functions in an interactive software system for volume rendering. In E. Fiume, editor, *Eurographics 2002 Short Presentations*, Annual Conference Series, pages 119–123, September 2002.

[83] J. J. Koenderink and A. J. van Doorn. Shading in the case of translucent objects. In *Proceedings of SPIE*, volume 4299, pages 312–320, 2001.

[84] O. Kreylos, N. Max, B. Hamann, S. N. Crivelli, and E. W. Bethel. Interactive protein manipulation. In *IEEE Visualization '03*, pages 581–588, 2003.

[85] P. Lacroute and M. Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In A. Glassner, editor, *SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference Series, pages 451–458, July 1994.

[86] A. R. Leach. *Molecular Modelling: Principles and Applications*. Prentice Hall, second edition, 2001.

[87] B. Lee and F. M. Richards. The interpretation of protein structures: Estimation of static accessibility. *J. Mol. Biol.*, 55:379 – 400, 1971.

[88] J. Leech, J. F. Prins, and J. Hermans. SMD: Visual steering of molecular dynamics for protein design. *IEEE Computational Science & Engineering*, 3(4):38–45, 1996.

[89] H. Lensch, M. Gosele, P. Bekaert, J. Kautz, M. Magnor, J. Lang, and H.-P. Seidel. Interactive rendering of translucent objects. In *Proc. IEEE Pacific Graphics 2002*, pages 214–224, 2002.

[90] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[91] J. Li and C. C. Kuo. Progressive coding of 3D graphics models. *Proceedings of the IEEE*, 86(6):1052–1063, June 1998.

[92] W. Lorensen and H. Cline. Marching cubes: a high resolution 3D surface construction algorithm. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):163–169, July 1987.

[93] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Conference Proceedings (Los Angeles, CA)*, Annual Conference Series, pages 198 – 208. ACM Press, August 1997.

[94] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, Inc, 2003.

[95] S. R. Marschner, S. H. Westin, E. P. F. Lafortune, K. E. Torrance, and D. P. Greenberg. Image-based BRDF measurement including human skin. In *Eurographics Workshop on Rendering*, pages 139–152, 1999.

[96] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[97] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3D scalar function. *San Diego Workshop on Volume Visualization, Computer Graphics*, 24(5):27–33, 1990.

[98] T. Mertens, J. Kautz, P. Bekaert, H.-P. Seidel, and F. Van Reeth. Interactive rendering of translucent deformable objects. In Per Christensen and Daniel Cohen-Or, editors, *Proceedings of the 14th Eurographics Symposium on Rendering*, pages 130–140, 2003.

[99] V. Milenkovic and L. Nackman. Finding compact coordinate representations for polygons and polyhedra. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, pages 244–252. ACM Press, June 1990.

[100] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2), April 1999.

[101] J. D. Mulder, J. J. van Wijk, and R. van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1):119–129, 1999.

[102] M. Nulkar and K. Mueller. Splatting with shadows. In *Volume Graphics 2001*, pages 35–50, 2001.

[103] H. Oberoi and N. M. Allewell. Multigrid solution of the nonlinear Poisson-Boltzmann equation and calculation of titration curves. *Biophys. J.*, 65:48 – 55, 1993.

[104] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January 2000.

[105] R. Pajarola and J. Rossignac. Squeeze: Fast and progressive decompression of triangle meshes. In *Proceedings Computer Graphics International CGI 2000*, pages 173–182, Los Alamitos, California, 2000. IEEE, Computer Society Press.

[106] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[107] F. Pellacini, J. A. Ferwerda, and D. P. Greenberg. Toward a psychophysically-based light reflection model for image synthesis. In *Siggraph 2000 Conference Proceedings*, Annual Conference Series, pages 55–64. ACM Press, 2000.

[108] K. Perlin. An image synthesizer. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19(3), pages 287–296, July 1985.

[109] M. Pharr and P. Hanrahan. Monte carlo evaluation of non-linear scattering equations for subsurface reflection. In Kurt Akeley, editor, *SIGGRAPH 2000, Computer Graphics Procedings*, Annual Conference Series, pages 75–84, July 2000.

[110] B.-T. Phong. Illumination for computer generated pictures. *CACM June 1975*, 18(6):311–317, 1975.

[111] S. M. Pizer and V. L. Wallace. *To Compute Numerically: Concepts and Strategies*. Little Brown Computer Systems Series, 1983.

[112] D. Pletinckx. Quaternion calculus as a basic tool in computer graphics. *The Visual Computer*, 5(1/2):2–13, March 1989.

[113] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[114] R. Ramamoorthi and P. Hanrahan. A signal-processing framework for inverse rendering. In E. Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, Annual Conference Series, pages 117–128, July 2001.

[115] R. Ramamoorthi and P. Hanrahan. Frequency space environment map rendering. In John Hughes, editor, *SIGGRAPH 2002, Computer Graphics Proceedings*, Annual Conference Series, pages 517–526, July 2002.

[116] K. Riley, D. Ebert, C. Hansen, and J. Levit. Visually accurate multi-field weather visualization. In *IEEE Visualization '03*, pages 279–286, 2003.

[117] W. Rocchia, E. Alexov, and B. Honig. Extending the applicability of the nonlinear Poisson-Boltzmann equation: Multiple dielectric constants and multivalent ions. *J. Phys. Chem. B*, 105:6507–6514, 2001.

[118] J. Rossignac. Edgebreaker: Connnectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1), January 1999.

[119] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June 1993.

[120] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[121] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[122] M. F. Sanner and A. J. Olson. Real time surface reconstruction for moving molecular fragments. In R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klein, editors, *Pacific Symposium on Biocomputing '97*, pages 385–396, 1997.

[123] Y. Sato, M. D. Wheeler, and K. Ikeuchi. Object shape and reflectance modeling from observation. In Turner Whitted, editor, *SIGGRAPH 97, Computer Graphics Proceedings*, Annual Conference Series, pages 379–388, August 1997.

[124] J. P. Schulze, M. Kraus, U. Lang, and T. Ertl. Integrating pre-integration into the shear-warp algorithm. In I. Fujishiro, K. Mueller, and A. Kaufmann,

editors, *Proceedings of the 2003 Eurographics/IEEE TVCG Volume Graphics Workshop (VG-02)*, pages 109–118, July 7–8 2003.

[125] F. X. Sillion, J. R. Arvo, S. H. Westin, and D. P. Greenberg. A global illumination solution for general reflectance distributions. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):187–196, July 1991.

[126] P. Sloan, J. Hall, J. Hart, and J. Snyder. Clustered principal components for precomputed radiance transfer. In *SIGGRAPH 2003, Computer Graphics Proceedings*, Annual Conference Series, page to appear, July 2003.

[127] P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In J. Hughes, editor, *SIGGRAPH 2002, Computer Graphics Proceedings*, Annual Conference Series, pages 527–536, July 2002.

[128] J. Stam. Diffraction shaders. In Alyn Rockwood, editor, *SIGGRAPH 99, Computer Graphics Proceedings*, Annual Conference Series, pages 101–110, August 1999.

[129] J. Stam. An illumination model for a skin layer bounded by rough surfaces. In S. J. Gortler and K. Myszkowski, editors, *Rendering Techniques '01*, pages 39–52. Springer Verlag, 2001.

[130] K. Sugihara. On finite-precision representations of geometric objects. *Journal of Computer and System Sciences*, 39:236–247, 1989.

[131] S. Sukharev, S. R. Durell, and H. R. Guy. Structural models of the MSCL gating mechanism. *J. Biophys*, 81(2):917–936, 2001.

[132] Y. Sun, F. D. Fracchia, M. S. Drew, and T. W. Calvert. Rendering iridescent colors of optical disks. In *Rendering Techniques '00*, pages 341–352, 2000.

[133] J. A. Tainer, E. D. Getzoff, J. Sayre, and A. J. Olson. Modeling intermolecular intractions: topography, mobility and electrostatic recognition. *J. Mol. Graphics*, 3:103–105, 1985.

[134] C. Tanford and J. G. Kirkwood. Theory of protein titration curves. I. General equations for impenetrable spheres. *J. Am. Chem. Soc.*, 79:5333 – 5339, 1957.

[135] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 123–132. ACM SIGGRAPH, 1998.

[136] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 123–132. ACM SIGGRAPH, 1998.

[137] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.

[138] B. Taylor and C. E. Kuyatt. Guidelines for evaluating and expressing the uncertainity of NIST measurement results. Technical Report Technical Note 1297, National Institute of Standards and Technology, Gaithersburg, MD, January 1993.

[139] G. Turk. Re-tiling polygonal surfaces. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26(2), pages 55–64, July 1992.

[140] F.-Y. Tzeng, E. Lum, and K.-L. Ma. A novel interface for higher-dimensional classification of volume data. In G. Turk, J. J. van Wijk, and R. Moorhead, editors, *IEEE Visualization '03*, pages 505–512, 2003.

[141] A. Varshney, F. P. Brooks, Jr., and W. V. Wright. Computing smooth molecular surfaces. *IEEE Computer Graphics & Applications*, 15(5):19–25, September 1994.

[142] G. J. Ward. Measuring and modeling anisotropic reflection. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26(2), pages 265–272, July 1992.

[143] J. Warwicker and H. C. Watson. Calculation of electrostatic potential in the active site cleft due to α-helix dipoles. *J. Mol. Biol.*, 155:53 – 62, 1982.

[144] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In M. Cohen, editor, *SIGGRAPH '98*, Computer Graphics Proceedings, Annual Conference Series, pages 169–178, July 1998.

[145] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):367–376, August 1990.

[146] P. Wonka, M. Wimmer, and F. Sillion. Instant visibility. *Computer Graphics Forum*, 20(3) (EG 2001 Proceedings):411–421, 2001.

[147] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3, No. 2:171 – 183, June 1997.

[148] Z. Xiang. Color image quantization by minimizing the maximum intercluster distance. *ACM Transactions on Graphics*, 16(3):260–276, July 1997.

[149] S. Yoon, B. Salomon, and D. Manocha. Interactive view-dependent rendering with conservative occlusion culling in complex environments. In *IEEE Visualization '03*, pages 163–170, October 2003.

[150] Y. Yu, P. Debevec, J. Malik, and T. Hawkins. Inverse global illumination: recovering reflectance models of real scenes from photographs. In Alyn Rockwood, editor, *SIGGRAPH 99, Computer Graphics Proceedings*, Annual Conference Series, pages 215–224, August 1999.

[151] H. Zhang and K. E. Hoff. Fast backface culling using normal masks (color plate S. 189). In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 103–106, New York, April27–30 1997. ACM Press.

[152] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. *Proceedings of SIGGRAPH'97*, pages 77–88, 1997.