Stony Brook University

Lattice Volume Rendering

A Dissertation Presented
by
Feng Qiu

to
The Graduate School
in Partial Fulfillment of the
Requirements
for the Degree of
Doctor of Philosophy
in
Computer Science
Stony Brook University

December 2008

Stony Brook University
The Graduate School

Feng Qiu

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

Arie E. Kaufman, Dissertation Advisor
Distinguished Professor, Computer Science Department

Klaus Mueller, Chairperson of Defense
Associate Professor, Computer Science Department

Xianfeng David Gu
Assistant Professor, Computer Science Department

Torsten Möller, External Committee Member
Associate Professor, Computer Science Department, Simon Fraser University

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Lattice Volume Rendering

by

Feng Qiu

Doctor of Philosophy

in

Computer Science

Stony Brook University

2008

Direct volume rendering renders volume datasets without intermediate surface representation. However, due to the large data size, high quality volume rendering is a challenging research problem. Researchers have proposed various volumetric lighting models that consider no scattering, single scattering and multiple scattering.

In practice, nearly all numerical solutions of graphics and visualization problems assume a discretization of the simulation domain into a grid of cells or points. In most cases, the discretization is regular and generates a lattice. This work presents several new techniques and algorithms for rendering lattice volumes. The 3D Cartesian Cubic (CC) lattice is the most frequently used lattice for volumes. Because of its simplicity, it is natively supported by many kinds of hardware, especially on the graphics processing unit (GPU). Three algorithms are presented for rendering 3D CC lattices without scattering: an object order cell projection algorithm based on min-max octree data structure that renders large datasets beyond GPU memory capacity, a hybrid method that improves the rendering speed by CPU/GPU parallelism, and a method of ray tracing height field on the GPU. For rendering the CC lattice with single scattering, a half angle splatting algorithm has achieved real time rendering of smoke. To overcome the popping artifact of half angle based method, a lighting volume based method is proposed and further accelerated with the GPU.

A novel volumetric global illumination framework based on the Face-Centered Cubic (FCC) lattice is proposed to compute multiple scattering for volumetric

global illumination. An FCC lattice has better sampling efficiency than the CC lattice. Furthermore, it has the maximal possible kissing number, which provides optimal 3D angular discretization. The proposed algorithms greatly simplify the computation of multiple scattering and minimize illumination information storage due to angular discretization.

Two distinct applications of lattice computation and rendering have been developed, dispersion visualization for the flow simulation of lattice Boltzmann Method (LBM) and computer aided polyp detection for virtual colonoscopy. The LBM simulates the flow field by the micro-scale Boltzmann kinetics of fluid elements on a lattice, and the simulation results are rendered on GPUs and GPU clusters. The second application focuses on a new pipeline for computer aided polyp detection for virtual colonoscopy. The new pipeline has a volume rendering stage to generate the electronic biopsy image of a conformal flattened colon. Then, the polyps are detected on the single 2D biopsy image with texture analysis, which is significantly faster than traditional shape analysis methods.

To my parents.

Contents

List of Tables

List of Figures

Acknowledgements

First of all, I would like to thank sincerely my parents, my sister, and my brother for their invaluable love in all my life. Without their support, trust, and inspiration, I could have not finished my research in the last few years.

I am deeply grateful to my adviser, Arie Kaufman, for years of guidance in pleasant and exciting journey. I have learned much from him, not only his insightful thinking and the way to do research but also the attitude towards life. Without his support and encouragement, I could not have achieved my research goals and makes this dissertation possible.

I would like to thank Prof. Klaus Mueller, Xianfeng David Gu, Hong Qin, Dimitris Samaras, Michael Ashikhmin in the Visualization lab and Prof. Jerome Liang at the Department of Radiology, for their great collaboration and valuable suggestions in various aspects of my research over the years.

I would like to thank Bin Zhang for his great work and technical support. I would like to thank the current and past members of the Visualization lab, especially Wei Hong, Zhe Fan, Ye Zhao, Xiaoming Wei, Wei Li, Neophytou Neophytos, Jianning Wang, Huamin Qu, Haitao Zhang, Suzanne Yoakum-Stover, Fang Xu, Shengying Li, Aili Li, Yiping Han, Yu-chuan Kuo, Kaloian Petkov, Joseph Marino, Miao Jin, Xiaotian Yin for extensive collaboration, joint publication, and friendship.

The work has been partially supported by the ONR grant N000140110034, NSF grants CCR-0306438, IIS-0097646 and CCF-0702699, and NIH grants CA082402 and CA110186.

Publications

1. W. Hong, X. Gu, F. Qiu, and A. Kaufman, Conformal Colon Flattening for Virtual Colonoscopy, *IEEE Trans. on Medical Imaging*, (submitted), 2008.
2. Z. Fan, Y. Kuo, Y. Zhao, F. Qiu, A. Kaufman, and W. Arcieri, Visual Simulation of Thermal Fluid Dynamics in a Pressurized Water Reactor, *The Visual Computer*, (submitted), 2008.
3. K. Petkov, Z. Fan, F. Qiu, K. Mueller, and A. Kaufman, Efficient Flow Simulation with LBM on Face-Centered Cubic Lattices, *IEEE Trans. on Visualization and Computer Graphics*, (submitted), 2008.
4. W. Hong, F. Qiu, and A. Kaufman, Hybrid Volumetric Ray-Casting, *The Visual Computer*, (to appear), 2008.
5. F. Qiu, B. Zhang, K. Petkov, L. Chong, A. Kaufman, K. Mueller, and X. Gu, Enclosed Five-Wall Immersive Cabin, *Lecture Notes in Computer Science*, (to appear), 2008.
6. F. Qiu, J. Marino, and A. Kaufman, Accelerated CAD Pipeline with Texture Analysis, *International Symposium on Virtual Colonoscopy*, (to appear), 2008.
7. F. Qiu, Z. Fan, X. Yin, A. Kaufman, and X. Gu, Colon Flattening with Discrete Ricci Flow, *MICCAI Workshop on Virtual Colonoscopy*, pp. 97-101, 2008.
8. F. Qiu, J. Marino, and A. Kaufman, Computer Aided Polyp Detection with Texture Analysis, *MICCAI Workshop on Virtual Colonoscopy*, pp. 148-152, 2008.
9. J. Marino, F. Qiu, and A. Kaufman, A Proposed Method for the Co-Registration of Virtual and Optical Colonoscopy Views, *MICCAI Workshop on Virtual Colonoscopy*, pp. 122-126, 2008.

10. Z. Fan, F. Qiu, and A. Kaufman, Zippy: A Framework for Computation and Visualization on a GPU Cluster, *Computer Graphics Forum*, 27(2):341-350, 2008.
11. J. Marino, F. Qiu, and A. Kaufman, Virtually Assisted Optical Colonoscopy, *SPIE Medical Imaging*, 6916:0J, 2008.
12. F. Qiu, F. Xu, Z. Fan, N. Neophytos, A. Kaufman, and K. Mueller, Lattice-Based Volumetric Global Illumination, *IEEE Trans. on Visualization and Computer Graphics*, 13(6):1576-1583, 2007.
13. Y. Zhao, F. Qiu, Z. Fan, and A. Kaufman, Flow Simulation with Multi-resolution LBM, *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 181-188, 2007.
14. W. Hong, J. Wang, F. Qiu, A. Kaufman, and J. Anderson, Colonoscopy Simulation, *SPIE Medical Imaging*, 6511:0R, 2007.
15. W. Hong, F. Qiu, J. Marino, and A. Kaufman, Computer-aided Detection of Colonic Polyps Using Volume Rendering, *SPIE Medical Imaging*, 6514:06, 2007.
16. Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y. Kuo, A. Kaufman, K. Mueller, Visual Simulation of Heat Shimmering and Mirage, *IEEE Trans. on Visualization and Computer Graphics*, 13(1):179-189, 2007.
17. A. Kaufman, W. Hong, X. Gu, and F. Qiu, Patent (pending): System and Method for Computer Aided Polyp Detection, 2006.
18. W. Hong, F. Qiu, and A. Kaufman, A Pipeline for Computer Aided Polyp Detection, *IEEE Trans. on Visualization and Computer Graphics*, 12(5):861-868, 2006.
19. W. Hong, X. Gu, F. Qiu, M. Jin, and A. Kaufman, Conformal Virtual Colon Flattening, *ACM Symposium on Solid and Physical Modeling*, pp. 85-93, 2006.
20. Y. Zhao, L. Wang, F. Qiu, A. Kaufman, and K. Mueller, Melting and Flowing in Multiphase Environment, *Computers & Graphics*, 30(4):519-528, 2006.
21. X. Wei, F. Qiu, W. Li, S. Yoakum-Stover, and A. Kaufman, Visual Simulation of Chemical Garden, *Computer Graphics International*, pp. 74-81, 2005.
22. W. Hong, F. Qiu, and A.Kaufman, GPU-based Object-Order Ray-Casting for Large Datasets, *Volume Graphics*, pp. 177-186, 2005.

23. Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, GPU Cluster for High Performance Computing, *ACM/IEEE Supercomputing*, pp. 47-54, 2004.
24. F. Qiu, Y. Zhao, Z. Fan, X. Wei, H. Lorenz, J. Wang, S. Yoakum-Stover, A. Kaufman, and K. Mueller, Dispersion Simulation and Visualization for Urban Security, *IEEE Visualization*, pp. 553-560, 2004.
25. H. Zhang, F. Qiu, and A. Kaufman, Fast Hybrid Approach for Texturing Point Models, *Computer Graphics Forum*, 23(4):715-725, 2004.
26. X. Wei, Y. Zhao, Z. Fan, W. Li, F. Qiu, S. Yoakum-Stover, and A. Kaufman, Lattice-Based Flow Field Modeling, *IEEE Trans. on Visualization and Computer Graphics*, 10(6):719-729, 2004.
27. H. Qu, F. Qiu, N. Zhang, A. Kaufman, and M. Wan, Ray Tracing Height Fields, *Computer Graphics International*, pp. 202-209, 2003.

Chapter 1

Introduction

In traditional computer graphics, images are synthesized from geometric primitives such as lines, points and polygons. However, they are not suitable for representing the inside of objects such as a CT scan of a human bone. Some objects such as clouds and smoke are too voluminous to be represented efficiently in geometric primitives. Therefore, a volume is proposed to represent three dimensional (3D) objects with information inside them [6, 68].

Over the last few decades, technologies and methods for acquiring volumetric datasets have been developing rapidly. For example, Magnetic Resonance Imaging (MRI) and Computed Tomography (CT) scanners used in hospitals scan patients and generate series of 2D slices, which are later 3D reconstructed into a volume model and visualized for diagnosis. Computational Fluid Dynamics (CFD) methods in mechanical engineering produce 3D flow densities and velocity fields. Researchers in computer graphics and visualization have developed many techniques to visualize 3D volume data. A volume can be rendered by converting the data into explicit surface representation by surface extraction methods such as the Marching Cubes iso-surface extraction [88], which can be rendered with traditional computer graphics methods. However, the surface extraction is a slow process and it produces huge amount of small triangles to represent the surfaces. Many of the generated triangles are even smaller than a pixel on the screen. Rendering such large geometric models is a big challenge for the most advanced graphics hardware. The surface model is essentially a 2D representation of a volumetric object. Further, the surface

extraction process inevitably loses some information. Therefore, direct volume rendering has been proposed to process volume data and produce images without an intermediate surface representation. It is capable of preserving and revealing more information of the volume data than surface extraction.

1.1 Volume Data Representation

Mathematically, a volume is a continuous function defined on a 3D domain (R^3). In practice, nearly all numerical solutions assume a discretization of the simulation domain into a grid of cells or points, because the computer can easily store and process discrete data. This discretization can be regular or irregular. The irregular grids can be further divided into two categories: unstructured grids and curvilinear grids. Figure 1.1(b)-(c) illustrates examples of curvilinear and unstructured grids.

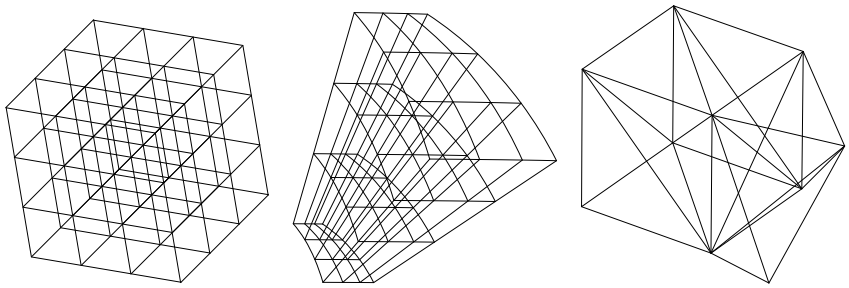


Figure 1.1: Three different volume data grids: (a) regular, (b) curvilinear, and (c) unstructured grids.

This work focuses on volumes on lattices, because most existing volume datasets and scanning modalities assume a regular discretization. A lattice (or a point lattice) is a set of points called lattice sites regularly positioned in space [22]. Mathematically, a lattice in R^n is a discrete subgroup [30] of R^n, which can be generated from a vector basis by a linear combination with integral coefficients. In other words, a lattice is the subgroup $\{a_1v_1 + a_2v_2 + \cdots + a_nv_n\}$, where $\{v_1, v_2, \cdots, v_n\}$ is the vector basis and a_i are integers. For example, $\{a_1\binom{1}{0} + a_2\binom{0}{1}\}$ is a lattice of R^2, which is a 2D cubic lattice as shown in Figure 1.2. The sites of a lattice are

connected with a set of lines (or links). In Figure. 1.2, the green vectors are the two basis vectors. The dots are the lattice sites and the blue one is the origin. The lines are the lattice links. Each site in this lattice has four neighbors.

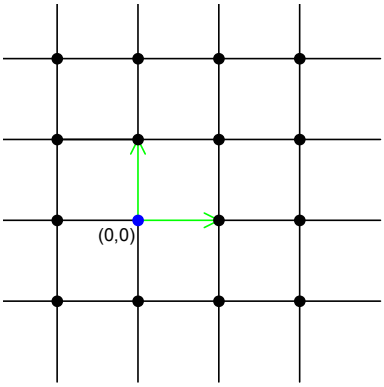


Figure 1.2: 2D cubic lattice.

A lattice may be constructed with different vector basis. For example, the vector basis for the 2D cubic lattice can be $\{\binom{0}{1}, \binom{1}{0}\}$ or $\{\binom{0}{1}, \binom{1}{1}\}$. Given a vector basis $\{v_0, v_1, \cdots, v_n\}$ where $v_i = (v_{i0}, v_{i1}, \cdots, v_{in})^T$, the matrix

$$M = (v_0, v_1, \cdots, v_n) = \begin{pmatrix} v_{00} & v_{10} & \cdots & v_{n0} \\ v_{01} & v_{11} & \cdots & v_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{0n} & v_{1n} & \cdots & v_{nn} \end{pmatrix} \tag{1.1}$$

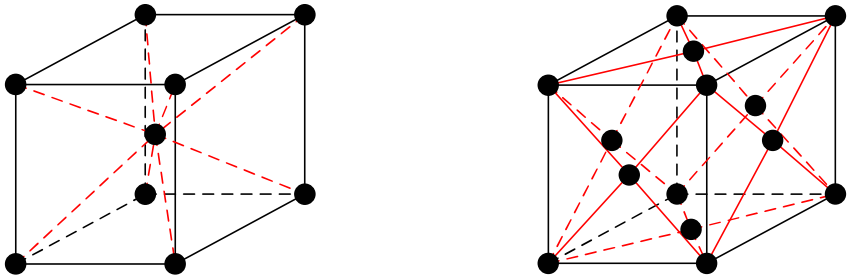
is called a *generator matrix* for the $(n+1)$D lattice. Because of this simple representation, a lattice can be compactly stored in a 1D array.

The lattice sites result from the discretization of space or space-time, and the lattice links are the results of angular discretization. In fact, space tiling and covering is a well-studied research topic in mathematics [46, 47], and researchers in the graphics community have also worked in this domain, studying geometric modeling with 3D solids [104]. In physics, a lattice model is a physical model defined on a lattice, as opposed to the continuum of space or space-time. Further, it has been widely recognized [12, 117] that atoms, molecules or ions in crystals are periodically positioned on a lattice. Then, within a lattice model, the lattice sites are associated with some physical and chemical properties or attributes, and the physical process is modeled by some equations or operations acting on the lattice sites.

The most commonly used lattice is the simple Cubic Cartesian (CC) lattice, but some researchers have also proposed discretization as well as voxelization algorithms on the Body Centered Cubic (BCC) lattice (Figure 1.3(a)), the Face Centered Cubic (FCC) lattice (Figure 1.3(b)), and other general grids [89, 90, 166]. A 3D BCC lattice can be constructed by adding one lattice site at the center of every cell of the simple cubic lattice. A 3D FCC lattice can be constructed by adding sites at the centers of the square surfaces of every cell of the simple cubic lattice. The generator matrix for BCC and FCC lattices are

$$M_{BCC} = \begin{pmatrix} 1 & 0 & \frac{1}{2} \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} \end{pmatrix}, \qquad M_{FCC} = \begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}$$

respectively, assuming the cubic cell size is 1.



(a) BCC Lattice. (b) FCC Lattice.

Figure 1.3: (a) 3D BCC and (b) 3D FCC lattices.

The BCC lattice in particular has been utilized in isosurface extraction and 3D/4D rendering [14,26,33,95,112,149,151], and in CT reconstruction [93,94,108, 109]. The main motivation in all of these applications was the lossless reduction (assuming radial frequency spectra) of the required grid points due to the better sampling efficiency of the BCC lattice.

Interpolation filters reconstruct a continuous function from the lattice points. In lattices of regular geometry, the appropriate shape of this interpolation filter is determined by the Voronoi cell of the lattice, which defines the *Nyquist Boundary Solid (NBS)* in frequency space. The NBS separates the signal spectrum from its aliases. The filter function, on the other hand, is dictated by signal quality constraints. An ideal filter function should satisfy:

1. Minimal contributions in the filter's frequency spectrum stop-band and non-scaled contributions in the spectrum's pass-band (the spectra outside and inside the NBS, respectively);
2. Packing the NBS of main spectrum and aliases most compactly, as this creates the sparsest spatial sampling [10] and saves storage space;
3. The distribution of the lattice points in the computational domain (space or space-time) should be uniform.

Assuming that the lattice sites results from an isotropic and band-limited sampling function, the support of the corresponding lattice in the frequency domain is a hyper-sphere, surrounded by a set of alias replicas. Hence, the most efficient sampling scheme arranges the replicated (hyper-spherical) frequency response as densely as possible in the frequency domain to avoid overlapping of the aliased spectra. As demonstrated in multi-dimensional signal theory [29] an optimal sampling scheme is obtained when the frequency response of the sampling lattice is an optimal sphere packing lattice [22]. Optimal sampling lattices can achieve up to 13.4%, 29.3%, and 50% of savings in 2, 3 and 4 dimensions, and they have been used in volume visualization [33, 112, 150] with high quality image results. It is shown [22, 50] that the FCC lattice achieves the sphere close packing. Theußl et al. have used this result to prove that the BCC lattice is the optimal regular lattice for volume sampling. The BCC grid has become quite popular [4, 28, 102, 112, 149, 150]. Much research has been devoted to design complex (non-isotropic) filters to capture the rhombic dodecahedral shape of the FCC NBS precisely [34] and the hexagonal shape of the 2D hexagonal lattice [26, 152].

1.2 Optical Model for Volume Rendering

Direct volume rendering does not produce resulting images from geometric primitives, therefore an optical model describing how light interacts with the volume is required [96]. There are three major processes for the interaction between light and volumetric objects:

1. Emission - light energy sent out from particles;
2. Absorption - light energy retained by particles;

3. Scattering - changing light direction by particle reflection.

Absorption is described by the absorption coefficient σ_a of the medium, which represents the probability of light being absorbed by the medium per unit distance. The light intensity changed by absorption at position is described by the differential equation:

$$\frac{dI(x,\omega)}{ds} = -\sigma_a(x,\omega)I(x,\omega) \tag{1.2}$$

where $I(x,\omega)$ is the light intensity at position x from direction ω. In computer graphics, the absorption coefficient is usually a function of the position and independent of light direction. Equation 1.2 can be solved and the intensity of the ray starting at point x with direction ω after traveling distance d is:

$$I(x+d\omega,\omega) = I_0(x,\omega)e^{-\int_0^d \sigma_a(x+t\omega,\omega)dt} \tag{1.3}$$

where I_0 is the light source intensity. Similarly, the emission coefficient σ_e is used to describe the light emitted from the medium:

$$\frac{dI(x,\omega)}{ds} = \sigma_e(x,\omega) \tag{1.4}$$

and the solution is

$$I(x+d\omega,\omega) = I_0(x,\omega) + \int_0^d \sigma_e(x+t\omega,\omega)dt. \tag{1.5}$$

One of the commonly used optical models in direct volume rendering involves only absorption and emission. For any point on a ray, the amount of light varies:

$$\frac{dI(x,\omega)}{ds} = -\sigma_a(x,\omega)I(x,\omega) + \sigma_e(x,\omega) \tag{1.6}$$

and the solution is:

$$I(x+d\omega,\omega) = I_0(x,\omega)e^{-\int_0^d \sigma_a(x+t\omega,\omega)dt} + \int_0^d \sigma_e(x+t\omega,\omega)e^{-\int_t^d \sigma_a(x+(d-s)\omega,\omega)ds}dt. \tag{1.7}$$

As light travels in the medium, it may change the direction due to scattering by the particles. The scattering process reduces the light intensity in its incoming direction ω, which is called *out-scattering* and increases the light intensity in its outcoming direction ω', which is called *in-scattering*. The scattering coefficient σ_s

represents the probability of light scattered in the medium. The radiance reduction due to out-scattering is:

$$\frac{dI(x,\omega)}{ds} = -\sigma_s(x,\omega)I(x,\omega). \tag{1.8}$$

For convenience, the effect of out-scattering is usually calculated with absorption together and the combination of these two effects is called *extinction*. The extinction coefficient σ_t is simply the sum of σ_a and σ_s:

$$\sigma_t(x,\omega) = \sigma_a(x,\omega) + \sigma_s(x,\omega). \tag{1.9}$$

The in-scattering increases the radiance at a point x from scattering from other directions. To account for in-scattering, the phase function is used to describe the conditional probability of light scatter from ω to be scattered in direction ω' assuming that the light is scattered, and it obeys:

$$\int_\Omega f(x,\omega,\omega')d\omega' = 1 \tag{1.10}$$

where Ω is the solid angle space at x. Most media in real world is *isotropic* and the phase functions only depend on the cosine of the angle θ between ω and ω'. Another important and frequently used optical model takes account of shadows. It assumes that the light is only scattered once, which occurs at the sampling point, as shown in Figure 1.4(a). Thus, this optical model is also a single scattering model. The intensity of light arriving at the sampling point can be solved with:

$$I(x+d\omega',\omega') = I_0(x,\omega')e^{-\int_0^\infty \sigma_a(x+t\omega',\omega')dt} \tag{1.11}$$

where ω' is the unit light direction. And the solution of light intensity arriving at a image pixel is:

$$\begin{aligned} I(x+d\omega,\omega) &= I_0(x,\omega)e^{-\int_0^d \sigma_a(x+t\omega,\omega)dt} + \int_0^d \Bigg(f(x+t\omega,\omega,\omega')I_0(x+t\omega,\omega') \\ &\quad e^{-\int_0^\infty \sigma_a(x+t\omega+s\omega',\omega')ds} e^{-\int_t^d \sigma_a(x+(d-s)\omega,\omega)ds} \Bigg) dt \end{aligned} \tag{1.12}$$

The single scattering model is only valid for low albedo media, where the probability for a photon to be scattered is low. For some volumetric objects such

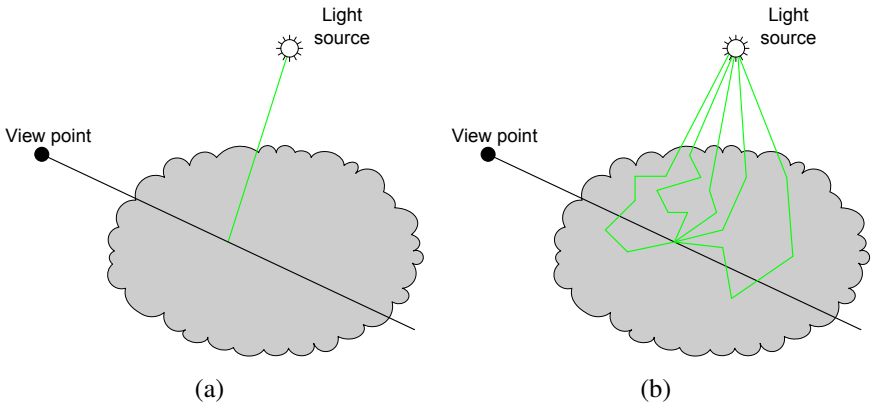


Figure 1.4: (a) Single and (b) multiple scattering.

as clouds, a beam of light is scattered multiple times before it is finally absorbed or arrives at the image plane. Multiple scattering is important for high albedo participating media. The differential equation accounting for multiple scattering is

$$\frac{dI(x,\omega)}{ds} = -\sigma_t(x,\omega)I(x,\omega) + \sigma_e(x,\omega) + \sigma_s(x,\omega)\int_\Omega f(x,\omega,\omega')I(x,\omega')d\omega', \tag{1.13}$$

which does not have an analytic solution. A two-pass numerical solution of Equation 1.13 is

$$\begin{aligned} I(x+d\omega,\omega) &= I(x,\omega)e^{-\int_0^d \sigma_t(x+t\omega,\omega)dt} + \\ &\quad \int_0^d R(x+t\omega,\omega)e^{-\int_t^d \sigma_t(x+s\omega,\omega)ds}dt, \qquad (1.14) \\ R(x,\omega) &= \sigma_e(x,\omega) + \sigma_s(x,\omega)\int_\Omega f(x,\omega,\omega')I(x,\omega')d\omega'. \qquad (1.15) \end{aligned}$$

1.3 Background

1.3.1 Existing Rendering Methods

In low-albedo rendering, the volume rendering integral [9, 67, 96] is evaluated along straight lines. The volume rendering methods can be classified into object-order, image-order, domain-based and hybrid techniques. Object-order techniques use a forward mapping scheme where the volume data is mapped onto the image

plane. In image-order algorithms, a backward mapping scheme is used where rays are cast from each pixel in the image plane through the volume data to determine the final pixel value. Hybrid rendering methods are combinations of object-order and image-order algorithms, in which the volume is traversed in object order while the contribution of each voxel to the image is computed in image order. In domain-based techniques the spatial volume data is first transformed into an alternative domain, such as compression frequency and wavelet, and then a projection is generated directly from that domain.

Splatting, proposed by Westover [165], is an object-order rendering techniques. The renderer calculates a 2D footprint for each data sample and uses the footprint to distribute the sample's energy onto the image plane. Each sample is transformed from input 3D space coordinates (x_i, y_i, z_i) to (u_i, v_i) screen position. Then, the sample is shaded with some rule such as the Phong illumination model and gets the color and opacity values. Next, the renderer determines the pixels affected by this sample and adds its contribution to those pixels. Mueller and Crawfis [105] have proposed a view-aligned sheet buffer method to remove the popping artifact. Mueller et al. [107] have analyzed the common approximation errors in the splatting process for perspective viewing and have presented an antialiasing extension to the basic splatting algorithm that mitigates the spatial aliasing for high-resolution volumes. They have introduced a resampling filter combining a reconstruction with a low-pass kernel to avoid aliasing artifacts. Mueller et al. [106, 113] have adapted the splatting pipeline by performing the classification and shading process after the voxels have been projected onto the screen to produce crisp edges and surface details. Also, time-varying data on BCC grids [112] has been efficiently rendered with the splatting method. Zwicker et al. [180] have presented a splatting approach based on elliptical Gaussian kernels. Splatting approach has been used for rendering points [136] and surfaces [181].

Cell projection is another popular object-space rendering technique, especially for unstructured volume grids [97, 138]. The basic idea of cell projection is to decompose the volume into tetrahedral cells. Then, the projected tetrahedra is further decomposed into one to four triangles. For the vertices of triangle(s) around the tetrahedron's silouette, they have zero color and opacity because of zero thickness.

The viewpoint and any other triangle vertex determines a line intersecting the surface of the tetrahedron at two points. The color and opacity values of both points can be interpolated from the tetrahedron vertices. Then, an approximated integration is applied on the segment to get the color and opacity of the triangle vertex. At last, the triangles are scan converted in the graphics hardware pipeline. This method requires that the cells are sorted in view-dependent depth order. A lot of algorithms have been proposed to solve this problem, such as the Mesh Polyhedra Visibility Ordering (MPVO) [168], which exploited the connectivity information in acyclic convex meshes; XMPVO [142], which removed the assumption of MPVO that the mesh be convex and connected; BSP-XMPVO [21], which introduced the BSP tree on the set of boundary faces of the mesh to improve speed. Stein et al. have described a $O(n^2)$ algorithm to sort n arbitrarily shaped convex polyhedra [145].

Farias et al. [36] have presented the ZSweep algorithm based on sweeping the data with a plane parallel to the viewing plane, in order of increasing z, projecting the faces of cells that are incident to vertices as they are encountered by the sweep plane. The efficiency arises from the fact that the algorithm exploits the implicit (approximate) global ordering that the z-ordering of the vertices induces on the cells that are incident on them. The algorithm projects cells by projecting each of their faces, with special care taken to avoid double projection of internal faces and to assure correctness in the projection order. The contribution for each pixel is computed in stages, during the sweep, using a short list of ordered face intersections, which is known to be correct and complete at the instant that each stage of the computation is completed.

Yagel et al. [172, 174] have proposed a fast approximation algorithm, which transformed the grid vertices to image space with graphics hardware for a given view direction, then incrementally computed the 2D polygon-meshes by letting a set of equidistant planes, parallel to the screen plane, intersect (slice) the transformed grid, finally used the graphics hardware to render (interpolate-fill) the polygon-meshes and composite them in a front-to-back order. Westermann [163] has proposed a technique which took advantage of hardware accelerated polygon rendering and 2D texture mapping and thus avoided any sorting of the tetrahedral elements.

Ray casting is the most famous image-order volume rendering method [78,79]. In these implementations, shading and classification is performed first with transfer

function and gradients. Then, for each pixel on the image plane, one ray is cast into the volume data along which the volume data is sampled and accumulated. Usually a trilinear interpolation is used to get color and opacity values of sample points while other interpolation filters are also available as discussed in Section 1.1. The color and opacity values are then composited to different pixels in either back-to-front and front-to-back order. The shading and classification stage can be implemented after interpolation in post-classification rendering.

Levoy's hierarchical data structure, octree, has been employed to decompose the volume data set according to the opacity values [79]. In the ray traversing procedure, those empty regions were skipped thus improving the rendering speed. The second optimization technique, called "early ray termination", was applied in front-to-back compositing. It stopped the traversing procedure for a ray when the corresponding pixel's opacity value was larger than certain user specified threshold.

One disadvantage of ray casting is aliasing. For parallel projections, the rays that are cast through the volume maintain a constant sampling rate on the underlying volume data. For perspective projections, however, the rays do not maintain such a continuous and uniform sampling rate. Levoy et al. have proposed to use a 3D-mipmap representation of the underlying volume data to create larger sampling kernels when the rays diverge [80]. Novins et al have presented a technique to split rays into four child rays once the neighboring rays diverge past some threshold [115]. Kreeger et al. have proposed the ER-Perspective algorithm by dividing the view frustum into regions based on exponentially increasing distances from the viewpoint [71]. Then, continuous rays are cast back-to-front (or front-to-back) and merge (or split) the rays once they become too close (or too far) from each other.

Ray tracing can also be applied in rendering unstructured volume data. The ray-segment, the part of a ray inside the volume, determines the contribution of data to a pixel. Following a ray-segment inside the volume can be efficiently achieved with the adjacency information of cells. The ray-segment is generated through identifying the first cell intersected with a ray. Garrity has sorted all boundary faces into a coarse mesh and only the faces in the mesh region intersected by the ray have been tested [41]. Bunyk et al. have solved the first cell problem by transforming boundary faces into screen space and sorting the intersection point in depth value for each screen pixel [11].

In hybrid methods, the volume is traversed in object order and the contributions of data are accumulated in image order. Yagel and Kaufman have proposed the template-based method for parallel projection [173]. The algorithm determines which of the volume faces is most perpendicular to projection direction as the base-plane. The same form for all rays is stored as the ray-template. For each pixel, one ray is cast into the volume by repeating a sequence of steps specified by the ray-template. Finally, the base-plane is warped to the image plane.

Lacrout et al. [74] have proposed a method based on a factorization of the viewing matrix into a 3D shear parallel to the slices of the volume data, a projection to form a distorted intermediate image, and a 2D warp to produce the final image. The view transformation matrix M_{view} can be defined as a factorization $M_{view} = P \cdot S \cdot M_{warp}$ where P is a permutation matrix which transposes the coordinate system to make the z-axis the principal viewing axis, S transforms the volume into sheared object space, M_{warp} transforms sheared object coordinates into image coordinates. A simple volume rendering algorithm based on the shear-warp factorization operates as follows. First, transform the volume data to sheared object space by translating and resampling each slice according to S. For perspective transformations, also scale each slice. P specifies which of the three possible slicing directions to use. Then, composite the resampled slices together and get a 2D intermediate image in sheared object space. At last, transform the intermediate image to image space by warping it according to M_{warp}. This second resampling step produces the correct final image. The shear-warp factorization allows to implement coherence optimizations for both the volume data and the image with low computational overhead because both data structures can be traversed simultaneously in scanline order. It is one of the fastest software rendering algorithms for regular grids. The shear-warp method needs three stacks of slices for different projection directions, each with one copy of volume data. When base-plane changes, the stacks used for projection also changes and causes popping. The sampling distance between slices is fixed and may cause artifacts in some cases.

Global illumination has not been widely employed in volume rendering because of the computation complexity. When a photon encounters an object, it might be reflected, refracted, and scattered many times before finally being absorbed or exiting the scene. It is closely related to the radiative transfer problem that has been

studied by physicists for decades [16]. The simulation of all kinds of interaction is time-consuming, and many simplified models have been proposed in computer graphics. Max [96] has evaluated several optical models for direct volume rendering and presented an integral equation for light transport in volumes including multiple scattering. Blinn [9] has analytically solved the transport equation for constant density medium with single scattering. Kajiya and von Herzen [67] have proposed tracing rays in inhomogeneous volumes. To calculate multiple scattering, spherical harmonics have been used. Radiosity [18, 44, 114] is a finite element method, modeling light inter-reflections between diffuse surfaces with equation

$$B_i dA_i = E_i dA_i + R_i \int_j B_j F_{ji} dA_j, \tag{1.16}$$

where B_i, E_i and R_i are the radiosity, emitted energy and reflectivity of patch i, respectively, and F_{ji} represents how much energy from patch j can arrive at patch i. The radiosity method was extended to glossy and mirror reflections [59, 139, 140, 153] and participating media [135]. Rushmeier and Torrance [135] have exploited the zonal method for isotropic scattering. Max [98] has extended the discrete ordinates method to capture anisotropic multiple scattering. Based on the Monte Carlo ray tracing method in which rays interact with objects stochastically [23, 24, 66], Jensen proposed the two-pass photon mapping algorithm [60]. In pass 1, photons are emitted and traced through the environment where the photons interact with objects in a stochastic way, and the illumination information generated is stored in a photon map. In pass 2, the photons are then used for estimating the irradiance of a given region. Photon Mapping has been applied to caustics [61], participating media [63] and subsurface scattering [64] (see also [19, 62]). Geist et al. [42] have revised the LBM to render participating media with only light diffusion. However, all these methods are still very slow.

1.3.2 Volume Rendering Hardware

The expensive computational cost of direct volume rendering makes it difficult for sequential implementations and general-purpose computers to deliver interactive or real-time performance. Motivated by traditional computer graphics which

can be accelerated by OpenGL graphics hardware, researchers have designed several different special-purpose volume rendering hardware architectures to address this challenge.

The underlying algorithm of Cube-4 [122, 123] is modified from the ray-casting algorithm and suitable for a parallel hardware implementation. The volumetric dataset is stored as a 3D regular grid of voxels. The face of the volume memory that is most perpendicular to the major component of the viewing direction is called the base-plane. Consecutive data slices parallel to the base-plane are traversed in scanline order. Beams of two adjacent data slices of voxels are processed simultaneously to compute a new slice of interpolated sample values in between these two slices. The orthogonal voxel neighborhoods between data slices allow for accurate 3D resampling using trilinear interpolation. To approximate the surface normals for shading and classification and avoid accessing voxel values more than once, the interpolated sample values are used to estimate the gradients on each sample position. The interpolated data slices from the trilinear interpolation stage are stored in the so-called ABC buffers. The current buffer stores the samples that are currently being shaded. The ahead and behind buffers store the samples one slice ahead and one slice behind in major viewing direction, respectively. After shading and classification, the compositing of samples onto the base-plane is performed. The distorted intermediate base-plane image is then warped onto the viewing plane to produce the final image.

A second generation VIZARD system, VIZARD II, has been presented by Meißner [100, 101]. It is a reconfigurable volume rendering hardware system for perspective ray casting. The core of the system is the ray processing unit (RPU), which calculates color pixel values using the start position and increment values for a given ray. The VIZARD II PCI card is built using off-the-shelf components such as the PCI interface chip, the SHARC ADSP 21160 DSP, and the Xilinx Virtex FPGA. The system running at 50MHz can render a 256^3 data set at 3-7 frames per second at 256^2 image resolution.

The RACE II engine uses a hybrid volume rendering methodology that combines algorithmic and hardware acceleration to improve ray casting performance [130]. It integrates space leaping with an empty space data-structure called the Transparent Voxel-Block (TVB) table.

In recent years, the demand for high-performance 3D computer graphics, mostly driven by computer game and entertainment industry, has led to powerful graphics processing unit (GPU) installed on almost every consumer PC. Some functions of these cards such as 3D texture are previously only available on expensive graphics workstation. Some functions such as a floating-point fragment program were not even implemented on those workstations. Thus, some volume rendering algorithms can be easily ported to PC platforms and new volume rendering methods were proposed on programmable graphics hardware.

The majority of 3D graphics hardware generate raster images through a fixed sequence of processing stages or pipeline [131, 137]. The input of the pipeline is a stream of geometric primitives described by vertices, which are generated by evaluating polynomial functions for approximating curve and surface geometry. In the geometry processing stage, the geometry engine computes the linear transformations of vertices such as rotation, translation and scaling. Then, local illumination models are evaluated for each vertex. The vertices are grouped to rendering primitives, such as lines and triangles. After clipping, rendering primitives are projected to image plane. In the rasterization stage, the geometric primitives are converted to fragments. Each fragment corresponds to one pixel on the image plane and is assigned with some attributes such as screen coordinates, depth, color, opacity and texture coordinates. Then, the texture coordinates are used to fetch proper texels from the texture maps which are combined to final color and opacity of the fragment. When a fragment is written to the frame buffer, certain tests are performed to determine whether it can be written and the actual value saved in the frame buffer is also modified by the previous content in frame buffer.

To add more features to this standard pipeline, many extensions are integrated in modern graphics accelerators. In the following sections, NVIDIA's Geforce family of cards are discussed as representatives. One of the major improvements in Geforce3 class cards is a flexible mechanism for fragment shading including texture shaders and register combiners [116]. The texture shader extension defines more than 20 pre-defined texture shader programs. In most recent graphics accelerators, the fixed pipeline for rendering only polygons with texture mapping has evolved to a flexible pipeline with programmable vertex, geometry, and fragment stages and therefore is called graphics processing unit (GPU), as shown in Figure

1.5(b). The fragment program stage has been generalized to include floating point computation and a complete, orthogonal instruction set. An application defined fragment program can perform mathematical computations and texture lookups using arbitrary texture coordinates on each fragment.

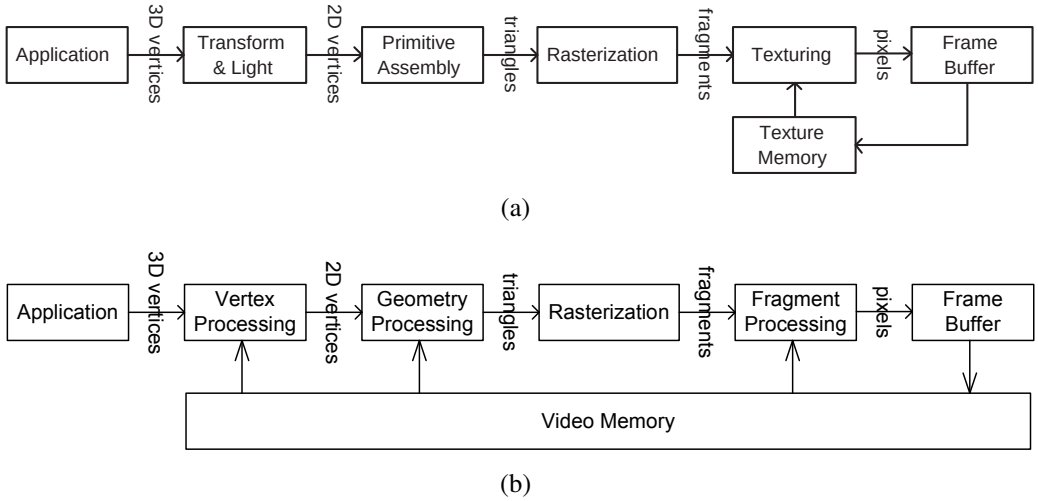


Figure 1.5: (a) Traditional OpenGL hardware pipeline. (b) GPU pipeline.

The GPU can be described with a streaming processing model [126]. Streaming computing differs from traditional computing in that the system reads the data required for a computation as a sequential stream of elements. Each element of a stream is a record of data requiring a similar computation. The system executes a program or kernel on each element of the input stream placing the result on an output stream. In this sense, a programmable graphics processor executes a vertex program on a stream of vertices, a geometry program on a stream of primitives, and a fragment program on a stream of fragments.

Crawfis and Max [25] have exploited texture hardware in traditional OpenGL graphics hardware to accelerate the splatting method. It stores the generic footprint table in a texture map and then uses the texturing hardware to interpolate sampled values from these maps. The compositing hardware is used to update the frame buffer. Three dimensional texture mapping hardware has been recognized as a very efficient acceleration technique for volume rendering, which was introduced in the SGI RealityEngine [1]. The basic idea of this method is to store the

volume as a 3D solid texture on the graphics hardware, then to sample the texture using planes parallel to the image plane and composite into the frame buffer with the blending hardware. Cabral et al. [13] have rendered datasets of 256^3 voxels at interactive frame rates on a four Raster Manager SGI RealityEngine Onyx with a single 150MHz CPU.

With evolving hardware, Rezk-Salama et al. [132] have described a method that implements trilinear interpolation and per-pixel illumination with multi-textures and register combiner extensions. To produce a high quality image, post-classification is required and can be implemented with dependent texture lookups. Pre-integrated classification method [32] calculates the ray integral in the pre-processing step, which is a function of the densities at the two end points of the ray segment. Li et al. [83] have applied empty space skipping and occlusion clipping techniques to accelerate 3D texture based volume rendering.

Kruger et al. [73] have further implemented the whole ray casting algorithm on most recent GPUs in a multi-pass approach. In the first pass, the front face of the volume bounding box is rendered to a 2D RGB texture. 3D texture coordinates of each vertex are issued as per-vertex color COL. The result is a 2D texture (TMP) having the same resolution as the current viewport. The color components in the texture are the coordinates of the first intersection point in texture space. In the second pass, the back faces are rendered and the normalized direction computed from the TMP texture and the position of the intersection point with the back faces is stored in the texture DIR. Then, in each main pass, M steps of ray traversal along the rays are performed, and rendering is directed to a 2D texture RES, which can be accessed in the consecutive passes.

To integrate early ray termination, after each main pass, an additional intermediate pass is executed. In this pass, the fragment program checks the opacity value of each fragment and replaces the depth value if the opacity value is larger than a certain threshold. Thus, in the later main passes, the corresponding fragment is culled by an early depth test and no fragment program is executed for it. To enable empty space skipping, an auxiliary data structure is precomputed, which stores the min/max density in one block containing 8^3 voxels. In the intermediate pass, the fragment program checks the auxiliary data structure at 8 times larger step size. If all blocks possibly accessed in the next main pass are empty, the fragment's depth

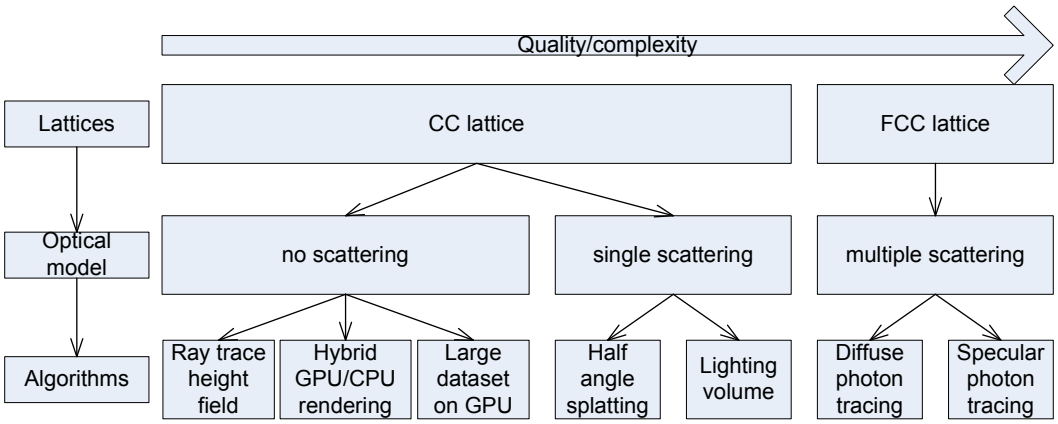


Figure 1.6: Lattice volume rendering techniques presented in this work.

value is replaced.

Hegeman et al. [52] have proposed a two-pass approach for strongly forward scattering with GPU acceleration. Harris and Lastra [51] have used a similar approach to render clouds. Kniss et al. [69] have introduced a volume lighting model for GPU-accelerated volume rendering with forward scattering using a single pass based on half angle slicing. Riley et al. [133] have extended this method to render atmospheric phenomena.

1.4 Contributions

The major contributions of this work to medical and scientific visualization research are various volume rendering techniques and methods based on lattice representation, which improves the speed, quality, and performance of volume rendering with different optical models. As shown in Figure 1.6, the techniques in this work use lattice for volume representation because of the simplicity and efficiency of the lattice structure explained in Section 1.1. Moving from left to right, the algorithms become more complex and the image quality becomes better. The CC lattice is the simplest and most popular lattice for volume representation because most scanning modalities only produce rectilinear grid data and many scientific computations use CC discretization. For rendering CC lattice volumes without scattering, three techniques of GPU acceleration are presented in this work:

- A ray tracing technique for rendering terrain data has been developed with GPU acceleration. An elevation map and a texture map on the 2D CC lattice are used to compactly store the terrain model. Real time rendering speed is achieved on the GPU by projecting the cell boundaries.
- A hybrid CC lattice volume rendering algorithm has been developed with CPU and GPU parallelism. Unlike pure GPU based methods, the hybrid rendering method explores the computation power of both CPU and GPU. The CPU is more flexible and can access complicated data structures in the system memory and the GPU is specifically designed for graphics computation as an SIMD machine. The algorithm uses the CPU for empty space skipping and early ray termination and the GPU for sampling, shading and compositing. Workload balancing is achieved by querying GPU idling status and changing the algorithm complexity on the CPU.
- A GPU based object order ray-casting algorithm has been developed for rendering large CC lattice volume datasets, such as the visible human CT datasets. The basic single pass GPU ray-casting algorithm cannot handle high resolution CC lattice volume beyond graphics card memory capacity. The proposed algorithm organizes the volume in a min-max octree data structure and projects octree nodes that can be stored in graphics memory in front-to-back order. Empty space skipping and early ray termination are employed to improve rendering speed.

For amorphous objects such as clouds and smoke, the self shadow effect is an important visual cue. Two techniques for rendering smoke with single scattering are presented. The smoke data is produced with the computational fluid dynamics (CFD) model, lattice Boltzmann method (LBM), which simulates the flow field on a 3D CC lattice.

- A half angle splatting technique has been developed, which is a multi-pass algorithm that slicing the lattice volume perpendicular to the half angle direction. The half angle direction is half way between the light direction and the view direction (or inverted view direction). The splats are projected twice for light map attenuation and viewing ray integration.
- A lighting volume technique has been accelerated on the GPU, which avoids the popping artifact of the half angle splatting method. An OpenGL based

method has been used to render the LBM simulation results distributed on a GPU cluster. A more efficient method with a CUDA implementation has been proposed for rendering smoke with a single GPU.

To further improve the image quality, global illumination effects such as soft shadows, indirect illumination and color bleeding must be computed, which is extremely slow because the computation of multiple scattering events on a single optical path is time consuming. A novel volumetric global illumination framework based on the FCC lattice is presented. The new method has two passes. In the first pass, photons are emitted from light sources and the photon energy is distributed in the scene, illuminating the media. In the second pass, a ray tracing method is used to generate the final image.

- For volumetric objects where the dominant effect is diffusion, a new algorithm to trace photons on the lattice links has been proposed. The photon direction is discretized to one lattice link direction and the optical events only occur on the lattice sites. The angular discretization greatly simplifies the computation of optical events, photon storage, and radiance estimation. Therefore, the new algorithm achieves 1-2 orders of magnitude faster rendering speed than conventional photon mapping method.
- To mitigate the ray effect caused by the discretization of scattering directions when accurate directions are needed for specularity, an enhanced algorithm, specular photon tracing, has been developed where every photon is associated with its accurate direction. The O-Buffer data structure has been exploited for compact photon storage and efficient photon query in radiance estimation.

Finally, a new computer aided polyp detection pipeline for virtual colonoscopy has been developed. The new pipeline is based on analyzing the electronic biopsy images produced with the volume rendering technique. Compared with the conventional shape based method, the new CAD pipeline reduces the information from 3D to 2D so that the analysis speed is greatly improved. And the shape analysis is only executed on suspicious regions obtained during the image analysis stage for false positive reduction. The resulting system has achieved 100% sensitivity and a comparable false positive rate of the best shape analysis method.

Chapter 2

Volume Rendering CC Lattices without Scattering

In this chapter, several new algorithms will be presented to accelerate the rendering of volumes on the Cartesian Cubic (CC) lattice without scattering. To render a volume without scattering, the most time consuming part is the application of the reconstruction filter (or the interpolation filter). The 3D CC lattice is the most frequently used lattice in practice. The generator matrix of the CC lattice is simply the 3×3 identity matrix. Each site of the CC lattice has 6 nearest (axial) neighbors, 12 secondary (minor diagonal) neighbors and 8 tertiary (major diagonal) neighbors. The 3D CC lattice can be directly stored in a 3D texture on the GPU and trilinear interpolation is natively supported by the texture hardware. Therefore, rendering volumes on 3D CC lattices can be efficiently accelerated on GPUs.

2.1 Ray Tracing Height Fields

Terrain rendering has many important applications, such as flight simulation, battlefield visualization, mission planning, and GIS. Terrain data usually come in the form of two complementing datasets: a color or texture image and an elevation map (i.e., height field). There are two approaches to render terrain. One approach is rasterization. A triangle mesh or other geometric primitives are constructed from the elevation map and can be rendered by commodity graphics hardware. Another

approach is ray tracing. The terrain model for ray tracing can be either simply an elevation map or a true 3D volume representation created from an elevation map. Rasterization is currently the more popular approach. It achieves fast rendering speed and high image quality by using state-of-the-art graphics hardware and various level-of-detail techniques [56, 86].

However, ray tracing has some advantages over rasterization. Terrain models for ray tracing are usually more compact than terrain models for rasterization. A typical terrain model for ray tracing is simply an elevation map, which requires less storage space and consumes less memory than a triangle mesh. The elevation map is defined on a 2D lattice, usually 2D Cartesian lattice. Every site of the elevation map is assigned a height value that can be represented with a floating point number or quantized to be a fixed point number. Instead, a vertex of the triangle mesh requires at least 3 floating point numbers to store the position and sometimes needs another 3 floating point numbers to store the normal information. More importantly, ray tracing provides more flexibility than hardware rendering. For example, ray tracing allows us to operate directly on the image/z-buffer to render special effects such as terrain with underground bunkers, terrain with shadows, and fly-through with fish-eye view. In addition, it is easy to incorporate clouds, haze, flames, and other amorphous phenomena with terrains by ray tracing.

Typically, terrain surfaces are either piecewise height planes [20] which may not provide satisfying image quality, or triangle meshes [111] rendered by conventional ray-triangle intersection algorithms which did not exploit the regularity of height-field data. Paglieroni and Petersen [119] developed a special cone-like volume data structure to accelerate ray traversal for height fields. The terrain surface is computed analytically by bilinear or bicubic interpolation. Lee et al. [76] proposed an efficient ray-casting algorithm by exploiting vertical ray coherence. Cohen-Or et al. [20] presented a comprehensive fly-through system for voxel-based terrain, where the terrain surface is represented by piecewise height planes which may not provide satisfying image quality if the voxel projection size is bigger than the pixel size. Wan et al. [154] used a prefiltered, antialiased true 3D volume terrain representation for high quality terrain rendering.

In this section, a rasterization-tracing hybrid terrain rendering method is presented which has the features of both rasterization and ray tracing. This algorithm

is specially designed for next generation graphics hardware.

2.1.1 Surface Reconstruction

Figure 2.1 shows the framework of the surface reconstruction method. Z_1, Z_2, Z_3, Z_4 are the four grid points and their heights of a cell in the elevation map. For each cell, there are two intersection points between the ray and the cell boundaries: one entry and one exit point. Let P_1 be the entry point and P_2 be the exit point. Figures 2.1(a) and 2.1(b) show two possible situations of a ray passing through a cell. To simplify the presentation, our algorithm is demonstrated for the situation illustrated at Figure 2.1(a). Extension to the situation of Figure 2.1(b) is straightforward. Let Q_1 and Q_2 be the projections of P_1 and P_2 on lines Z_1Z_2 and Z_3Z_4, respectively, along the height direction. Therefore, P_1Q_1 and P_2Q_2 are the offsets of P_1 and P_2, respectively, to the terrain surface along the height direction. If the heights of point P_1 or P_2 are below any heights of Z_1, Z_2, Z_3, and Z_4, the ray possibly hits the terrain surface in this cell. Then, the offsets of P_1 and P_2 are computed.

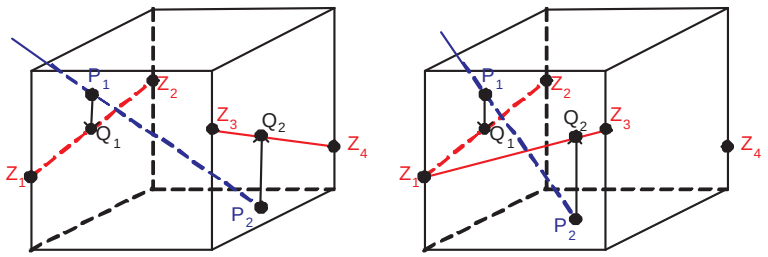


Figure 2.1: Two cases of a ray passing through a cell.

Figure 2.2 shows the method for reconstructing triangles in the cell. There are two triangles in a cell, triangle $Z_1Z_2Z_3$ and triangle $Z_2Z_3Z_4$. To compute the intersection point of the ray and triangle $Z_1Z_2Z_3$, point Z_4 to Z_5 so that Z_1, Z_2, Z_3, Z_5 are coplanar. It is easy to see that $Z_5 = Z_3 + Z_2 - Z_1$. Let C be the central point of line Q_1Q_2. If the intersection point I is inside triangle $Z_1Z_2Z_3$, then

$$\frac{P_1Q_1}{P_2Q_2} = \frac{Q_1I}{Q_2I} \leq \frac{Q_1C}{Q_2C}. \tag{2.1}$$

Project line P_1P_2 onto the base plane of the terrain. Let a be the projection of line Q_1C, b the projection of line Q_2C. Let dy_1 be the projection of Q_1Z_2 and dy_2 the projection of Q_2Z_3. Then,

$$\frac{Q_1C}{Q_2C} = \frac{a}{b} = \frac{dy_1}{dy_2}. \tag{2.2}$$

Therefore, if $P_1Q_1 \times dy_2 \leq P_2Q_2 \times dy_1$, the intersection point is inside the triangle $Z_1Z_2Z_3$, and the real intersection point of the ray and the triangle mesh is found. Otherwise, repeat the process for triangle $Z_2Z_3Z_4$.

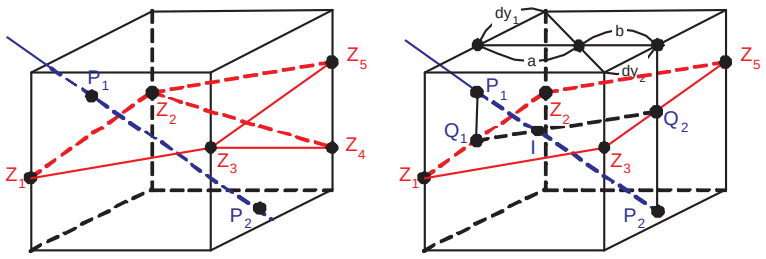


Figure 2.2: Triangle reconstruction. (a) Z_4 is moved to Z_5; (b) test whether the intersection point is inside the triangle $Z_1Z_2Z_3$.

2.1.2 Rasterization-Tracing Hybrid Rendering on GPU

In this sub-section, the method of rendering the triangle mesh with programmable GPU is presented. As shown in Figure 2.2, only the two intersection points, P_1 and P_2, of each ray with cell boundaries need to be computed, where P_2 is the first intersection point which is below the terrain surface and P_1 is the last one which is above the terrain surface. P_1 and P_2 can be computed efficiently using rasterization and fragment processors of the GPU.

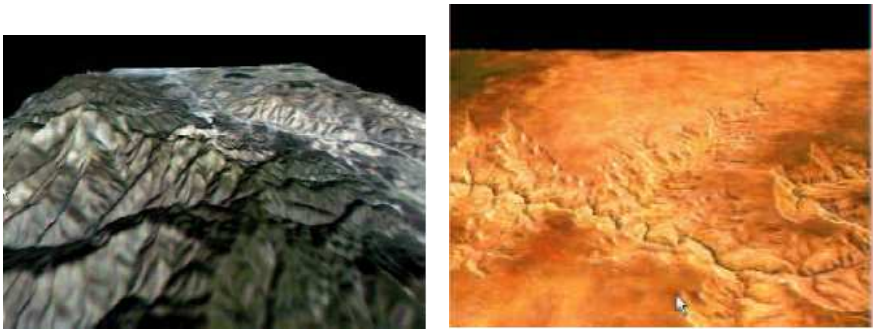
All the cell boundaries can be combined into $4n$ rectangles for an $n \times n$ height field, n rectangles perpendicular to the x axis, n rectangles perpendicular to the y axis, and $2n$ rectangles perpendicular to the $x = y$ plane. When projecting all these rectangles onto the screen, each fragment is an intersection point of the ray and a boundary plane of a cell. Each vertex of the rectangle is assigned a 3D texture coordinate relative to the bounding box of the height field. The r texture coordinate

of fragment P is the distance from the fragment center to the base plane of the height field. In the fragment program, the s and q texture coordinates are used to retrieve the height of Q from the elevation texture, where line PQ is perpendicular to the base plane and Q is on the terrain surface. If P is higher than Q, the fragment is discarded. Otherwise, we find an intersection point which is below the terrain surface. The fragment with the s and q texture coordinates, length of PQ and depth value is written to the floating point pixel buffer without loss of precision. The depth test is used to get the nearest fragment to the view point. Thus, after the first pass, the first sampling point P_2 is obtained which is below the terrain surface. The ray must intersect with the triangle in the cell containing P_2.

In the second pass, all the rectangles are projected again. In the fragment program, the depth value of each fragment is subtracted by the depth value computed at the first pass for this fragment. The depth test guarantees that the intersection point P_1 which is the nearest to P_2 is saved in the buffer. Finally, in the third pass, the fragment program uses the attributes of P_1 and P_2 to interpolate the position of the intersection point and retrieve the color value from the texture.

This algorithm has the following features:

1. It has the features of both rasterization methods and ray tracing methods. Compared with traditional projection methods, it dramatically reduces the number of geometric primitives to be rasterized. In our algorithm, only $O(n)$ rectangles are rendered, while traditional triangle mesh methods need to render $O(n^2)$ triangles. Compared with conventional ray tracing methods, it avoids the expensive ray traversal process.
2. It takes full advantage of the GPU. Except for the rasterization program and texture mapping which have been widely used in practice, the GPU also provides powerful fragment processors which are specially designed for vector operations. Traditional triangle mesh methods do not take advantage of this. This algorithm has the trivial parallel nature of ray tracing methods. Thus, it can take advantage of the newly available fragment programs.



(a) California terrain. (b) Grand canyon terrain.

Figure 2.3: Rendering results of the rasterization-tracing hybrid algorithm.

2.1.3 Results

The rasterization-tracing hybrid rendering algorithm is demonstrated with two real terrains: a 512×512 southern California terrain (as shown in Figure 2.3(a)), and a 4096×2048 Grand Canyon terrain (as shown in Figure 2.3(b)). Each terrain model consists of an elevation map and a corresponding aerial photographic map of the same resolution. The image resolution is 500×400.

The rasterization-tracing hybrid rendering algorithm has been implemented on the software simulator of NVIDIA GeforceFX. The rendering cost of our algorithm includes two parts. One is the cost of rectangle rasterization. The other is the cost of the fragment program. Assume the resolution of the height field is $n \times n$. In our algorithm, only about $4n$ rectangles are rendered. Commodity graphics hardware can render more than 1M rectangles per second. Thus, the cost of the rasterization can be ignored and the total cost is determined by the efficiency of the fragment program and the number of fragments generated. In our implementation, the fragment program in the first pass includes 4 instructions and in the second pass less than 100 instructions. Therefore, the total number of fragment instructions executed in one frame is less than $4f_1 + 100f_2$, where f_1 and f_2 are the number of fragments generated in each pass. With the screen resolution of 640×480, f_2 is approximately 30K. The total number of fragments generated by our algorithm in the first pass with the Grand Canyon terrain is about 10M. Thus, the total number of fragment instructions executed for one frame is 70M. For the GeforceFX card which has 8 fragment processors running at 500MHz, it can execute about 4G fragment

instructions per second. Thus, the hardware is expected to render about 60 frames per second. This is much faster than the CPU version of the algorithm.

2.2 GPU-CPU Hybrid Volume Ray-casting

Many techniques have been exploited for accelerating volume rendering in software, such as empty space skipping and early ray termination [79]. Knittel [70] has described an architecture and implementation that makes extensive use of MMX and streaming SIMD instructions for perspective ray-casting on a PC.

Empty space skipping and early ray termination techniques have also been implemented on graphics hardware. Li and Kaufman [82] have used bounding contours to skip empty and invisible voxels to accelerate 2D texture-based volume rendering. They [83] also used a box growing method to partition the volume into sub-volumes for empty space skipping. The sub-volumes are then culled and clipped against an orthogonal opacity map. The opacity map is efficiently updated by the GPU in every frame. Krüger and Westermann [73] have proposed a method to accelerate volume rendering based on early ray termination and empty space skipping in a GPU-based multi-pass ray-casting approach. Roettger et al. [134] have proposed an adaptive pre-integration method to implement hardware accelerated ray casting, which automatically subsumes the space leaping acceleration techniques. The early ray termination is implemented based on the occlusion query and the early z-test using an intermediate rendering pass. Xue et al. [171] have employed isosurface-aided acceleration techniques for slice-based volume rendering. Stegmaier et al. [144] have presented a framework for the hardware accelerated visualization of volumetric data based on a single-pass ray-casting approach. Their system exhibits very high flexibility and allows for an easy integration of non-trivial volume rendering techniques. However, these GPU-based methods do not use any computational power of the CPU.

CPU-based direct volume rendering methods are not suitable to generate high-quality images in real-time due to the lack of parallelism and hardware support for trilinear interpolation and local illumination. The GPU-based methods can provide comparable image quality with high rendering frame rates, however they are not flexible enough to be used in different applications, due to the limit of graphics

hardware. These pure CPU-based and pure GPU-based direct volume rendering methods have their weaknesses. Westermann and Sevenich [164] have proposed to combine the processing power of the CPU and the GPU to accelerate volume ray-casting. They have computed the ray entry points and exiting points using texture mapping, and the results are read back from the GPU. Then, the ray traversal is performed in software. The main drawback of this method is that on current graphics hardware the performance of read back is poor and the GPU pipeline is stalled by data transferring. Thus, CPU and GPU cannot work in parallel in this method.

The GPU-CPU hybrid volumetric ray-casting algorithm presented in this section is different from Westermann and Sevenich's method [164]. It uses the CPU to compute the ray entry points and determine when a ray is terminated. The GPU is used for ray traversal. In this way, the power of the CPU and the GPU are fully exploited, meanwhile the CPU and the GPU can work in parallel. Compared with other pure GPU-based methods, the main difference is that our algorithm uses the CPU to do empty space skipping, which is overlapped with the other computation on the GPU.

2.2.1 Algorithm Overview

The CPU programming models are generally serial and do not adequately expose data parallelism in their applications. The recent CPUs allow some data parallel execution, but the degree of parallelism exploited by the CPU is much less than that of the GPU. The CPU targets general-purpose programs. Therefore, they do not contain specialized hardware for particular functions, such as trilinear interpolation. The GPU, however, implements special-purpose hardware for particular tasks, which is far more efficient than a general-purpose programmable solution could ever provide. Consequently, the GPU is more suitable for tri-linear interpolation and local illumination computation. The CPU memory system is optimized for minimum latency, and it contains several levels of cache memory to minimize this latency. Moreover, a large fraction of the CPU's transistors and wires are used to implement complex control functionalities such as branch prediction and out-of-order execution. Therefore, the CPU is more suitable to implement some complex and elaborate algorithms.

Our algorithm fully exploits advantages of both the CPU and the GPU and makes them work in parallel to accelerate the volumetric ray-casting. The basic idea of the GPU-CPU hybrid volumetric ray-casting method is straightforward. It uses the GPU to do streamed trilinear interpolation, local illumination and compositing, and uses the CPU to do ray traversal and maintain elaborate data structures. For example, empty space leaping and early ray termination are done on the CPU side, because additional special data structures can make these techniques more efficient.

Various pre-computed data structures have been proposed in software to rapidly traverse or skip over the empty voxels that have no contribution to the rendered image, such as octree [74, 79], K-d tree [146], bounding convex polyhedrons [5], and proximity clouds [17]. The min-max octree structure [74] allows changing the classification interactively, hence is used here. One leaf cell of the min-max octee corresponds to a cubical region with one voxel on each of its eight corners. For each cell of the min-max octree, the minimum and maximum density values of the voxels belonging to the node are stored. Given a transfer function, a cell can be efficiently classified as *empty*, *opaque*, or *translucent*. A leaf cell is empty, if its eight voxels have zero opacity values. A leaf cell is opaque if the opacity values of its eight voxels are greater than some threshold, such as 0.99 in the current implementation. Otherwise, the cell is classified as translucent. For a non-leaf cell, if its children cells are all empty, it is classified as an empty cell. If its children cells are all non-empty, it is classified as an object cell. Otherwise, it is classified as a partial cell.

In order to take the advantage of the parallelism between the CPU and the GPU, the rays are grouped into small tiles [72]. Each tile of rays corresponds to a square region on the image plane. Our algorithm is applied on each tile in sequence. Given a view point and a volumetric object enclosed by a bounding box, the algorithm of constructing the tile structure is described as follows:

1. A tile T_0 corresponding to the whole image plane is initialized and put into the queue Q.
2. If Q is empty, the algorithm is terminated.
3. Pop a tile T_i from the head of Q.
4. Cast four rays at the corner of the tile T_i to test whether they intersect with the bounding box B.

1. If all of four rays do not intersect the bounding box, and the projection of the bounding box is outside the tile, goto Step 2.
2. If all of four rays intersect the bounding box and the size of T_i is greater than 64×64, the tile is subdivided into four smaller tiles.
3. If only one or two rays intersect with the bounding box and the size of T_i is greater than 16×16, the tile is subdivided into four smaller tiles which are then pushed into Q.
4. Otherwise, the tile is put into a list and goto Step 2.

As a result, a list of tiles are generated with a size varied from 64×64 to 16×16. Moreover, most of the rays within the tiles intersect the bounding box. The tile whose four corner rays all intersect with the bounding box is marked as *full*, which will be processed first in the following steps. If the camera is located inside the object, the rays are simply grouped into tiles with the size of 64×64.

Early ray termination is efficient only when the transfer function is opaque, in which most of the rays will be terminated before leaving the volume. Thus, if the transfer function is semi-transparent, the early ray termination is not employed, which makes the algorithm even simpler.

The flowchart of the GPU-CPU hybrid volumetric ray-casting algorithm is shown in Figure 2.4. After the tile construction, a ray determination step is executed on the CPU to compute ray entry points and normalized ray directions. In this step, space leaping is employed to make the ray entry points close to the object boundary. Then, a multi-slab rendering algorithm shown in yellow in Figure 2.4 is applied. At last, the holes are filled in the last step. For a semi-transparent transfer function, the early ray termination is not employed, which makes the ray termination step a little different from that for an opaque transfer function. The tile construction, ray determination, and ray termination algorithms are executed on the CPU. The slab rendering and hole filling steps are mainly executed on the GPU. In these two steps, the CPU is only used to issue some non-block OpenGL commands. The detail of the ray determination, multi-slab rendering, and hole filling algorithms are described in the following sections.

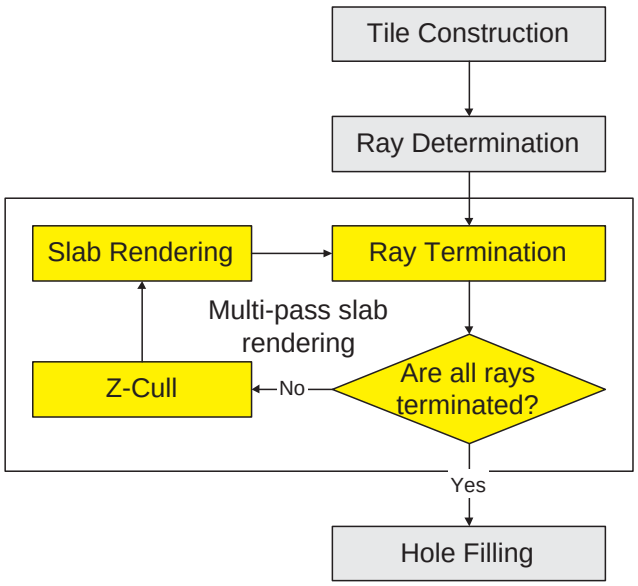


Figure 2.4: The flowchart of our GPU-CPU hybrid volumetric ray-casting algorithm.

2.2.2 Ray Determination

After the tile construction, the ray entry point and the normalized ray direction for each ray of the tile are calculated. Although the GPU is more suitable and efficient for this kind of linear computation [164], our method still uses the CPU to do these computations to avoid transferring data back from the GPU to the CPU, because the ray directions are needed to do ray traversal on the CPU. The readback from the GPU is very slow on the current graphics hardware. For an image with the resolution of 512×512, 6MB of data needs to be read back from the GPU, which is even slower than directly computing them on the CPU using SSE instruction sets. Moreover, the readback operation will stall the graphics pipeline. Thus, the GPU is fed from the CPU with a stream of rays stored in two 2D floating point textures: one for ray entry points and the other for normalized ray directions. This means that the GPU is working as a coprocessor of the CPU.

First, the position of each pixel on the image plane is calculated, along with the normalized ray direction. The main problem is the normalization of the ray direction, which requires many instructions to compute the distance between the camera and each pixel position. The good thing is that if the distance from the

camera to the image plane and the angle of the field of view are both fixed, the distance between the camera and each pixel on the image plane is also fixed for every frame. Thus, the reciprocals of distances between the camera and pixels on the image plane are pre-computed and used to scale the ray direction to obtain the normalized ray direction.

Second, the first intersection point of each ray with the object boundary is computed. Using the min-max octree structure, this computation can be done very efficiently. Instead of computing the first intersection point of each ray with the real object boundary, the intersection point of each ray with the cell nodes of the min-max octree is first computed. The parameter l_{min} is used to control the minimal level of cells that the ray can reach, which means that when the ray hits a non-empty cell with level equal to l_{min}, the ray traversal is stopped. Whether moving these points to the exact object boundary is based on the workload of the CPU.

Lakare and Kaufman [75] have proposed a method to exploit the ray coherence to accelerate ray-casting using *detector rays*. It is observed that a group of rays usually traverse the same distance before intersecting the object boundary. To compute the first intersection points for a tile of rays, a group of detector rays are cast first. For each detector ray, it returns the distance from the view point to the first intersection point. The distance information is then used to conservatively estimate the depth values for the neighboring rays. The detector rays are cast interspersedly from the pixels on the image plane.

This method can be easily incorporated with the min-max octree. The traversal of the detector rays can be accelerated with the min-max octree. After the depth values are returned by the detector rays, the algorithm for spreading the depth values for other rays is as follows. For each detector ray, its depth value is propagated to its eight neighboring rays. The depth values for the non-detector rays are initialized with a very large value. While propagating the depth value, it is compared with the depth value of the neighboring ray, and the smaller of the two values is kept. After the depth values are obtained, the first intersection points are estimated.

This method combined with the min-max octree only achieves an efficient conservative estimation for the first intersection points. Whether applying an accurate empty space leaping after this step or not depends on the workload of the CPU and the GPU.

2.2.3 Multi-pass Slab Rendering and Hole Filling

After the ray determination step, all of the ray entry points and normalized ray directions are computed and stored in two 2D floating point textures. After these two textures are uploaded to the GPU, the ray integration can be done by the GPU. However, it is not known how far each ray of the tile will travel before being terminated or leaving the volume at this step. The simple and efficient solution is using the GPU to do multi-pass slab rendering, and using the CPU to decide when the multi-pass slab rendering should be terminated on the GPU in the meantime.

In order to make the multi-pass slab rendering more efficient, the rays within each tile are further divided into quads. Each ray quad consists of 2×2 rays. It is observed that if one of the four rays is terminated, the other rays are likely terminated in the following slab rendering pass. Consequently, the ray quad is the basic unit in the multi-pass slab rendering step.

After the empty space leaping in the ray determination step, some rays may already leave the volume. For each quad of the tile, it is checked whether its four rays have already left the volume or not. If so, the quad is marked as *terminated*. If only some part of the four rays leave the volume, the quad is marked as a *hole*. Otherwise, the quad is marked as *non-terminated*. If a tile does not contain any terminated quads and hole quads, the tile is called a *full tile*. Otherwise, it is called a *partial tile*. The multi-pass slab rendering algorithm is applied on the full tiles until all full tiles become partial tiles. This is done by employing slab rendering on the GPU, and in the meantime the ray termination is performed on the CPU. The ray termination and slab rendering algorithms are the same for full tiles and partial tiles. The only difference is that the early Z-cull step is skipped for the full tiles, because it does not need to modify the depth values to cull the terminated quads for the full tiles in the slab rendering.

2.2.3.1 Early Z-Cull

On current graphics hardware, before a fragment reaches the fragment processor, the z-cull unit is used to compare the fragment's depth with a corresponding value that already exists in the depth buffer. If the fragment's depth is greater, the

fragment will not be visible, and the fragment program is not executed by the fragment processor. The depth buffer is initialized to one at the beginning. In the ray termination step, it will generate two lists containing the terminated quads and hole quads respectively. Because the depth buffer is modified, the three color channels are masked in this step. For the terminated quad, its corresponding quad is rendered with a depth value of zero. As a result, the depth values corresponding to the terminated rays are all set to zero. And, the corresponding fragments will be culled before they reach the fragment processors in the following slab rendering passes. The hole quad is rendered with depth value of 0.4, because the hole quads still need to be rendered to trigger fragment programs in the hole filling step. By this technique a quad with the depth value smaller than 0.4 can be rendered to trigger hole filling fragment programs for the hole quad in the final step. After the depth values are modified to enable early z-cull, the quads corresponding to the partial tiles are rendered to trigger the slab rendering fragment program.

2.2.3.2 Slab Rendering

When the fragments pass the early z-cull test and reach the fragment processor, the fragment program is executed on the GPU. In the fragment program, N uniformly sampled points along the rays of sight are processed. At each sampling point, trilinear interpolation is used to obtain the density value and gradient, and then post-classification is used to obtain the color for the sampling point. The gradient is computed for each voxel with central difference and uploaded to the GPU along with its density values using a 3D RGBA texture. Two lights are used to calculate the local illumination at each sampling point. One light is put at the camera position, the other is from the opposite direction of view up.

In one slab rendering pass, some rays may saturate their opacity values or leave the volume, which should be terminated, while the others still need to be processed in the following slab rendering pass. After the CPU issues the OpenGL commands to render the quads to trigger the fragment program, it starts to detect which non-terminated quad becomes terminated, and which non-terminated quad becomes a hole quad after the slab rendering on the CPU. The slab rendering fragment program and ray termination are performed in parallel on the GPU and the CPU respectively.

2.2.3.3 Ray Termination

In order to accurately determine where a ray should be terminated on the GPU, the ray also needs to be uniformly sampled on the CPU using the same sampling distance as that on the GPU. At each sampling point, the density value should also be tri-linear interpolated. The opacity value is then queried through the same opacity transfer function and accumulated along the ray. When the accumulated opacity value exceeds the predefined threshold, the ray should be terminated on the GPU. It is obvious that the CPU and the GPU do some overlapping work in this method, which is inefficient. Moreover, performing the tri-linear interpolation on the CPU causes a loss in performance.

As mentioned before, each leaf cell of the min-max octree is defined as the cubical region with voxels on its eight corners. Given a transfer function, it is classified as opaque, when the density values of its eight voxels are all greater than 0.99. The following equation is used to do front-to-back compositing on the GPU:

$$\alpha_{dst} = \alpha_{src}(1-\alpha_{dst}) + \alpha_{dst} = \alpha_{src} + \alpha_{dst}(1-\alpha_{src}) \tag{2.3}$$

It is noted that the accumulated opacity value is greater than the source opacity value. When a ray passes through an opaque cell, the sampling point in this cell has an opacity value greater than 0.99, so does the accumulated opacity value. Thereby, a ray should be terminated if it passes through an opaque cell. In this method, the time consuming trilinear interpolation is avoided. The main task of this method is to compute the cells that are pierced by the ray, which can be efficiently obtained using a 3D digital differential analyzer (3DDDA) [2]. Given two endpoints of the ray, this algorithm generates a 6-connected line, which includes all of the cells pierced by the ray.

For each non-terminated quad, our method checks whether the quad contains any ray that will be terminated after the corresponding slab rendering pass. If all four rays of the quad should be terminated after this slab rendering pass, it is marked as terminated and put into a list storing the new generated terminated quads. If all four rays of the quad are not terminated after this slab rendering pass, it is unchanged. Otherwise, the quad is marked as a hole and put into a list storing the new generated hole quads. The two lists will be used in the early Z-Cull step to modify the corresponding depth values.

For each partial tile, if the number of the non-terminated quads is less than a predefined threshold, the whole tile of rays is terminated, and the tile is removed from the partial tile list. This threshold value can also be used to control the balance between the CPU and the GPU. If the partial tile list is empty, the multi-pass slab rendering algorithm is terminated.

For semi-transparent transfer functions, the early ray termination is not as efficient as for opaque transfer functions. Consequently, it is unnecessary to test whether a ray pierces an opaque cell, because the volume only contains very few opaque cells, or it may not contain any opaque cells. To make this checking more efficient, the length for each ray is computed in the ray determination step. This value is subtracted by $0.5N$ for each slab rendering pass, where 0.5 is the sampling distance. If this value is less than $0.5N$ after the current slab rendering pass, the corresponding ray is terminated.

2.2.3.4 Hole Filling

After the multi-pass slab rendering, the hole quads still need to be processed. For each non-terminated ray within the hole quads, the point where the ray leaves the volume is computed. Then, the space leaping is employed from both ends of the ray based on the workload of the CPU and the GPU. And the length of the ray segments are stored in a 2D texture and uploaded to the GPU.

Before the hole filling fragment program on the GPU, the depth values of the terminated rays within the hole quads are modified to cull the corresponding fragments. Then a bounding box enclosing all hole quads is computed and rendered with a depth value of 0.2 to trigger the hole filling fragment program. The hole filling fragment program is very similar to the slab rendering fragment program. The only difference is that the hole filling fragment program has different travel steps based on the length of the corresponding ray segment.

2.2.4 Dynamic Workload Balancing

The workload balance of the CPU and the GPU is crucial to the GPU-CPU hybrid algorithm. The ideal situation is that the programs running on the CPU and the GPU take almost the same time for rendering one frame. In our method,

NVIDIA's performance toolkit is used to access the *gpu_idle* counter to determine if the GPU is underload. The *gpu_idle* counter contains the percentage of time the GPU is idle since the last call. If the *gpu_idle* counter is greater than zero, the workload of the CPU should be reduced, and some work is passed to the GPU. On the other hand, if the GPU is always busy, some work needs to be passed to the CPU. The basic idea to balance the workload of the CPU and the GPU is controlling the degree of the empty space skipping on the CPU. The more empty voxels are skipped, the less work needs to be done by the GPU.

The ways to adjust the workload of the CPU and the GPU are described as follows:

1. Ray traversal in the min-max octree: To reduce the workload on the CPU, stop the rays at high level partial cell before reaching the object cells, where the first intersection points are computed.
2. Computing the first intersection point: when computing the intersection point between the ray and the cell, the ray does not go inside the cell. To increase the workload on the CPU, let the ray move into the cell and reach the real object boundary.
3. Computing the existing point and employing empty space skipping from the existing point in the reverse direction: this way increases the workload on the CPU and efficiently decreases the workload on the GPU.
4. For a partial tile, if the number of the non-terminated quads is less than a threshold, the whole tile rays are terminated. A larger threshold can be used if the workload of the GPU need to be reduced. A smaller threshold results in the tiles being terminated quickly on the CPU. Therefore, the workload of the GPU is increased.

2.2.5 Implementation and Results

All images shown in this subsection have a resolution of 512×512. The sampling distance is 0.5, which is good enough to generate high quality images for all tested data sets. Most of the experiments have been conducted on a 3.0GHz Intel Pentium IV PC, with 1G RAM and a NVIDIA Quadro FX 3400 graphics card (PC1). Another 2.4GHz Intel Pentium IV PC, with 1G RAM and a NVIDIA

Geforce 6800 Ultra graphics card (PC2) is used to demonstrate workload balance. Both PCs are running the Windows XP operating system.

(a) Opaque Engine (b) Semi-transparent Engine (c) Opaque foot (d) Semi-transparent foot

Figure 2.5: Volume rendering of the engine and human foot data sets.

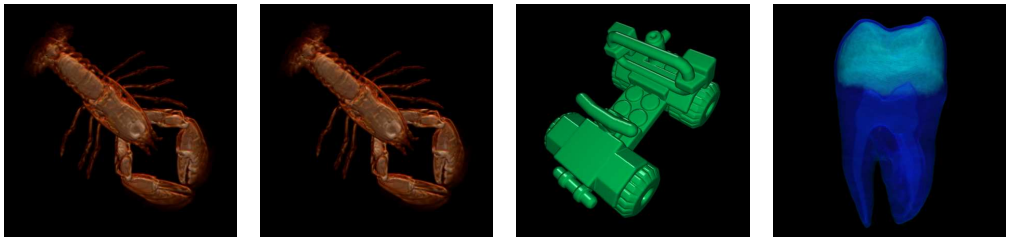
The two data sets in Figure 2.5 are rendered both with an opaque transfer function ((a) and (c)) and a semi-transparent transfer function ((b) and (d)) on both PC1 and PC2. Table 2.1 lists the rendering time in frames per second (fps). The CPU on PC1 is faster than that on PC2. Thus, accurate empty space skipping is employed on PC1 and coarse empty space skipping is employed on PC2. Because the GPU on PC2 is faster than that on PC1, similar performance has been achieved on both PC1 and PC2. The performance on PC1 is a little better than that on PC2, because the PC1 uses PCI Express which is faster than AGP8 used by PC2, and the CPU on PC1 is also faster than that on PC2. When the semi-transparent transfer function is applied on PC1, the performance drops a little, because the early ray termination is not efficient at this situation, while the performance on the PC1 for the two cases is nearly the same. Because the 3DDDA algorithm is not performed on the CPU when the semi-transparent transfer function is applied.

Table 2.1: Average rendering speed for the engine and human foot data sets.

Data Set	Size	Opaque		Transparent	
		PC1	PC2	PC1	PC2
Engine	$256 \times 256 \times 128$	21.9	18.9	17.8	18.0
Foot	$152 \times 256 \times 220$	19.8	16.5	16.3	13.6

NVIDIA Geforce 6 series cards support dynamic branching in the fragment

program, which makes it possible to implement a single-pass GPU-based volumetric ray-casting algorithm. The lego car, lobster and human tooth data sets are used to compare the GPU-CPU hybrid algorithm with the pure GPU-based ray-casting algorithm. In Figures 2.6(a) and 2.6(b), two semi-transparent lobsters are rendered with the two different methods from the same view point, respectively. The difference between the two images can not be observed. For the lego car and human tooth data sets, the resulting images by the GPU-CPU hybrid method are shown in Figures 2.6(c) and 2.6(d). The performance of the GPU-CPU hybrid volumetric ray-casting algorithm (HRC) and pure GPU-based volumetric ray-casting algorithm (GRC) is listed in Table 2.2 in frames per second (fps), which shows that the GPU-CPU hybrid algorithm is faster than the pure GPU-based ray-casting algorithm. The experiments are conducted on PC1.



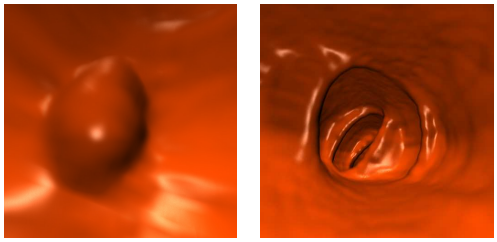
(a) Lobster (HRC) (b) Lobster (GRC) (c) Lego Car (d) Human Tooth

Figure 2.6: Volume rendering of the lobster, lego car, and tooth data sets.

Table 2.2: Average rendering speed for the lobster, lego car, and tooth data sets.

Data Set	Size	HRC	GRC	Speedup
Lego Car	$256 \times 256 \times 128$	19.1	16.5	15.8%
Lobster	$152 \times 256 \times 220$	28.3	20.4	39.1%
Tooth	$128 \times 128 \times 256$	32.8	16.9	94.1%

Virtual colonoscopy uses a computer visualization system to virtually navigate within a colon model. The volume rendering method can display more details than the surface-based rendering method when the camera is located close to the colon surface. Moreover, the volume rendering method does not need to extract a surface model in pre-processing. The GPU-CPU hybrid volumetric ray-casting is efficient



(a) 20.4fps (b) 18.3fps

Figure 2.7: (a) A close up view of a polyp; (b) A view of the colon from a camera parallel to the centerline.

for the virtual colonoscopy applications, because the colon surface is rendered with an opaque transfer function. The data sets used to test the GPU-CPU hybrid approach for the virtual colonscopy system are all real patients' CT data sets. The size of the CT data sets is usually 512^3. It is impossible to pre-compute the gradient and store them on the GPU memory. Thus, the gradient is estimated on-the-fly at each sampling position with central difference. Figure 2.7 shows two different views inside a human colon. Figure 2.7(a) rendered at 20.4 fps is a close up view of a polyp, in which the camera is very close to the colon wall. The viewing direction for Figure 2.7(b) is nearly parallel to the centerline, which is rendered at 18.3 fps. Because the camera for the left image is much closer to the colon surface than that for right image, the ray determination time for Figure 2.7(a) is shorter than that for Figure 2.7(b). The number of hole pixels for Figure 2.7(a) is $18,369$ (7.0%), and the number for Figure 2.7(b) is $23,913$ (9.1%). Because there are fewer hole pixels in Figure 2.7(a) than those in Figure 2.7(b), the rendering time of Figure 2.7(a) is a little less than that of Figure 2.7(b).

For the virtual colonoscopy system, the camera moves along a pre-defined centerline. All camera parameters of the fly-through navigation can be pre-computed from the centerline. The GPU-CPU hybrid volumetric ray-casting algorithm can benefit from these pre-computed camera parameters. After the CPU renders the list of quads to issue the hole filling fragment program, it can do ray determination for the next frame. Thus, the GPU-CPU hybrid method can fully take the advantage of the parallelism between the CPU and the GPU to achieve a real-time rendering speed with high quality for virtual colonoscopy.

2.3 Ray-casting Large Datasets with GPUs

In this section, a GPU-based object-order ray-casting algorithm [54] is presented for the rendering of large volumetric datasets, such as the Visible Human CT datasets. High resolution CT data is highly demanded by many current medical applications. The typical size of contemporary clinical 16bit CT data is about 256MB (512^3 voxels). The photographic volumetric datasets have color information, which are usually larger than the CT and MRI datasets of the same resolution. Moreover, the size of datasets will likely keep increasing at a high rate due to the advance of scientific devices. The rendering of large volumetric datasets is a classical problem in visualization.

Volumetric datasets used in a variety of fields usually contain many regions that are classified as transparent or empty. Object-order approaches are well-suited for skipping empty regions. However, the hidden volume removal is inefficient compared with the ray-casting method. Mora et al. [103] proposed a CPU-based object-order ray-casting algorithm to take the advantages of both image-order and object-order approaches for orthogonal projection.

In the algorithm presented in this section, a volumetric dataset is decomposed into small sub-volumes called *cells*, which are organized using a min-max octree structure. The cells are classified as empty cells or non-empty cells. The non-empty cells are loaded into video memory or AGP memory, as many as possible. Then, the cells are projected from front to back and composited using the GPU. In order to make the cell projection more efficient, a propagation method is investigated to sort the cells into layers such that all cells in one layer can be projected simultaneously. Weiskopf et al. [162] have proposed to split the volume into bricks to achieve constant frame rates in 3D texture based volume rendering. Parker et al. [120] have used the similar idea of volume bricks to render isosurface in large volume datasets on the CPU. In this section, the presented method renders large volume datasets that are beyond GPU memory capacity with ray-casting method. A cell grouping method is used to reduce the OpenGL context switches for early ray termination.

2.3.1 Algorithm Overview

A cell is a cubical region of a sub-volume containing $N \times N \times N$ voxels. A cell is classified as empty, if all voxels of the cell are invisible based on the transfer function. Otherwise, it is classified as non-empty. The min-max ocree [167] is used to organize the cells for efficient classification. Each leaf node of the min-max octree contains a cell, as well as the minimum and maximum density values of the cell. Each interior node only contains the minimum and maximum density values found in that node's subtree.

The non-empty cells are projected onto the image plane in a front-to-back order. A fragment program is used to do ray integration for each projected cell on-the-fly, in which a volumetric ray-casting algorithm is performed. Each cell is stored in a 3D texture. Since the volumetric ray-casting algorithm requires a neighborhood of voxels for proper interpolations and gradient calculations, the neighboring voxels of the cell need to be stored in the 3D texture. Thus, for each cell the resolution of the corresponding 3D texture is $(N+2) \times (N+2) \times (N+2)$.

Although the cells can be hierarchically sorted using the min-max octree structure, a more efficient propagation algorithm is used to sort cells. The cells are front-to-back sorted and grouped into layers. The cells within the same layer can be projected simultaneously, which dramatically improves the performance of the cell projection algorithm on the GPU. The cell sorting and projection algorithms take the advantage of the parallelism between the CPU and the GPU. When a layer of cells is determined, they can be projected immediately to trigger fragment programs to be executed on the GPU. The CPU then can be used to generate the next layer of cells.

Although a large number of cells are classified as empty cells, which do not need to be uploaded to the GPU, the 3D textures corresponding to the non-empty cells are still too large to be fitted in video memory. Some non-empty cells need to be transferred to video memory on-the-fly. The OpenGL extension pixel_buffer_object (PBO) defines an interface to using buffer objects for pixel data, which dramatically improves the texture uploading performance. By using this extension, the GPU can asynchronously pull the data from the AGP memory using DMA (Direct Memory Access). Therefore, a three-level structure is used to store the cell data in video memory, AGP memory, and system memory as shown in

Figure 2.8. Suppose that M 3D textures can be allocated in video memory and N buffers of the same size can fit in AGP memory, and the first 20 buffers are used as a memory pool for transferring data on-the-fly. First M random non-empty cells are uploaded to video memory. Then, the other $M-20$ non-empty cells are copied into AGP buffers. The rest of non-empty cells are still resident in system memory. For each cell, a flag is used to indicate whether its corresponding data is resident in video memory, AGP memory, or system memory. Thus, the size of the dataset that can be rendered by the algorithm is only limited by the size of the system memory.

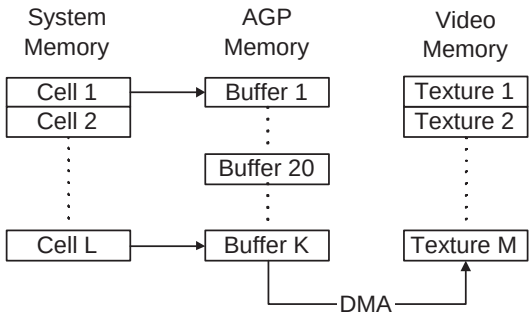


Figure 2.8: The three-layer structure used to store the cell data.

The overview of our algorithm is shown in Figure 2.9. The min-max octree construction, classification, and texture loading are performed in the pre-processing step, which is view independent. The cell sorting algorithm organizes cells into layers. When a layer of cells are generated, it first checks whether all non-empty cells reside in video memory. If any non-empty cell within the layer is not resident in video memory, it is uploaded on-the-fly. Before uploading, it must be determined which 3D texture object is used to store the data, and the current data stored in that 3D texture is replaced. A replacement queue is used to hold the cells that are already projected and can be switched out. When a layer of cells is sent to the GPU, it cannot be put into the replacement queue immediately, because the fragment program executed on the GPU might not have finished. The NVIDIA OpenGL extension NV_fence is used to determine whether the cell projection of a layer of cells is finished on the GPU. This extension introduces the concept of a ”fence” to the OpenGL command stream. Once the fence is inserted into the command stream,

it can be queried whether it is finished. After all OpenGL commands for cell projection of the layer of cells are issued, a fence is appended to the commands. Then, the state of the fence is queried after every layer of cells is projected. If the fence is completed, the cells before the fence are inserted into the replacement queue, and a new fence is appended to the OpenGL commands stream. In case the replacement queue is empty, a random 3D texture is chosen, of which the corresponding cell has not been projected.

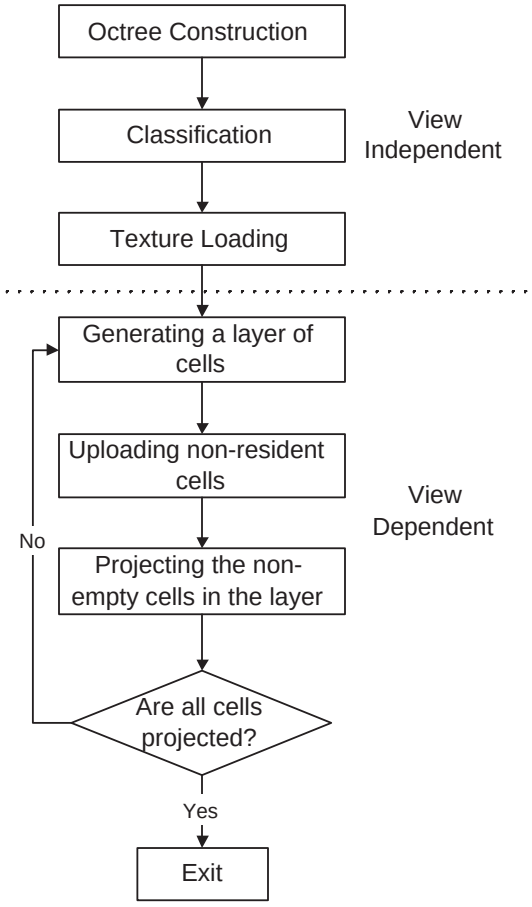


Figure 2.9: Overview of GPU-based object-order ray-casting algorithm.

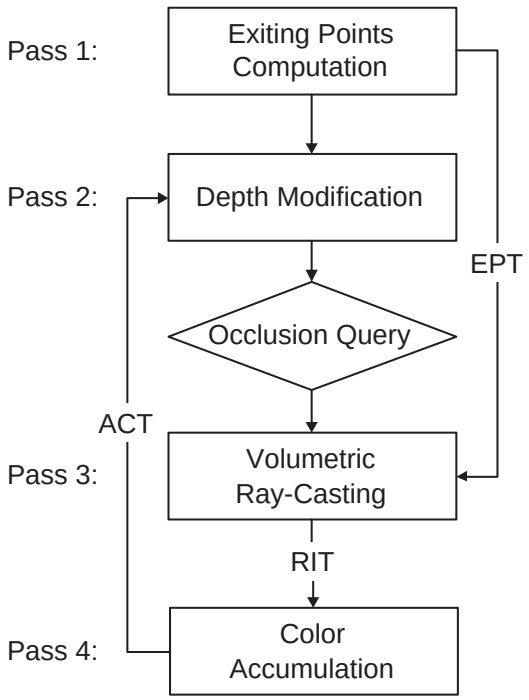


Figure 2.10: Cell projection pipeline.

2.3.2 Cell Projection

When orthogonal projection is used, every cell projection on the image plane is given by the same hexagon shape per viewing direction. This projection can be computed once, and then used as a template for all cells, which can be obtained by translation. The rays intersecting with the cell are then determined by the cell projection efficiently. However, when perspective projection is applied, the situation becomes more complicated. The cell projections on the image plane are different, and the pre-computed template can not be used any more, which make the CPU-based object-order ray-casting algorithm inefficient. The good thing is that the cell projection can be efficiently implemented on the recent graphics card even when perspective projection is used, which makes it possible to implement a fast object-order ray-casting algorithm on the GPU.

The cell projection algorithm is implemented using fragment programs running on the GPU. When a cell is rendered, a number of fragments are generated, which corresponds to the rays intersecting with that cell. For every non-empty

cell that has to be projected, the rendering pipeline is shown in Figure 2.10. The algorithm consists of four rendering passes for each cell. The modelview matrix and projection matrix remain unchanged for all four rendering passes. Hence, the fragments generated at the same window position in the four rendering passes correspond to the same ray intersecting with the cell.

OpenGL provides pixel buffers (pbuffer for short) for off-screen rendering. Combined with the render_texture extension, it allows the color buffer of the pbuffer to be used for both rendering and texturing. Three pbuffers are used as rendering targets for different render passes in the algorithm. The first pbuffer, the rendering target of the first rendering pass, is used to store the exiting points of the rays that intersect with the projected cell. It is also bound to a 2D RGB floating point texture, named *exiting points texture* (EPT), which is accessed in the third rendering pass to compute the length of each ray segment and normalized ray direction. The second pbuffer is made up of a depth buffer and a color buffer, which are the rendering targets of the second and third rendering pass, respectively. The depth buffer is used to implement early ray termination with the early-z test technique [73]. This optimization is valid only when the fragment program does not modify the fragment depth. Thus, a separate rendering pass is used to modify the depth values based on the opacity values. The color buffer is used to store the result of the ray integration, which is bound to a 2D RGBA floating point texture, named *ray integration texture* (RIT) and accessed in the last rendering pass. The third pbuffer is the rendering target of the last rendering pass, which is used to accumulate the color values. It is bound to a 2D RGBA floating point texture in the second rendering pass, which is named *color accumulation texture* (CAT). Its opacity values are accessed in the second rendering pass to modify the depth values accordingly for culling the fragments whose corresponding rays have already saturated their opacity values. The cell projection algorithm is described as follows:

- **Pass 1** (Exiting Points Computation): In the first rendering pass, the exiting points for the rays intersecting with the projected cell are computed by only rendering the back faces of the cell. For each vertex of the cell, its texture coordinates in the corresponding 3D texture space are assigned as its primary color. The fragment program is straightforward, which just passes the fragment's primary color as output. In the rasterization stage, the texture

coordinates for each fragment are interpolated, which are the coordinates for the exiting point of the ray in the texture space.

- **Pass 2** (Early Ray Termination): The opacity value of the fragment is accessed through the CAT. For any fragment whose opacity value exceeds 0.99, the depth value is set to one. As a consequence, if the depth test is set to GREATER, the corresponding fragment in the third rendering pass is discard.
- **Pass 3** (Volumetric Ray-Casting): The front faces of the cell are rendered to compute the entry points for the rays using the same method as Pass 1. In the fragment program, the exiting point is obtained through accessing the exiting point texture (EPT). The normalized ray direction and length of ray segment are computed in the 3D texture space. The ray is then evenly sampled with a sampling distance 0.5 to do ray integration. It is impossible to pre-compute the gradient information and store them on the GPU for large datasets. Thus, the gradient on each sampling point is calculated on the fly. The densities at six neighboring positions are fetched to estimate the gradient with central difference.
- **Pass 4** (Color Accumulation): The front faces of the cell are rendered again to generate corresponding fragments. In the fragment program, the color value and opacity value of the projected cell are accessed through ray integration texture (RIT), and returned as color output directly. OpenGL blending is enabled in this rendering pass for accumulating the color and opacity values.

When all non-empty cells are projected, the CAT holds the final image. In fact, the rendering Pass 2 is not need to be executed for every layer. In the current implementation, it is enabled every other two layers.

Because the rendering context is switched three times during the cell projection, this may cause a significant loss in performance on current GPUs. In order to decrease the number of rendering context switching, more cells need to be projected in each rendering pass to improve the performance of the cell projection. Thus, a cell sorting algorithm is devised, which allows to project a layer of cells each pass.

2.3.3 Cell Sorting

For a given viewing direction vector in the octree coordinate system, the signs of the coordinates determine the order in which the eight children are visited when parallel projection is used. When perspective projection is used, visibility order of the eight children can still be determined by the location of the camera relative to the octree. However, the octree structure only allows to project at most four cells in one pass for some viewing directions. As many cells as possible needed to be projected in each pass to decrease the number of rendering context switching.

The main idea of the algorithm is to divide the cells into layers and only determine the visibility order of layers. The cells within the same layer can be projected at the same time. It is observed that the cells that have the same distance to the camera can be projected together. However, using the Euclidean distance from the cells to the camera to do the cell sorting is inefficient. In order to improve the performance, the Manhattan distance is used instead of the Euclidean distance. Moreover, the Manhattan distance between a source cell and the other cells is used to group the cells into layers. A source cell is determined first for a given view point, which is the closet cell to the camera. Then a propagation method is used to compute the Manhattan distance for the other cells. The cells that have the same Manhattan distance to the source cell are put into the same layer. In the following, the cell sorting algorithm is first described in the 2D case, and then extend it to 3D.

In the 2D case, the whole object can be represented with a square, and the camera can be set up around the square. The closest cell to the camera is determined based on the camera's location with respect to the square. If the camera is located at the corner region as shown in the right image of Figure 2.11, the closest cell is the corresponding corner cell shown in grey. Otherwise, the closest cell is on the edge of the square that is opposite to the camera as shown in the left image of Figure 2.11. The closest cell can be obtained by shooting a ray perpendicular to the edge. The intersected cell is the closest cell. If the ray intersects two cells, the two cells are both used as source cell. In Figure 2.11, the Manhattan distance of each cell is shown as the number in each square box. It is observed that the cells show a very clear layer structure. It is also noted that each layer consists of more cells by using the Manhattan distance than using the Euclidean distance. In fact, it is unnecessary to explicitly compute the Manhattan distance. From the Figure 2.11, it is clear that

the source cell is made up of the first layer. And, the second layer consists of the edge neighboring cells of the source cell. Thus, a propagation method is used to group the cells into layers from the source cell C_0.

1. Let $C_0.visited = 1$ and put C_0 into a list L_0. Set the other cells to be un-visited.
2. For each cell C_i in the list L_0
    1. Obtain the four edge neighboring cells $C_{ij}(j = 0,1,2,and3)$ of C_i. If $C_{ij}.visited$ is 0, let $C_{ij}.visited = 1$ and put C_{ij} into the list L_1.
2. Project the non-empty cells of L_0. If all non-empty cells are projected, the algorithm is terminated.
3. Copy L_1 to L_0, and goto 2.

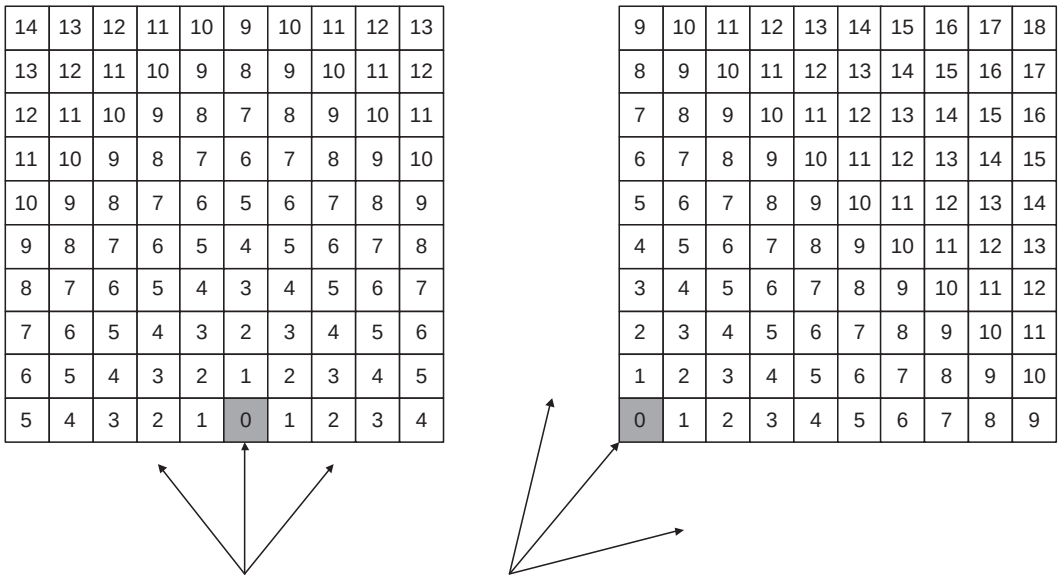


Figure 2.11: A layer of cells with same Manhattan distance can be projected together.

By using this sorting algorithm, each layer of the cells have the same Manhattan distance to the source cell. The cells within the same layer do not occlude each other, and can be projected at the same time. This algorithm can be easily extended to the 3D case. In the 3D case, the closest cell still can be found efficiently based on the region where the camera is located with respect to the volumetric dataset. The propagation process is almost the same, except that six nearest cells are used for propagation in the 3D case.

If the camera is located inside the dataset, the sorting algorithm becomes even simpler. The source cell is the cell where the camera is located. Moreover, the order information is only propagated along with the viewing direction of the camera. The cell projection of the starting cell is implemented a little different from that of the other cells. Only one rendering pass is needed to implement the projection of the starting cell. The rendering target is the third pbuffer used for color accumulation. The back faces of the starting cells are rendered to trigger the fragment program, which also give the exiting points of the corresponding rays. The camera position is passed to the fragment program as an uniform parameter. The ray direction is computed by using the exiting points and camera position. Then, the ray is evenly sampled to do ray integration from the camera position. Thus, the presented algorithm can be used for fly-through applications, such as virtual colonoscopy.

2.3.4 Implementation and Results

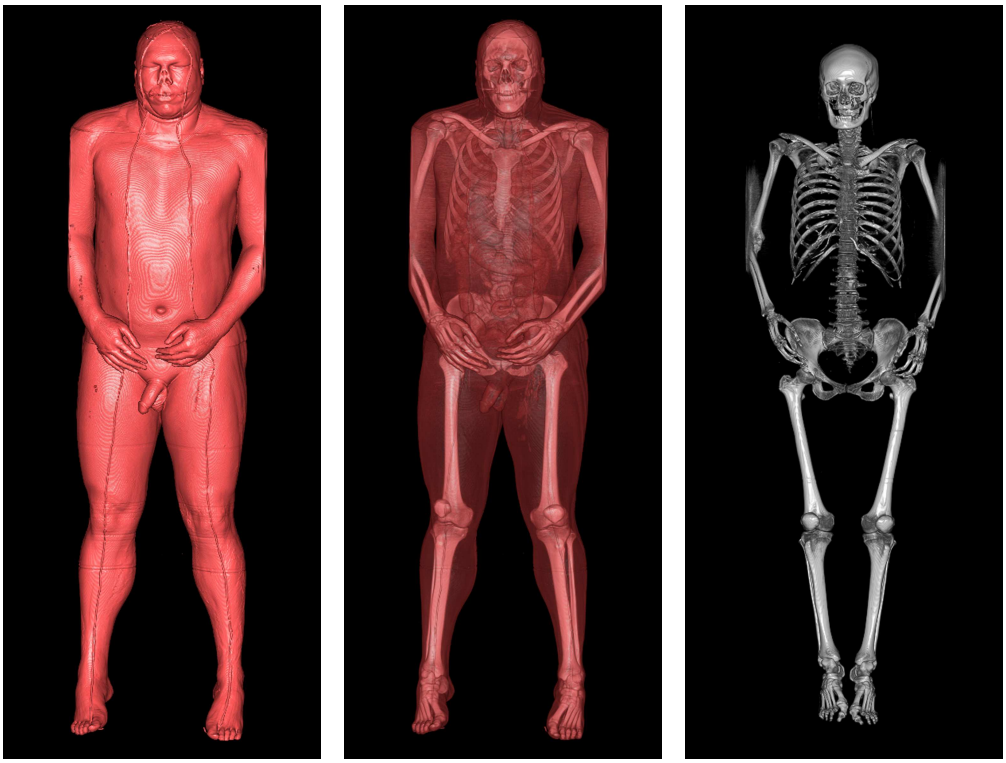
Our object order ray casting algorithm is implemented using C/C++, and fragment programs are implemented using Cg [92]. The experiments have been conducted on a 3.0GHz Intel Pentium IV PC, with 2G RAM and a NVIDIA Quadro FX 3400 graphics card. The information of the datasets used in the experiments is listed in Table 2.3.

The size N of the cell is crucial to our algorithm. A smaller N is efficient for empty space skipping, but inefficient for the cell projection executed on the GPU. Because using a smaller N will increase the number of rendering context switches, which decreases the performance. It also increases the number of texture object switches because the cells are stored in separate 3D textures. A smaller N will result in the projection of the cell covering less pixels on the image plane, which degrades the efficiency of the volumetric ray casting because of poor caching. Moreover, for each cell normalized ray direction and length of the ray segment are needed to be computed for the rays intersecting with that cell. A larger N can decrease such computation. In our current implementation, $N = 64$ for the purpose of the trade-off between empty space skipping and cell projection on the GPU.

The full resolution Visible Human CT datasets is used to test the algorithm with various transfer function. About half of the cells are empty and skipped after

Table 2.3: Datasets used in the experiments.

Dataset	Dimension	Size
Visible Male	$512 \times 512 \times 1887$	0.71GB
Visible Female	$512 \times 512 \times 1734$	0.65GB
Brain	$1080 \times 1110 \times 158$	0.93GB



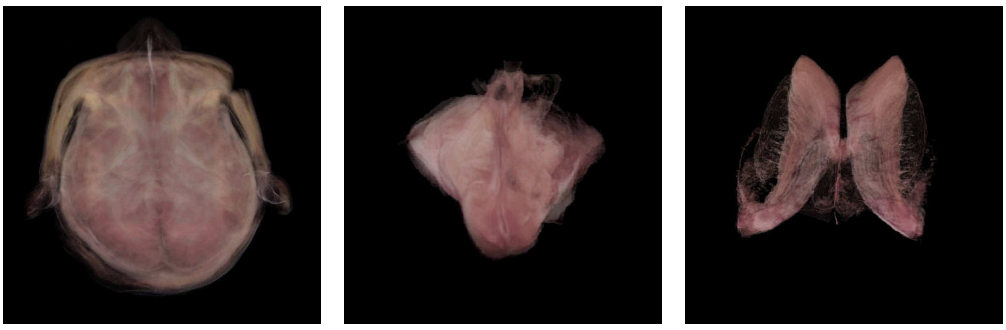
(a) 3.1 fps (b) 1.8 fps (c) 3.5 fps

Figure 2.12: Visible Human CT datasets rendering results.

the classification. Thus, most cells are fitted into video memory and AGP memory. Only a small number of cells are still resident in system memory. Our object order ray casting algorithm is capable of rendering such large datasets at several frames per second on a commodity PC. Some resulting images are shown in Figure 2.12, which are all rendered at the resolution of 512×1024. In Figure 2.12(a), the skin of the Visible Male is shown with an opaque transfer function. In Figure 2.12(b), the bone structure and some organs are shown using a semi-transparent transfer

function. In Figure 2.12(c), the bone of the Visible Female is shown with an opaque transfer function. It is natural that higher rendering speed has been achieved with the opaque transfer functions than a translucent transfer function. Because more cells are skipped in object-space and fewer cells need to be projected.

Another segmented photographic volumetric dataset is used to demonstrate the efficiency of the algorithm. Compared with the CT datasets, volume rendering for photographic datasets requires an opacity transfer function in non-linear color space, which is more complicated than that for CT datasets. The CIE *Luv* color model is used to obtain a perceptually uniform representation of the color volume, and assign an opacity value for each voxel using the method proposed by Ebert et al. [31]. Thus, each voxel of this dataset contains RGB color, opacity and segmentation information. In order to render segmented datasets, each cell is assigned with a list of labels that are used for labeling the voxels in the cell. When an organ is chosen for rendering, only the cells containing the corresponding label are loaded into video memory for projection. These cells usually can be fitted into the video memory without on-the-fly transferring data, which allows to interactively explore the segmented organs. Figure 2.13 shows some resulting images rendered from the segmented brain dataset with a resolution of 512×512. A top view of the full resolution brain dataset is shown in Figure 2.13(a). The segmented brain stem and ventricle can be rendered in real-time shown in Figures 2.13(b) and (c), because all the related cells can be fitted in video memory.



(a) 1.5 fps (b) 21.8 fps (c) 23.1 fps

Figure 2.13: Brain dataset of the Korean Visible Human.

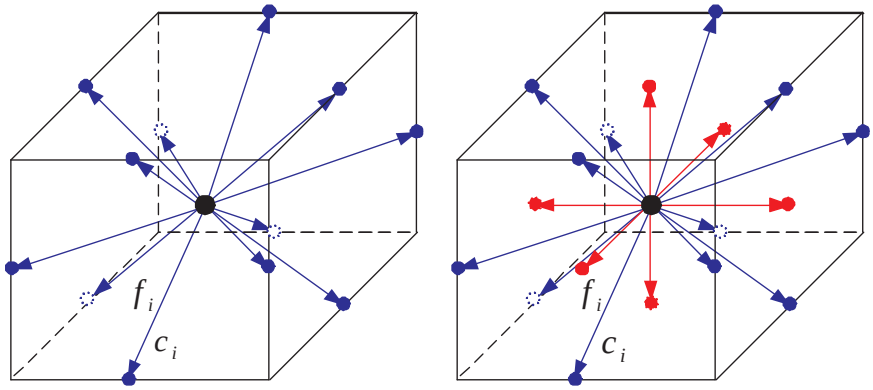
Chapter 3

Volume Rendering CC Lattices with Single Scattering

The volume optical model without scattering effects is simple and efficient to implement. However, it does not have any shadow effects. In practice, shadow is an important visual cue for many amorphous objects such as clouds, and smoke. This kind of natural phenomena is becoming more and more important in recent years. The game and movie industry need physically based flow simulation to generate realistic scenes with wind, and flowing fluids. Moreover, scientists in various fields such as mechanical engineering requires physically accurate simulation and visualization techniques to understand the characteristics and properties of flows in the real world. In this chapter, algorithms for rendering volumes with single scattering on CC lattices are presented.

The volumes being rendered are produced with the Lattice Boltzmann Method (LBM), which is a relatively new computational fluid dynamics (CFD) model [147, 169]. Inspired by cellular automata, it models Boltzmann particle dynamics on a lattice [99]. The Boltzmann equation expresses how the average number of flow particles with a given velocity changes between neighboring sites due to inter-particle interactions and ballistic motion. The LBM is second-order accurate in both time and space, and thus in the limit of zero-time step and lattice spacing, it yields the Navier-Stokes equation for an incompressible fluid. The LBM uses 3D cubic lattices of various link configurations. Figure 3.1 shows two 3D lattices used

in LBM; both lattices have the same configuration of sites but different link configurations connecting neighboring sites. The D3Q13 LBM has 12 linked neighbors for each site, while in the D3Q19 LBM, each site has 18 linked neighbors. The attributes associated with each lattice site are the particle distributions representing the probability of particle presence with a given velocity. Particles stream synchronously along lattice links in discrete time steps. Between streaming steps, they undergo collision.



(a) D3Q13 lattice. (b) D3Q19 lattice.

Figure 3.1: (a) D3Q13 and (b) D3Q19 LBM.

The particle distribution f_i is associated with the link i corresponding to the velocity vector c_i. The macroscopic fluid density $\rho(x,t)$ and velocity $u(x,t)$ are computed from the particle distributions:

$$\rho = \sum_i f_i \tag{3.1}$$

$$u = \frac{1}{\rho}\sum_i f_i c_i. \tag{3.2}$$

With the BGK collision model [169], the Boltzmann dynamics can be represented as a two-step process of collision and ballistic streaming. Taken together they can be represented as:

$$f_i(x+c_i,t^+) = f_i(x,t) - \frac{1}{\tau(f_i(x,t) - f_i^{eq}(\rho,u))} \tag{3.3}$$

$$f_i(x+c_i,t+1) = f_i(x+c_i,t^+) \tag{3.4}$$

where the local equilibrium particle distribution is given by

$$f_i^{eq}(\rho, u) = \rho(A + B(c_i \cdot u) + C(c_i \cdot u)^2 + Du^2). \tag{3.5}$$

The constant τ represents the relaxation time scale determining the viscosity of the flow, while *A*-*D* are constant coefficients specific to the chosen lattice geometry. The equilibrium distribution is a local distribution whose value depends only on conserved quantities: mass ρ and momentum ρu. The attributes of lattice site x are updated synchronously with the above equations at each time step t.

The LBM method excels due to its very efficient and simple computing process, unmatched by solver-based CFD simulators. This simplicity also makes the LBM very amenable to GPU acceleration [35, 81, 84, 128, 159, 160].

Because the LBM is defined on 3D CC lattices, the simulation results are 3D volumes defined on CC lattices. Several methods presented in this chapter uses the GPU and GPU cluster for volume rendering with single scattering. In addition, a ray tracing method for continuous ray refraction on the CC lattice is developed to render the heat shimmering and mirage phenomena.

3.1 Dispersion Visualization with Half Angle Splatting

In this section, a system is presented for visualizing the propagation of dispersive contaminants with an application to urban security [128]. In particular, airborne contaminant propagation in open environments characterized by sky-scrapers and deep urban canyons is rendered. The data is the simulation result based on the Multiple Relaxation Time Lattice Boltzmann Model (MRTLBM), which can efficiently handle complex boundary conditions such as buildings. In the simulation, massive amounts of results are generated. These numbers are hard to understand by most scientists. With visualization, the user can better analyze the simulation results of flow fields through streamlines. Realistic visualization in real time can help trainees and emergency services personnel (end users) better understand the situation and make decisions in real events. The visualization has two parts. The first is to render buildings with textures. Because the simulation is executed on the

GPU and most of the texture memory is used to store simulation data, there is little room to store textures of the buildings. Instead, noise textures and a smart shader are used to help texturing the buildings. The second part of the visualization is to render smoke with self-shadows in real-time.

3.1.1 Texturing Buildings

Textures for city models are usually captured together with the geometry. For example, Wang et al. [156] have generated both geometry and texture from a large set of registered images taken automatically. On the other hand, Früh and Zakhor [39] have implemented texture capturing as a separate video based process parallel to geometry scanning. All these methods, however, generate huge amounts of texture data since every building gets its own texture. There are several approaches to reduce the number of textures. Wonka et al. [170] have done so by creating a detailed semantic based geometry using grammars, which is textured with few repeated textures. Legakis et al. [77] have concentrated on brick patterns by synthesizing textures using a cell based method. Another approach common to commercial solutions is to concentrate on landmarks, which are postprocessed by hand using CAD applications or taken from libraries. Other parts of the city model are left without texture. In contrast, texturing is not part of the model generation process in the presented system. Instead, pictures of the real buildings are incorporated into the geometry.

The city model consists of plain geometry only. To improve the visual appearance, building façade textures are used to resemble the look of the actual city. This leads to two problems: the textures themselves need to be created and rendered and the geometry has to be augmented with texture coordinates.

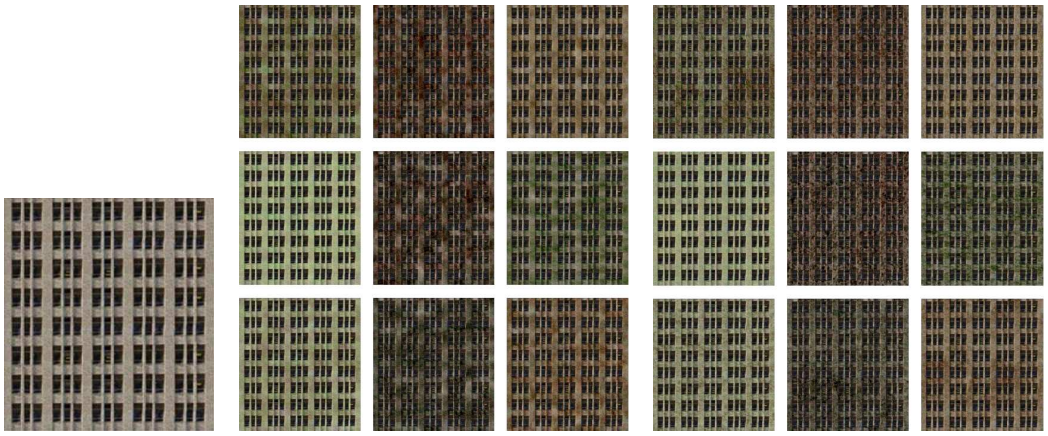
Façade textures are prepared by hand from pictures taken on site. Since texture memory is a scarce resource, only a very small amount of actual distinct façade textures can be used. The trade-off is between a larger number of low resolution textures and a smaller number of high resolution textures. Since blurry artifacts introduced in the former case are more disturbing than visual repetitiveness in the latter one, a few high resolution textures are used.

To reduce the repetitiveness of this approach, the programmable fragment

shading capability of modern graphics hardware has been exploited by implementing a texture-aging-and-variation shader. This shader changes the overall appearance of a façade texture by adding dirt and cracks without affecting major features such as windows. To do so, it needs an appropriate opacity map stored in the alpha channel of the façade texture. In total it uses five textures per façade, one of which is the façade texture itself.

Since texture memory is the most important constraint, the shader and its data are designed for versatile use. Thus, information about color and intensity of dirt added to the façade is split into color parameters and grey scale textures. This enables the shader to produce very different results with the same textures, and thus reduces the overall number of textures needed.

For each façade three grey scale "noise" textures are used to add three differently colored layers of dirt. The term noise is quoted since statistical or Perlin noise [121] usually does not give best results. Patterns of corrosion or erosion found in nature are more suitable, as shown in Figure 3.2. Dirt is added by means of color replacement. For each dirt layer the shader has a base color and a dirt color. The more the local original façade color is similar to the base color, the more the actual fragment color is dragged towards the dirt color. The similarity is attenuated by the respective noise texture.



(a) Plain façade. (b) Variations with Perlin noise. (c) Variations with corrosion patterns

Figure 3.2: Façade variation using one set of textures.

Additionally, one more grey scale texture is used to attenuate the façade texture's intensity directly, simulating cracks. Bump maps were tested for this purpose, but results indicated that, besides the necessary three color channels instead of one, high resolution maps are needed in order to make them visually effective. Again, for versatile use a parameter attenuates the impact of the intensity texture.

Due to the layering, the perceived final texture resolution is higher than the individual layer resolutions. Since the façade texture already has high resolution, the noise textures do not need to (see Figure 3.2). Additionally, due to the nature of a noise texture it can be shrunk and tiled across the whole facede without obvious artifacts. Shrinking factors up to 2 give good results. Thus, the impact of noise textures on texture memory can be kept to a minimum without sacrificing effectiveness.

The second problem to be addressed is texture coordinate generation. This includes the following steps: separation of buildings into façades, choice of a façade texture, and finally the actual texture coordinate generation for all five textures per façade.

The first step has no general solution. Its implementation depends highly on the input geometry. In the city model, buildings generally follow a box shape. This allows to associate the building's triangles with a façade based on their normals. Therefore, the k-means clustering algorithm [3] is used to get four groups of similarly aligned triangles. These four groups form two opposite wide façades and two narrow ones.

Subsequently, a façade texture has to be chosen. Since a limited number of original façade textures are available, they must be fit to multiple buildings. This is done by first registering the prepared façade textures with their respective original buildings to get an estimate of the respective physical floor height h_f and window width w_f. The façade texture assigned to a building is the one that can be fitted best using only multiples of h_f and w_f. The four noise textures are chosen randomly from a given set.

Finally, texture coordinates and shader parameters have to be generated. The façade texture coordinates are computed directly from the number of floors and windows that the chosen façade yields. Noise textures on the other hand are less

restricted. To increase vividness of the result, starting from the façade texture coordinates, the coordinates are transformed randomly using rotation, translation, and scaling in a given range. The shader parameters are generated randomly as well except for the base colors, which are attributes of the façade textures. Dirt colors are varied in the red and green channels only since blue colors are not found in natural dirt.

(a) Plain facade closeup view.

(b) Facade variation closeup view.

Figure 3.3: Closeup view using nearest neighbor interpolation.

3.1.2 Smoke

The LBM simulation computes the position and velocity of smoke particles with the coarse interactions of the fluid with the scene. The particles can be rendered using OpenGL points after reading back the positions from the GPU to the main memory and sorting them into slices by the CPU. Each particle is projected onto the image plane as a textured splat, which can be accomplished on graphics hardware efficiently. Textured splats add the small-scale interactions and visual details to the final image. However, the original textured splats method does not take into account the shadows among splats, although the shadows of all the splats can be cast onto other scene objects. Kniss et al. [69] have proposed a shading model for volumetric shadows and translucency. Instead of slicing the volume in the view direction, this method adopts the half angle slicing technique, as shown in Figure 3.4. The angle between the light direction l and the half angle direction h and the angle between h and the view direction (or inverted view direction if the angle between l and v is larger than $\frac{\pi}{2}$) v are both θ. For each slice, the light map is computed. All slices are

projected to the image plane in a front to back or back to front order as in texture based volume rendering, using a light map for shading. In Kniss et al.'s model, the volume is stored in a 3D texture and the 3D texture hardware can be exploited to reconstruct each slice efficiently. In this case, the volume is a series of particles and the slices are reconstructed by splatting.

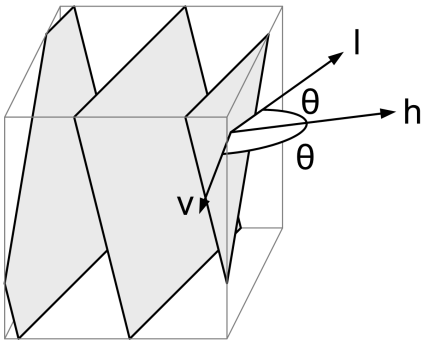


Figure 3.4: Half angle slicing.

Fig. 3.4 demonstrates the smoke rendering algorithm. First, the view direction v and the light direction l are determined and the half direction h is computed. To reconstruct the volume, the half space coordinate system must be established. h is the z-axis. The cross product of h and v is the x-axis. The origin is the center of the bounding box of the simulation. Then, the bounding box of the volume in the half space coordinate system is computed. This bounding box is sliced into n slabs with slicing planes perpendicular to the z-axis of the half space coordinate system. Thus, each slab has a start and an end z-value. For each particle, the z coordinate in the half space is computed and used to sort it into one slab. This bucket sorting costs $O(m \log n)$ time, where m is the number of particles. In each slab, the particles are rendered using the textured splats method into the density map for the current slice. The slice is projected onto the image plane and its density map and light map are used for shading. In the half space, the light map of the next slice is computed by attenuating the current light map with the density map.

Because a particle is treated as a Gaussian sphere of diameter d, the final area covered by one splat on the image plane should be a circle. Therefore, the area of one splat projected onto the slicing plane is an ellipse with minor axis of length d and major axis of length $d/\cos(\theta)$, where θ is the angle between slicing plane and

viewing plane. In the half space coordinate system, the major axis is parallel to the y-axis and the minor axis is parallel to the x-axis. $\cos(\theta)$ is the dot product of the half direction and the view direction. When projected onto the light plane (plane perpendicular to the light direction), the area covered by this ellipse is a circle of diameter d. This is because the angle between the light plane and the half plane is also θ. Therefore, the light transport is correctly computed in the half space. Figure 3.5 shows how the Gaussian reconstruction kernel for one splat is projected on the three planes.

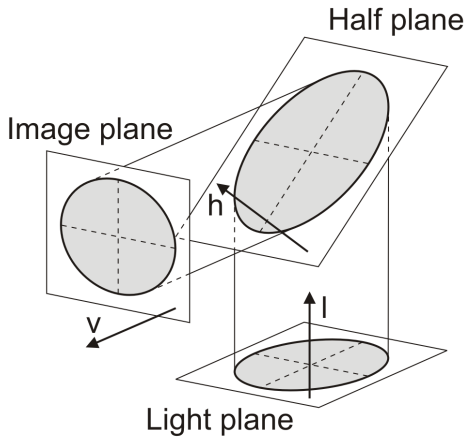


Figure 3.5: The projected spherical gaussian kernel on different planes.

3.1.3 Results

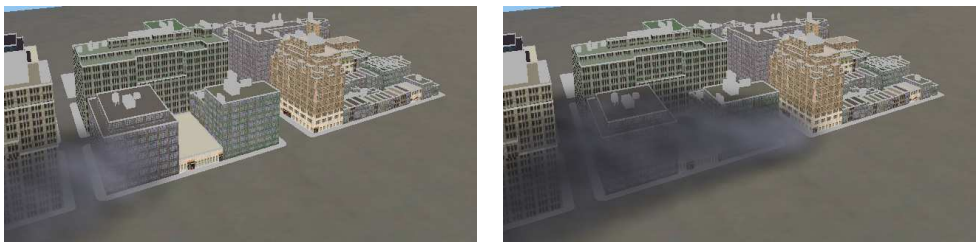


Figure 3.6: Snapshots of smoke dispersion simulation in the West Village area of New York City.

Figure 3.6 shows several snapshots of the dispersion simulation procedure at time steps 247 and 319. Figure 3.7 shows closeup views of the buildings and smoke during the simulation. Figure 3.8 shows the simulation results of a 10-block area rendered by our visualization program, where red (blue) streamlines indicate upward (downward) streaming. The LBM model consists of $90 \times 30 \times 60$ lattice sites with lattice spacing of less than 5 meters. The building GIS models are at 1 meter resolution in the West Village. The smoke particles with initial temperature and velocity are generated at the upper left corner of the bounding box. The air flows from left to right. The 6 images are snapshots of the scene at 6 different time steps. For a 640×640 image, each time step, the simulation costs 81 milliseconds, rendering the buildings costs 16 milliseconds, and rendering smoke costs 31 milliseconds.

For texturing our program uses 4 different façade textures of size up to 512×512 consuming 2.25MB in total. Additionally 10 different noise textures of size 256×256 are used, adding 640KB. Thus, less than 3MB of texture memory are used for visualizing the buildings.

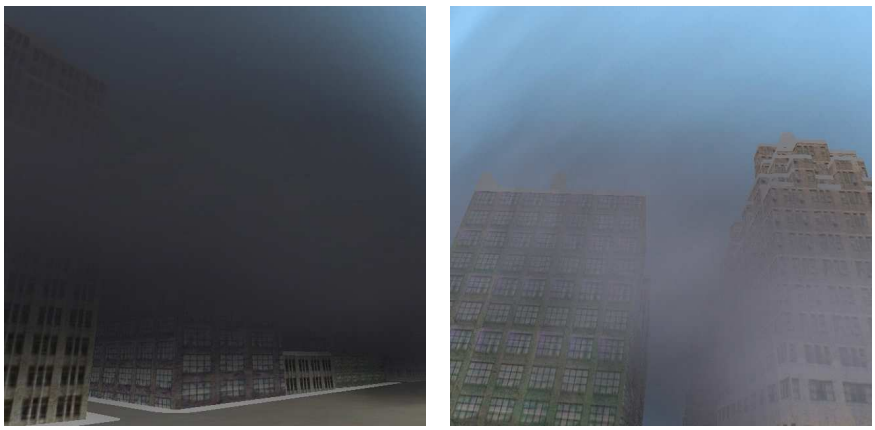


Figure 3.7: Closeup views of buildings and smoke.

3.2 Smoke Rendering with Lighting Volume

In physically-based flow modeling, previous work usually employs a uniform grid to discretize the simulation domain, and then applies numerical computations

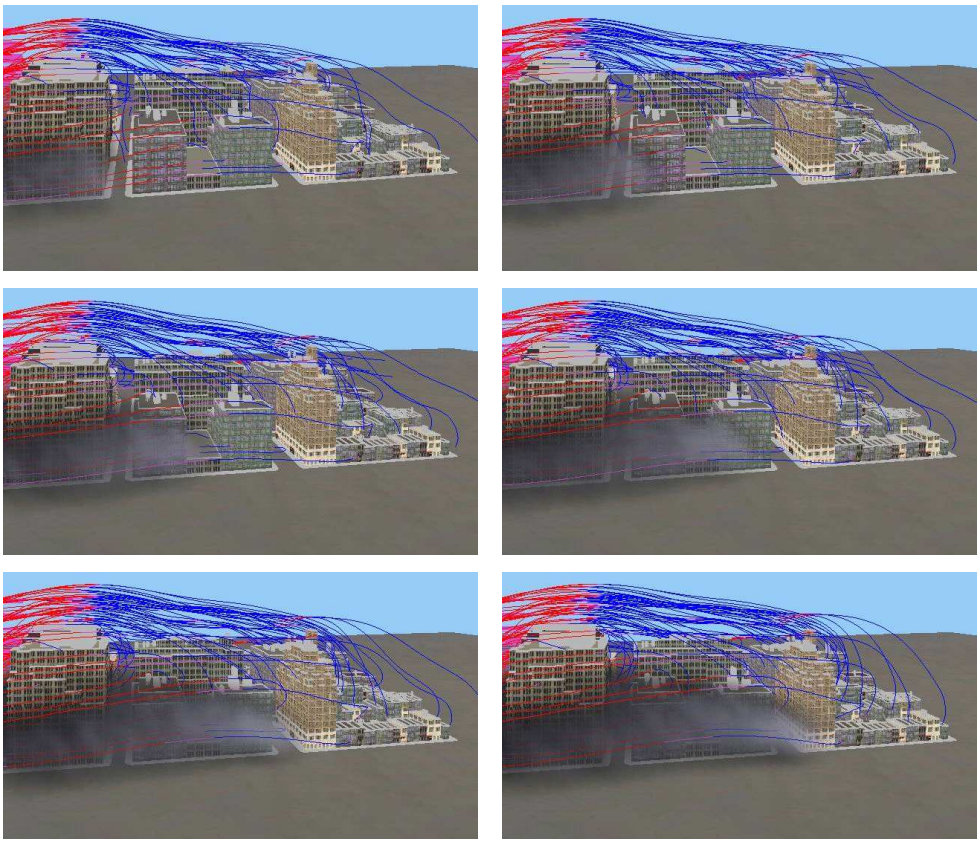


Figure 3.8: Smoke and streamlines representing dispersion simulation results in the West Village area of New York City.

to solve the Navier-Stokes equations. For large-scale simulations, it is inefficient to maintain a uniform grid with high resolution spanning the entire domain. To achieve interactive performance and at the same time optimize the use of resources, the multi-resolution LBM has been exploited that offers high resolution computation in areas of interest (for example, near a solid body) and places low resolution grids in other areas or faraway boundaries. Interfaces between the grids with different resolutions are properly treated to satisfy the continuity of mass and momentum.

This level-of-detail scheme is implemented by a 3D block-based grid structure consisting of a coarse grid and one or more fine grids. The global flow behavior in the whole simulation space is roughly modeled by the LBM simulation on the coarse grid with relatively low consumption of resources. For regions of interest, the LBM computation is performed on the corresponding fine grids superposed on the coarse one. These grids are implemented as separate blocks instead of tree-style recursive structures. The global simulation on the coarse grid determines the flow properties at grid interfaces and then defines boundary conditions of the fine grids at each time step. Therefore, the simulation on the fine grids obeys the correct global flow behavior. Meanwhile, it supplies rich visual details in the regions of interest by utilizing small grid spacing and small time intervals, and by introducing vorticity confinement. A fine grid is easily initiated and terminated at any time while the global simulation is running. Moreover, a fine grid is able to move along a moving object, to model small-scale turbulence caused by the object-fluid interactions.

The multi-resolution LBM computes the flow field in the simulation domain [179]. When a fluid source (for example, a smoke inlet) is placed and begins to release smoke, the smoke density constructs a scalar volumetric dataset. The evolution of this density volume is modeled by an advection-diffusion equation and computed by a back-tracing algorithm based on the method of characteristics. The monotonic cubic interpolation [37] is used for computing the back-tracing density values at positions off the regular grid sites. This high-order interpolation scheme slows the generation of the density volume, however, it fixes the over-shooting problem of the trilinear interpolation method and provides clearer visual details.

Because the flow field is computed on the adaptive structure, the velocities

used in the back-tracing algorithm are chosen from the computation results of different grids. Obviously, the velocities computed in the fine grids have higher priority than those of the coarse grid. We use the rectangular bounding box of the fine grid to find if the back-tracing point is inside or outside of the fine grid. Then, the velocity computed from the fine grid or from the coarse grid are used in back-tracing, respectively. Consequently, the smoke density values in regions of interest are computed and stored in a higher-resolution volume, which provides more details of the smoke dispersion. We have also implemented the back-tracing algorithm and the monotonic cubic interpolation on the GPU. This greatly accelerates the smoke volume generation compared with the CPU version.

Volumetric objects in the real world, such as smoke, and gas, are illuminated by light sources. The global illumination of participating media requires substantial computing resource due to its intrinsic complexity and it cannot achieve interactive speed in our case. Therefore, ray-tracing with single scattering is used to render the volume, which computes shadows - one of the most important illumination effects. The presented method is based on the method suggested by Kajiya and Von Herzen [67], which is a two-step numerical algorithm for volume rendering with self-shadows.

The algorithm has two steps: (1) a lighting volume is calculated that stores the light intensity of each voxel; (2) one ray is cast from each pixel on the image plane to compute the color of that pixel. As shown in Figure 3.9, the lighting volume (blue box) is oriented facing the light source so that the light direction is perpendicular to the lighting volume slices (green lines), and tightly bounds the density volume (black box). For point v_i on slice i, its lighting intensity is calculated by attenuating the lighting intensity of v_{i-1} on slice $i-1$, where v_{i-1} is the projection of v_i on the slice $i-1$ and the straight line pass through v_i and v_{i-1} is parallel to the light direction.

In order to achieve interactive speed, it is imperative to accelerate the rendering process with the GPU. In the first step, the lighting volume is computed slice by slice. The pixels on every slice are calculated in parallel. The first slice is initialized with the intensity of the light source. Every other slice is calculated by attenuating the intensity of the previous slice with opacities defined by the density values interpolated from the density volume. In the ray-casting pass, each ray starts

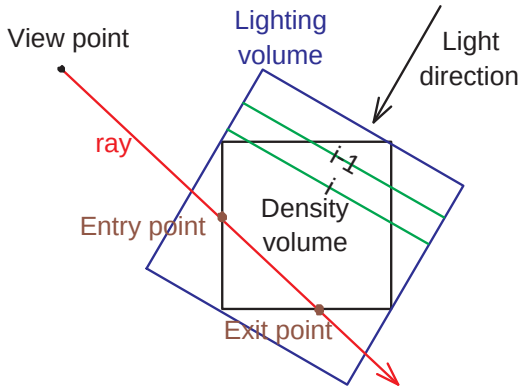


Figure 3.9: Lighting volume calculation.

Table 3.1: Smoke rendering performance.

Experiment	Volume Resolution	Rendering Time (msec)
Figure 3.10	$82 \times 82 \times 82$	52
Figure 3.11	$100 \times 50 \times 50$	42
Figure 3.12	$50 \times 50 \times 50$	36

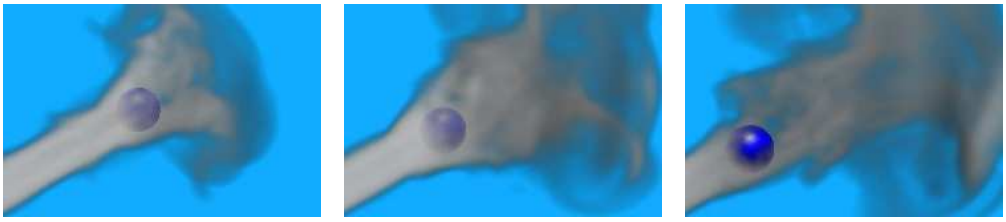
from the front faces of the volume bounding box and stops at either the back faces or the surface of the polygonal objects inside the volume. A face of the bounding box is a front (back) face if and only if the dot product of its normal and the view direction is less (greater) than 0. To calculate the termination points, the polygonal objects and the back faces are projected onto the image plane with depth test. This method computes the depth information of the possible ray termination points. When a ray is cast into the volume, color and opacity values are accumulated at each sampling point, where the lighting intensity is interpolated from the lighting volume for shading.

Figures 3.10, 3.11 demonstrate the rendering results of simulating smoke passing through a sphere at different time steps, with a fine grid surrounding the sphere, superposed on the coarse grid. In Figure 3.12, the fine grid is controlled by the user. (a) At first, no fine grid is used; (b)-(c) A fine grid is used to model small-scale details around the sphere; (d) The fine grid is terminated and small-scale details disappear. The rendering performance is reported in Table 3.1.

The speed of the multi-pass lighting algorithm is reduced because every slice

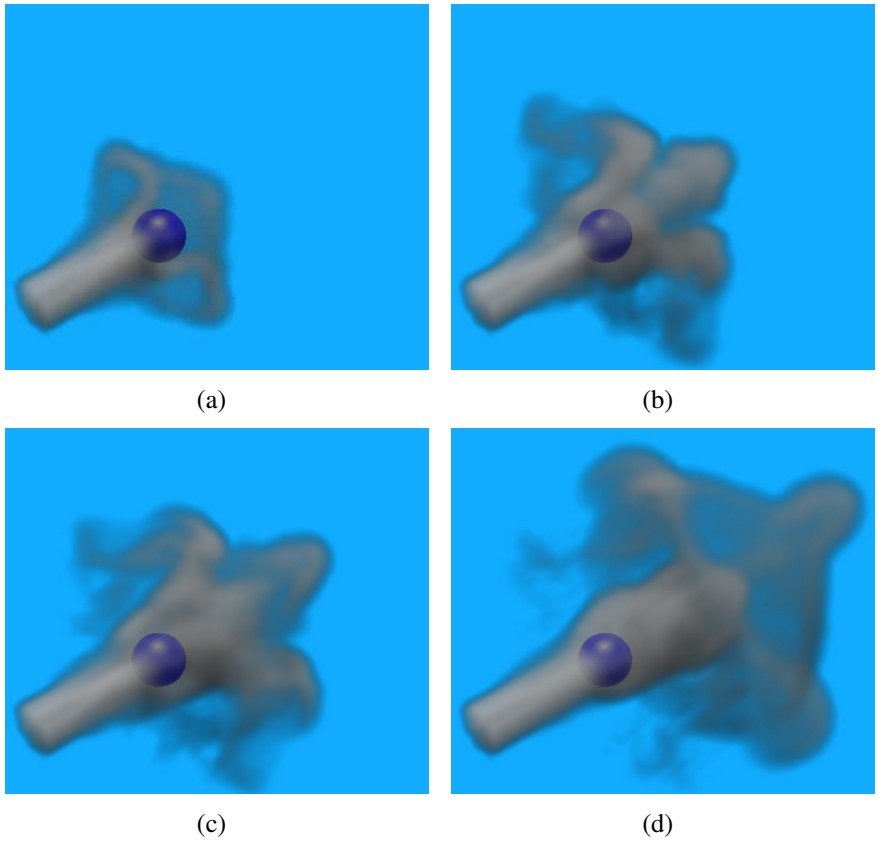
(a) (b) (c)

Figure 3.10: Smoke passing a static sphere.



(a) (b) (c)

Figure 3.11: Smoke passing a sphere moving towards the smoke inlet.



(a) (b)

(c) (d)

Figure 3.12: Smoke passing a sphere with user controlled fine resolution grid.

is written to the framebuffer, copied to the 3D texture and read back in the fragment program. Moreover, the OpenGL pipeline is stalled at the end of every pass waiting for texture copy. Therefore, a new method with the NVIDIA CUDA toolkit has been implemented. CUDA uses the C programming language with some extensions for multi-threading computation on GPU. It allows the program to write data to multiple memory addresses in one thread (scattering), which is the feature needed to compute the lighting volume more efficiently. For a lighting volume of resolution $x \times y \times z$, a CUDA kernel program is called with $x \times y$ threads. Each thread calculates one lighting ray. z voxels on a single lighting ray are calculated in a loop of the kernel program and written to the lighting volume. The kernel program is called only once and the entire volume is obtained. Therefore, it is more efficient than the multi-pass OpenGL algorithm. Currently, CUDA cannot share texture with OpenGL. Thus, the resulting lighting volume is copied from the CUDA memory to an OpenGL pixel buffer object and then to a 3D texture. This procedure is extremely fast because only high-bandwidth GPU memory is involved and the comparatively slower PCI-express bus is not used for data transfer.

In the ray-casting step, previous methods trace rays in both the density volume and the lighting volume. Each ray needs to maintain the current sampling position and ray direction in both volumes. At each sampling point, the program calculates two trilinear interpolations, which is inefficient. We propose a new method of tracing rays only in one volume. Recall that when calculating the lighting volume, the density volume must be sampled with trilinear interpolation. We store the density $s(p)$ along with the light intensity $L(p)$ for each point p in the lighting volume. The light texture is thus changed to store both light intensity and density. Krüger and Westermann [73] have implemented ray-casting for volume rendering on the GPU with empty space skipping and early ray termination. However, empty space skipping requires an additional data structure (such as min-max octree) to be calculated and stored on the GPU. Our simulation generates different density volumes in each step. Calculating the data structure for empty space skipping costs more time than rendering one frame. The dispersion volume is highly transparent and the opacity of most rays is not saturated. Early ray termination is not efficient in this case. Therefore, our ray-casting algorithm is a simple one-pass algorithm without empty space skipping or early ray termination.

The CUDA based method has been tested to render the LBM simulation results using a model of New York City from 7th Avenue to 6th Avenue and from 59th Street to 57th Street. This small region contains 6 blocks and tens of buildings. The test platform is a PC with a Geforce 8800 GTX graphics card with 768MB memory and two Xeon 3.6GHz CPU (although the experiment code is not multi-threaded). The resolution of the simulation grid is $128 \times 64 \times 64$. Figure 3.13 shows the navigation snapshot images of resolution 600×800 inside the city blocks. The average time for producing one frame is 57 milliseconds, in which the simulation costs 30 milliseconds, rendering smoke costs 7 milliseconds, and rendering buildings and roads costs about 20 milliseconds.

3.3 Volume Rendering for Urban Security on GPU Cluster

In an emergency of airborne hazardous releases in urban environments such as New York City, quick response is necessary to save lives and reduce property damages. Emergency management personnel need to get the information of plumes in real-time for possible remediation and evacuation. Furthermore, it is extremely helpful to predict the propagation of airborne contaminants, even hours and days in advance running the simulation in accelerated real-time. The potential dispersion propagation under different conditions can also be used for training purposes. In practice, the airborne releases are moved by the air flows and propagate in the environment. The flow field of the atmosphere high above the buildings is roughly described by the local meteorological conditions. The meteorological data usually has the resolution in about several to 30 kilometers and the frequency in many minutes or hours. This very coarse data is hard to be used for estimating the dispersion propagation, because of its low resolution. The complicated hypsography in urban environments characterized by sky-scrapers and deep urban canyons makes subtle changes in the flow field in the scale of several meters, which is not captured in the meteorological model. The flow details in this fine resolution have great impact on the dispersion propagation and cannot be neglected. Therefore, a system has been developed based on LBM, which uses the meteorological data or other

Figure 3.13: Snapshots of navigation in New York city blocks on a single GPU.

user defined data on a very coarse grid as the boundary condition of the simulation. However, it simulates the flow in the fine resolution at several meters and tracks the dispersion in the detailed flow field.

3.3.1 Background

Propelled by the fast advancement in recent years of the GPUs, modern GPU's high computational-power, and high-level programming interface, general-purpose computation on GPUs (GPGPU) have become an active research area. Researchers have mapped several simulations, linear algebra operations, and a broad range of other computations on the GPU [45, 118]. Many of these GPU-based implementations have achieved around an order of magnitude speed-up over their software-version counterparts. Some researchers have also used multiple GPUs either in a GPU cluster [35, 57] or on a multiple-graphics-card PC [40, 48] to solve larger-scale computations. In Section 3.1, the simulation results were read out from the GPU, stored to disks and later rendered off-line. In this section, a more complicated volume rendering is presented for multi-resolution LBM simulation on the GPU cluster. The real-time online rendering is coupled with the simulation on the same GPU cluster. In so doing, there is no need for reading out the whole simulation volume from the GPUs and dumping them to disks, further allowing the user to monitor the simulation in running-time.

Since the simulation data is located on a cluster of GPUs, the GPUs are also used to render the volume and each GPU renders a partial frame for one subvolume. Many methods have been proposed for compositing the partial frames with general or special-purpose graphics hardware for distributed volume rendering [58,87,110]. In this section, the compositing tree [141] is used for better network performance.

The simulation computes the flow field in each step and traces the dispersion particles to generate density volumes. Although the flow field can be visualized as streamlines, realistic rendering of the dispersion volume gives the user an intuitive understanding of the dispersion distribution. The buildings of the simulated urban area are represented by polygons and textured with facadlets created from photos of the real buildings. The final result composites the dispersion volume and the buildings with their facade.

3.3.2 Volume Rendering on a GPU Cluster

The density volumes generated in the simulation are distributed on the cluster nodes. Transferring volume data to a single node for rendering is inefficient and

unnecessary. Because the simulation is implemented on GPUs, the volume can also be rendered on the same GPUs to avoid transferring large amounts of data between GPUs and main memory. In the rest of this subsection, the rendering algorithm on a single GPU is explained, followed by the discussion of the architecture of distributed volume rendering on a GPU cluster.

The algorithm on a single GPU is similar to the one described in Section 3.2 except the ray casting step. In the ray-casting step, previous method traces rays in both the density volume and the lighting volume. Each ray needs to maintain the current sampling position and ray direction in both volumes. At each sampling point, the program calculates two trilinear interpolations, which is inefficient. In this section, a new method of tracing rays only in one volume is presented. Remember that when calculating the lighting volume, the density volume must be sampled with trilinear interpolation. Density $s(p)$ is stored together with the light intensity $L(p)$ for each point p in the lighting volume. This can be easily achieved by saving density s to the alpha bits of the framebuffer. The light texture is thus changed to store both light intensity and density. The lighting volume calculations is summarized as follows:

1. The light coordinate system is constructed with z being the light direction and its origin is the center of the density volume. The resolution of the lighting volume $Res_l(x,y,z)$ is calculated according to the user defined sampling rate and the size of the density volume bounding box. An off-screen pbuffer is created with the lighting volume slice resolution.
2. The polygonal mesh of the buildings is transformed to this space and projected onto the pbuffer. Because only depth is needed in calculating the lighting volume, modern GPUs can process the mesh very fast.
3. The density volume and lighting volume are stored in two 3D textures texDensity and texLight respectively, and the transfer function is stored in a 1D texture texTransFunc. Initialize the light intensity of slice 0 with the intensity of light source. Sample the texDensity and save it to slice 0. Let i=1.
4. For slice i, draw a rectangle of the pbuffer size associated with the proper coordinates in the texDensity and texLight. The fragment program retrieves the light intensity L and density s of the previous slice. Then, s is translated to opacity τ by lookup texTransFunc. Attenuate L with s to get the light

intensity L' of the current pixel. Sample texDensity for the density s' of the current pixel. Output s' and L' to pbuffer.

5. Copy the contents in pbuffer to slice i of texLight.
6. $i = i + 1$. If $i < Res_l.z$, go back to step 4.

Krüger and Westermann [73] have implemented ray-casting for volume rendering on the GPU with empty space skipping and early ray termination. However, empty space skipping requires an additional data structure to be calculated and stored on the GPU. The simulation generates different density volumes in each step. Calculating the data structure for empty space skipping costs more time than rendering one image. The dispersion volume is highly transparent and the opacity of most rays is not saturated. Early ray termination is not efficient in this case. Therefore, a new ray-casting algorithm has been designed without empty space skipping and early ray termination for this application. The algorithm [73] draws the front faces of the volume bounding box to trigger the fragment program, which does not work when the image plane intersects with the front face. It also draws the intersection polygon of the image plane with the bounding box. The algorithm further [73] reads the direction of the rays in every rendering pass, which flushes the volume data in the texture cache. The proposed algorithm calculates the direction in the vertex program and normalizes the direction vector in the fragment program. Although it uses 3 additional instructions for vector normalization, it reduces memory bandwidth consumption. In this application, the result of volume rendering must be correctly blended with the mesh of the urban environments. Starting from the image plane, each ray may or may not enter the volume before reaching the building surface. The occluded part of the volume need not be sampled in the volume rendering. This can be accomplished by comparing the polygon depth with the depth of the sampling point in the fragment program. The early z test is exploited to cull occluded fragments, which requires writing one fragment for one sampling point. In general, our ray-casting algorithm on the GPU has the following steps:

1. Draw the mesh and the back face of the density volume's bounding box and transform the vertices to the eye space in the vertex program. Use depth test to store the nearest fragment's depth value of each pixel in a 2D texture named texExit, which is the end point of the ray.
2. If the bounding box intersects with the image plane, calculate the intersection

polygon. This polygon and the front face of the bounding box are the proxy geometry. Draw the proxy geometry with the same vertex program used in step 1. The generated fragments are the start points of the rays. For each pixel, the fragment program retrieves the end point depth from texExit, which is subtracted from the input depth. The result is the length of the ray segment inside the volume and not occluded by the mesh. This length is divided by the step size to calculate the number of sampling points or steps on the ray segment. The result is normalized by the maximum possible sampling points m and stored in the depth buffer.

3. Calculate the coordinates of the proxy geometry vertices in texLight. Also, calculate the view point coordinates eyeLS in the texLight, which is a uniform parameter in the vertex program of the next step.
4. Given the maximum number of steps n of the rays, draw the proxy geometry n times. In the vertex program, calculate the ray direction dirLS with input coordinates posLS in texLight and eyeLS. Output posLS, dirLS and the current step number j as texture coordinates. The output clip coordinate z_c is carefully computed by $z_c = (2\frac{j}{m} - 1)w_c$ so that the normalized device coordinate z_w (or depth) is $\frac{j}{m}$. The meaning of z_c, z_w are defined in OpenGL specification. If a ray segment has $k(k < n)$ sampling points, the $(k+1)$th to nth fragments of the pixel are culled by early z test. In the fragment program, normalize the input dirLS and calculate the sampling position pos=posLS+dirLS*stepNumber*stepSize. Fetch the light intensity and density from texLight to calculate the color and opacity. The color and opacity are written to the framebuffer for front-to-back compositing.

The described algorithm does not calculate fragments occluded by the mesh or outside the light volume and thus is very efficient.

The GPU cluster has one master node, a group of simulation and rendering nodes (i.e., work nodes), and a group of compositing nodes. Figure 3.14 shows the architecture. The master node sends a message of rendering parameters such as a transformation matrix to rendering nodes in each frame. The rendering nodes exchange light images in a chain. Each rendering node renders a partial frame of the density volume of one block. The generated images are read from the GPU and transferred to the compositing nodes on the lowest level of the compositing tree.

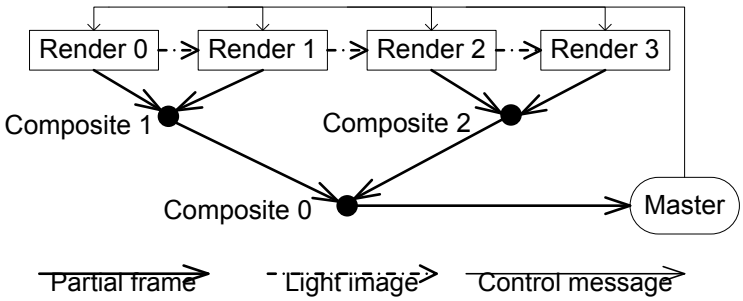


Figure 3.14: A sample GPU cluster of 4 work nodes for simulation and rendering, 3 compositing nodes and 1 master node.

Every compositing node receives two partial frames and composites them together on the GPU. If the compositing node is not the top level node, then the compositing result is sent to the parent node. The master node is responsible for rendering the buildings with facadlets and compositing them with the volume rendering result.

The simulation of flow field partitions the space into $x \times y \times z$ blocks of the same resolution, each assigned to the GPU of one node. A sort-last strategy for distributed volume rendering has been exploited. The first task is to compute the lighting volume of each block. Due to space partitioning, a block might be occluded by other blocks in the light direction. The light intensity of the blocks first slice cannot be initialized with the light source. Instead, it is the last slice of the neighboring block in the light direction that decides the initial lighting. In the worst case, one block can be occluded by $O(max(x,y,z))$ blocks. This means that the lighting volume of certain blocks cannot be computed before the computation of $O(max(x,y,z))$ blocks finishes, which is inefficient. It is observed that the simulation traces the dispersion in the flow field and each simulation step corresponds to a short time in the real world. Therefore, the density volume does not change too much in several steps. The algorithm sorts the blocks in ascending order $B_i(i = 0,1,...,x \times y \times z - 1)$ of the distance from block centers to the light source. For each frame, the last slice of block B_i's lighting volume is read from the GPU to compute the light intensity of the first slice of B_{i+1}. The result is transferred to the node of block B_{i+1}. This method is not 100% accurate but still produces visually pleasant results. Initially, every node gets one copy of the geometry of buildings and calculates the same shadow volume independently. The shadow volume is stored on the GPU and used

in the following lighting volume calculation, as long as the light direction does not change.

As demonstrated in Figure 3.15, each block is occluded by one or more blocks from the light except block 1. The blue dashed boxes are the lighting volumes of block 0 and 1. The red triangle of lighting volume 1 is outside block 1 and the density is 0 inside it. Therefore, the light intensity of the last slice of lighting volume 1 is equivalent to that of the interface (brown line) between block 0 and 1. Similarly, the green triangle of lighting volume 0 is outside block 0. The light intensity on the brown line is equivalent to that of the first slice of lighting volume 0. Here, the first slice of lighting volume 0 is decided by the last slice of lighting volume 1.

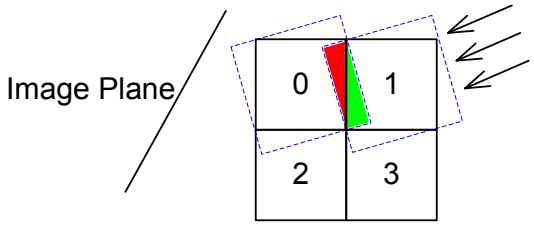


Figure 3.15: Example configuration of 2×2 nodes in 2D.

There are two types of network communication for rendering except for the control messages from the master node to the rendering nodes. One is transferring partial frames, the other is lighting images. To exploit the parallelism between network communication and GPU computation and rendering, the non-blocking message passing routines of MPI are used. Each compositing node has two fan-in and one fan-out, which reduces the possibility of network collision.

3.3.3 Results

The simulation and visualization system is demonstrated with the Times Square area in New York City, from 8th Avenue to Park Avenue and from 42nd Street to 60th Street. This region is approximately 1.46 kilometers$\times$1.19 kilometers and has 75 blocks and more than 800 buildings. The GPU cluster is composed of 16 simulation/rendering nodes, 15 compositing nodes and 1 master node for a user interface. Each node is equipped with two 3.2GHz Xeon CPUs, but only one

is used. Each node has 2GB memory and one NVIDIA QuadroFX 4500 GPU of 512MB graphics memory. The nodes are connected with a Gigabit Ethernet. Note that the compositing nodes have the same configuration as computation nodes, although they are used mainly for their network bandwidth. The finer grid of the simulation is arranged to tightly enclose the Times Square area and its resolution is $320 \times 100 \times 280$ at the grid spacing of 4.25 meters. The spacing of the coarse grid is twice that of the finer grid and the resolution is $180 \times 100 \times 160$.

Figure 3.16 shows the overview of the simulation results at different time steps, where the wind is blowing from the right to the left. Figure 3.16(a)-(c) are the snapshots during navigation at different time steps. Figure 3.16(d)-(f) shows the bird-eye views at 3 ascending time steps. For each step, the simulation on the GPU cluster runs in 485 milliseconds including 2 steps in the finer grid and 1 step in the coarse grid. Thus, it computes 42.9 million LBM cells per second with 16 nodes. As a comparison, the previous system [35] calculates 49.2 million LBM cells per second on 32 nodes without integrated rendering. Note that the system uses ZippyGPU for fast development and can be further optimized. The volume rendering costs 107 milliseconds and the most time-consuming step is calculating the lighting volume. Because current GPUs do not support rendering to 3D texture, each slice of the lighting volume must be calculated and copied back to the texture with glCopyTexSubImage3D function call, which is very slow on current GPU architecture.

3.4 Volumetric Refraction for Heat Shimmering and Mirage

Various natural phenomena involve hot objects, dynamic flows and heat transfers, such as melting, dissolving, shimmering and mirage, which are of great interest to researchers in computer graphics and scientific simulations. For simulating these phenomena, it is imperative to provide a correct and efficient modeling of the heat transfer as well as the interaction between the objects and the flow. Zhao et al. [178] have presented a physically-based method that provides a basic framework for modeling these thermal phenomena. A heat transfer model has been introduced between the heat source objects and the ambient flow environment, which includes

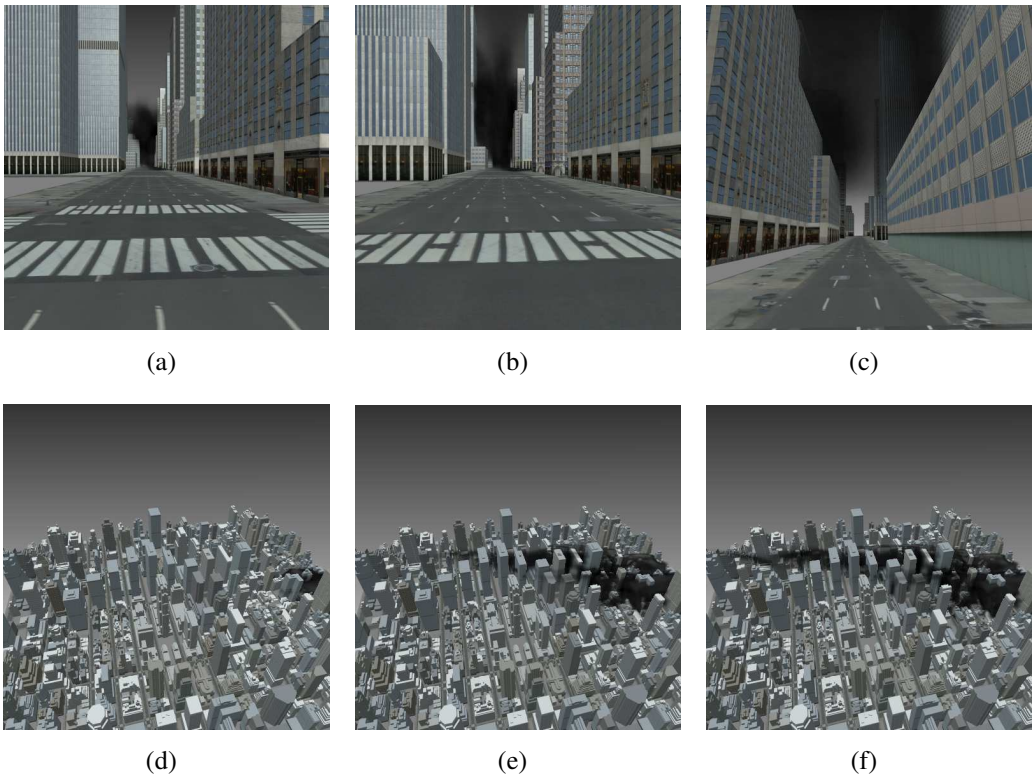


Figure 3.16: Smoke dispersion simulated in the Times Square Area of New York City.

conduction, convection and radiation. The heat distribution of the objects is represented by a novel temperature texture. The simulated thermal flow dynamics models the air flow interacting with the heat by a hybrid thermal lattice Boltzmann model (HTLBM). The computational approach couples a multiple-relaxation-time LBM (MRTLBM) with a finite difference discretization of a standard advection-diffusion equation for temperature.

In this framework, the temperature variation resulting from the interaction between the heat sources and the surrounding air is computed from the method described above. The changes in the index of refraction of the air are attributed to such temperature variation. Refraction, which produces the shimmering phenomena, occurs when light rays cross the interface between regions that have different indices of refraction. The relation between the angle of incidence θ_1 and the angle of refraction θ_2 is described by Snell's Law:

$$\frac{n_1}{n_2} = \frac{\sin\theta_2}{\sin\theta_1} \tag{3.6}$$

where n_1 and n_2 are the corresponding indices of refraction of the two materials, and the incident ray and the refracted ray stay in the same plane.

For air, the dependence of the index of refraction on temperature and pressure can be empirically described by the following equation [85]:

$$n = \frac{c_1 * Pa * (1.0 + Pa * (60.1 - 0.972 * T) * 10^{-10})}{1.0 + c_2 * T} \tag{3.7}$$

where $c_1 = 0.0000104$ and $c_2 = 0.00366$ and n is the index of refraction of air. The constant pressure of the air, Pa, is measured in Pascal and the temperature, T, in Celsius.

A light ray traverses the temperature volume with a small step size. At each step, the gradient of the temperature field is calculated by trilinear interpolation at the hit point, which defines the normal N of the interface. Then, the index of refraction is determined by Equation 3.7. By bending the light ray using Snell's Law (Equation 3.6), the new resulting ray direction is obtained, and the ray is traversed to the next hit point. When bending the ray, total reflection may occur, which causes a mirage. The presented algorithm includes this situation: When calculating θ_2 in Equation 3.6, if $|\sin\theta_2| > 1$, total reflection occurs. As a consequence, the ray

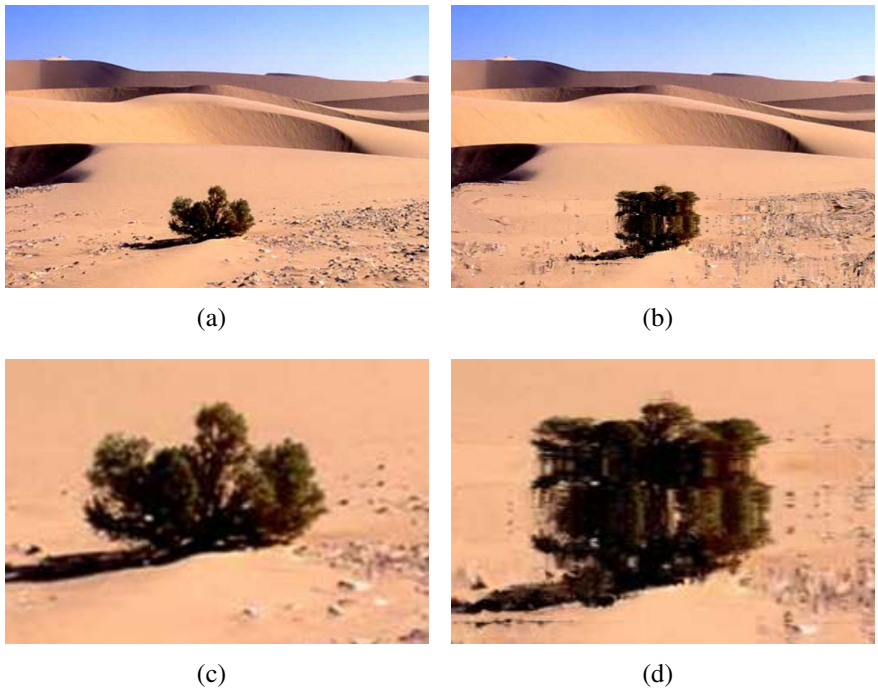
direction is changed to the total reflection direction at the point. Therefore, the effects of a mirage is naturally included in this model.

A heat source object is voxelized and each voxel is assigned a segmentation flag: inside or outside. For each ray, if a sampling point is inside the object, the nonlinear ray tracing stops and returns the color of the object texture.

The nonlinear ray tracing through the temperature volume is implemented on the GPU and coupled with the simulation. By executing the whole simulation cycle (including both computation and rendering) on the GPU, the data does not need to be read from the GPU, which could be a major bottleneck on current GPU architecture. For every image pixel, a ray is shot from the eye to its position. The information of all rays is stored in a 2D texture (each texel corresponds to one ray) and is processed by a fragment program. On current GPUs, the Shader Model 3.0 allows loops, dynamic branching, and program lengths of up to 65535 instructions. Using these facilities, the GPU implementation only needs a single pass of fragment processing to iteratively forward the rays and compute their refractions until they terminate. This allows for a much easier GPU implementation than a previous GPU-based non-linear raycaster [161] which required multiple rendering passes.

Figure 3.17 illustrates the shimmering effects easily observed in a desert on a sunny day. The ground is heated up rapidly by the sun and the heat rises to the air. Due to the non-uniform and dynamic distribution of the air temperature, the original background landscape in Figure 3.17(a) and its zoom-in view in Figure 3.17(c) appear distorted to the observer. In Figure 3.17(b) and in the corresponding zoom-in view in Figure 3.17(d), heat comes up from the ground and shimmering is clearly visible on the bush at the center of the scene. The images in Figure 3.17 are rendered with a NVIDIA Geforce 6800 Ultra card. The 3D simulation lattice size is $50 \times 50 \times 50$. For rendering an image of 400×400 pixels, the GPU-based nonlinear ray-casting takes 104 milliseconds for each frame. For comparison, the software implementation of the same algorithm takes 3034 milliseconds and the GPU version is 29.1 times faster.

Mirage occurs when some rays are bent by total reflection, a situation which is handled naturally in the presented algorithm. In Figure 3.18, comparing with a static desert scene (Figure 3.18(a)), a phantom body of water appears in the desert



(a) (b)

(c) (d)

Figure 3.17: Desert shimmering.

(a) (b) (c)

Figure 3.18: Mirage in a desert.

(a) (b) (c)

Figure 3.19: Mirage over water.

with shimmering (Figure 3.18(b)) and may become larger (Figure 3.18(c)). In Figure 3.19, a real photo is compared with the simulated mirage effect. Figure 3.19(a) is a real photo taken in Finland. Figure 3.19(b) shows an original scene with no mirage. Using it as the background and starting the simulation, the mirage effect similar to the real photo (in Figure 3.19(a)) appears, as shown in Figure 3.19(c).

Chapter 4

Volumetric Global Illumination on FCC Lattice

In this chapter, a novel volumetric global illumination framework based on the Face-Centered Cubic (FCC) lattice is described. An FCC lattice has important advantages over a Cartesian lattice. It has higher packing density in the frequency domain, which translates to better sampling efficiency. Furthermore, it has the maximal possible kissing number (equivalent to the number of nearest neighbors of each site), which provides optimal 3D angular discretization among all lattices. A new GPU-accelerated two-pass (illumination and rendering) global illumination scheme on an FCC lattice is presented. This scheme exploits the angular discretization to greatly simplify the computation in multiple scattering and to minimize illumination information storage. The GPU has been utilized to further accelerate the rendering stage. The new framework is demonstrated with participating media and volume rendering with multiple scattering, where both are significantly faster than traditional techniques with comparable quality.

4.1 Background

Direct volume rendering algorithms reconstruct a continuous function, which is projected to a 2D image. This procedure involves dimension reduction thus inevitably loses some information. To capture more details, many lighting and illumination methods have been developed. Local illumination models omit sophisticated effects such as multiple scattering and indirect illumination for the sake of rendering speed. However, they are the dominant light-object interaction for participating media (smoke, clouds) and many translucent materials. To cope with these effects, volumetric global illumination techniques are required in order to present important visual features [143].

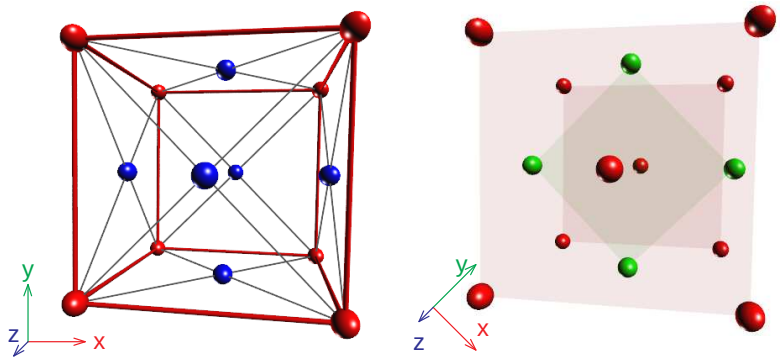
This chapter describes a new volumetric global illumination framework, which exploits both spatial and angular discretization on lattices. In computer graphics, spatial discretization has been well-studied to simplify the calculation of light-object interaction, but angular discretization has not been fully exploited. Specifically, the FCC lattice is adopted because it has better sampling efficiency compared with the CC lattice and it provides optimal angular discretization among all lattices. Furthermore, a new two-pass algorithm is presented to render participating media and volumes with multiple scattering. The idea of this algorithm is that the traced photons only move along the lattice links. We call these photons "diffuse photons". Here, the phase function is discretized on the lattice links to simplify the diffuse photon tracing and radiance estimation. The storage of diffuse photons is minimized by storing the number of photons on lattice links. For flexibility and extensibility, we also implemented tracing photons with accurate direction, which are called "specular photons" in this paper. The O-Buffer data structure proposed by Qu and Kaufman [129] has been exploited to reduce the storage space of specular photons. Our volumetric global illumination framework is capable of producing high quality images and is significantly faster than traditional methods. This general and flexible framework can be extended to render hybrid scenes with both volumes and surface objects.

The proposed volumetric global illumination framework is a new two-pass rendering algorithm on FCC lattices. In the first pass, photons are emitted from light sources and the photon energy is distributed in the scene, illuminating the media. In

the second pass, a ray tracing method is used to generate the final image. At each sampling point x on the ray of direction $-\omega$, the radiance $R(x,\omega)$ is estimated by the photons in a small neighborhood of x for shading.

4.2 FCC Data Structure

As shown in Figure 4.1(a), an FCC lattice consists of simple CC cubic cells with additional sampling points (blue) located at the center of each cell face. According to the lattice definition described in Section 1.1, any FCC lattice site can be constructed via linear combination of three basis vectors. As shown in Figure 4.1(b), another construction of the FCC lattice has basis vectors $v_x = (1,0,0)$, $v_y = (0,1,0)$ and $v_z = (0.5,0.5,\sqrt{2}/2)$, which can be obtained by defining an appropriate rotated coordinate system. This construction can be viewed as interleaving 2D square grids (red and green semi-transparent slices), where slice $2i+1$ is shifted from slice $2i$ by v_z. The second construction in Figure 4.1(b) can be obtained by rotating the first one in Figure 4.1(a) $\frac{\pi}{4}$ about the z axis. The distance between two adjacent lattice sites is 1 and the origin $O = (0,0,0)$.



(a) (b)

Figure 4.1: Two constructions of an FCC lattice.

4.2.1 Storage and Indexing

An FCC lattice is stored in a 3D array of dimension (n_x, n_y, n_z).The array stores the lattice layer by layer and each layer has $n_x \cdot n_y$ sites, which are stored in row major. Given a lattice site that is on layer k, row j, and is the i-th lattice site, its coordinates in the FCC lattice are represented as a 3-tuple (i, j, k). This lattice site is the $(k \cdot n_y \cdot n_x + j \cdot n_x + i)^{th}$ element of the array.

This definition scheme is adopted due to its simplicity, where the FCC lattice can be decomposed into two interleaved sub cubic lattices with a deviation vector of $v = (0.5, 0.5, \sqrt{2}/2)$. This representation provides a framework that enables quick indexing and efficient implementation of many basic lattice operations. For instance, the mapping from an arbitrary FCC lattice site of index (i, j, k) to its corresponding CC coordinates (x, y, z) can be defined by the following equations:

$$\begin{aligned} x &= i + (k \bmod 2)/2, \\ y &= j + (k \bmod 2)/2, \\ z &= \sqrt{2}k/2. \end{aligned} \tag{4.1}$$

4.2.2 Nearest Site

Finding the nearest FCC lattice site can be implemented by first looking for two neighbors having the shortest Euclidean distance to the sample point in the two sub cubic lattices using Equation 4.1, then selecting the closer one between them. Given a point $P_C = (x, y, z)$ in Cartesian space, find the nearest FCC lattice site $P_{FCC} = (i, j, k)$ that has the minimum distance d_{min} to P_C. The two nearest layers to P_C are $k_0 = FLOOR(z/v_z.z)$ and $k_1 = k_0 + 1$. Construct a 2D coordinate system on layer k_0, where the origin is the first lattice site of k_0 at $((k_0 \bmod 2) \cdot v_z.x, (k_0 \bmod 2) \cdot v_z.y, k_0 \cdot v_z.z)$. The coordinate of P_C in this coordinate system is (x_0, y_0):

$$\begin{aligned} x_0 &= x - (k_0 \bmod 2) \cdot v_z.x \\ y_0 &= y - (k_0 \bmod 2) \cdot v_z.y. \end{aligned} \tag{4.2}$$

Therefore, the nearest site on layer k_0 is $P_{FCC0} = (i_0, j_0, k_0)$:

$$\begin{aligned} i_0 &= FLOOR(x_0 + 0.5) \\ j_0 &= FLOOR(y_0 + 0.5) \end{aligned} \tag{4.3}$$

and the square distance d_0^2 from P_C to P_{FCC0} is

$$d_0^2 = (x_0 - i_0)^2 + (y_0 - j_0)^2 + (z/v_z.z - k_0)^2 * v_z.z^2. \tag{4.4}$$

Similarly, the coordinate of P_C in layer k_1 is (x_1, y_1):

$$\begin{aligned} x_1 &= x - (k_1 \textbf{ mod } 2) \cdot v_z.x \\ y_1 &= y - (k_1 \textbf{ mod } 2) \cdot v_z.y. \end{aligned} \tag{4.5}$$

The nearest site on layer k_1 is $P_{FCC1} = (i_1, j_1, k_1)$:

$$\begin{aligned} i_1 &= FLOOR(x_1 + 0.5) \\ j_1 &= FLOOR(y_1 + 0.5) \end{aligned} \tag{4.6}$$

and the square distance d_1^2 from P_C to P_{FCC1} is

$$d_1^2 = (x_1 - i_1)^2 + (y_1 - j_1)^2 + (z/v_z.z - k_1)^2 * v_z.z^2. \tag{4.7}$$

The nearest site P_{FCC} is:

$$P_{FCC} = (d_0^2 < d_1^2) * P_{FCC0} + (d_0^2 >= d_1^2) * P_{FCC1}. \tag{4.8}$$

4.2.3 Links and Neighbors

Given a lattice site $P_{FCC} = (i, j, k)$, the 12 links l_i and neighbors N_i ($i = 0, 1, \cdots, 11$) are defined in Table 4.1. The 12 links can be grouped into 3 sets. One set is located on layer i. The other two sets are located on layer $i-1$ and $i+1$. In Figure 4.2, the 3 sets of links are represented by red, green and blue arrows, respectively. Figure 4.2(b), 4.2(c) and 4.2(d) show the projection of the 3 sets on the plane $z = 0$. Note that the links are carefully numbered so that the inverse link of l_i is l_{11-i}, which simplifies the diffuse photon tracing.

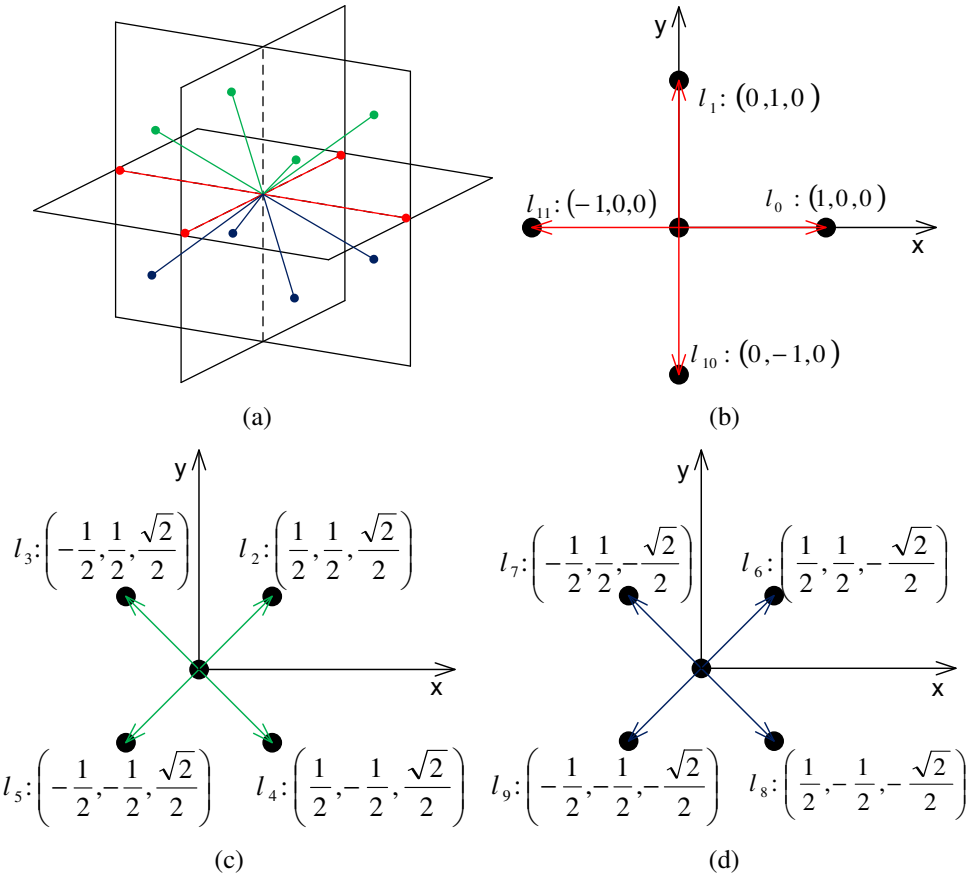


Figure 4.2: Projection of 12 links and neighbors of an FCC lattice site on the plane $z = 0$.

Table 4.1: Link vectors and neighbors of FCC lattice sites.

No.	Link vector	Neighbor
0	v_x	$(i+1, j, k)$
1	v_y	$(i, j+1, k)$
2	$v_z * (1,1,1)$	$(i+k \bmod 2, j+k \bmod 2, k+1)$
3	$v_z * (-1,1,1)$	$(i+k \bmod 2-1, j+k \bmod 2, k+1)$
4	$v_z * (1,-1,1)$	$(i+k \bmod 2, j+k \bmod 2-1, k+1)$
5	$v_z * (-1,-1,1)$	$(i+k \bmod 2-1, j+k \bmod 2-1, k+1)$
6	$v_z * (1,1,-1)$	$(i+k \bmod 2, j+k \bmod 2, k-1)$
7	$v_z * (-1,1,-1)$	$(i+k \bmod 2-1, j+k \bmod 2, k-1)$
8	$v_z * (1,-1,-1)$	$(i+k \bmod 2, j+k \bmod 2-1, k-1)$
9	$v_z * (-1,-1,-1)$	$(i+k \bmod 2-1, j+k \bmod 2-1, k-1)$
10	$-v_y$	$(i, j-1, k)$
11	$-v_x$	$(i-1, j, k)$

4.2.4 Closest Link

Given an arbitrary normalized vector $d = (x, y, z)$ in Cartesian space and a lattice site $P_{FCC} = (i, j, k)$, find the link l_i of P_{FCC} that has the minimum angle between d and l_i. (This can be used to trace specular photons.) This can be accomplished with the method described below:

First, construct a new coordinate system $x'y'z'$ by rotating the x and y axes $-\frac{\pi}{4}$ through axis z. The transformation matrix from coordinate system xyz to $x'y'z'$ is denoted as M:

$$M = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{4.9}$$

In this coordinate system, the 12 FCC lattice links can be grouped into 3 sets. As shown in Figure 4.3, the links in each of the sets are located in the $x'y'$, $y'z'$ and $z'x'$ plane, and colored in red, green and blue, respectively. Figure 4.3(a) shows the $x'y'$, $y'z'$ and $z'x'$ planes. Figure 4.3(b), 4.3(c), 4.3(d) show the 3 sets of links in red, green and blue, respectively.

Let $d' = M \cdot d$. Then check the projection of d' on the planes $x'y'$, $y'z'$ and $z'x'$. The links l_0, l_1, l_{11} and l_{10} are in the quadrant 0, 1, 2 and 3 of plane $x'y'$, respectively. The links l_2, l_5, l_9 and l_6 are in the quadrant 0, 1, 2 and 3 of plane $y'z'$, respectively.

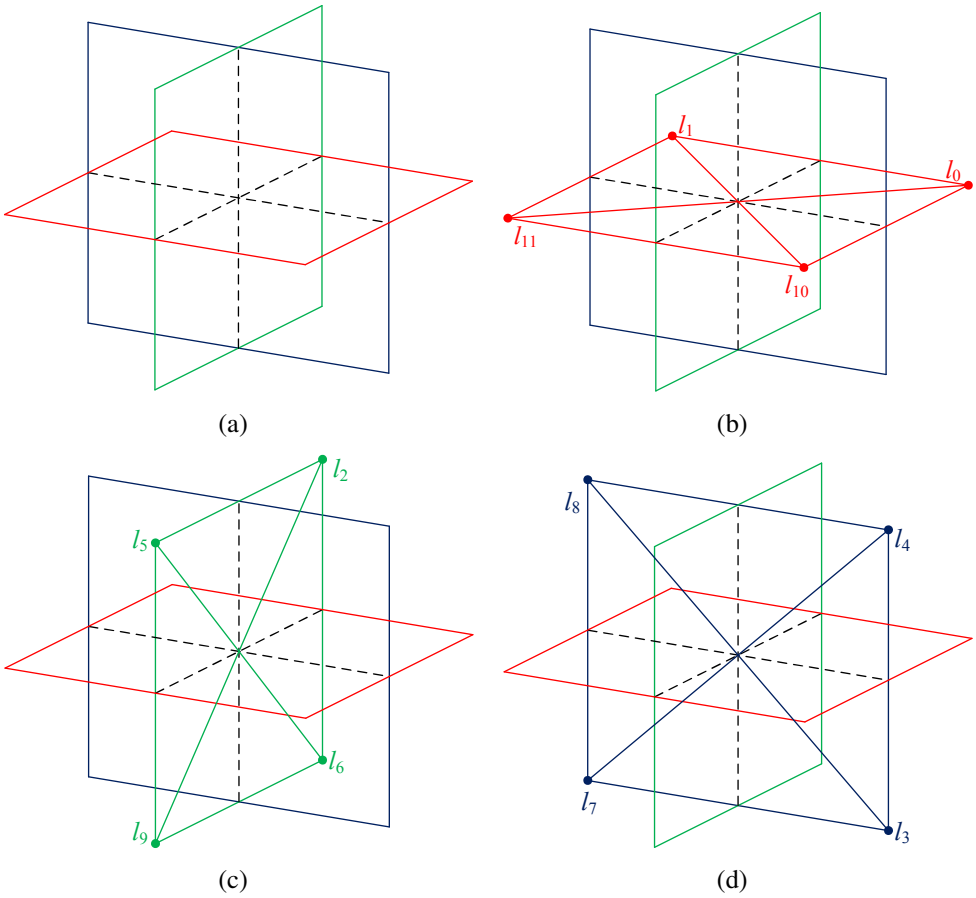


Figure 4.3: 12 links grouped into 3 sets in the transformed coordinate system.

Table 4.2: The positions of links in projection planes.

plane	quadrant	link
$x'y'$	0	l_0
$x'y'$	1	l_1
$x'y'$	2	l_{11}
$x'y'$	3	l_{10}
$y'z'$	0	l_2
$y'z'$	1	l_5
$y'z'$	2	l_9
$y'z'$	3	l_6
$z'x'$	0	l_4
$z'x'$	1	l_8
$z'x'$	2	l_7
$z'x'$	3	l_3

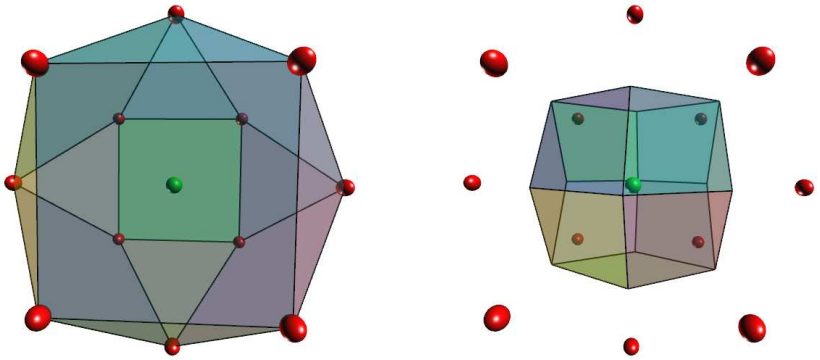
The link l_4, l_8, l_7 and l_3 are in the quadrant 0, 1, 2, 3 of plane $z'x'$, respectively. The correspondence is summarized in the Table 4.2.

The closest links $l_{x'y'}$, $l_{y'z'}$ and $l_{z'x'}$ on the three projection planes can be easily calculated by checking the sign of the transformed coordinate d'. The cosines between d' and $l_{x'y'}$, $l_{y'z'}$ and $l_{z'x'}$ can be calculated with dot products and the maximum cosine corresponds to the closest link.

Another simple method is to group all links in two sets: one including l_0, l_1, l_{11} and l_{10}, the other including all others. For the second set, each of them lies in one quadrant of the coordinate system xyz and the angle between the link and any of the xy, yz and zx planes is $\frac{\pi}{4}$. Therefore, the closest link of d is the one in the same quadrant of d. Define a 3D vector $b = d < 0$ such that $b.x = d.x < 0$, $b.y = d.y < 0$ and $b.z = d.z < 0$. The closest link is $l_{xyz} = l_{2+b.x+b.y\cdot 2+b.z\cdot 4}$. For the first set, the method described above can be used and the closest link is $l_{x'y'} = l_{d'.x<0+(d'.y<0)*10\cdot}$ Then, compare these two closest links in the two sets and choose the one with maximum dot product.

The geometric layout of the FCC lattice also gives rise to its higher angular discretization granularity than both CC and body-centered cubic (BCC) lattices, which is important for our rendering framework. Each site in the FCC lattice has direct links to a total of 12 nearest neighbors, in contrast to 8 and 6 in the BCC and

CC lattices, respectively. This is the best angular discretization rate that any 3D regular lattice can achieve, since in R^3 the maximum number of spheres of radius 1 that can simultaneously touch the unit sphere (i.e., the kissing number) is 12 [22]. This unique feature has important implications for sampling and interpolation, as we will discuss further below. For example, when a particle is scattered at an FCC lattice site, it has 12 possible moving directions, which is 50% more than a BCC lattice and 100% more than a CC lattice. In addition, the 12 links of an FCC lattice site are symmetric under rotation and reflection, which supports a relatively simple computational framework. Figure 4.4(a) shows the cuboctahedron defined by the 12 closest neighbors (red spheres) of a FCC site (green sphere), while Figure 4.4(b) shows the Voronoi cell of an FCC lattice site (green sphere), which is essentially a rhombic dodecahedron.



(a) (b)

Figure 4.4: (a) Cuboctahedron composed of 12 neighbors; (b) The Voronoi cell of the FCC lattice.

Finally, the reciprocal lattice of FCC yields its representation in the frequency domain, which is essentially a BCC lattice [150]. As it will be described in the next section, this property gives rise to its near-optimal sampling behavior that is capable of reconstructing original signals with a minimum number of samples. It has been shown in the literature [150] that the FCC sampling scheme requires 13.4% and 23% fewer samples in R^2 and R^3 domain compared to CC lattices, respectively. Thus, the FCC lattice presents a much more efficient spacial sampling scheme over the traditional CC lattice.

4.3 Sampling on FCC

In any lattice used for volume visualization, the lattice sites are discretized or sampled from R^n, and an efficient, preferably optimal, sampling scheme is paramount. An optimal lattice structure captures information in the hyper-volume R^n using the least number of sampling points. Assuming an isotropic and band-limited sampling function, the resulting frequency support of a sampling point is a hyper-sphere, surrounded by a set of alias replicas. Hence, the most efficient sampling scheme arranges the replicated (hyper-spherical) frequency response as densely as possible in the frequency domain to avoid overlapping of the aliased spectra. As demonstrated in multi-dimensional signal theory [29] an optimal sampling scheme is obtained when the frequency response of the sampling lattice is an optimal sphere packing lattice [22]. Optimal sampling lattices can achieve up to 13.4%, 29.3%, and 50% of savings in 2, 3 and 4 dimensions, and they have been used in volume visualization [33, 112, 150] with high quality image results.

In three dimensions there are infinite optimal sphere packings including the FCC lattice and the HCP (hexagonal closed packing). The spatial equivalent of the FCC lattice in the frequency domain is the BCC lattice, which is the inverse Fourier transform of the FCC and vice-versa. The FCC lattice in the spatial domain corresponds to the BCC lattice in the frequency domain which is not an optimal sphere packing, and the FCC lattice achieves about 23% of savings over the CC lattice in terms of sampling efficiency. It was chosen for our global illumination framework because it is the lattice that maximizes uniform angular discretization with its kissing number of 12. The HCP is another candidate with an optimal kissing number of 12. In strict mathematical definition, HCP is not a lattice but can be defined as the union of the lattice L with generation vectors $(1,0,0)$, $(1/2,\sqrt{3}/2,0)$ and $(0,0,\sqrt{8/3})$ and the translate $L+(1/2,1/\sqrt{12},\sqrt{2/3})$. However, it has a bias towards the z-direction. For any link with direction $d=(d_x,d_y,d_z)$ such that $d_z \neq 0$, the link of direction $-d$ does not exist. When a ray or photon moves from one of such links and is not absorbed or scattered at the lattice site, it cannot continue straight along its original direction. Therefore, HCP is not symmetric and thus unsuitable for use in our framework.

Given the initial assumption that the represented function is hyper-spherically

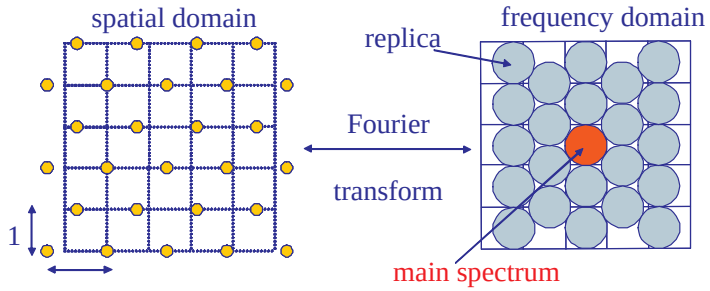


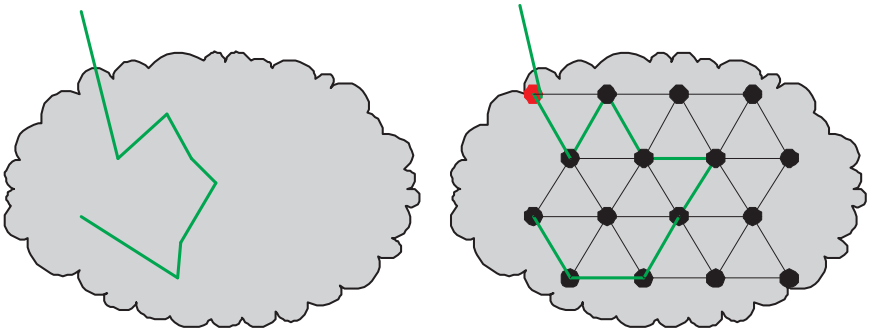
Figure 4.5: Hexagonal lattice and its Fourier transformation.

band-limited, the ideal choice for the reconstruction function is also a radially symmetric kernel. We have studied a set of Gaussian reconstruction filters and have found that the relatively narrow Gaussian $f(r) = e^{-2r^2}$, offers good frequency behavior and reasonable overlap between neighboring sites.

4.4 Diffuse Photon Tracing

For volumetric objects where the dominant effect is diffusion, a new algorithm to trace photons on the lattice links has been developed. The volumetric objects are sampled in a finite region of the FCC lattice. As shown in Figure 4.6 on a hexagonal lattice, when a photon is emitted from a light source, the nearest lattice site to the first hit point on the lattice boundary is calculated and the moving direction of the photon is discretized to one of the link directions. The photon will be traced on the links between lattice sites and its path is composed of line segments of lattice links. Figure 4.6(a) shows the photon path in green color in the real medium. Figure 4.6(b) shows the photon path on a hexagonal lattice. The circles represent lattice sites and the black line segments represent lattice links. The red circle is the nearest lattice site to the photon intersection point with the medium boundary. The photon path is composed of the green lattice links.

A photon emitted from the light sources has an arbitrary direction ω. The accurate photon direction is stochastically converted to one of the lattice links at the volume boundary. An FCC lattice site x_i has 12 closest neighboring sites of equal



(a) (b)

Figure 4.6: Illustration of tracing photons on a hexagonal lattice.

distance, which forms a cuboctahedron as shown in Figure 4.4(a). The ray originated from site x_i with direction ω intersects the cuboctahedron centered at point v on one of its 14 faces. Denoting the vertices of the face containing v as $v_0, v_1, \cdots, v_m$ $(m = 2, 3)$ in counterclockwise order and letting $\omega_k = v_k - x_i$, the probability of the photon direction to be changed to ω_k is defined as the barycentric coordinates [38]:

$$p_k = \frac{w_k}{\sum_k w_k}, \quad w_k = \prod_{t \neq k-1,k} A(v, v_t, v_{t+1}) \tag{4.10}$$

where $A(v, v_t, v_{t+1})$ is the area of triangle v, v_t, v_{t+1}. The barycentric coordinates of v are well defined with Equation 4.10 because all the faces of the cuboctahedron are regular polygons. Equation 4.10 implicitly replies on ω because the intersection point v is decided by ω. p_k is a continuous function of ω. It guarantees that $p_k = 1$ when $\omega = \omega_k$.

When arriving at a lattice site x_i from direction ω_j the photon might be absorbed, be scattered or continue moving along the extended link of ω_j. The absorption and the scattering coefficients, $\sigma_a(x_i, \omega_j)$ and $\sigma_s(x_i, \omega_j)$, represent the probabilities of such events. (Please note that this approximation is only correct when the link length l is small compared to $1/\sigma_t$. The actual absorption probability should be $exp(-\int_0^l \sigma_a(x_i, \omega_j))ds$. Because the links on an FCC lattice are of identical length, σ_a, σ_s and σ_t are used here for convenience.) The Russian roulette technique [124] is used to determine whether the photon is absorbed, scattered or transmitted. In detail, the program generates a random number $\xi \in [0, 1)$ and the photon:

$$\begin{cases} \text{is absorbed} & \text{if } \xi \in [0, \sigma_a(x_i, \omega_j)), \\ \text{is scattered} & \text{if } \xi \in [\sigma_a(x_i, \omega_j), \sigma_a(x_i, \omega_j) + \sigma_s(x_i, \omega_j)), \\ \text{is transmitted} & \text{if } \xi \in [\sigma_a(x_i, \omega_j) + \sigma_s(x_i, \omega_j), 1). \end{cases} \tag{4.11}$$

In contrast to the deterministic procedures of absorption and transmission, stochastic scattering requires further processing. In traditional methods, a phase function $f(x, \omega, \omega')$ is utilized to describe the probability of a photon being scattered at location x with an input direction ω and an output direction ω'. The computation of the new direction is performed using importance sampling. In the widely used models such as the Schlick model [8], the probability depends on $\cos(\theta) = \omega \cdot \omega'$ only, thus the importance sampling returns the value of $\cos(\theta)$. In order to compute ω', a local coordinate system at the scattering position has to be constructed to convert spherical angles to direction vectors, which is computationally expensive.

In the lattice illumination framework, the computation of the scattering process is greatly simplified because photons only move along discretized lattice links. Here, the continuous phase function $f(x, \omega, \omega')$ is discretized to $f(i, j, k)$, which represents the probability of a photon located at site x_i being scattered from the input link ω_j to the new output link ω_k. In practice, this discrete phase function is constructed as a 2D table of $n \times n$ resolution to represent all possible combinations of input/output links on a lattice site ($n = 12$ for FCC lattices). The generation of the discrete phase function via sampling and normalizing the continuous phase function is described by the following equation:

$$f(i, j, k) = \frac{f(x_i, \omega_j, \omega_k)}{\sum_{t=0}^{n-1} f(x_i, \omega_j, \omega_t)}. \tag{4.12}$$

A more accurate method is to calculate the integral over the angular space Ω defined in Equation 4.13, which can be solved numerically for arbitrary continuous phase function.

$$\begin{aligned} f(i, j, k) &= \int_{\Omega} closest(\omega, \omega_k) f(x_i, \omega_j, \omega) d\omega \\ closest(\omega, \omega_k) &= \begin{cases} 1 & \text{if the closest link to } \vec{d} \text{ is } \omega_k, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{4.13}$$

Importance sampling of discrete phase functions is simply a binary search for a given random number ξ such that $\sum_{t=0}^{k} f(i,\omega_j,\omega_t) \leq \xi$ and $\sum_{t=0}^{k+1} f(i,\omega_j,\omega_t) > \xi$. The complexity of this binary search is $O(\log(n))$, which is very efficient because n is usually very small ($n = 12$ for FCC lattices).

After determining the photon behavior of each encounter, the activity information on the lattice sites is saved. Basically, a stored photon represents a possible light path from the light sources to its location. This information is used in the following rendering pass, where the irradiance of sampling positions is estimated with the stored light paths within a small neighborhood. The lattice-based framework enables us to store photons in a 3D array and the position of the photons is implicitly defined by their index in the array. Moreover, the quantized directions are encoded in a byte using the link index. Since the link vectors are known a-priori, an optimized solution for photon direction storage is adopted, where photons with the same direction are grouped together. Here, a 1D array $E(\omega_j)$ of n elements is used for a lattice site x_i, such that $E(\omega_j)$ is the total energy of the photons at x_i with direction ω_j. Due to the employed Russian roulette technique [124], the photon energy does not change until it is absorbed. Therefore, the stored photons all have the same energy and only the integral number of photons need to be stored at link (i, j). Given the maximum possible number of photons stored on the links, the unsigned byte or short format can be used to represent the actual photon counts instead of storing individual floating point energy values. In other words, all photons are stored in a 4D integer array $E(i,\omega_j)$ with three dimensions of site position and one dimension for link direction.

After the recording of diffuse photons, rays are traced from the image plane into the FCC lattice to collect irradiance values. At each sampling point x, the radiance is estimated by the photons inside a small spherical region centered at x. With the 4D array of photon numbers, the radiance in Equation 1.15 is calculated with the following simplified formula:

$$R(x,\omega) = \sum_{X} \sum_{j} \sigma_s(x,\omega_j) f(x,\omega,\omega_j) g(x'-x) E(x',\omega_j) dx' \tag{4.14}$$

where ω is the reverse ray direction, f is the continuous phase function and X is the set of lattice sites in the search region. g is a normalized smoothing filtering

function used for removing high-frequency noise. Because the lattice sites are positioned regularly, searching for the lattice sites in a sphere is simple and efficient. In the experiments, the medium is isotropic and the phase function only depends on the angle between the ray direction and the lattice link ω_j. The dot product of ω and ω_j can be pre-computed and reused for all the sampling points on one ray in the rendering process, which yields an efficient implementation of the radiance estimation.

This new algorithm greatly simplifies the computation of photon-volume interaction and photon storage. Therefore, the program can trace millions of photons in a short time, which is good for improving image quality by removing the stochastic noise and variance without excessive smoothing in the radiance estimation. Moreover, this method is general and can use arbitrary phase functions including those of strong backward scattering.

4.5 Specular Photon Tracing

The method described in Section 4.4 is capable of calculating multiple scattering events for participating media and volumes where diffusion is the dominant effect. However, specular reflection and refraction may exhibit ray effects when discretized rays hit smooth specular surfaces. This effect has been discovered by the radiative heat transfer community [15] and found to be caused by the discretization of scattering directions when accurate directions are needed for specularity.

To mitigate this ray effect, an enhanced algorithm called specular photon tracing has been developed, where every photon is associated with its accurate direction ω and start position x. In each time step, the photon moves on the FCC lattice and the new sampling position is calculated by $x = d \times \omega$ where d is the step size. The lighting properties σ_a and σ_s are sampled to decide whether the photon is absorbed, scattered or transmitted at x (Equation 4.11). If the photon is scattered, the new direction ω is computed with the continuous phase function. The Russian roulette technique is again used to avoid photon energy change.

The O-Buffer data structure [129] is used to store the photon information compactly, where each lattice site stores a sequence of photons in its Voronoi cell (Figure 4.4(b)). For each stored photon at position x, the nearest lattice site x_i is computed and only the offset o from x to x_i is stored. The offset o is quantized into 256 levels in each axis so that o can be compactly represented in 3 bytes. This 3-byte representation increases the photon position accuracy by 256 times of the lattice resolution, while it only needs 25% of the storage space of the traditional floating point representation. Because the search radius for the radiance estimation is usually much larger than the link length, it is good enough for most rendering applications. The Voronoi cell (Figure 4.4(b)) of an FCC lattice site is a rhombic dodecahedron. Assuming a unit distance between neighboring sites, the distance from the lattice site to the vertices of its Voronoi cell is $\frac{\sqrt{2}}{2}$. The maximum error introduced by this offset quantization scheme is $\frac{\sqrt{3}}{2} \cdot \frac{1}{255} \cdot \frac{\sqrt{2}}{2}$.

For photon direction encoding, vectors ω are converted to spherical coordinates and represented with 2 bytes [60]. Because the photon energy does not change, only one byte is used to record the color channel of the photon. In total, the storage space of one photon is just 6 bytes.

In the rendering pass, the radiance at each sampling point is estimated with the photons stored in the lattice photon O-Buffer. For a spherical search neighborhood S with radius r, the lattice sites in the sphere S' of radius $r+\frac{\sqrt{2}}{2}$ are retrieved because the maximum distance from a lattice site to the photons stored in it is $\frac{\sqrt{2}}{2}$. Then, the photons stored in these lattice sites are visited. If the distance to the sampling point x is larger than r, the photon is discarded. The radiance at x is summed over all the photons inside S with:

$$R(x,\omega) = E\sum_{p} \sigma_s(\omega_p,x) f(x,\omega,\omega_p) g(x_p - x) \tag{4.15}$$

where x_p and ω_p are the photon position and direction, respectively. The term E is the energy of the photon, which is the same for all photons since the Russian Roulette scheme is used.

The diffuse photon tracing algorithm is capable of tracing multi-million photons in seconds. With the FCC lattice, the photons move on the lattice links and are scattered only on the lattice sites. Therefore, the most time consuming steps in traditional methods such as sampling the lighting properties, calculating scattered

directions with phase functions are greatly simplified. Its major disadvantage is the ray effect, which cannot be neglected in certain cases, for example, in reflection, refraction or scattering on specular surfaces, and hard shadows. The specular photon tracing solves this problem but is much slower.

4.6 Implementation

The algorithms presented in Sections 4.4 and 4.5 have been implemented to render participating media and volumetric datasets. Since the reconstruction process in the current scanning modalities such as MRI and CT only produce rectilinear data, the FCC data we used are generated by resampling existing rectilinear volumes or voxelizing geometric objects. Currently, a windowed sinc filter has been used, although a better filter such as B-spline could be used.

The incremental triangle voxelization method [27] has been to voxelize polygonal models to the FCC lattice. The original method was proposed for volumes of CC lattices, but the employed distance-based method enables its direct application to the FCC lattice.

For surface voxelization, the volume density of a lattice site p is determined by the distance between p and its closest triangle. Each triangle is processed in the following manner. For each lattice site p in the neighborhood of the triangle, the distance d between p and the triangle is computed. The distance d is positive if p is in the normal direction of the triangle, or negative if p is in the reverse direction. If $|d|$ is smaller than the previously stored absolute value of the distance, $|d|$ replaces the previous stored value and the density of p is updated with the following equation:

$$density = \begin{cases} 1 & \text{if} \quad d < -W, \\ 0 & \text{if} \quad d > W, \\ 0.5 \times (1 - \frac{d}{W}) & \text{otherwise}, \end{cases} \tag{4.16}$$

where W is the width of the oriented box filter. While for a CC lattice the optimized width was estimated to be $2\sqrt{3}$ voxel units [27], we estimate that for an FCC lattice, the optimized width of the filter is decreased to 2 lattice units. Based on the surface voxelization result, the interior of the solid is voxelized using seed growing.

For inhomogeneous participating media, such as clouds and smoke, the absorption coefficient $\sigma_a(x)$ and scattering coefficient $\sigma_s(x)$ typically do not rely on the light direction, although our algorithm is capable of rendering anisotropic media. The volume data of the medium usually defines the density field $\rho(x)$ of particles. We assume that the $\sigma_a(x)$ and $\sigma_s(x)$ are proportional to the local density $\rho(x)$. For FCC lattices where the lattice links have unit lengths, it implies:

$$\sigma_a(x,\omega_j) = \sigma_a\rho(x), \text{ and } \sigma_s(x,\omega_j) = \sigma_s\rho(x). \tag{4.17}$$

where σ_a and σ_s are user-defined scaling coefficients. In real world phenomena, most practical participating media are isotropic and the phase function $f(x,\omega,\omega')$ does not vary upon position x. In our implementation, $f(x,\omega,\omega') = f(\omega\cdot\omega')$ is described with the Schlick model [8]. The participating medium is represented with an FCC lattice of densities and all the coefficients are calculated by scaling $\rho(x)$.

Our algorithms support chromatic participating media, where the coefficients $\sigma_a(x,\lambda)$ and $\sigma_s(x,\lambda)$ are wavelength-dependent. It is implemented by defining scaling factors of absorption and scattering in RGB channels. The photons emitted from the light sources can be red, green or blue randomly. The photon tracing program calculates $\sigma_a(x,\lambda)$ and $\sigma_s(x,\lambda)$ with proper scaling factors in the color channel of the traced photons. In the rendering pass, the opacity value α of a sampling point is the average extinction of those in three channels:

$$\alpha = \frac{1}{3}(\sigma_t(x,\lambda_r)^d + \sigma_t(x,\lambda_g)^d + \sigma_t(x,\lambda_b)^d) \tag{4.18}$$

where d is the step size.

For general volume datasets, transfer functions have been exploited to define lighting properties from the density field. Some 2D transfer functions might also use the gradient information. Our framework is general and capable of integrating any transfer function as long as the input data (density, gradient or any other data) of the transfer function is defined on the lattice. In our current implementation, a 1D transfer function is defined for σ_a and σ_s in each RGB channel. Equation 4.18 can be used to compute the opacity α or a separate transfer function can be defined for α.

Following the photon tracing computation, a single-pass ray-casting approach is employed on the GPU to render the diffuse photon tracing results. An algorithm similar to [73] is used, except that the sampled volume density is used for obtaining the scattered coefficient as well as transparency values to composite the final radiance results. Here, in addition to the density volume, an additional photon storage volume that records the diffuse photon distribution on each lattice site is sampled to composite ray values. Both the density and the diffuse photon volumes are FCC lattices, stored and indexed as described in Section 4.2. The diffuse photon storage table is essentially a 12-element array, each of which records the number of photons stored along 12 different lattice links. A two-byte unsigned short is allocated for each such node to provide a count of up to 65536. To compute the radiance estimation at each sampling point, dot-products of each lattice link with the current viewing ray are used to weight individual diffuse radiance value, which is given by indexing the photon distribution previously computed on the lattice. The 12 weighted values are then summed up to yield the final contribution. Filtering on all lattices uses the Gaussian kernel of size 2 and the sum is normalized at the end.

4.7 Results

The presented algorithms have been implemented on a 3.4GHz PC with 3GB memory and a Geforce 8800 GTX graphics card. All the resulting images are of 512×512 resolution and cropped in Figures 4.7-4.11.

Figure 4.7 demonstrates the rendering results of participating media with our algorithms. A single light of white color is placed on top of the smoke dataset. The original data is $100 \times 100 \times 40$ and we sampled it into a $108 \times 100 \times 56$ FCC lattice with a windowed sinc filter. The algorithms used are: (a) ray casting; (b) our diffuse photon tracing; (c) our diffuse photon tracing with strong backward scattering; (d) our specular photon tracing; (e) and (f) our multi-channel diffuse photon tracing. Table 4.3 gives the time of different algorithms used to render the corresponding images. From left to right, the columns represent the algorithm used, the number of photon traced (in millions), photon tracing time (in seconds), and rendering time (in seconds) on the CPU and the GPU. (DPT: diffuse photon tracing; SPT: specular photon tracing.)

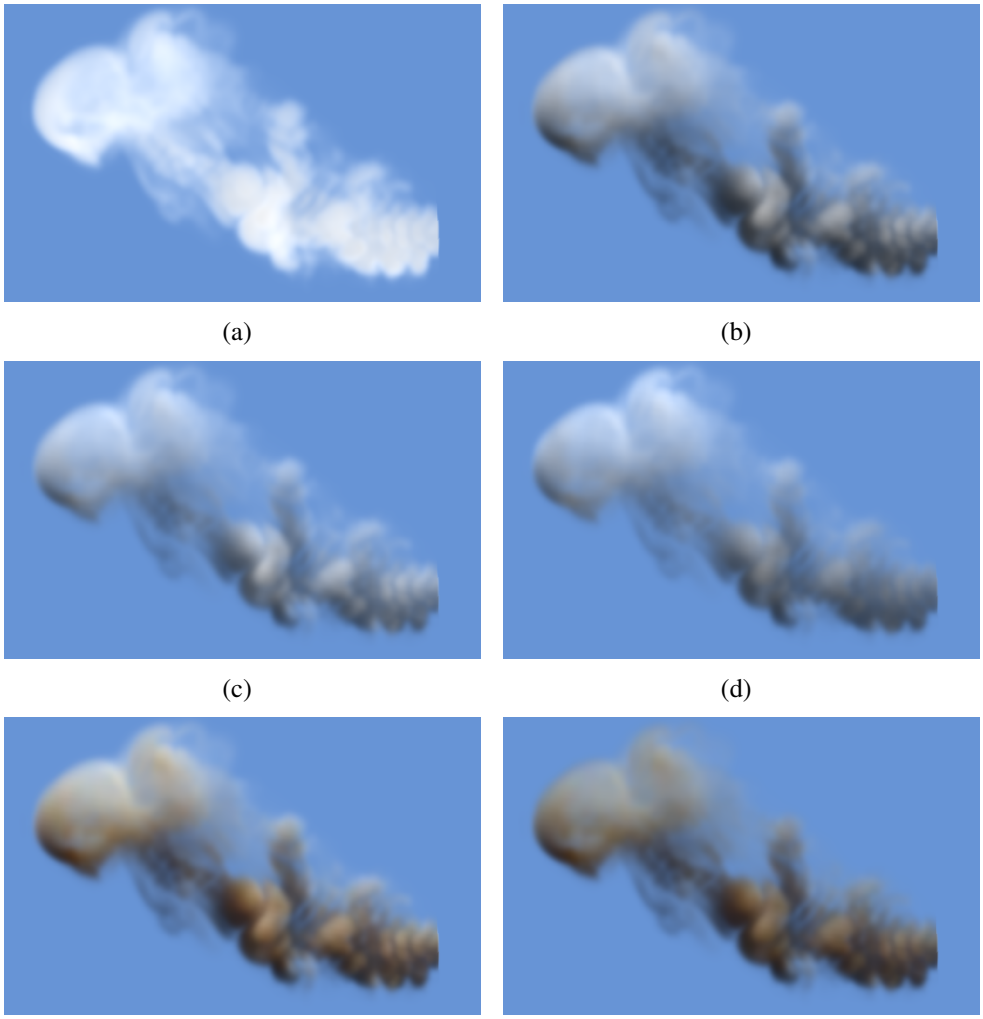


Figure 4.7: Inhomogeneous smoke rendered with global illumination (multiple scattering) and an anisotropic phase function.

Table 4.3: Times used to render the smoke in Figure 4.7.

	Algorithm	Photon count	Photon tracing	CPU rendering	GPU rendering
Figure 4.7(a)	ray casting	N/A	N/A	20.6	1.0
Figure 4.7(b)	DPT	1.0	11.3	48.1	2.4
Figure 4.7(c)	DPT	1.0	11.7	48.1	2.4
Figure 4.7(d)	SPT	0.1	27.0	729.0	N/A
Figure 4.7(e)	multi-channel DPT	3.0	48.5	52.7	2.6
Figure 4.7(f)	multi-channel DPT	3.0	48.6	52.7	2.6

In Figure 4.7(b), the eccentricity coefficient k of the Schlick phase function is set to 0.2. The absorption and scattering coefficients are $\sigma_a = 0.08$ and $\sigma_s = 0.2$, respectively. Figure 4.7(c) uses the same coefficients except that the eccentricity k is -0.5 for strongly backward scattering. The photon tracing of 1 million photons and subsequent rendering passes cost about 11 and 48 seconds on the CPU, respectively. With GPU acceleration, the time of the rendering stage is reduced to 2.4 seconds. This performance is significantly faster than the original photon mapping method [63], where tracing 10K photons in a cloud model of similar size takes 8 seconds on an HP computer of 16 180MHz PA-8000 processors, while rendering a 1024 pixel wide image takes 92 seconds. Note that although a higher resolution is used in Jensen and Christensen's method [63], the complexity of the algorithms is mainly determined by the number of photons, of which our example generates 100 times more. A major further enhancement of our implementation can be achieved by incorporating empty space skipping or adaptive sampling techniques that are used by Jensen and Christensen [63].

Figure 4.7(d) has been rendered using specular photon tracing with the same medium properties as Figure 4.7(b), and the rendering time is similar to traditional photon mapping methods [63]. With our compact FCC O-Buffer data structure, 5.8 million stored photons only consume 35MB of memory space. The search radius for radiance estimation is changed from 2.0 to 3.0 to remove the stochastic variance. Given the same number of photons, diffuse photon tracing is approximately 21 times faster than specular photon tracing and the corresponding rendering pass is 15 times faster. We observed that the image produced from our lattice-based framework using diffuse photon tracing (Figure 4.7(b)) is comparable to the image using

specular photon tracing (Figure 4.7(d)), and has a similar quality and appearance of those computed with traditional photon mapping methods (such as Figure 12.10 of [124]). However, our framework has much better performance.

In Figure 4.7(e), the absorption coefficient is wavelength dependent and the values in RGB channels are $\sigma_a = (0.08, 0.15, 0.3)$, while the scattering coefficient is the same as in Figure 4.7(b). The time of the rendering pass increases to 52.7 seconds mainly because the radiance estimation is performed in 3 channels. In Figure 4.7(f), the scattering coefficient in the blue channel has been changed to 0.4 and the eccentricity coefficient k has been changed to 0.5. In Figures 4.7(e) and 4.7(f), the upper part of the smoke is grey but the lower part under the shadow of the upper part becomes orange because of the different absorption and scattering coefficients in the RGB channels.

Figure 4.8: Cloud rendered with our diffuse photon tracing.

Figure 4.8 demonstrates another example of participating media, in which the resolution of the data is $96 \times 74 \times 143$. The eccentricity coefficient k of the Schlick phase function is 0.2. The absorption and scattering coefficients are $\sigma_a = 0.05$ and $\sigma_s = 0.1$, respectively. One million photons have been traced in 12.7 seconds and the rendering pass amounted to 98.5 seconds on the CPU and 5.5 seconds on the GPU.

Figure 4.9 displays the foot of the visible human CT data. The original data is 128^3 and the resampled FCC lattice is $128 \times 128 \times 180$. Figure 4.9(a) is rendered using a ray casting method with local illumination. Figure 4.9(b) has been rendered using our diffuse photon tracing algorithm. The bone appears semi-translucent and brighter and has soft self-shadows due to multiple scattering. In Figure 4.9(c), the muscle and soft tissue are displayed with red color with the absorption coefficient

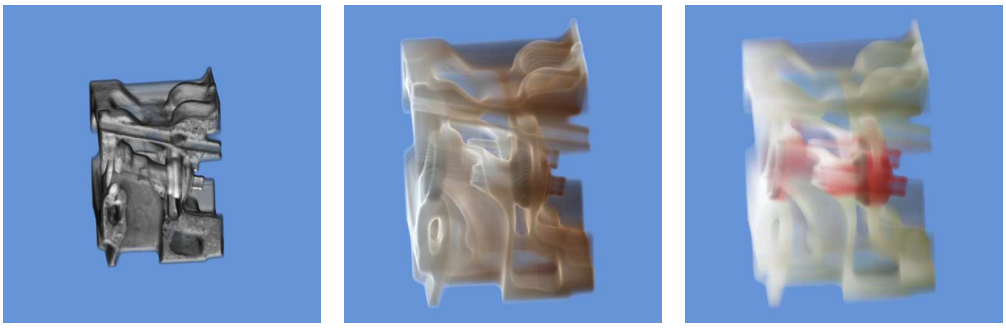
similar to 4.9(a).

Figure 4.10 and Figure 4.11 are the rendering results of the engine and lobster data, respectively. The original engine data is $128 \times 128 \times 64$ and the resampled FCC lattice is $136 \times 136 \times 98$. Figure 4.10(a) has been rendered using ray casting with local illumination and Figure 4.10(b) and 4.10(c) have been rendered with our framework and indeed the objects appear substantially different. In Figure 4.10(b), the material absorbs green and blue photons faster and becomes more red gradually through the light direction. In Figure 4.10(c), the high density region appears saturated red for emphasis and the surrounding region is less saturated red due to color bleeding. The material of the objects is easily controlled and changed using user specified transfer functions. The original lobster data is $128 \times 127 \times 28$ and the resampled FCC lattice is $114 \times 113 \times 35$. In Figure 4.11(b), the lobster shell absorbs green and blue photons faster than red ones and scatters red photons more than green and blue ones. The shell appears red and the muscle casts soft shadows onto itself. Figure 4.11(c) displays the data from a different camera position.

(a) (b) (c)

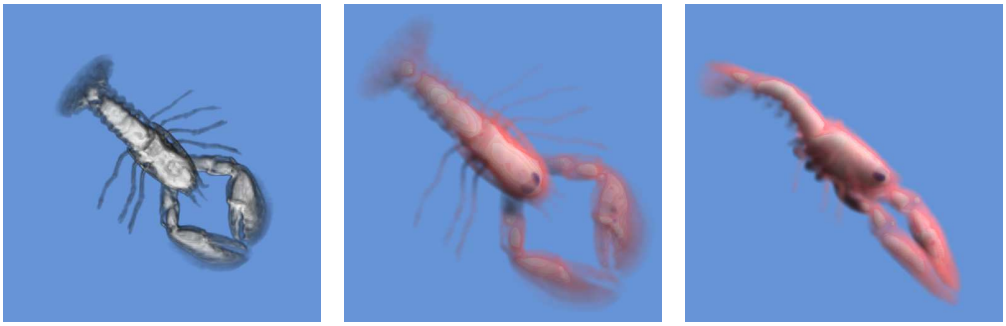
Figure 4.9: Global illumination of a CT scan of the visible human foot.

In the presented framework, a photon is saved at every step of the first pass, regardless of whether it is scattered or not. A stored photon represents a possible path from the light sources. In the ray tracing pass, the radiance estimation actually calculates the density of photon paths at the sampling positions. In contrast, the traditional photon mapping method uses the probabilistic sampling technique to calculate the step size, and the expected step size is $1/\sigma_t$. Usually σ_t is small and tracing a photon can generate many more stored photons in our algorithms than in photon mapping. Moreover, the FCC lattice provides a more efficient data structure to store photons. In diffuse photon tracing, each lattice site stores multiple photons.



(a) (b) (c)

Figure 4.10: Global illumination of an industrial CT scan of an engine.



(a) (b) (c)

Figure 4.11: Global illumination of a CT scan of a lobster.

Table 4.4: Rendering time of the foot, engine and lobster data in Figures 4.9, 4.10 and 4.11.

	Algorithm	Photon count	Photon tracing	CPU rendering	GPU rendering
Figure 4.9(b)	DPT	1.0	27.0	98.0	4.7
Figure 4.9(c)	multi-channel DPT	3.0	86.1	118.6	5.8
Figure 4.10(b)	multi-channel DPT	9.0	136.4	110.6	5.4
Figure 4.10(c)	multi-channel DPT	9.0	147.9	110.7	5.4
Figure 4.11(b)	multi-channel DPT	3.0	26.9	40.7	2.0
Figure 4.11(c)	multi-channel DPT	6.0	53.9	37.9	1.9

In radiance estimation, the contribution of multiple photons on a lattice link is computed jointly. Suppose the radiance estimation searches photons in the spherical neighborhood S, and there are k_0 sites and k_1 photons inside S. It costs $O(k_0)$ time for diffuse photon tracing and $O(k_0+k_1)$ for specular photon tracing. However, it costs $O(k_1+\log n)$ time with the k-d tree data structure in photon mapping, where n is the total number of photons. Also, the k-d tree need to be built before the rendering pass, which costs $O(n\log n)$ time. However, our algorithms do not need such a preprocessing step.

There are other simplified lighting models for participating media and volumes [51, 52, 69, 133]. For all these methods, only forward scattering is considered and lighting values are propagated from slice to slice. The value of each pixel is calculated by sampling and attenuating neighboring pixels (up to 4) on the previous slice. In other words, forward scattering is calculated in several directions in the 2π solid angle. Our algorithms can handle scattering within the entire 4π solid angle. Also, our methods can store photons from multiple light sources in one volume, while previous methods do not support multiple light sources. The method of Max [96, 98] is more accurate than ours but runs slowly.

The idea of tracing photons on the lattice links is general and might be applied to other lattices such as the CC lattice. However, a CC lattice site only has 6 nearest neighbors, which is not enough for discretizing some kinds of phase function. Consider a strongly forward scattering phase function (for example, the eccentricity coefficient of the Schlick phase function is large). On the CC lattice, a photon arriving from a link can only move forward on its original direction or be scattered to 4 directions perpendicular to its incoming direction. The resulting image will be very similar to that rendered with single scattering. Although we can add links between the secondary or tertiary neighbors, this solution needs to calculate and store the absorption and scattering probabilities of links with varying length (1, $\sqrt{2}$ and $\sqrt{3}$), thus making the photon tracing algorithm more complicated and time consuming. Instead, a FCC lattice site has 12 nearest neighbors, which is sufficient in photon tracing (in previous systems, only 4 or 5 directions are used for forward scattering). The link length in the FCC construction is uniform, which greatly simplifies the computation and storage: the absorption and scattering probabilities on 12 links of a site are the same and stored only once on the site. Moreover, the FCC lattice has

better sampling efficiency than the CC lattice. With the same number of sites, FCC captures 23% more information. In addition, the maximum distance of an arbitrary point to its nearest site is $\frac{\sqrt{2}}{2}$ in FCC instead of $\frac{\sqrt{3}}{2}$ in CC, which means 18.4% less quantization error of photon positions with the O-Buffer data structure.

Chapter 5

Volume Rendering for Virtual Colonoscopy

In this chapter, an important application of 3D CC lattice volume rendering, computer aided polyp detection (CAD) for virtual colonoscopy, is presented. Virtual colonoscopy is based visualizing the CT scanning of the patient's abdomen, which is a 3D CC lattice volume. The CAD pipeline uses a GPU accelerated 3D CC lattice volume rendering technique to calculate the electronic biopsy image of the colon, which reduces the information from a 3D CC lattice to a 2D CC lattice. Then, the image analysis methods are used on the 2D CC lattice for polyp detection. Because of the regularity of 2D CC lattice, analyzing the electronic biopsy image is more efficient than the conventional shape analysis methods executed on the triangle mesh of the colon surface.

Colorectal cancer is the second leading cause of cancer-related deaths in the United States. Most colorectal cancers are believed to arise within benign adenomatous polyps that develop slowly over the course of many years [125]. Evidence-based guidelines recommend the screening of adults who are at average risk for colorectal cancer, since the detection and removal of adenomas has been shown to substantially reduce the incidence of cancer and cancer-related mortality. Therefore, some researchers have advocated screening programs to detect polyps with a diameter of less than one centimeter [91]. However, most people do not follow this advice because of the discomfort and inconvenience of the traditional optical colonoscopy (OC).

To encourage people to participate in screening programs, virtual colonoscopy (VC), also known as computed tomographic colonography (CTC), has been proposed and developed to detect colorectal neoplasms by using a computed tomography (CT) scan. VC is minimally invasive and does not require sedation or the insertion of a colonoscope, though a minimal bowel preparation is necessary. VC exploits computers to reconstruct a 3D colon model from the CT scans taken of the patient's abdomen, and create a virtual fly-through of the whole colon to help radiologists navigate the model for diagnosis. Pickhardt et al. [125] have demonstrated that the performance of a VC compares favorably with that of a traditional optical colonoscopy. Because of the complex structure of the colon surface, the inspection is prone to errors, and the radiologist needs to navigate forth (from rectum to cecum) and back (from cecum to rectum) to improve the accuracy of the inspection. A single examination usually generates 400-700 512×512 CT images, which need 10-15 minutes to be interpreted by a radiologist conducting 3D VC [65].

The anticipated long interpretation effort of the VC screening procedure suggests a computer-aided detection (CAD) approach. A CAD scheme that automatically detects the locations of the potential polyp candidates could substantially reduce the radiologists' interpretation time and improve their diagnostic performance with higher accuracy. However, the automatic detection of colonic polyps is a very challenging task because the polyps can have various sizes and shapes. Moreover, there are numerous colon folds and residual leftover colonic materials on the colon wall that mimic polyps and could result in false positives (FPs). A CAD scheme should have the ability to identify true polyps and eliminate the FPs.

A novel pipeline for computer aided polyp detection based on CC lattice volume rendering has been developed. Traditionally, the polyps are detected by the shape features such as shape index and curvedness. However, the computation of shape features is very expensive on the colon triangular mesh. The new CAD pipeline uses direct volume rendering to generate electronic biopsy images, which reduces the information on a 3D CC lattice to a 2D CC lattice. Then the polyp detection is implemented by image analysis techniques on the 2D CC lattice, which is much faster than calculating shape features in 3D.

5.1 The CAD Pipeline

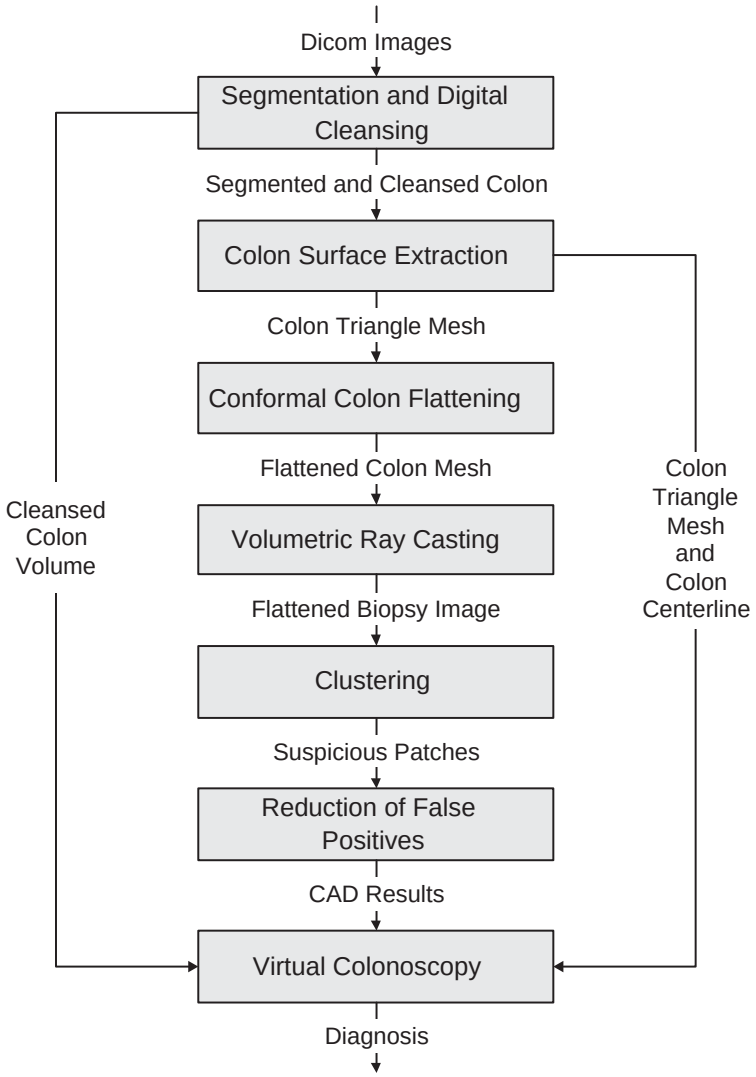


Figure 5.1: CAD pipeline.

Figure 5.1 shows the flow of the pipeline CAD. First, for segmentation and digital cleansing of the colon, an iterative partial volume segmentation algorithm is applied. Then, a topologically simple colon surface is extracted for conformal colon flattening. The electronic biopsy colon image is then generated using the flattened colon and a volumetric ray casting algorithm. After that, our clustering

algorithm and reduction of FPs are performed. All of these processes are performed automatically in the pipeline.

Digital cleansing aims to segment the colon lumen from a patient abdominal data set acquired using CT and an oral contrast agent for colonic material tagging, and to cleanse the colon lumen of all tagged material, so that a cleansed virtual colon model can be constructed. In this pipeline, an iterative partial volume segmentation algorithm [158] is applied first. The voxels in the colon lumen are classified as air, mixture of air with tissue, mixture of air with tagged materials, or mixture of tissue with tagged materials. Then, the interface layer is identified by the dilation and erosion method.

For colon surface extraction, a new volume based topological denoising algorithm is used to remove tiny handles (i.e., topological noise) from the segmented colon [53, 55]. Then, an enhanced dual contour method [177] is applied to extract a simplified smooth colon surface while preserving the topology of the finest resolution colon surface.

Virtual dissection is an efficient visualization technique for polyp detection, in which the entire inner surface of the colon is displayed as a single 2D image. The straightforward method [155] starts with uniformly resampling the colonic central path. At each sampling point, a cross section orthogonal to the path is computed. The central path is straightened and the cross sections are unfolded and remapped into a new 3D volume. The isosurface is then extracted and rendered for polyp detection. However, this method results in severe distortions. Several methods have been developed that are either area preserving [7] or angle preserving [49, 53].

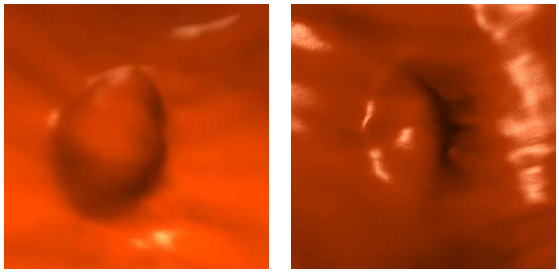
An angle preserving method is applied in the pipeline, because radiologists identify polyps mainly based on the shape information, and the lost area and volume information can reconstructed by referring back to the original volumetric data. Haker et al.'s method [49] can only handle genus 0 surfaces and maps the colon to a parallelogram. Nevertheless, our method [53] is more general and capable of handling high genus surfaces, and it maps the colon surface to a rectangle. Moreover, because the method of Haker et al. [49] is based on the shape information computed from the colon surface, it requires a highly accurate and smooth surface mesh to achieve good mean curvature estimation.

In order to compute the conformal map between the colon surface and a rectangle, its gradient field is computed first. Mathematically, this gradient field is called holomorphic 1-form. Then, the conformal mapping can be obtained by integration. Each gradient field of a conformal map is a pair of tangential vector fields with special properties, such as the curl and laplace are zero everywhere. All such vector fields form a linear space. A basis of this linear space is constructed by solving a linear system derived from these properties. The global distortion from the colon surface to the parametric rectangle is minimized, which is measured by harmonic energy. The details of our flattening algorithm can be found in the paper [53].

The holomorphic 1-form based conformal mapping method is pretty slow. A discrete Ricci flow based conformal mapping method for colon flattening has been developed. Ricci flow has a simple physical intuition. Given a surface S with a Riemannian metric $\mathbf{g}$, the metric induces the Gaussian curvature function. If the metric $\mathbf{g}$ is changed, then the Gaussian curvature will be changed accordingly. The Ricci flow deforms $\mathbf{g}$ in the following way: at each point, $\mathbf{g}$ is locally scaled to a new metric $\bar{\mathbf{g}}$ such that the scaling factor is proportional to the curvature at the point. Because of this locally isotropic deformation, the deformation is a *conformal* metric deformation such that angles measured by $\mathbf{g}$ are equal to that measured by $\bar{\mathbf{g}}$. After the deformation, the new metric $\bar{\mathbf{g}}$ induces a new curvature function. Both the metric and the curvature evolve while the deformation process is repeated. And the curvature evolution is like a heat diffusion process. Eventually, the Gaussian curvature function is constant everywhere. For a surface with a cylinder topology such as the colon, the result Gaussian curvature function is 0 everywhere. In practice, the colon surface is represented with a triangular mesh. Therefore, the discrete Ricci flow is computed where the Riemannian metric is replaced with the circle packing metric. The details of the algorithm can be found at [127].

We have used CUDA to accelerate the discrete Ricci flow on the GPU. With the Geforce 8800GTX graphics card, a colon mesh with 160,000 faces can be flattened in less than 10 seconds. In comparison, the holomorphic 1-form method in [53] needs 7 minutes to flatten a colon.

As assumed in previous CAD literature, the colonic polyps usually have an elliptic curvature of the peak subtype, i.e., the shape at the top section of a regular polyp (toward the colon wall) is more likely to be a spherical cap. Because of the



(a) (b)

Figure 5.2: (a) Closeup endoscopic view of a polyp; (b) Zoom-in view of the same polyp in the flattened colon image.

angle preservation of our colon flattening algorithm, the elliptic shape of a colonic polyp shown in Figure 5.2(a) is preserved in the flattened image as shown in Figure 5.2(b).

After a high resolution flattened electronic biopsy image is rendered, the RGB values of the given pixel and its twelve neighboring pixels form a 39-dimensional local feature vector. The principal component analysis (PCA) is applied to the local vector series to determine the dimension of the feature vectors and the associated orthogonal transformation matrix (i.e., the K-L transformation matrix). The PCA on the training data sets shows that a reasonable dimension of the feature vectors is 7, where the summation of the first 7 principal components variances is more than 96.5% of the total variance. The mean vector of these feature vectors is computed and used as the representative vector V of the feature vectors belonging to polyps. The square root of the variance of these feature vectors is also computed and used as a threshold T for vector similarity in the clustering.

After the clustering algorithm, the pixels classified belonging to a polyp are marked. A labeling algorithm is used to extract the connected components on this image. Since only polyps with a diameter larger than 5 millimeters are significant in diagnosis, a component whose pixel number is below such a threshold is classified as a false-positive finding. Consequently, many small components are removed.

The false-positive findings can be further reduced by analyzing the shape features, such as the shape index and curvedness [176] and volumetric texture features [175]. The computation of these features for the entire volume is time consuming. In this pipeline, these features are computed on the suspicious areas marked in

cluster step for FP reduction, rather than for the entire colon.

5.2 Direct Volume Rendering of Flattened Colon

The result of the flattening algorithm is a triangulated rectangle where the polyps are also flattened. The rendering of the flattened colon image is crucial for the detection of polyps. Haker et al. [49] use color-coded mean curvature to visualize the flattened colon surface. Although it can show the geometry information of the 3D colon surface, it is still unnatural for the physicians to detect the polyps. The shape of the polyps is a good clue for polyp detection. In this section, two direct volume rendering techniques is used to render the flattened colon image. Each pixel of the flattened image is shaded using a fragment program executed on the GPU, which allows the physician to move and zoom a viewing window to inspect the entire flattened inner colon surface. The idea of the rendering algorithm is to map each pixel of the flattened image back to the 3D colon surface, i.e., the volume space. The pixel is shaded using a volumetric ray-casting algorithm in the volume space.

To render the endoscopic views, the transfer function is designed so that the colon wall appears to be red, just like the endoscopic views in traditional virtual colonoscopy. In order to perform the ray-casting algorithm, the ray direction needs to be determined for each vertex of the 3D colon surface first. A number of cameras are uniformly placed on the central path of the colon. The ray direction of a vertex is then determined by the nearest camera to that vertex.

The camera registration algorithm starts with approximating the central path with a B-spline and resampling it into uniform intervals. Each sampling point represents a camera. Each vertex is then registered with a sampling point on the central path. The registration procedure is implemented efficiently by first dividing the 3D colon surface and central path into N segments. The registration is then performed between the correspondent segments of colon and the central path. The division of the 3D colon is done by classifying the vertices of the flattened 2D mesh into N uniform segments based on their height. As a consequence, the vertices of the 3D colon mesh are also divided into N segments. Then, $N-1$ horizontal lines on the flattened 2D mesh, which uniformly divide the 2D mesh into N segments. Each

traced horizontal line corresponds to a cross contour on the 3D mesh. In fact, it is unnecessary to really trace the horizontal lines. For each horizontal line, the intersection points of the horizontal line and edges intersecting with it are computed. For each intersection point, the corresponding 3D vertex of the 3D colon mesh is then interpolated. The centroid of these interpolated 3D vertices is computed and registered with a sampling point of the central path. Therefore, the central path is also divided into N segments, and each segment of the 3D colon mesh corresponds to a segment of the central path. Although the division of the 3D colon surface and the central path is not uniform as that of the 2D mesh, it does not affect the accuracy of the camera registration.

For each vertex of a colon surface segment, the nearest sampling point in its corresponding central path segment is computed and the neighboring two segments are obtained. This algorithm is efficient because for each vertex, the comparison is performed only with a small number of sampling points on the central path. For each vertex, the B-spline index of the sampling points is stored, instead of its 3D coordinates.

To generate a high-quality image of the flattened colon, only coloring the vertices of the polygonal mesh and applying linear interpolation is not sufficient. The color for each pixel of the 2D image is computed using a fragment program on the GPU. For each vertex of the flattened polygonal mesh, its corresponding 3D coordinates and camera index are passed through texture coordinates to the fragment program. When the flattened polygonal mesh is rendered, each pixel of the flattened image will obtain its barycentric interpolated 3D coordinates and camera index. Its 3D position may not be exactly on the colon surface, but very close to the colon surface. Because the direct volume rendering method is used, it does not affect the image quality. The interpolated camera index is used to look up its corespondent sampling point on the central path. Then, the ray direction is determined and volumetric ray casting algorithm is performed using an opaque transfer function.

Since the flattened image is colored per-pixel, it can provide the physician with a high-quality zoom-in view of a suspicious area on the flattened image in real-time. Because each vertex is registered with a sampling point on the central path, the flattened colon image can be easily correlated with the navigation of a virtual colonoscopy system. The correlated 3D view of the suspicious area can be

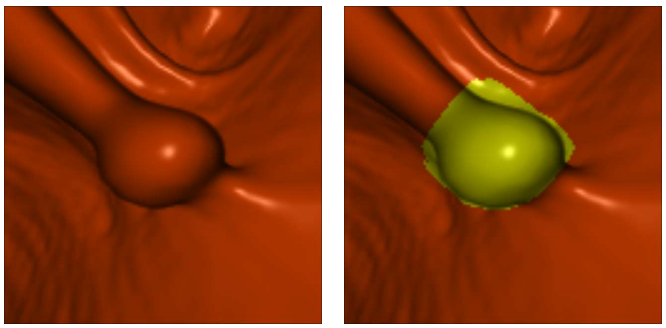
also shown simultaneously.

In the canonical volumetric ray casting algorithm, a ray is shot for each pixel on the image plane. The direction of the ray is defined by the positions of the viewpoint and the pixel. When the ray hits the boundary of the volume, the ray starts to accumulate color and opacity values while stepping inside the volume. In the CAD pipeline, a constrained volumetric ray casting algorithm can be used to generate the 2D biopsy image. In this method, the gradient at the intersection point is defined as the direction of the ray. In the volumetric ray-casting algorithm, the sampling distance is 0.5mm. Because the physicians are only interested in a thin layer (20mm) beneath the colon surface, each ray is only allowed to traverse up to 40 steps. Moreover, because the colon wall protrudes into the lumen, some rays may enter the colon lumen again. In order to avoid rays re-enter the colon lumen, these rays are terminated in the ray-casting algorithm using the segmentation information of the colon lumen.

5.3 Polygon Assisted Colon Rendering

When navigating or flying through the colon interior, the colon wall is rendered with a direct volume rendering method. Because of the large size of the colon volume data and the inherent complexity of volume rendering, it is very hard to achieve interactive frame rates with a software implementation. 3D texture-based volume rendering [13] is a popular volume rendering method that can achieve real-time speed on commodity graphics hardware (GPU). However, the rays shot from the image plane have different sampling rates due to the planar proxy geometry. Ray casting [73] has been implemented on the GPU, which has a coherent sampling rate for all rays. To achieve interactive speed, the two common acceleration techniques, empty-space skipping and early ray termination, have been used.

The polygonal mesh has been exploited by us to help direct volume rendering [5]. The polygonal mesh representing the object boundary is extracted from the volume in the second step of our pipeline. Each vertex is associated with its coordinates in volume texture space. The mesh is projected onto the image plane for calculating the entry points of rays, and the empty space between the image plane and the object boundary is skipped. The detection result is also stored in a



(a) (b)

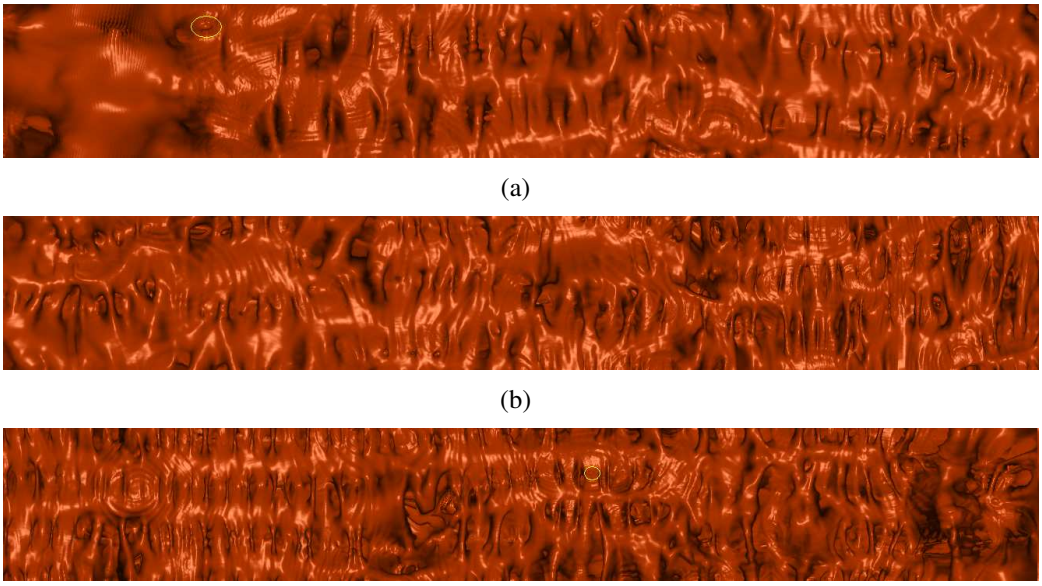
Figure 5.3: A closeup view of a polyp rendered (a) without coloring, and (b) with coloring.

2D image, in which polys are colored in yellow and normal colon wall is colored in red. When the flattened mesh is projected on the image, 2D texture coordinates are computed by interpolation, which is used to access the resulting image to determine the color of the ray. This method is very efficient because the GPU is very fast in rasterizing triangles onto the image plane. The algorithm has two passes. In the first pass, the mesh is rendered and the rasterization hardware interpolates the texture coordinates for each fragment. In this pass, the depth test is enabled so that only the nearest intersection points are preserved in the framebuffer. In the second pass, the fragment shader reads back the intersection point for each pixel on the image plane and a standard ray casting is performed from this point. A polyp rendered with our method with and without coloring is shown in Figure 5.3. The rendering frame rates for a 512 by 512 image is about 17-20 per second.

5.4 Results

The polyp detection pipeline has been implemented in C/C++ and all of the experiments have been run on a 3.6 GHz Pentium IV PC with 3G RAM and one NVIDIA Quadro 4500 graphics board. Figure 5.4 shows a colon dataset of $512 \times 512 \times 460$ resolution rendered with a transfer function for endoscopic view. Figure 5.4(a) includes the rectum of the colon at the left end. Figure 5.4(c) includes the cecum of the colon at the right end.

Figure 5.5(a) shows the electronic biopsy image of the flattened colon of the



(a)

(b)

(c)

Figure 5.4: A flattened image for a whole colon data set.

same dataset. For a 4000×196 image, the GPU renders the results images in about 300 milliseconds. Figure 5.5(b) shows the result of the clustering algorithm. The result of the reduction of FPs with shape analysis and 3D texture analysis is given in Figure 5.5(c).

The CAD pipeline has been demonstrated and tested with 52 CT data sets from the National Institute of Health (NIH). Along with the raw colon CT images, there are VC reports, OC reports, pathology reports, and OC videos available for viewing. In addition, another 46 CT data sets along with VC reports and OC reports obtained from Stony Brook University Hospital (SBUH) has also been used in testing. The specialists' VC and OC reports for the NIH and SBUH data sets have been used to evaluate the CAD pipeline.

Ten of the 52 NIH data sets are used for training the clustering algorithms, to compute the K-L matrix and the representative vector V. The rest of the data sets are used for testing, which exhibits consistent results. The electronic biopsy images are all generated with the same biopsy transfer function. The clustering algorithm is 100% sensitive to polyps, and no polyp from the 42 NIH data sets and the 46 SBUH data sets was missed. The polyps are colored using the volumetric

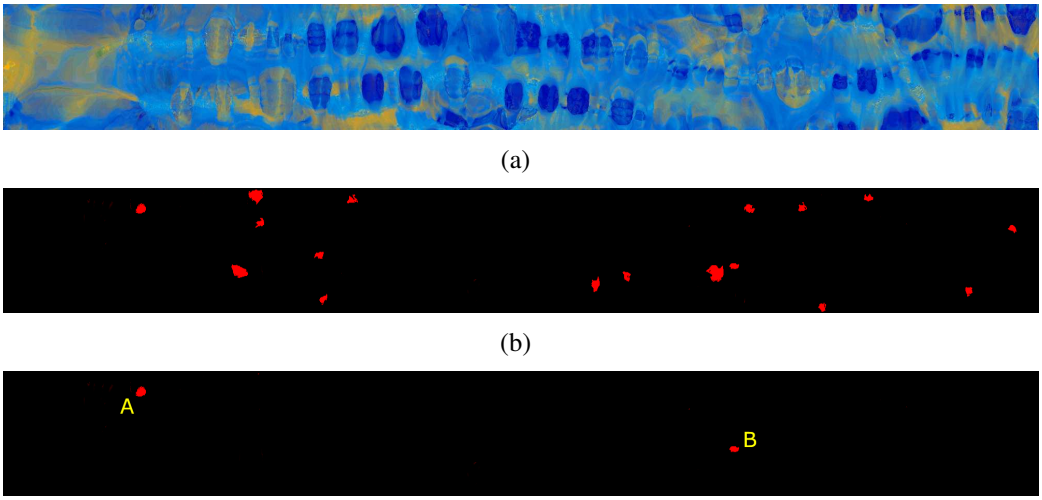


Figure 5.5: Results of (a) rendering, (b) clustering and (c) FP reduction.

ray casting algorithm with a translucent biopsy transfer function. The polyps are shown in similar colors on the 2D image, which will not be missed by the clustering algorithm.

Table 5.1: Experimental results of the CAD pipeline.

Data Source	Total Polyps	FP per data set	FP Reduction
NIH	58	3.1	96.3%
SBUH	65	2.9	97.1%

The experimental results are shown in Table 5.1, which are confirmed using VC reports and OC reports. There are 58 polyps in the 42 NIH data sets. 96.3% FPs are eliminated in the reduction step. Our method has an average FP number of 3.1 per NIH data set after the FP reduction. There are 65 polyps in the 46 SBUH data sets. 97.1% FPs are eliminated in the reduction step. The presented method has an average FP number of 2.9 per SBUH data set. The best shape analysis based systems [43, 148, 157, 176] achieved $2-3$ FPs per dataset with 100% sensitivity. The experiment results show that the proposed method achieved similar results as these systems.

In summary, our 3D CC lattice volume rendering based polyp detection is

different from previous shape based methods, as we detect suspicious patches on a 2D CC lattice electronic biopsy image of the colon. The FPs are further reduced in a subsequent step by shape analysis, which are only performed on the suspicious patches. The adenomatous and malignant polyps in the volume rendered biopsy images have different densities compared with normal tissues. Our system is 100% sensitive to these polyps with a very low FP rate. The detection results are stored on a 2D CC lattice, which can be easily integrated into the VC system to highlight the polyp locations on the colon wall during navigation. The regularity of 2D and 3D CC lattices simplifies the computation, thus our CAD pipeline is more efficient than traditional shape based methods.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work, several techniques and algorithms have been presented for rendering volumes on lattices with various optical models. For the simple absorption and emission model without scattering, the GPU has been exploited for improved rendering speed:

- A rasterization-tracing hybrid height field rendering method on GPU that exploits the advantages of both image-based and object-based techniques, which reconstructs a triangle mesh from the elevation map of a height field.
- An efficient GPU-CPU hybrid volumetric ray-casting algorithm for direct volume rendering, which exploits both the computation power and flexibility of the GPU and the CPU. Space space skipping and early ray termination are performed on the CPU, and ray traversal and compositing are implemented on the GPU. The parallelism between the CPU and the GPU has been exploited and a workload balancing algorithm has been exploited to keep both the CPU and GPU busy.
- A GPU-based object-order ray-casting algorithm for rendering large volumetric datasets that cannot be stored in the GPU memory. The volume dataset is decomposed into small cells, and organized using a min-max octree structure. The empty cells are skipped immediately after the classification, while non-empty cells are rendered in front-to-back order. A cell sorting algorithm

is designed allowing to project a layer of cells in parallel.

The 3D CC lattice volume rendering algorithms exploit the parallel architecture, high computation power, and native support of 3D CC lattice storage and interpolation of the GPU. The described algorithms produce high quality images at interactive or real time speed, which previously was only available on high-end workstation or specially designed graphics hardware.

The LBM models Boltzmann particle dynamics on a CC lattice and generates simulation results on the same CC lattice. The efficient and simple computing process of LBM makes it easy to be parallelized on the GPU and GPU cluster. Therefore, algorithms have been developed to render the smoke and amorphous objects produced by the LBM flow simulation on the same GPU or GPU cluster. To incorporate the shadow effect, the single scattering optical model is used. Specifically, a half angle splatting method has been accelerated on a single GPU. Also, ray tracing with single scattering has been accelerated with OpenGL both on a single GPU and on a GPU cluster using the lighting volume method. To further improve the rendering speed, a CUDA based method has been implemented, which does not have the frequent context switching as in the OpenGL based implementation. The described methods have achieved interactive speed and been integrated with the LBM simulation into a complete system.

To further improve image quality in volumetric global illumination, it has been accelerated with a novel framework based on FCC lattices. Benefitting from the unique geometric and sampling properties of FCC lattices, algorithms have been developed that can render participating media and volumes with multiple scattering effects. The new diffuse photon tracing algorithm renders high quality images at a speed significantly faster than conventional methods. To solve the ray effects in the diffuse photon tracing method, a memory efficient specular photon tracing algorithm has been described, which can be easily plugged into the traditional rendering software package.

The 3D CC lattice volume rendering technique has also been applied for computer-aided polyp detection for virtual colonoscopy. Different from previous shape based methods, the system first detects polyps on 2D electronic biopsy images rendered from conformally flattened colons. Then the expensive shape analysis is applied on suspicious areas to reduce false positives. Our system is 100%

sensitive to malignant polyps and the false positive rate is as low as the best shape-based method, while it is much faster than shape-based methods. The detection results can be easily integrated to the virtual colonoscopy system to highlight polyps and save the diagnostic time of radiologists. The 3D CC lattice volume rendering of the entire flattened colon costs about 300 milliseconds. It reduces the information from a 3D CC lattice to a 2D CC lattice. The image analysis is much faster than conventional shape analysis methods because of two reasons. First, the image analysis processes 2D information, while the shape analysis methods processes information in 3D. Second, the electronic biopsy image is stored on a 2D CC lattice, while the shape analysis methods processes the triangle mesh of the colon surface. The 2D CC lattice is much simpler and more efficient to process because of its regularity. In comparison, the data structure of a triangle mesh is more complex.

6.2 Future Work

In the future, the global illumination framework on FCC lattices can be extended to render hybrid scenes of volumetric datasets and surface objects with specular reflection and refraction. The current implementation only uses 12 links to the nearest sites on the FCC lattice. For future efforts, it is possible to incorporate the links connecting the secondary and tertiary neighbors to increase the angular discretization granularity. With these additional links, the phase function can be discretized with even higher precision thus more accurate radiance estimation will be obtained. In doing so, a total of 42 neighbors with distance less than $\sqrt{3}$ will have to be considered and different link lengths will participate in the computation of the absorption and scattering coefficients. Hence, a more efficient data structure will be required for diffuse photon storage.

The GPU and the GPU cluster can be further investigated to accelerate the rendering of FCC lattices. Currently, only the rendering pass in the diffuse photon tracing algorithm is accelerated on the GPU. With CUDA, accelerating the photon tracing pass on the GPU is possible and may significantly improve the performance. Because the global memory accessing instruction is two orders of magnitude slower than normal computation instructions on the CUDA-enabled GPU, directly mapping the photon tracing pass to the GPU might be inefficient. One possible idea is

to decompose the entire volume into many small blocks. Each of the blocks is small enough to be stored on the shared memory (or in-chip cache) of a multiprocessor. Also, because the photons are stored on the lattice sites, the 3D/4D photon array can also be stored in the shared memory. The global memory will be used for the communication between blocks. There is one photon queue for each multiprocessor in the global memory, which saves all the photons entering the block at the boundary. The CPU will be used to schedule the photon tracing process of blocks. During the tracing process, the density array will be loaded to shared memory first. Then, the computation kernel reads the densities or lighting properties and writes the photon arrays in the shared memory. When a photon leaves one block, it can be written to the photon queue of the block to be entered. This algorithm can also be generalized for ray tracing surface objects with the bounding volume hierarchy data structure.

The bottleneck of the ray tracing pass is the expensive Gaussian filter used for radiance estimation, which is slow even on the GPU. A possible solution is to use the photon splatting method. First, in the ray space, the transmittance volume can be calculated using CUDA with an algorithm similar to the lighting volume calculation algorithm. Then, the photons of each lattice site can be splatted together with a Gaussian kernel.

The FCC and BCC lattices have not been frequently used in computer graphics and visualization. One major reason is the lack of multi-resolution and hierarchical data structure on FCC and BCC lattices. Preliminary study has shown that hierarchical FCC and BCC lattices can be constructed from the hierarchical CC lattice. For example, one CC lattice can be decomposed into two interleaving FCC lattices. Given a CC lattice where a lattice site has index (i, j, k), the sites with odd index sum $i+j+k$ (odd sites) form an FCC lattice, while the even sites form another FCC lattice. Therefore, given a hierarchical CC lattice, a hierarchical FCC lattice can be easily obtained by decomposing levels of CC lattices. The BCC lattice is similar. The next step is to study how to resample a fine level FCC to get a coarse level FCC. More interestingly, the CC, FCC and BCC lattices can be mixed together for a multi-resolution representation. This requires to study the algorithms and methods of resampling one lattice to get another lattice, and the reconstruction filter at the boundaries between two lattices. A possible method is to study the Voronoi cells of CC, FCC and BCC lattices. Similar to the mipmap of 2D/3D textures, the

overlapping volume of the Voronoi cells in coarse level and fine level can be calculated and used as the weights. More complicated resampling kernel can also be used.

In traditional computer graphics, light is treated as bunches of rays. A ray is simply a path along which energy is transmitted from one point to another in the environment. The optical path of a ray is a sequence of straight-line segments. At the vertices of line segments, certain optical events such as emission, reflection, refraction, and absorption occur and change the ray direction. This treatment neglects the wave properties of light and is described with laws of geometric optics. However, geometric optics cannot model other important light effects such as diffraction, interference, polarization, and Doppler effect. All these effects due to the the wave character of the light can be described with physical optics or wave optics. The Huygens-Fresnel Principle states that each point of an advancing wave front is in fact the center of a fresh disturbance (or wavelet) and the source of a new train of waves. The advancing wave as a whole may be regarded as the sum of all the secondary waves arising from points in the medium already traversed. The amplitude of the wave at any given point equals the superposition of the amplitudes of all the secondary wavelets at that point. A wide range of optical phenomena including reflection, refraction, and diffraction can be simulated on the FCC lattices with the Huygens-Fresnel Principle.

Angular discretization has not been well studied in computer graphics, although it has important applications such as BRDF, phase function in computer graphics. One commonly used technique to represent angle dependent functions is the spherical harmonics. However, spherical harmonics is not a local representation. A spherical function $f(\theta,\phi)$ is expanded as a linear combination of an infinite set of orthonormal spherical functions $Y_l^m(\theta,\phi)$:

$$f(\theta,\phi)=\sum_{l=0}^{\infty}\sum_{m=-l}^{l} f_l^m Y_l^m(\theta,\phi). \tag{6.1}$$

This representation becomes less accurate when l is truncated. In practice, l is always truncated at some value. If a better accuracy is needed, the more basis functions are needed, and the resampling operations becomes more complicated. This is inconvenient, especially compared with the image representation. When a higher

resolution of the image is used, the complexity of the interpolation operation on the image does not change. A possible method for angular discretization is to define a set of sampling points and a good filter on the sphere domain. However, (θ, ϕ) (latitude and longitude) is not a good representation of the sphere domain because of singularity and oversampling. A better method is to study the regular tessellation (discretization) of the unit sphere surface, which does not have the singularity and oversampling problems. There are only three regular tessellations on a sphere: square tiling, triangular tiling and hexagonal tiling (dual to triangular tiling). Then, spherical Fourier transformation can be used to study the properties of three regular tessellation in the frequency domain and design a good filter.

Tessellation, discretization, and lattice are three closely related topics. A lattice represents a regular discretization of an R^n domain. The Voronoi cells of the lattice sites tessellate the R^n domain with a regular pattern. CC lattices are widely used for multi-dimensional image representations. Other lattices such as FCC and BCC lattices have their own advantages, but have not been well studied. There are many possibilities to exploit different lattices to discretize or tessellate the definition domain of general computation problems beyond volume rendering, for example, LBM on FCC lattices, hexagonal image processing, and finite element methods.

Bibliography

[1] K. Akeley. Reality engine graphics. *SIGGRAPH*, pages 109–116, 1993.

[2] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. *Eurographics*, pages 3–9, 1987.

[3] M. R. Anderberg. *Cluster analysis for applications*. Number 19 in Probability and Mathematical Statistics. Academic Press, New York, 1973. xiii+359 pages.

[4] M. Artner, T. Möller, I. Viola, and M. E. Gröller. High-quality volume rendering with resampling in the frequency domain. *EuroVis*, pages 85–92, June 2005.

[5] R. S. Avila, L. M. Sobierajski, and A. E. Kaufman. Towards a comprehensive volume visualization system. *Visualization*, pages 13–20, 1992.

[6] Barthold Lichtenbelt, Randy Crane, and Shaz Naqvi. *Introduction to Volume Rendering*. Prentice Hall PTR, Upper Saddle River, NJ07458, 1998.

[7] A. V. Bartroli, R. Wegenkittl, A. Konig, and E. Groller. Nonlinear virtual colon unfolding. *Visualization*, pages 411–418, October 2001.

[8] P. Blasi, B. L. Saëc, and C. Schlick. A rendering algorithm for discrete volume density objects. *Computer Graphics Forum*, 12(3):201–210, 1993.

[9] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *SIGGRAPH*, pages 21–29, 1982.

[10] R. N. Bracewell. *The Fourier Transform and Its Applications*. McGraw-Hill Book Company, New York, NY, USA, 1986.

[11] P. Bunyk, A. E. Kaufman, and C. T. Silva. *Scientific Visualization*, chapter Simple, Fast, and Robust Ray Casting of Irregular Grids, pages 30–36. IEEE Computer Society, 1999.

[12] G. Burns. *Solid State Physics*. Academic Press, 1985.

[13] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *Symposium on Volume Visualization*, pages 91–98, 1994.

[14] H. Carr, T. Theußl, and T. Möller. Isosurfaces on optimal regular samples. *Symposium on Data Visualisation*, pages 39–48, 2003.

[15] J. C. Chai, H. S. Lee, and S. V. Patankar. Ray effect and false scattering in the discrete ordinates method. *Numerical Heat Transfer Part B*, 24:373–389, 1993.

[16] S. Chandrasekhar. *Radiative Transfer*. Dover Publications, 1960.

[17] D. Cohen and Z. Sheffer. Proximity clouds: An acceleration technique for 3D grid traversal. *The Visual Computer*, 11:27–38, 1994.

[18] M. F. Cohen and D. P. Greenberg. The hemi-cube: A radiosity for complex environments. *SIGGRAPH*, 19(3):31–40, July 1985.

[19] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Boston, MA, 1993.

[20] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar. A real-time photo-realistic visual flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):255–265, 1996.

[21] J. Comba, J. T. Klosowski, N. Max, J. S. B. M. C. T. Silva, and P. L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Eurographics*, 18(3):369–376, 1999.

[22] J. H. Conway, N. J. A. Sloane, and E. Bannai. *Sphere Packings, Lattices, and Groups*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[23] R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.

[24] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *SIGGRAPH*, pages 137–145, 1984.

[25] R. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. *Visualization*, pages 261–266, 1993.

[26] B. Csébfalvi. Prefiltered Gaussian reconstruction for high-quality rendering of volumetric data sampled on a body-centered cubic grid. *Visualization*, pages 311–318, 2005.

[27] F. Dachille and A. Kaufman. Incremental triangle voxelization. *Graphics Interface*, pages 205–212, May 2000.

[28] D. Dudgeon and R.Mersereau. *Multidimensional Signal Processing*. PrenticeHall, New Jersey, NJ, USA, 1984.

[29] D. E. Dudgeon and R. M. Mersereau. *Multidimensional Digital Signal Processing*. Prentice Hall Professional Technical Reference, 1990.

[30] D. S. Dummit and R. M. Foote. *Abstract Algebra*. John Wiley and Sons, second edition, 1999.

[31] D. S. Ebert, C. J. Morris, P. Rheingans, and T. S. Yoo. Designing effective transfer functions for volume rendering from photographic volumes. *IEEE Transactions on Visualization and Computer graphics*, 8:183–197, April 2002.

[32] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. *Workshop on Graphics Hardware*, pages 9–16, 2001.

[33] A. Entezari, R. Dyer, and T. Moller. Linear and cubic box splines for the body centered cubic lattice. *Visualization*, pages 11–18, 2004.

[34] A. Entezari and T. Möller. Extensions of the Zwart-Powell box spline for volumetric data reconstruction on the Cartesian lattice. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1337–1344, 2006.

[35] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of ACM/IEEE Supercomputing*, page 47, 2004.

[36] R. Farias, J. S. B. Mitchell, and C. T. Silva. ZSWEEP: An efficient and exact projection algorithm for unstructured volume rendering. *Symposium on Volume Visualization*, pages 91–99, 2000.

[37] R. Fedkiw, J. Stam, and H. Jensen. Visual simulation of smoke. *SIGGRAPH*, pages 15–22, 2001.

[38] M. S. Floater, K. Hormann, and G. Kós. A general construction of barycentric coordinates over convex polygons. *Advances in Computational Mathematics*, 24(1–4):311–331, January 2006.

[39] C. Früh and A. Zakhor. Constructing 3D city models by merging aerial and ground views. *IEEE Computer Graphics and Applications*, 23(6):52–61, November/December 2003.

[40] J. Fung and S. Mann. Using multiple graphics cards as a general purpose parallel computer : Applications to computer vision. *International Conference on Pattern Recognition*, 1:805–808, 2004.

[41] M. P. Garrity. Raytracing irregular volume data. *Workshop on Volume Visualization*, pages 35–40, 1990.

[42] R. Geist, K. Rasche, J. Westall, and R. J. Schalkoff. Lattice-Boltzmann lighting. *Rendering Techniques*, pages 355–362, 2004.

[43] S. B. Göktürk, C. Tomasi, B. Acar, C. F. Beaulieu, D. S. Paik, R. B. Jeffrey, J. Yee, and S. Napel. A statistical 3D pattern processing method for computer aided detection of polyps in CT colonography. *IEEE Transactions on Medical Imaging*, 20(12):1251–1260, 2001.

[44] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modelling the interaction of light between diffuse surfaces. *SIGGRAPH*, 18(3):213–222, 1984.

[45] GPGPU. http://www.gpgpu.org.

[46] B. Grünbaum. The emperor's new clothes: full regalia, g-string, or nothing? *Mathematical Intelligencer*, 6(4):47–56, 1984.

[47] B. Grünbaum and G. C. Shephard. *Tilings and Patterns: An Introduction*. W. H. Freeman, 1989.

[48] R. Gulde, M. Weeks, S. Owen, and Y. Pan. Parallel computing with multiple GPUs on a single machine to achieve performance gains. *Workshop on General-Purpose Computing on Graphics Processors*, 2004.

[49] S. Haker, S. Angenent, A. Tannenbaum, and R. Kikinis. Nondistorting flattening maps and the 3D visualization of colon CT images. *IEEE Transactions on Medical Imaging*, 19:665–670, December 2000.

[50] T. C. Hales. Cannonballs and honeycombs. *Notices of the American Mathematical Society*, 47(4), April 2000.

[51] M. J. Harris and A. Lastra. Real-time cloud rendering. *Computer Graphics Forum*, 20(3), 2001.

[52] K. Hegeman, M. Ashikhmin, and S. Premože. A lighting model for general participating media. *Symposium on Interactive 3D Graphics and Games*, pages 117–124, 2005.

[53] W. Hong, X. Gu, F. Qiu, M. Jin, and A. Kaufman. Conformal virtual colon flattening. *Symposium on Solid and Physical Modeling*, pages 85–93, 2006.

[54] W. Hong, F. Qiu, and A. Kaufman. GPU-based object-order ray-casting for large datasets. In *Proceedings of International Workshop on Volume Graphics*, pages 177–185, 2005.

[55] W. Hong, F. Qiu, and A. Kaufman. A pipeline for computer aided polyp detection. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):861–868, September–October 2006.

[56] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. *Visualization*, pages 35–42, 1998.

[57] D. Horn, M. Houston, and P. Hanrahan. ClawHMMer: A streaming HMMer-Search implementation. *Supercomputing*, pages 11–19, 2005.

[58] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A scalable graphics system for clusters. *SIGGRAPH*, pages 129–140, 2001.

[59] D. S. Immel, M. F. Cohen, and D. P. Greenberg. A radiosity method for non-diffuse environments. *SIGGRAPH*, pages 133–142, 1986.

[60] H. W. Jensen. Global illumination using photon maps. *Workshop on Rendering*, pages 21–30, 1996.

[61] H. W. Jensen. Rendering caustics on non-lambertian surfaces. *Computer Graphics Forum*, 16(1):57–64, 1997.

[62] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[63] H. W. Jensen and P. H. Christensen. Efficient simulation of light transport in scences with participating media using photon maps. *SIGGRAPH*, pages 311–320, 1998.

[64] H. W. Jensen, S. R. Marschner, M. Levoy, and P. Hanrahan. A practical model for subsurface light transport. *SIGGRAPH*, pages 511–518, 2001.

[65] C. D. Johnson and A. H. Dachman. CT colonography: The next colon screening examination? *Radiology*, 216(2):331–341, 2000.

[66] J. T. Kajiya. The rendering equation. *SIGGRAPH*, pages 143–150, 1986.

[67] J. T. Kajiya and B. P. V. Herzen. Ray tracing volume densities. *SIGGRAPH*, pages 165–174, 1984.

[68] A. E. Kaufman. Volume visualization: Principles and advances. *SIGGRAPH Course notes*, July 1997.

[69] 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, April–June 2003.

[70] G. Knittel. The ULTRAVIS system. *Symposium on Volume Visualization*, pages 71–79, 2000.

[71] K. Kreeger, I. Bitter, F. Dachille, B. Chen, and A. Kaufman. Adaptive perspective ray casting. *Symposium on Volume Visualization*, pages 55–62, 1998.

[72] S. Krishnan, C. T. Silva, and B. Wei. A hardware-assisted visibility-ordering algorithm with applications to volume rendering. *EG/IEEE TVCG Symposium on Data Visualization*, pages 233–242, 2001.

[73] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. *Visualization*, pages 287–292, 2003.

[74] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *SIGGRAPH*, pages 451–458, 1994.

[75] S. Lakare and A. Kaufman. Light weight space leaping using ray coherence. *Visualization*, pages 19–26, 2004.

[76] C. Lee and Y. G. Shin. An efficient ray tracing method for terrain rendering. *Pacific Graphics*, pages 180–193, 1995.

[77] J. Legakis, J. Dorsey, and S. J. Gortler. Feature-based cellular texturing for architectural models. *SIGGRAPH*, pages 309–316, 2001.

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

[79] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[80] M. Levoy and R. Whitaker. Gaze-directed volume rendering. *Symposium on Interactive 3D Graphics*, pages 217–223, 1990.

[81] W. Li, Z. Fan, X. Wei, and A. Kaufman. Flow simulation with complex boundaries. *GPU Gems II: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 747–764, 2005.

[82] W. Li and A. Kaufman. Accelerating volume rendering with texture hulls. *Symposium on Volume Visualization*, pages 115–122, 2002.

[83] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. *Visualization*, pages 317–324, October 2003.

[84] W. Li, X. Wei, and A. Kaufman. Implementing lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19(7–8):444–456, 2003.

[85] D. R. Lide. *Handbook of Chemistry and Physics*. CRC Press LLC, 84 edition, 2003.

[86] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. *SIGGRAPH*, pages 109–118, 1996.

[87] S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich. Scalable interactive volume rendering using off-the-shelf components. *Symposium on Parallel and Large-Data Visualization and Graphics*, 2001.

[88] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH*, pages 163–169, 1987.

[89] C. H. Luis Ibanez and C. Roux. Ray-tracing and 3D objects representation in the BCC and FCC grids. *International Workshop on Discrete Geometry for Computer Imagery*, pages 235–242, 1997.

[90] C. H. Luis Ibanez and C. Roux. A vectorial algorithm for tracing discrete straight lines in n-dimensional generalized grids. *IEEE Transactions on Visualization and Computer Graphics*, 7:97–108, April–June 2001.

[91] J. S. Mandel, J. H. Bond, T. R. Church, D. C. Snover, G. M. Bradley, L. M. Schuman, and F. Ederer. Reducing mortality from colorectal cancer by screening for fecal occult blood. *New England Journal of Medicine*, 328(19):1365–1371, 1993.

[92] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *SIGGRAPH*, pages 896–907, 2003.

[93] S. Matej and R. Lewitt. Efficient 3D grids for image reconstruction using spherically-symmetric volume elements. *IEEE Transactions on Nuclear Science*, 42(4):1361–1370, 1995.

[94] S. Matej and R. Lewitt. Practical considerations for 3-d image reconstruction using spherically symmetric volume elements. *IEEE Transactions on Medical Imaging*, 15(1):68–78, 1996.

[95] O. Mattausch. Practical reconstruction schemes and hardware-accelerated direct volume rendering on body-centered cubic grids. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, May 2004.

[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 functions. *Workshop on Volume Visualization*, 24(5):27–33, 1990.

[98] N. L. Max. Efficient light propagation for multiple anisotropic volume scattering. *Workshop on Rendering*, pages 87–104, 1994.

[99] G. G. McNamara and G. Zanetti. Use of the Boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, 61:2332–2335, 1988.

[100] M. Meißner, U. Kanus, and W. Straßner. Vizard II, a PCI-card for real-time volume rendering. *Workshop on Graphics Hardware*, pages 61–67, 1998.

[101] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. Vizard II: A reconfigurable interactive volume rendering system. *Workshop on Graphics Hardware*, pages 137–146, 2002.

[102] R. M. Mersereau. The processing of hexagonally sampled two-dimensional signals. *IEEE Proceedings*, 67:930–949, June 1979.

[103] B. Mora, J. P. Jessel, and R. Caubet. A new object-order ray-casting algorithm. *Visualization*, pages 203–210, October 2002.

[104] J. Morey and K. Sedig. Archimedean kaleidoscope: A cognitive tool to support thinking and reasoning about geometric solids. *Geometric Modeling: Techniques, Applications, Systems and Tools*, pages 376–393, 2004.

[105] K. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. *Visualization*, pages 239–245, 1998.

[106] K. Mueller, T. Möller, and R. Crawfis. Splatting without the blur. *Visualization*, pages 363–370, 1999.

[107] K. Mueller, T. Möller, J. E. Swan II, R. Crawfis, N. Shareef, and R. Yagel. Splatting errors and antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 4(2), April–June 1998. ISSN 1077-2626.

[108] K. Mueller and R. Yagel. The use of hexagonal grids to improve the efficiency of the algebraic reconstruction technique (ART). *Annual Conference of the Biomedical Engineering Society*, 1996.

[109] K. Mueller and R. Yagel. Anti-aliased 3D cone-beam reconstruction of low-contrast objects with algebraic methods. *IEEE Transactions on Medical Imaging*, 18(6):519–537, 1999.

[110] S. Muraki, E. B. Lum, K. Ma, M. Ogata, and X. Liu. A PC cluster system for simultaneous interactive volumetric modeling and visualization. *Symposium on Parallel and Large-Data Visualization and Graphics*, pages 95–102, 2003.

[111] F. K. Musgrave. Grid tracing: Fast ray tracing for height fields. Technical Report Technical Report YALEU/DCS/RR-639, Yale University, 1988.

[112] N. Neophytou and K. Mueller. Space-time points: 4D splatting on efficient grids. *Symposium on Volume Visualization*, pages 97–106, 2002.

[113] N. Neophytou and K. Mueller. Post-convolved splatting. *EuroVis*, pages 223–230, May 2003.

[114] T. Nishita and E. Nakamae. Continuous tone representation of 3-D objects taking account of shadows and interreflection. *SIGGRAPH*, 19(3):23–30, July 1985.

[115] K. L. Novins, F. X. Sillion, and D. P. Greenberg. An efficient method for volume rendering using perspective projection. *Workshop on Volume Visualization*, pages 95–102, 1990.

[116] NVIDIA Corporation. NVIDIA OpenGL extension specifications, 2003.

[117] M. O'Keeffe and B. Hyde. *Crystal Structures I: Patterns and Symmetry*. Mineralogical Society of America, 1996.

[118] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Eurographics State of the Art Reports*, pages 21–51, August 2005.

[119] D. W. Paglieroni and S. M. Petersen. Height distributional distance transform methods for height field ray tracing. *ACM Transactions on Graphics*, 13(4):376–399, 1994.

[120] S. Parker, P. Shirley, Y. Livnat, C. Hansen, P.-P. Sloan, and M. Parker. Interacting with gigabyte volume datasets on the Origin 2000. *The 41st Annual Cray Users Group Conference*, 1999.

[121] K. Perlin. An image synthesizer. *SIGGRAPH*, pages 287–296, 1985.

[122] H. Pfister and A. Kaufman. Cube-4: a scalable architecture for real-time volume rendering. *Symposium on Volume Visualization*, page 47, 1996.

[123] H. Pfister, A. Kaufman, and F. Wessels. Towards a scalable architecture for real-time volume rendering. *Workshop on Graphics Hardware*, pages 123–130, August 1995.

[124] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[125] P. J. Pickhardt, J. R. Choi, I. Hwang, J. A. Butler, M. L. Puckett, H. A. Hildebrandt, R. K. Wong, P. A. Nugent, P. A. Mysliwiec, and W. R. Schindler. Computed tomographic virtual colonoscopy to screen for colorectal neoplasia in asymptomatic adults. *The New England Journal of Medicine*, 349(23):2191–2200, December 2003.

[126] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *SIGGRAPH*, pages 703–712, 2002.

[127] F. Qiu, Z. Fan, X. Yin, A. Kaufman, and X. Gu. Colon flattening with discrete ricci flow. *MICCAI Workshop on Virtual Colonoscopy*, pages 97–101, 2008.

[128] F. Qiu, Y. Zhao, Z. Fan, X. Wei, H. Lorenz, J. Wang, S. Yoakum-Stover, A. Kaufman, and K. Mueller. Dispersion simulation and visualization for urban security. *Visualization*, pages 553–560, 2004.

[129] H. Qu and A. Kaufman. O-buffer: A framework for sample-based graphics. *IEEE Transactions on Visualization and Computer Graphics*, 10(4), July–August 2004.

[130] H. Ray and D. Silver. The race II engine for real-time volume rendering. *Workshop on Graphics Hardware*, pages 129–136, 2000.

[131] C. Rezk-Salama. *Volume Rendering Techniques for General Purpose Hardware*. PhD thesis, University of Erlangen-Nuremberg, 2002.

[132] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. *Workshop on Graphics Hardware*, pages 109–118, 2000.

[133] K. Riley, D. S. Ebert, M. Kraus, J. Tessendorf, and C. D. Hansen. Efficient rendering of atmospheric phenomena. *Symposium on Rendering*, pages 374–386, 2004.

[134] S. Roettger, S. Guthe, D. Weiskopf, T. Erlt, and W. Strasser. Smart hardware-accelerated volume rendering. *EuroVis*, pages 231–238, 2003.

[135] H. E. Rushmeier and K. E. Torrance. The zonal method for calculating light intensities in the presence of a participating medium. *SIGGRAPH*, pages 293–302, 1987.

[136] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. *SIGGRAPH*, pages 343–352, 2000.

[137] M. Segal and K. Akeley. The OpenGL graphics system: A specification (version 1.4), 2002.

[138] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Workshop on Volume visualization*, pages 63–70, 1990.

[139] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. *SIGGRAPH*, pages 335–344, 1989.

[140] F. X. Sillion, J. R. Arvo, S. H. Westin, and D. P. Greenberg. A global illumination solution for general reflectance distributions. *SIGGRAPH*, pages 187–196, 1991.

[141] C. T. Silva, A. Kaufman, and C. Pavlakos. PVR: High-performance volume rendering. *IEEE Computational Science & Engineering*, 3(4):18–28, Winter 1996.

[142] C. T. Silva, J. S. B. Mitchell, and P. L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. *Symposium on Volume Visualization*, pages 87–94, 1998.

[143] L. M. Sobierajski and A. E. Kaufman. Volumetric ray tracing. *Symposium on Volume Visualization*, pages 11–18, 1994.

[144] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexiable volume rendering framework for graphics-hardware-based raycasting. *Volume Graphics*, pages 187–195, 2005.

[145] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. *Symposium on Volume Visualization*, pages 83–90, 1994.

[146] K. R. Subramanian and D. S. Fussell. Applying space subdivision techniques to volume rendering. *Visualization*, pages 150–159, 1990.

[147] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Numerical Mathematics and Scientific Computation. Oxford University Press, 2001.

[148] R. M. Summers, C. D. Johnson, L. M. Pusanik, J. D. Malley, A. M. Youssef, and J. E. Reed. Automated polyp detection at CT colonography: Feasibility assessment in a human population. *Radiology*, 219(1):51–59, 2001.

[149] J. Sweeney and K. Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. *Symposium on Data Visualisation*, pages 95–104, 2002.

[150] T. Theußl, T. Möller, and M. E. Gröller. Optimal regular volume sampling. *Visualization*, pages 91–98, 2001.

[151] T. Theußl, T. Möller, and M. E. Gröller. Reconstruction schemes for high quality raycasting of the body-centered cubic grid. Technical Report TR-186-2-02-11, Vienna University of Technology, December 2002.

[152] D. V. D. Ville, T. Blu, M. Unser, W. Philips, I. Lemahieu, and R. V. de Walle. Hex-splines: A novel spline family for hexagonal lattices. *IEEE Transactions on Image Processing*, 13(6):758–772, June 2004.

[153] J. R. Wallace, M. F. Cohen, and D. P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. *SIGGRAPH*, pages 311–320, 1987.

[154] M. Wan, H. Qu, and A. Kaufman. Virtual flythrough over a voxel-based terrain. *Virtual Reality*, page 53, 1999.

[155] G. Wang and M. W. Vannier. GI tract unraveling by spiral CT. *SPIE*, 2434:307–315, 1995.

[156] X. Wang, S. Totaro, F. Taillandier, A. Hanson, and S. Teller. Recovering facade texture and microstructure from real-world images. *Workshop on Texture Analysis and Synthesis at ECC*, pages 145–149, 2002.

[157] Z. Wang, Z. Liang, L. Li, X. Li, B. Li, J. Anderson, and D. Harrington. Reduction of false positives by internal features for polyp detection in CT-based virtual colonoscopy. *Medical Physics*, 32(12):3602–3616, 2005.

[158] Z. Wang, Z. Liang, X. Li, L. Li, D. Eremina, and H. Lu. An improved electronic colon cleansing method for detection of colonic polyps by virtual colonoscopy. *IEEE Transactions on Biomedical Engineering*, 53(8):1635–1646, 2006.

[159] X. Wei, W. Li, K. Mueller, and A. Kaufman. The lattice Boltzmann method for gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, March–April 2004.

[160] X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover, and A. Kaufman. Blowing in the wind. *Symposium on Computer Animation*, pages 75–85, July 2003.

[161] D. Weiskopf, T. Schafhitzel, and T. Ertl. GPU-based nonlinear ray tracing. *Computer Graphics Forum*, 23(3):625–634, 2004.

[162] D. Weiskopf, M. Weiler, and T. Ertl. Maintaining constant frame rates in 3d texture-based volume rendering. *Computer Graphics International*, pages 604–607, 2004.

[163] R. Westermann. The rendering of unstructured grids revisited. *Symposium on Visualization*, pages 65–74, 2001.

[164] R. Westermann and B. Sevenich. Accelerated volume ray-casting using texture mapping. *Visualization*, pages 271–278, 2001.

[165] L. Westover. Footprint evaluation for volume rendering. *SIGGRAPH*, pages 367–376, 1990.

[166] H. Widjaya, T. Möller, and A. Entezari. Voxelization in common sampling lattices. *Pacific Graphics*, page 497, 2003.

[167] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11:201–227, July 1992.

[168] P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[169] D. A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models: An Introduction*. Springer-Verlag, 2000.

[170] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(3):669–677, July 2003.

[171] D. Xue, C. Zhang, and R. Crawfis. iSBVR: Isosurface-aided hardware acceleration techniques for slice-based volume rendering. *Volume Graphics*, pages 207–215, 2005.

[172] R. Yagel. Volume rendering polyhedral grids by incremental slicing. Technical Report OSU-CISRC-10/93-TR35, Ohio State University, 1993.

[173] R. Yagel and A. Kaufman. Template-based volume viewing. *Computer Graphics Forum*, 11(3):153–167, 1992.

[174] R. Yagel, D. M. Reed, A. Law, P. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. *Symposium on Volume Visualization*, pages 55–62, 1996.

[175] H. Yoshida, Y. Masutani, P. MacEneaney, D. T. Rubin, and A. H. Dachman. Computerized detection of colonic polyps in CT colonography based on volumetric features: A pilot study. *Radiology*, pages 327–336, January 2002.

[176] H. Yoshida and J. Näppi. Three-dimensional computer-aided diagnosis scheme for detection of colonic polyps. *IEEE Transactions on Medical Imaging*, 20(12):1261–1274, 2001.

[177] N. Zhang, W. Hong, and A. Kaufman. Dual contouring with topolgy-preserving simplification using enhanced cell representation. *Visualization*, pages 505–512, October 2004.

[178] Y. Zhao, Y. Han, Z. Fan, F. Qiu, Y.-C. Kuo, A. Kaufman, and K. Mueller. Visual simulation of heat shimmering and mirage. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):179–189, 2007.

[179] Y. Zhao, F. Qiu, Z. Fan, and A. Kaufman. Flow simulation with locally-refined LBM. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 181–188, 2007.

[180] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA volume splatting. *Visualization*, pages 29–36, 2001.

[181] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. *SIGGRAPH*, pages 371–378, 2001.