# VOLUMETRIC RENDERING TECHNIQUES FOR SCIENTIFIC VISUALIZATION

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Erhan Okuyan
June, 2014

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Uğur Güdükbay (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. İsmail Hakkı Toroslu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Assoc. Prof. Dr. İbrahim Körpeoğlu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Assoc. Prof. Dr. Ceyhun Bulutay

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

# ABSTRACT

## VOLUMETRIC RENDERING TECHNIQUES FOR SCIENTIFIC VISUALIZATION

Erhan Okuyan
Ph.D. in Computer Engineering
Supervisor: Prof. Dr. Uğur Güdükbay
June, 2014

Direct volume rendering is widely used in many applications where the inside of a transparent or a partially transparent material should be visualized. We have explored several aspects of the problem. First, we proposed a view-dependent selective refinement scheme in order to reduce the high computational requirements without affecting the image quality significantly. Then, we explored the parallel implementations of direct volume rendering: both on GPU and on multi-core systems. Finally, we used direct volume rendering approaches to create a tool, *MaterialVis*, to visualize amorphous and/or crystalline materials.

Visualization of large volumetric datasets has always been an important problem. Due to the high computational requirements of volume-rendering techniques, achieving interactive rates is a real challenge. We present a selective refinement scheme that dynamically refines the mesh according to the camera parameters. This scheme automatically determines the impact of different parts of the mesh on the output image and refines the mesh accordingly, without needing any user input. The view-dependent refinement scheme uses a progressive mesh representation that is based on an edge collapse-based tetrahedral mesh simplification algorithm. We tested our view-dependent refinement framework on an existing state-of-the-art volume renderer. Thanks to low overhead dynamic view-dependent refinement, we achieve interactive frame rates for rendering common datasets at decent image resolutions.

Achieving interactive rates for direct volume rendering of large unstructured volumetric grids is a challenging problem, but parallelizing direct volume rendering algorithms can help achieve this goal. Using Compute Unified Device Architecture (CUDA), we propose a GPU-based volume rendering algorithm that itself is based on a cell projection-based ray-casting algorithm designed for CPU implementations. We also propose a multi-core parallelized version of the cell-projection algorithm using

OpenMP. In both algorithms, we favor image quality over rendering speed. Our algorithm has a low memory footprint, allowing us to render large datasets. Our algorithm support progressive rendering. We compared the GPU implementation with the serial and multi-core implementations. We observed significant speed-ups, that, together with progressive rendering, enabling reaching interactive rates for large datasets.

Visualization of materials is an indispensable part of their structural analysis. We developed a visualization tool for amorphous as well as crystalline structures, called *MaterialVis*. Unlike the existing tools, *MaterialVis* represents material structures as a volume and a surface manifold, in addition to plain atomic coordinates. Both amorphous and crystalline structures exhibit topological features as well as various defects. *MaterialVis* provides a wide range of functionality to visualize such topological structures and crystal defects interactively. Direct volume rendering techniques are used to visualize the volumetric features of materials, such as crystal defects, which are responsible for the distinct fingerprints of a specific sample. In addition, the tool provides surface visualization to extract hidden topological features within the material. Together with the rich set of parameters and options to control the visualization, *MaterialVis* allows users to visualize various aspects of materials very efficiently as generated by modern analytical techniques such as the Atom Probe Tomography.

*Keywords:* Volume visualization, direct volume rendering, view-dependent refinement, progressive meshes, unstructured tetrahedral meshes, Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA), OpenMP, material visualization, crystals, amorphous materials, crystallography, embedded nano-structure visualization, crystal visualization, crystal defects.

# ÖZET

# BİLİMSEL GÖRÜNTÜLEME İÇİN HACİM BOYAMA YÖNTEMLERİ

Erhan Okuyan
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Prof. Dr. Uğur Güdükbay
Haziran, 2014

Doğrudan hacim boyama saydam ya da kısmen saydam olan üç boyutlu hacim verilerinin içini görüntülemeyi gerektiren pek çok uygulamada kullanılan bir yöntemdir. Biz bu problemi değişik boyutları ile inceledik. Öncelikle, yüksek işlemci gereksinimini resim kalitesini önemli ölçüde bozmadan azaltmak amacıyla, bir bakışaçısına bağlı seçici sadeleştirme mekanizması önerdik. Daha sonra doğrudan hacim görüntüleme probleminin, grafik işleme ünitesi (GPU) ve çok çekirdekli işlemcili sistemler üzerindeki paralel uygulamalarını inceledik. Ve son olarak, doğrudan hacim görüntüleme tekniklerini kullanarak, amorf ve kristal yapıları görüntülemeyi amaçlayan, *MaterialVis* aracını geliştirdik.

Büyük hacim veri kümelerinin görüntülenmesi her zaman önemli bir problem olmuştur. Hacim görüntüleme tekniklerinin yüksek işlemci zamanı gereksinimleri dolayısıyla görüntülemeyi interaktif seviyelere çıkarmak kolay bir iş değildir. Biz, hacim veri kümesini bakış açısına bağlı olarak dinamik bir şekilde seçici sadeleştiren bir mekanizma önerdik. Bu mekanizma, hacim veri kümesinin farklı kısımlarının sonuç resim üzerinde ne kadar etkisi olacağını otomatik olarak tahmin eder ve veri kümesini buna göre sadeleştirir. Sonuç resim üzerinde çok etkisi olacak kısımlar daha detaylı temsil edilirken, az etkisi olan kısımlar daha az detayla temsil edilir. Görüş bağımlı sadeleştirme mekanizması, kenar göçertme tekniği tabanlı tetrahedral ağ sadeleştirme algoritması üzerine kurulu bir ilerlemeli tetrahedral ağ veri yapısı kullanır. Önerdiğimiz görüş bağımlı sadeleştirme mekanizmamızı en gelişmiş hacim görüntüleme araçlarında test ettik. Görüş bağımlı sadeleştirme mekanizmamızın düşük ek iş yükü sayesinde, yaygın veri kümelerinde yeterli çözünürlükler için interaktif seviyelere çıkmayı başardık.

Büyük hacim veri kümelerinin görüntülenmesinde etkileşimli hızlara ulaşmak kolay değildir. Ancak, hacim görüntüleme algoritmalarının paralelleştirilmesi faydalı olacaktır. Bu amaçla, Birleşik Cihaz Hesaplama Mimarisi (CUDA) kullanarak grafik işlem ünitesi üzerinde çalışacak, hücre izdüşümü ve ışın fırlatım tabanlı bir hacim görüntüleme algoritması önerdik. Aynı zamanda, OpenMP kullanarak bu algoritmanın çok çekirdekli işlemciler üzerinde çalışacak versiyonunu da geliştirdik. İki algoritmada da, sonuç resim kalitesini, işleme hızının önünde tuttuk. Algoritmalarımızın düşük hafıza kullanımları büyük veri kümelerini işleyebilmemize olanak sağladı. Grafik işlemci tabanlı algoritmayı ve çoklu çekirdek tabanlı algoritmayı seri tek çekirdek tabanlı algoritmayla karşılaştırdık ve ciddi hız artışları gözlemledik. Aşamalı örüntü işleme yöntemiyle beraber, büyük veri kümeleri için etkileşimli işleme hızlarına ulaşmayı başardık.

Materyallerin görüntülenmesi analizlerinin önemli bir parçasını oluşturur. Amorf ve kristal yapıların görüntülenmesi amacıyla, *MaterialVis* adında bir araç geliştirdik. Hem amorf hem kristal yapılar topolojik özellikler sergiler. Kristal yapılarda, ayrıca kristal hataları da bulunabilir. *MaterialVis* hem topolojik özellikleri hem kristal hatalarını görüntülemek amacıyla birçok işlev içerir. *MaterialVis* materyalleri düz atomik koordinatlara ek olarak hem hacim hem de yüzey manifoldu olarak tanımlar. Direk hacim görüntüleme teknikleri materyallerin hacimsel özelliklerini görüntülemek için idealdir. Kristal hataları, kristal özellikleri olarak tanımlanıp görüntülenebilir. *MaterialVis* aracı aynı zamanda yüzey görüntüleme tekniklerini de destekler. Kullanıcıların görüntülemeyi kontrol etmesini sağlayan zengin parametre ve seçenekler sayesinde, materyallerin çeşitli özellikleri etkili bir şekilde görüntülenebilir. Bu sayede, amorf ve kristal yapıları çeşitli işleme biçimlerinde interaktif olarak işleyip topoloji ve kristal hataları gibi önemli materyal özellikleri ortaya konulabilir.

*Anahtar sözcükler*: Hacim görüntüleme, doğrudan hacim görüntüleme, bakış açısına bağlı sadeleştirme, kademeli ağlar, düzensiz tetrahedral örüntüler, grafik işleme birimi (GPU), Birleşik Cihaz Hesaplama Mimarisi (CUDA), OpenMP, materyal görüntüleme, kristaller, amorf materyaller, kristalografi, gömülü nano-yapı görüntüleme, kristal görüntüleme, kristal kusurları.

# Acknowledgement

I would like to express my gratitude to my thesis supervisor Prof. Dr. Uğur Güdükbay for his encouragement, support and belief in my work.

I would like to thank our collaborators; Assoc. Prof. Dr. Ceyhun Bulutay, Prof. Dr. Veysi İşler and Prof. Dr. Karl-Heinz Heinig for their valuable contributions. I would especially like to thank Dr. Bulutay for his guidance, which significantly improved the quality of our work.

I would like to thank my thesis monitoring commitee and jury members Prof. Dr. Özgür Ulusoy and Prof. Dr. İsmail Hakkı Toroslu and my jury member Assoc. Prof. Dr. İbrahim Körpeoğlu, for spending their time to read and comment on my thesis. I am grateful for their constructive comments.

I am thankful to Erkan Okuyan, Enver Kayaaslan and Ümit Keleş for their discussions and valuable comments on my thesis work.

I am grateful to Koji Koyamada for the volumetric datasets. The Comb dataset is courtesy of NASA. The Sf1 and Sf2 datasets are courtesy of David R. O′ Hallaron and Jonathan R. Shewchuk (CMU). The *Sponge* dataset is the courtesy of Dr. Karl-Heinz Heinig of Helmholtz-Zentrum Dresden-Rossendorf. Quantum dot datasets are the courtesy of Dr. Ceyhun Bulutay.

Finally, I would like thank my wife Ceyda, father Mehmet, mother Gülseren and brother Erkan for their support and understanding throughout my thesis study. Last but not least, I want to thank my little daughter Nihal whose mere existance helped me a lot to finish my thesis study.

*To my beloved wife Ceyda and our precious daughter Nihal...*

# Contents

# List of Figures

# List of Tables

# List of Publications

This dissertation is based on the following publications. The rights to use the whole content of these publications in this thesis are obtained from the publishers.

1. E. Okuyan, U. Güdükbay, and V. İşler. Dynamic view-dependent visualization of unstructured tetrahedral volumetric meshes. Journal of Visualization, 15:167–178, 2012.

2. E. Okuyan and U. Güdükbay. Direct volume rendering of unstructured tetrahedral meshes using CUDA and OpenMP. Journal of Supercomputing, 67(2):324–344, 2014.

3. E. Okuyan, U. Güdükbay, C. Bulutay, K.H. Heinig. *MaterialVis*: Material Visualization Tool Using Direct Volume and Surface Rendering Techniques. Journal of Molecular Graphics and Modelling, 50:50–60, 2014.

# Chapter 1

# Introduction

Direct Volume Rendering is a useful method for examining volumetric datasets. However, it is computationally expensive making it impractical for any reasonable size datasets. We studied this problem from different perspectives: devising a novel view-dependent selective refinement technique and exploring the GPU and multi-core parallelization of the volume rendering algorithms. We developed a material visualization tool, *MaterialVis*, based on direct volume rendering techniques.

We introduced a novel view-dependent selective refinement approach. We aim to selectively reduce the detail level of the dataset at regions that do not significantly contribute to the rendered image quality. This way we can reduce the dataset size, thus speeding up the rendering process significantly, without adversely affecting the image quality noticeably. In order to achieve this goal, we proposed a progressive mesh representation which support LOD (Level of Detail) representation of the dataset. We also proposed a view-dependent selective refinement algorithm, that estimates contributions of each regions in the volume on the rendered image. The algorithm sets the detail level for each volume region according to their importance. This work is published [1].

We explored the parallel implementations of direct volume renderers. We focused on parallel implementations for multi-core CPU's and GPU's, which are widely available nowadays. These parallel implementations utilize the available hardware efficiently and significantly increases the performance. We published this work in [2].

Lastly, we employed direct volume rendering techniques on a field where it has not been used before, but fits in naturally. We developed a volume rendering based material visualization tool, *MaterialVis*, that utilize these techniques. Volume rendering techniques, are quite useful to visualize the embedded topology and defects in materials. Thus, *MaterialVis* offers a quite useful tool that can demonstrate these features of materials better than existing tools. This work is published in [3]. We also created *BilKristal 2.0* tool, which is an extended version of our earlier work *BilKristal* [4], as a supporting tool for *MaterialVis*. *BilKristal 2.0* is published as a new version announcement [5].

This dissertation is based on three research mentioned above. Each of these is summarized in the following sections.

## 1.1 View-Dependent Refinement Techniques

Visualization of large volumetric datasets is an important and challenging area. We focus on the view-dependent refinement of unstructured tetrahedral meshes, widely used in computational fluid dynamics. A representation to store the volume data that allows progressive refinement is crucial for this purpose. A good decimation algorithm is an important factor in constructing the levels of detail of the original mesh to obtain a progressive representation. With a progressive mesh representation, the mesh should be refined during runtime in a view-dependent fashion.

We propose a framework for dynamic view-dependent visualization of unstructured volumetric meshes. The framework uses a progressive representation of the volumetric data that supports view-dependent refinement. The progressive mesh representation is based on the representation presented in [6], with a few key differences. We propose an algorithm that dynamically refines the progressive volumetric data in a view-dependent fashion, without requiring user input. Since the volume data can be highly transparent, a simple view test based on a screen space error threshold will not work. To accurately determine the importance of different regions of the volume data, the whole volume should be rendered. This should be done quickly; we propose a heuristic algorithm that performs a fast simplified rendering of the volumetric mesh. In this way, we can

roughly calculate the importance of the different parts of the mesh for the final image with a small computational overhead.

Two notable studies on selective refinement of tetrahedral meshes are by Cignoni et al. [6] and Callahan et al. [7]. Our work differs from these in one key aspect; regions of the mesh are refined automatically according to their expected impact on the rendered image. We estimate the importance of different regions of the mesh according to the camera parameters, transfer functions etc. and refine the regions with higher importance while coarsening other regions. Cignoni et al. use user input to determine the regions to refine. Users specify certain spatial regions or field values of the mesh and refinements are performed according to this input. Callahan et al. propose several heuristics to determine the resampling of the faces. Most of these heuristics do not consider dynamic properties such as camera parameters; thus they are static resampling methods. Only view-aligned resampling method uses camera parameters, but in a limited way. Our method automatically refines the mesh regions occluded by transparent regions while coarsening the transparent regions. Similarly, it refines opaque mesh regions while coarsening the occluded regions by this opaque region. Our method can be used with any volume renderer that use irregular tetrahedral meshes.

## 1.2 Parallel Implementations on Multi-Core CPUs and GPUs

Volume visualization is useful in many areas. Medical fields extensively benefit from this method, and computational fluid dynamics uses it to inspect several properties of fluid flow. In general, any discipline that studies the internal structure of a volume benefits from volume visualization. Volumetric data can be represented in different forms, depending on the application and data-capture technologies. We focus on directly rendering unstructured tetrahedral meshes where volume's interior needs to be visualized. There are many approaches to rendering unstructured grids and accuracy is usually important. Ray-casting-based methods, which our work focuses on, are widely accepted. These techniques provide accurate results but are computationally costly;

many actual volume datasets contain millions of tetrahedra. Rendering such complex data in a timely manner is a real challenge.

We propose direct volume rendering (DVR) algorithms for parallel architectures that achieve interactive rates for large datasets with good image quality and memory efficiency. We modified the *cell-projection* algorithm described in [8] to exploit parallelization and used OpenMP to parallelize the implementation for multi-core systems. We extended our *cell-projection* algorithm for GPUs using CUDA and focused on enhancing the highly parallelizable characteristic of the algorithm.

There are many works done on volume rendering. A large proportion of recent volume renderers are based on shader programming in order to achieve high rendering rates. Although, such approaches are fast, they have various drawbacks. First of all, shader programming is quite restrictive. There are memory and instruction limitations. The total memory available to each shader unit is quite low. Also the available instruction set is limited and the total number of instructions in a shader program have an easily reachable upper bound. Accordingly, many algorithms are too complex to be implemented with a shader program. Thus, shader-based volume renderers usually use approximation in order to satisfy shader restrictions, leading to inferior graphical quality. These volume renderers have very limited support to incorporate additional features, such as LOD approaches, lighting effects, iso-surface effects.

On the other hand, CPU-based volume renderers have great flexibility. Such restrictions on shader-based renderers do not exist for these renderers. Very accurate images can be rendered and there are no restriction on adding new features. However, CPU-based volume renderers are much slower than shader programming based renderers. They cannot give enough performance to be used interactively.

Our main motivation was to develop a volume renderer without the restrictions of shader-based renderers and is much faster than CPU-based renderers. Our goal was to focus on image quality first. We performed the computations as accurately as possible. For example, some shader-based volume renderers use pre-integration tables, represented as 3D textures, to compute the effects of tetrahedra. The size limitation on the 3D texture limits the accuracy of the computation. On the other hand, we perform the actual computation leading to accurate results. We also did not allow any

visual artifacts. Our second goal is to reach interactive rates while rendering decent image resolutions. We developed a CUDA-based volume renderer to satisfy both of our goals. CUDA provides a rich programming environment that allows the implementation of complex algorithms. Thus, we can implement accurate rendering algorithms and produce high quality images. Since CUDA provides GPU acceleration, the performance would be much higher than CPU implementations. In order to improve the interactive usability we also supported progressive rendering. Progressive rendering allows rendering the volume in low-resolution first and then progressively improving the image to the desired resolution. Since the low-resolution image can be rendered much faster, it can be displayed much earlier, improving the interactivity significantly. Progressive rendering is particularly useful for very high resolutions, where rendering takes much longer.

Our volume renderer can be integrated with additional features, like lighting and iso-surface effects. The modular architecture and the flexible programming environment provided by CUDA, allow the implementation of complex algorithms and easy integration. Our volume renderer can also be integrated with level-of-detail (LOD) approaches. It can be used in a dynamic view-dependent refinement setup ([1]), where the volumetric data can be selectively refined based on viewing parameters. View-dependent refinement can significantly reduce rendering costs without noticeable degradation of image quality.

Memory efficiency is crucial for volume renderers, especially if they are GPU accelerated. Our implementation focuses on keeping the memory overhead as low as possible without significantly affecting the performance. We have also employed mechanisms to allow rendering a volume in several iterations, reducing the memory requirement significantly. Accordingly, our volume renderer can handle large volume datasets.

## 1.3 *MaterialVis*: Material Visualization Tool based on Direct Volume and Surface Rendering Techniques

Extracting the underlying atomic-level structure of natural as well as synthetic materials is vital for materials scientists, working in the fields such as electronics, chemistry, biology, geology etc. However, as the topology and other important properties are buried under a vast number of atoms piled on top of one another, this inevitably conceals the targetted information. Without any doubt, the visualization of such embedded materials can help to understand what makes a certain sample unique in how it behaves. However, rudimentary visualization of atoms would fall short because it will not reveal any topological structure or crystalline defects.

In order to visualize the material topology, the data must be represented as a *surface manifold*, whereas, visualization of crystalline defects require extracting and quantifying defects and representing the data *volumetrically*. Current visualization tools lack such features, and hence, they are not very effective for visualizing the material topology and crystalline defects.

Material visualization tools require atomic coordinates of the materials as input. Acquisition of real-space atomic coordinates of a sample has been a a major obstacle, until recently mainly restricted to the surfaces. One can call this period as the *dark ages* of material visualization. However, recent techniques, such as *Atom Probe Tomography* [9], can extract atomic coordinates much easier than before. This is also a very active research field, with the promise of many new advances in the near future. Accordingly, as the data acquisition phase for materials gets more efficient and accurate, the necessity for sophisticated material visualization tools becomes self-evident.

Our motivation on *MaterialVis* is to provide such a visualization tool that can reveal the underlying structure and various properties of materials through several rendering modes and visualization options. In this way, we intend to provide a good material analysis tool that will be useful in a wide range of related disciplines. *MaterialVis* supports visualization of both amorphous and crystalline structures. Amorphous structures only present the topological features while crystalline structures present both

topological features and defects. The structure of a material can be best visualized using surface rendering methods. The underlying surfaces of the material should be extracted and visualized. On the other hand, defects such as the disposition of some atoms, vacancies or interstitial impurity atoms in the structure, cannot be visualized by simply drawing the atoms or rendering the surface of the crystal. These defects can be best visualized using *direct volume rendering* techniques. *MaterialVis* supports direct volume rendering and surface rendering, as well as combining them in the same visualization. It provides the functionality-driven visualization of the same structure with several techniques; thus it helps the user to analyze the material structure by combining the output of individual rendering modes.

We tested the tool with three real-world and seven synthetic datasets with various structural properties, sizes and defects. For instance, the sponge dataset [10] is a material produced from silicate, which has interesting nano-technological properties. Very recently, it has been experimentally shown that a silicon-rich oxide film can decay into a silicon nanowire network embedded in $SiO_2$ by spinodal decomposition during rapid thermal treatment [11], which has also been confirmed by accompanying kinetic Monte Carlo simulations [12]. The underlying goal in such a line of research is to achieve a nano-scale feature control and transfer it to inexpensive large-scale thin-film technology for silicon-based optoelectronics through growth kinetics. However, the direct imaging of such structures through transmission electron microscopy has not been satisfactory due to low contrast between Si and $SiO_2$ regions. We believe that it forms an ideal candidate for demonstrating the need for a direct volume imaging tool.

Direct volume rendering so far has not been widely used in material visualization, even though it is a well-studied subject in other application domains, such as medicine and computational fluid dynamics. Direct volume rendering algorithms render the volumetric data without generating an intermediate surface representation; they are useful when the inside of a material, such as a translucent fluid or gas, should be rendered. The 3D representations of the human body parts can be constructed from the images obtained using magnetic resonance imaging (MRI) techniques. These representations can be visualized using direct volume rendering techniques where the partially transparent body fluids are visible. The temperature and pressure variations in an engine block can be visualized using direct volume rendering techniques. Volume rendering

techniques are especially useful for the visualization of attribute variations in the volume. Crystals are usually homogeneous structures that lack of such features. However, we observed that direct volume rendering is a suitable way to visualize the crystal defects because one can easily accentuate the errors by changing the viewing and lighting parameters and colors.

# Chapter 2

# Related Work

Volume rendering is a well-studied subject. There are two main types of volume data: regular and irregular grids. Regular grid representations are widely used in medical imaging, with texture-based techniques dominating. Earlier approaches sampled volume with parallel planes along the view direction [13, 14]. The nature of graphics card allows storing the volume data in the GPU as 3D textures; Ertl et al. used a pre-integration approach to efficiently render volume using 3D texture representation [15, 16].

Although regular grids can be efficiently rendered, they can be large, limiting the detail level of the volume data. Unstructured grids can be represented in much more compact form, thus they can reach much higher detail levels. Iso-surface techniques allow fast rendering of volume data, which can be useful if surfaces are the critical regions in the volume. Lorensen and Kline proposed the Marching Cubes algorithm [17], which became the basis for many later algorithms.

In our work, we focus on direct volume rendering algorithms, of which visibility ordering is an important part. If the mesh primitives (faces or polyhedra) are ordered in a way that no primitive is occluded by an earlier primitive in the list, the list is visibility ordered. Such lists can be efficiently rendered by graphics cards. Cook et al. [18] and Kraus and Ertl [19] proposed methods for efficient visibility sorting. Shirley and Tuchman proposed a projected tetrahedra algorithm [20], which was later extended to GPUs using vertex shaders by Wylie et al. [21]. Maximo et al. [22] present some

methods to render tetrahedra as primitives using vertex shaders.

Garrity [23] and Koyomada [24] exploited connectivity to achieve fast cell traversals. Koyamada's algorithm [24], is one of the earlier influential algorithms that is more suitable for software implementations. This approach was later extended to GPUs by Weiler et al. [25], where the mesh and the connectivity information are represented as 3D and 2D textures, respectively. Callahan et al. introduced a visibility ordering algorithm, HAVS [26, 7], which performs a rough sorting on the CPU and finalizes the sorting in the GPU. The initial CPU sorting phase sorts the face primitives according to their center-to-eye distances. The resulting list contains errors but they are corrected in the GPU using the *k-buffer* approach. See Silva et al. [27] for an extensive survey of volume rendering techniques.

Mesh simplification is an important part of our work. There are various types of mesh representations and simplification algorithms proposed for them. Many triangular mesh simplification algorithms could be used as base for tetrahedral mesh simplification algorithms. Hoppe proposes the progressive mesh representation in [28, 29]. This representation is efficient and well-accepted, allowing view-dependent refinement of the mesh in a progressive fashion. The error metric used in a simplification algorithm is very crucial. Garland and Heckbert propose a quadric error metric in [30], which is used in many simplification algorithms.

Trotts et al. simplify tetrahedral meshes via repetitive collapses of the tetrahedra's edges [31]. Tetfusion collapses a tetrahedron into a vertex in one step [32], iteratively selecting the tetrahedron that will cause minimal error to the mesh. Staadt and Gross propose dynamic tests to avoid tetrahedron flips altogether [33]. These algorithms do not support level-of-detail adjustments or view-dependent refinement.

Visibility ordering is an important part of volume rendering algorithms. Cook et al. [18] and Kraus and Ertl [19] propose methods for performing visibility sorting efficiently. Shirley and Tuchman proposed a projected tetrahedra algorithm [20] for visibility sorting. Wylie et al. [34] later extend this algorithm to GPUs using vertex shaders.

Cignoni et al. [35] develop a multiresolution representation for volume data, using refinement-based and decimation-based approaches. Their model supports view-dependent refinement. They select mesh regions from different detail levels and merge them, and correct inconsistencies on the connecting surfaces. Cignoni et al. [6] also propose a progressive mesh representation that supports view-dependent refinement. This approach refines the mesh based on selective refinement queries specified by the user whereas our approach automatically refines the mesh based on camera parameters. Du et al. [36] propose an out-of-core simplification algorithm and crack-free LOD volume rendering. This approach also support selective refinement with user queries. Sondershaus et al. [37] propose a segmentation-based mesh representation of volume data, which allows view-dependent refinement using a hierarchy of pre-constructed segments. Our framework allows view-dependent refinement based on the progressive mesh representation.

As for the material visualization, there are many commercial and free tools. *CrystalMaker* [38], *Shape Software* [39], *XtalDraw* [40], *Vesta* [41], *Diamond* [42] and *Mercury* [43] are some examples. There are also some studies on the analysis of crystals that also provide some visualization functionality, such as the work of by Ushizima et al. [44]. These tools are essentially crystal analysis tools, which also provide some visualization functionality. Their visualization capabilities are not very advanced. They mostly offer just atom-ball models with some variations. Some of the tools support primitive surface rendering, which allows examining the crystal on the unit cell level. However, they are not sufficient to examine the underlying topology of a dataset.

There are also general visualization tools such as *AtomEye* [45], *VisIt* [46], and *XCrySDen* [47]. These tools provide sophisticated visualization capabilities but they lack the ability to create volumetric representations of materials, cannot use direct volumetric rendering techniques, and cannot quantify defects of crystal structures.

Iso-surface rendering techniques provide fast surface rendering of the volume data. They are especially useful when the surfaces are the regions of interest for the volumetric data. Doi and Koide [17] propose an efficient method for triangulating equi-valued surfaces by using tetrahedral cells based on the Marching Cubes algorithm [48].

*MaterialVis* is primarily based on direct volume rendering. There are mainly two types of volume data. The first type is the regular grid representation, which is widely used in medical imaging. Mostly texture-based techniques are used for the visualization of regular grids. Earlier approaches use sampling the volume along the view direction with parallel planes [13, 14]. New graphics cards allow storing the volume data as 3D textures in the GPU. Ertl et al. [15, 16] use a pre-integration mechanism to render the volume using 3D textures. Regular grid representation can be rendered efficiently, but the datasets using this representation are very large. The second type of data, unstructured grid representation, can be significantly compacted, so it can give much higher detail levels for the same size.

# Chapter 3

# View-Dependent Selective Refinement

## 3.1 Proposed Framework

We propose a framework for the view-dependent refinement of unstructured volumetric models. The framework supports direct volume visualization and selective refinement of different parts of the model for different viewing parameters. The framework is based on a new progressive volume-data representation that supports selective refinement. The detail level of the mesh can be set independently at different parts of the model depending on the viewing parameters. The framework consists of three stages. The first stage constructs the progressive mesh representation that will store the volume data. The second stage determines the detail levels of different parts of the volume according to the viewing parameters via the selective refinement algorithm. The last stage uses the direct volume renderer to support direct volume visualization for different viewing parameters with the proposed progressive representation.

Figure 3.1 gives an overview of the proposed framework. The tetrahedral mesh is the input of the framework, containing the vertices, which includes the position and scalar values, and the tetrahedra. The mesh simplification tool works on the input and creates the progressive mesh representation (PMR). It uses a decimation algorithm to obtain lower detail levels of the mesh and creates the vertex hierarchy that is the backbone of the PMR. The construction of the PMR is the preprocessing step of the

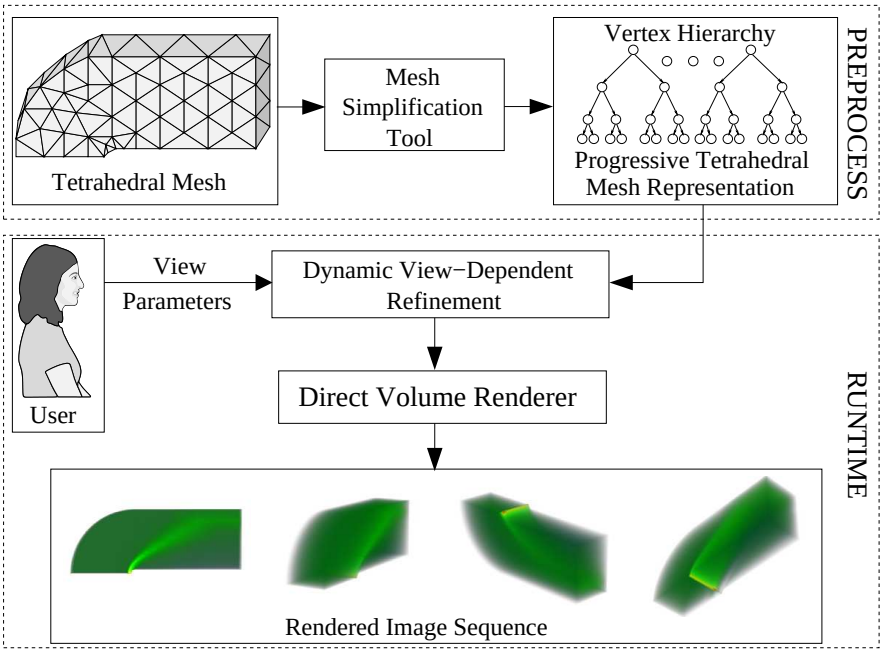



Figure 3.1: The overview of the proposed framework.

framework; the PMR is used as input to the volume renderer.

In order to render the PMR, the dynamic view-dependent refinement algorithm refines the PMR according to the viewing parameters. This algorithm first estimates which regions will have a higher impact and which regions will have a lower impact on the output image. Then the costs and benefits of refining and coarsening different parts of the mesh can be estimated. The important parts of the mesh are represented in high detail, and the unimportant parts in lower detail. Using this method, the simplification ratios can be much higher than using non-view-dependent detail adjustment approaches for the same target quality.

### 3.1.1 Volumetric Data Representation

We use a volumetric data representation that allows view-dependent volume rendering. Our representation is based on the data representation presented by Cignoni et al. [6] with some key differences. Similar to Cignoni et al.'s work, we use an edge-collapse based decimation algorithm to obtain lower detail levels. Repeated edge collapses

construct binary vertex trees, which are the backbone of the progressive mesh representation.

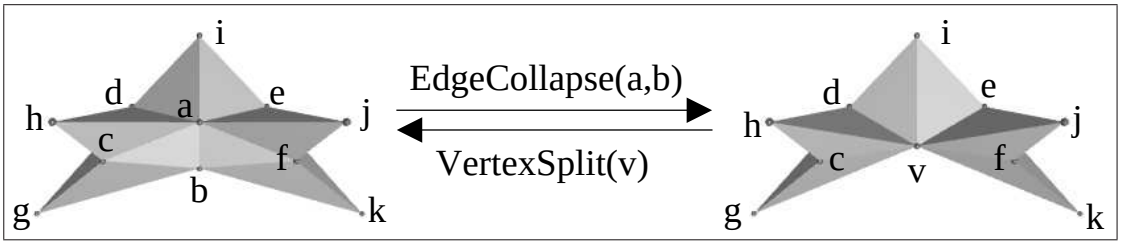



Figure 3.2: Vertex split and edge collapse.

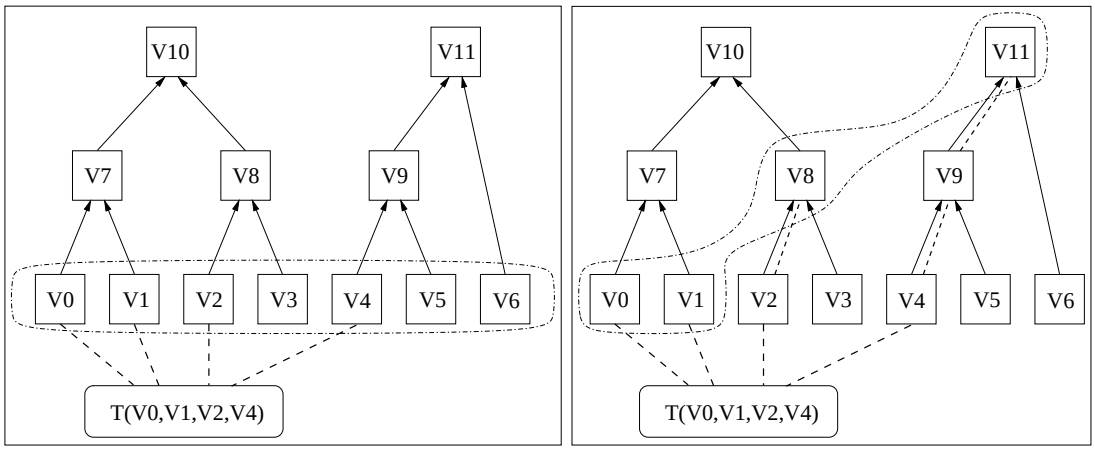
Figure 3.2 shows the two basic operations of the edge-collapse decimation algorithm. Edge collapses and vertex splits, which are the inverse of each other, are used to coarsen or refine the mesh. Figure 3.2 presents a simple tetrahedral mesh with eleven vertices and eight tetrahedra. The tetrahedra are *abcd*, *abde*, *abef*, *acdh*, *adei*, *aefj*, *befk*, and *bcdg*. The edge *ab* is collapsed into vertex *v*. As a result of this collapse, the tetrahedra that use both *a* and *b* vertices are collapsed as well. The tetrahedra that use one of the *a* and *b* vertices are modified to use vertex *v* instead. The resulting mesh contains five tetrahedra: *vcdh*, *vdei*, *vefj*, *vefk*, and *vcdg*. Splitting the vertex *v* restores the vertices *a* and *b*, obtaining the initial version of the mesh.

There are two key differences between Cignoni et al.'s representation and ours. The first difference is the active vertex mechanism, which eliminates the necessity of maintaining tetrahedral information during the refinement. The second one is related to handling possible tetrahedral flips during selective refinement.

### 3.1.2 Active Vertex Mechanism

In Cignoni et al. [6], whenever a vertex splits or an edge collapses, affected tetrahedra are updated accordingly. This brings significant overhead. We avoid this using active vertex mechanism. Before the mesh simplification begins, the tetrahedra contain pointers to the vertices in the original mesh. During the simplification, pairs of vertices are collapsed into a newly created parent vertex, and tetrahedra that use one of these collapsed vertices must be modified to use the newly created parent vertex. Figure 3.3 (a) shows an example vertex hierarchy. In this example, the vertices of the original mesh

are all active; i.e., the mesh is in its finest state. The tetrahedron $T$ consists of four vertices, $v_0$, $v_1$, $v_2$, and $v_4$, which are all active in this example. Assume that we change the detail level of this mesh by performing some collapses and thus obtain the mesh shown in Figure 3.3 (b). In this case, only the vertices $v_0$, $v_1$, $v_8$, and $v_{11}$ are active. The vertices $v_2$ and $v_4$ of $T$ are no longer active. We must now use a mechanism that will map $v_2$ to $v_8$ and $v_4$ to $v_{11}$ because $v_8$ and $v_{11}$ are the vertices that represent the inactive ones.




(a) (b)

Figure 3.3: Active vertex mechanism: (a) initial mesh, (b) simplified mesh.

Our idea is to keep the references in the tetrahedron structure and find the active vertex that represents the vertex stored in the tetrahedron structure during runtime. For example, when the volume renderer processes $T$ in Figure 3.3 (b), it requests vertices $v_0$, $v_1$, $v_2$, and $v_4$. $v_0$ and $v_1$ are active vertices and can be used. $v_2$ and $v_4$ have to be mapped to the active vertices. The mapping is done by following the parent links, until an active vertex is found. The dashed lines in Figure 3.3 (b) show such link traversals. With a simple caching mechanism, the overhead of these traversals is reduced significantly. Whenever an active vertex of a vertex is accessed, first the validity of the cached information is checked. If the cached information is valid, then it is used. Otherwise, with the described link traversals, correct information is found. The cached information along the traversal path are also updated. Thus, a link traversal would be required only if there has been a related vertex hierarchy change which invalidate the cached information. Accordingly, active vertex mechanism does not bring any redundant link traversal overhead.

The volume renderer traverses each tetrahedron and finds its active vertices. If all active vertices of a tetrahedron are different, then the tetrahedron is active. Otherwise, the resulting geometry is not a tetrahedron. While refining or coarsening the mesh with this mechanism, simply maintaining the vertex hierarchy is sufficient; maintaining the active tetrahedra, which could take up a significant time, can be avoided. The active vertex mechanism also groups the task of finding the active tetrahedra together. If the tetrahedral information were to be updated, the list of active tetrahedra could be maintained during the refinement. However, this job would be distributed among many refinement operations. With active vertex mechanism, such list can be maintained with a single traversal of tetrahedra. Such traversal can be very efficiently parallelized with GPU whereas tetrahedral updates have to be done serially on CPU. Thus, even though active vertex mechanism increases the total work volume of finding active tetrahedra, since all the tetrahedra have to be traversed not just affected ones, due to parallelism the process would be faster.

### 3.1.3 Progressive Mesh Representation of Volumetric Data

The two data structures defining the unstructured tetrahedral meshes are the vertex and tetrahedron structures. The vertex structure minimally contains the coordinates and the scalar value of the vertex. In order to support selective refinement, our representation adds additional fields; active vertex id, pointer to edge-collapse/vertex-split record, parent and child vertex pointers. Parent and child vertex pointers define the vertex hierarchy and active vertex id stores the cached active vertex information. Edge-collapse/vertex-split record stores the information that will be used to split a vertex or collapse an edge, such as error values and the affected vertices. The tetrahedron structure contains ids of the tetrahedron's vertices in the finest mesh. Since the number of vertices is significantly smaller than the number of tetrahedra, the memory overhead of the added fields is relatively small.

We use an edge-collapse based decimation algorithm. The decimation algorithm iteratively selects the edges and collapses them until the desired simplification level is reached. At each iteration, a prey edge is selected, collapsed and the mesh consistency is maintained. The success of the algorithm depends on the collapse order of the edges,

which is based on geometric and attribute errors. The quadric error metric proposed by Garland and Heckbert [30] is used to determine collapse errors. To ensure mesh consistency, tetrahedron flips should be handled. Collapse operations affect some tetrahedra by moving one of their vertices to the opposite side of their unaffected face, thus flipping their volume (cf. Figure 3.4). The edges that would cause tetrahedron flips are not collapsed.

The boundary surface geometry is extremely important for mesh quality. Deformations of the surface produce significant visual impairments. Not allowing any surface edge to collapse eliminates surface deformations. However, it is quite restrictive and adversely affects the achievable simplification ratios. We classify surface vertices as *sharp* and *smooth*. A surface vertex $v$ is considered sharp, if the angle between the normals of any two faces on the surface using the vertex is more than a certain threshold. We do not allow the collapse of any edge that contain a sharp vertex. This issue could also be solved by including boundary preservation into the error metric.

Another issue of mesh consistency is self-intersections, which cause similar problems like tetrahedron flips. Surface preservation eliminates most of the self-intersection cases. During our experiments, we did not observe any artifacts due to self-intersecting mesh. However, our framework does not guarantee to eliminate all the cases. In order to reduce computational complexity, we prefer not to perform extra checks to eliminate all these cases. The approaches described by Cignoni et al. [49] and Staadt and Gross [33] could be used to avoid self-intersections during decimation.

In the preprocessing stage, the decimation algorithm constructs the PMR. The PMR supports selective refinement of the mesh during runtime using vertex splits and edge collapses. The algorithm can only collapse edges whose vertices are both active and siblings in the hierarchy. Figure 3.3 (b) provides examples of allowed edge-collapse and vertex-split operations. The vertices $v_0$, $v_1$, $v_8$ and $v_{11}$ are active in the given example. Because no other pair of sibling vertices are active, only the edge between $v_0$ and $v_1$ can be collapsed. If that edge is collapsed, then $v_7$ becomes active, thus the edge between $v_7$ and $v_8$ becomes a candidate for collapse. Similarly, only the vertices $v_8$ and $v_{11}$ are candidates for vertex splits, since they are the only active vertices who have children.

The vertex-split operation activates the child vertices and deactivates the input vertex. Then the vertex split candidates and the edge collapse candidates structures are maintained. The edge-collapse operation is executed in a similar manner to the vertex-split operation. Basically, these two operations are inverses of each other.

These two operations simply activate or deactivate vertices in the vertex hierarchy. The tetrahedra information is not updated with these operations as the direct volume renderers must check if a tetrahedron is active or inactive before processing it. Thus, updating the vertex hierarchy is sufficient to refine or coarsen the mesh selectively.

## 3.2 View-Dependent Refinement

The proposed progressive mesh representation supports selective refinement during runtime. The regions of the mesh that have higher impact on the rendered image are automatically refined, while other regions coarsened. This ability is the main difference between this work and other related works. To this end, the detail levels of different parts of the mesh should be determined according to the viewing parameters. However, determining the important regions of the mesh is not an easy task. The volume is rendered through tetrahedra but the refinement is performed on vertex hierarchies. Thus the effect of the refinement of a vertex will be distributed through the volume of tetrahedra that use it. Furthermore, the refinement of seemingly unrelated parts of the mesh will affect the geometries of several tetrahedra and change the effects of previous refinements. Mapping vertex hierarchies to tetrahedra is non-trivial, making selective refinement a non-trivial task. We develop a heuristic algorithm, taking into account several parameters that affect the importance of a vertex-split/edge-collapse operation; i.e., how much the output image quality changes after performing the operation. Five parameters contribute to the importance metric. The first two represent the normalized mesh error values introduced with an edge collapse. The other values represent the weight of the mesh error, affecting the output image. The formulations of the parameters will be given on the vertex split operation $v \rightarrow (p,q)$.

*Color error:* The scalar values of vertices are used to calculate the color values using

the transfer function, which is then used to calculate the color error. The color error can be defined as $|p.color - v.color|^2 + |q.color - v.color|^2$, where colors are normalized RGBA vectors.

*Geometric error:* The geometric error can be defined as $|p.position - v.position|^2 + |q.position - v.position|^2$, where the positions are coordinates of the vertices. Since vertex coordinates does not change via the selective refinement, geometric errors can calculated in the preprocessing stage.

*Light intensity:* The regions of a volume affect the final image in a way directly proportional to the intensity of the light reaching these regions.

*Affected volume:* The affected volume for a vertex represents the total volume of tetrahedra that will be affected by the refinement of that vertex. The larger the volume, the bigger the affected image segment will be.

*Camera distance:* The regions of the volume close to the camera usually have high impact on the rendered image, especially for opaque surfaces.

Ray-casting-based volume renderers send rays through each pixel. While passing through the volume, rays lose some of their intensity depending on the transparencies of the tetrahedra on the path. The effect of each tetrahedron that the ray visits are combined to calculate the pixel's intensity. The intensity of the light reaching a region directly affects how much that region can contribute to the color of the ray. Thus, calculating light intensities in each part of the volume is necessary to determine the importance of that region, which is computationally intensive. As a solution, we approximate the intensities to determine the importance of different regions.

In order to answer light intensity queries in a timely manner, we construct an octree representation of the volume data. The octree does not replace the proposed PMR, but is a low-resolution representation of the original data. The octree structure is constructed bottom-up using only the vertices. The approximate light intensity calculations are performed during runtime before rendering. In principle, the calculations are similar to any ray-casting-based volume renderer. Due to the regularity of the octree structure, the approximate light intensity calculations can be done very efficiently. The octree should be updated during refinement operations. We use flooding-based

techniques to update the octree during the refinement. Whenever a refinement operation changes the vertex hierarchy, the corresponding octree cell is found. The owner vertices of the cells are updated starting from this cell and continuing to its neighbors.

### 3.2.1 Importance Metric

The selective refinement algorithm must decide the vertices to split and edges to collapse in the vertex hierarchy. Since the algorithm must determine the edge collapse/vertex split operations that will be executed, the importance metric should be defined for edge collapse/vertex split records. Since the vertex hierarchies do not directly reflect the specific effects of the vertices, defining an ideal importance metric is very difficult; thus we employ a heuristic approach. We use the weighted combination of a few parameters as the importance metric. We multiply the parameters in order to combine them and use the exponents as weights. The importance metric is given in Equation 3.1.

$$\begin{aligned} I(v) =& ColorError(v)^{\alpha} \times GeometricError(v)^{\beta} \times LightIntensity(v)^{\gamma} \times \\ & Volume(v)^{\theta} \times CameraDistance(v)^{\sigma} \end{aligned} \tag{3.1}$$

Selective refinement splits the vertices with the highest importance and collapse the edges with lowest importance. However, since refinements can only be done on active vertices in the vertex hierarchy, the refinement of a seemingly unimportant vertex can enable the refinements of more important vertices. The importance metric given in Equation 3.1 is updated to take this into account by taking a weighted average of the importance of a vertex and the importances of its children (Equation 3.2).

$$I_{updated}(v) = \quad 0.50 \times I(v) + 0.25 \times I(v.child_0) + 0.25 \times I(v.child_1) \tag{3.2}$$

Because the importance metric is used during the view dependent refinement, it must be calculated on the fly. Even though we use approximations for many parameters, the rendered images do not contain notable artifacts. Determining exponent

weights in the importance metric is an important problem on its own. The optimal weights highly depend on the mesh characteristics and viewing parameters. Since the importance values are used for comparisons, the respective values of weights are important; i.e., scaling all weights up or down will have no effect. Color errors and geometric errors are computed only using the vertices of edge collapse/vertex split operations. For the collapse of an edge of a very thin tetrahedron, color errors and geometric errors can be very high. However, since the affected volume will be small, the effect of such high errors would be small. The weights of geometric errors ($\beta$) should be higher when the tetrahedra of the mesh are more regular; i.e., closer to Delaunay tetrahedralization, since the affected volume will be more related to geometric errors. The weight of the color errors ($\alpha$) also depend on the affected volume. It should be higher for meshes with regular and uniform sized tetrahedra, since the affected volume sizes of the vertices would be closer to each other. The affected volume parameter is included in the importance metric as a support mechanism for geometric and color errors. For tetrahedral meshes with high regularity and uniformity, $\alpha$ and $\beta$ values should be higher compared to the weight of the affected volume parameter ($\theta$). However, for more irregular meshes, $\theta$ should be higher compared to $\alpha$ and $\beta$ values.

The weight of the light intensity parameter ($\gamma$) depend on the accuracy of light intensity calculations. Light intensity calculations use a low-resolution regular grid rendering mechanism, which will introduce a blending effect on the light intensity estimations. If the opacity of the mesh is more uniform, the accuracy will be better. Otherwise, due to the blending effect, the accuracy will be worse. The $\gamma$ value should be higher for more accurate estimations.

Although the affected volume parameter is very important it does not directly reflect the affected image area. If the volume is closer to the camera, the affected image area would be larger. The camera distance parameter is used to correct this effect. The vertex camera distance is converted to a coefficient that will relate the affected volume parameter to the size of the affected image region. The weight of the camera distance parameter ($\sigma$) should be equal to $\theta$. However, the camera distance parameter also have correction effect on light intensity parameter. Usually, the blurring effect of the intensity calculations builds up for distant parts of the mesh, making intensities of these parts less accurately estimated than closer parts. Favoring the closer parts of the

mesh results in better refinement, due to higher accuracy of light intensity estimations.

We employed a semi-automatic method to determine the weights. First, a few sample camera parameter values are selected to perform the tests. Then the mesh is rendered at the highest detail with these values to obtain the reference images. Initial weights are determined as described above or can be set to 1. We perform a convergence-based tuning approach. During this step, we set $\sigma$ equal to $\theta$ for simplicity. After we determine the converged weights, we refine the $\sigma$ value. To quantify the success of a certain weight set, we refine the mesh with these weights to 15%, and render with the sample camera parameters. The output images are compared with the reference images using PSNR (peak signal-to-noise ratio) as the quality metric. PSNR is an exponential metric and higher PSNR values indicate higher quality. The average of PSNR values indicates the performance of the tested weight set. At each iteration of the convergence algorithm, we individually scale up and down the $\alpha$, $\beta$, $\gamma$ and $\theta$ parameters, and obtain eight new weight sets. We test these sets and compare them with the current set. We select the best performing weight set and continue the convergence algorithm with it. The iterations end when the current weight set performs better than all of the newly tested weight sets.

The optimal weights are sensitive to the changes in opacity mapping due to the change in transfer functions. However, only $\alpha$ and $\gamma$ values are affected. Thus, re-running convergence algorithm by starting from the previously optimal weights and working only on these two parameters will re-optimize the weights much faster. Also pre-computing the optimal weights for a few opacity map profiles would work well. Since the weights are not very sensitive to small changes, such approach would work eliminating any need to re-optimization of weights during runtime.

The selective refinement algorithm is a heuristic algorithm that sets different regions of the volume data to the appropriate level of detail. It is called just before the rendering and refines the mesh according to the viewing parameters. The algorithm keeps track of every vertex that can be split and every edge that can be collapsed. The importance of these primitives are calculated and the mesh is updated accordingly for the desired detail level.

### 3.2.2 Required Modifications for Volume Renderers

The proposed selective refinement framework can be used with a wide variety of direct volume renderers. To this end, two modifications are needed on volume renderers. The first modification is to use the active vertex mechanism for selective refinement. The second modification is handling tetrahedron flips during selective refinement. When a tetrahedron flip occurs, the volume of the flipped tetrahedron is covered by more than one tetrahedron. That inconsistency can cause artifacts. Figure 3.4 presents a simple example to demonstrate this point. The collapse of the edge $(a,e)$ causes tetrahedron $T_{abcd}$ to flip and cover some volume below its base face. The tetrahedra $T_{abch}$ and $T_{bcde}$ are also affected with the edge collapse and they are stretched to cover the volume of $T_{abcd}$.

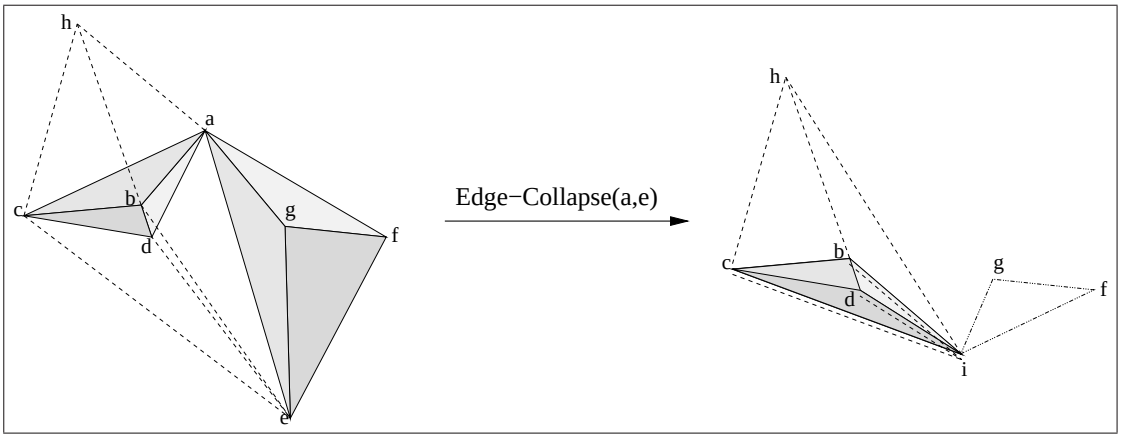



Figure 3.4: Covering a volume multiple times in a tetrahedral mesh due to tetrahedron flips.

Flipped tetrahedra cause artifacts for two reasons. They can cause over-rendering of the flipped volume and mis-representations of the volume. Flips can be handled in different ways, depending on the volume renderer and rendered datasets. One approach is to allow tetrahedron flips. It is suitable for volume renderers, which process the mesh as a set of faces, such as HAVS [26]. HAVS extracts the faces from the tetrahedra and sorts them in visibility order. It determines the contribution of a face on a pixel according to the distance between the face and the next face; thus, the flipped tetrahedra cannot cause over-rendering. They may cause mis-representations. Another approach is subtracting the contributions of the flipped tetrahedra. Since the flipped tetrahedra

can be considered to have negative volumes, subtracting their contributions prevents over-rendering. It also cannot eliminate mis-representations. This approach is suitable for ray casting-based volume renderers. The mis-representations caused by tetrahedron flips do not cause notable artifacts for most datasets. If the artifacts are noticeable, tetrahedron flips should be eliminated, e.g., using the approach described by Cignoni et al. [6] and El-Sana et al. [50]. However, it introduces extra computational and memory overhead for selective refinement, since directed acyclic graphs have to be constructed and maintained. These approaches can be incorporated into our framework without any difficulty.

## 3.3 Results

We analyze the performance of the proposed framework on different datasets. We use HAVS [26] for volume rendering to measure rendering times for selective and non-selective refinement. The *k-buffer* size is set to 6. We also use a software-based volume renderer, which is slow but generates high quality images, with the proposed framework to compare the image quality of selective and non-selective refinement schemes. Transfer functions are selected to highlight the important features of the volumetric datasets. Transfer functions leading to blurry images or very opaque transfer functions are avoided. Some sophisticated techniques, such as visibility-driven transfer functions [51], could also be used for this purpose. We use a wide range of camera parameters, in order to highlight the dynamic view-dependent refinement property. We compare the selective refinement scheme with non-selective refinement scheme. Non-selective version refines the mesh in reverse decimation order.

With the datasets used in experiments, HAVS generated some artifacts due to insufficient *k-buffer* size, particularly for simplified meshes where tetrahedra become more irregular. Since higher *k-buffer* sizes are not supported with current implementation, we were not able to compare our simplification method against HAVS's LOD methods [7]. We also were not able to compare our approach to Cignoni et al.'s selective refinement queries [6], since the implementation is not publicly available.

We used four well-known datasets in our experiments. The *Bucky-Ball* dataset represents the $C_{60}$ molecule. It is a cube shaped dataset with internal structure. The *Comb* dataset represents the temperature and pressure fields inside a combustion chamber. The *Aorta* dataset represent the structure and pressure fields of a human aorta. It has a very irregular shape. And the *Sf2* dataset presents geographic information about certain parts of San-Francisco city.

In the first group of tests, we set a certain level of image quality as the target quality. Then the mesh is selectively and non-selectively refined until the target quality is reached. The rendering times indicate the success of the refinement schemes. The quality of the mesh is measured in terms of *Peak- Signal-To-Noise Ratio* (PSNR) values, a widely accepted logarithmic scale for image comparisons. PSNR values are not perfectly accurate, however they are a good tool for general evaluations. For selective refinement, the average per-frame overhead is added to the rendering times. For comparison, the finest and coarsest meshes are also included in the tests. Please note that the quality comparisons of coarsest, selectively refined and non-selectively refined meshes are done using finest meshes as the reference. Accordingly, PSNR values for finest meshes are not available.

| Dataset | Bucky | | | | Comb | | | |
|---|---|---|---|---|---|---|---|---|
| LOD | F | S | N | C | F | S | N | C |
| Refinement ratio | 100.0 | 12.3 | 47.1 | 0.5 | 100.0 | 10.2 | 70.8 | 3.0 |
| No. tetrahedra | 1250.2 | 153.9 | 588.4 | 6.5 | 215.0 | 21.9 | 152.2 | 6.4 |
| PSNR value | N.A. | 32.43 | 32.37 | 10.41 | N.A. | 37.46 | 37.36 | 16.57 |
| Rendering times | 718 | 406 | 609 | 203 | 125 | 64 | 108 | 46 |

| Dataset | Aorta | | | | Sf2 | | | |
|---|---|---|---|---|---|---|---|---|
| LOD | F | S | N | C | F | S | N | C |
| Refinement ratio | 100.0 | 19.4 | 38.0 | 2.0 | 100.0 | 9.6 | 83.6 | 0.9 |
| No. tetrahedra | 1386.9 | 255.1 | 526.5 | 28.0 | 2067.7 | 199.3 | 1728.5 | 19.3 |
| PSNR value | N.A. | 38.19 | 38.16 | 18.39 | N.A. | 40.94 | 40.39 | 14.71 |
| Rendering times | 593 | 327 | 405 | 187 | 936 | 344 | 842 | 266 |

Table 3.1: The PSNR values and rendering times for various volumetric datasets. F: Finest; S: Selective Refinement; N: Non-selective; C: Coarsest. No. tetrahedra are given in thousands and Rendering times are given in milliseconds

The tests are performed on a PC with an *nVidia 8800GT* graphics card and an

*Intel Core 2 Duo 2.66GHz* CPU. The resolution is $512 \times 512$. The preprocessing step takes less than 30 minutes for the largest dataset, Sf2 with over two million tetrahedra. Table I show that selective refinement gives significant improvement over non-selective refinement. Depending on the dataset, up to 60% speed-ups are observed. Selective refinement also significantly reduces the number of rendered tetrahedra (up to 88%), which greatly reduces the memory requirements on the GPU.

The selective refinement is more successful for datasets where the parts of the mesh that define the features in the output image are spatially localized. Selective refinement can find and refine such localized regions where non-selective refinement cannot focus on a certain part of the mesh. Due to the irregular topology of the Aorta dataset, the octree representation is not as successful as in other datasets. Accordingly, selective refinement performance is affected. Using a higher octree size could produce better results for this dataset.

In the second group of tests, we compare the quality of images that selective and non-selective refinement schemes generate for a fixed budget of rendering time. The finest and coarsest meshes are also rendered to give a comparison. Figures 3.5, 3.6, 3.7, and 3.8 show that significant quality improvements are obtained with selective refinement. For some datasets, the quality of the selectively refined mesh reaches just below the finest mesh, for a fraction of the rendering time. Figure 3.9 shows frame rendering times and PSNR values for the visualization of Bucky and Sf2 datasets. The resolution for the animations are $1024 \times 1024$.

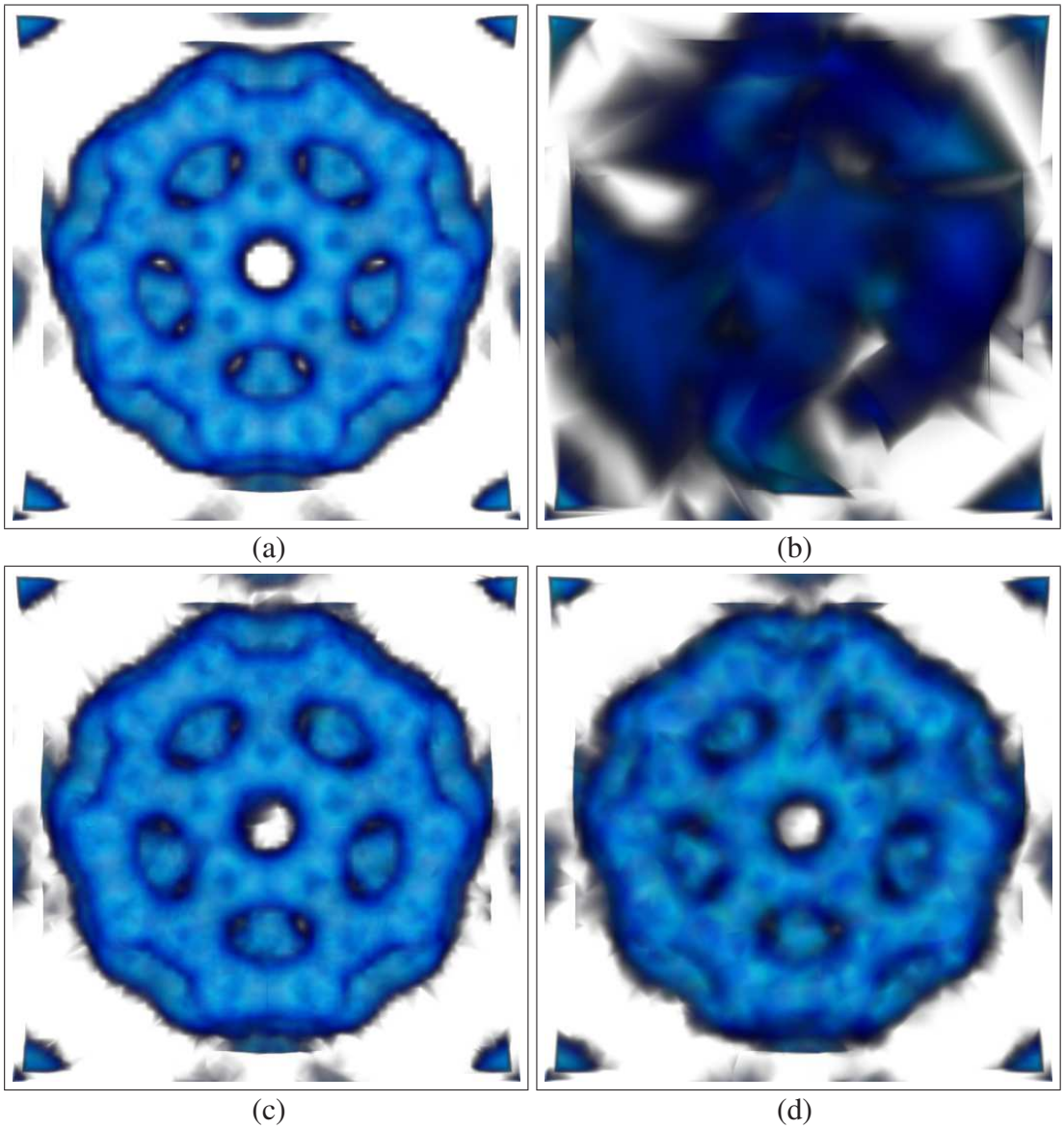
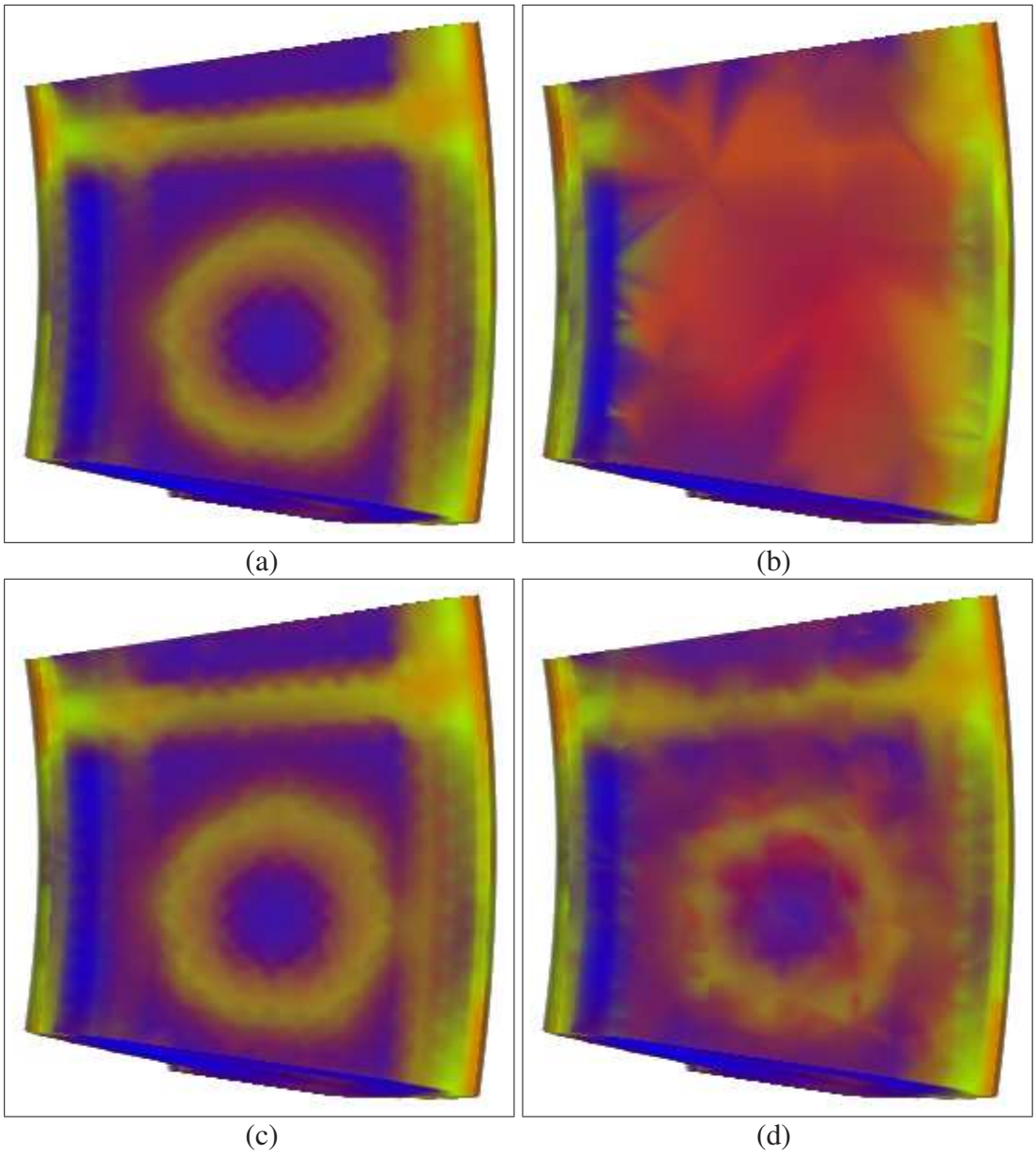



Figure 3.5: Rendered images of the Bucky dataset. a) finest, b) coarsest, c) selectively-refined, d) non-selectively refined



(a) (b)

(c) (d)

Figure 3.6: Rendered images of the Comb dataset. a) finest, b) coarsest, c) selectively-refined, d) non-selectively refined

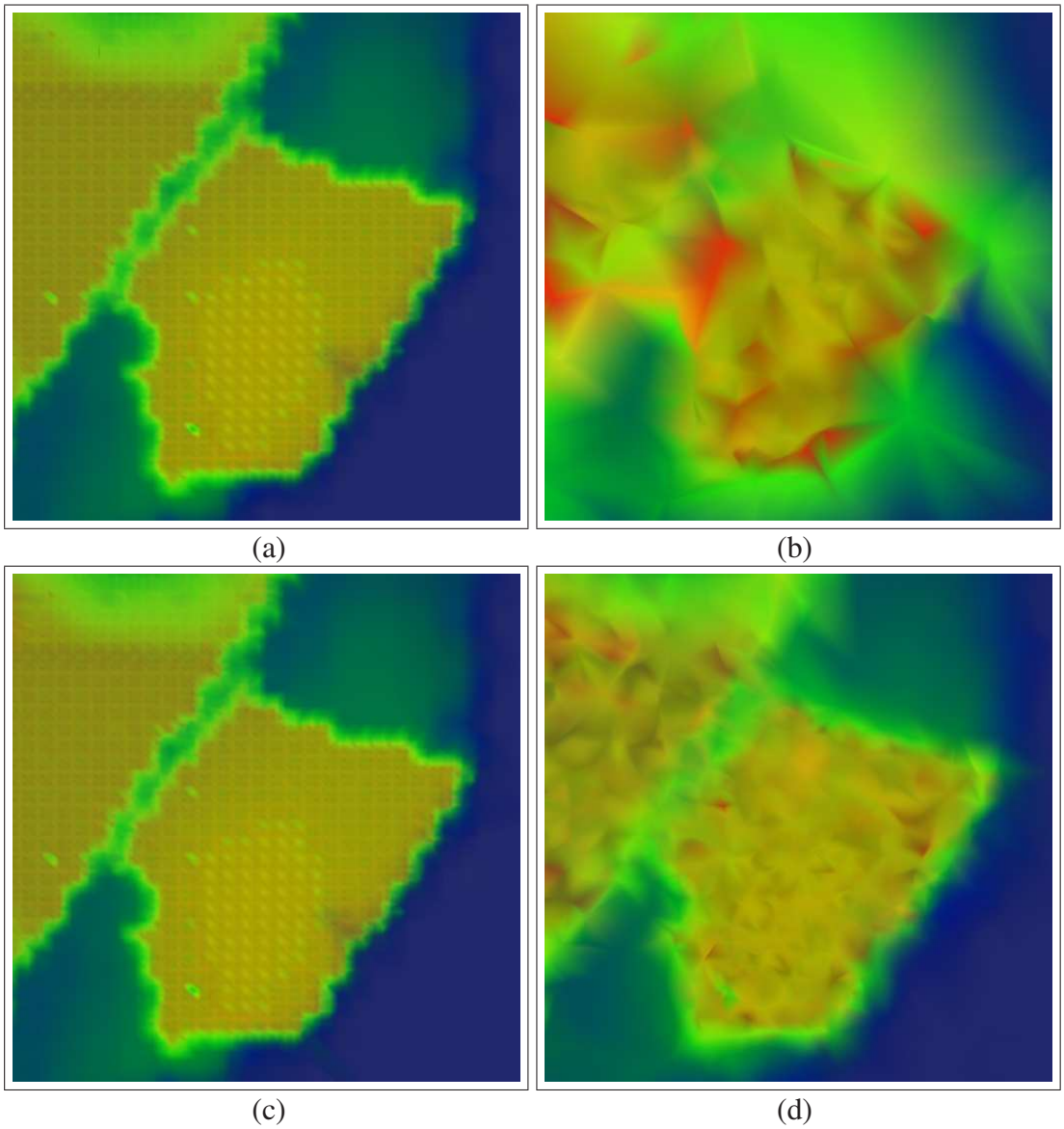



Figure 3.7: Rendered images of the Sf2 dataset. a) finest, b) coarsest, c) selectively-refined, d) non-selectively refined

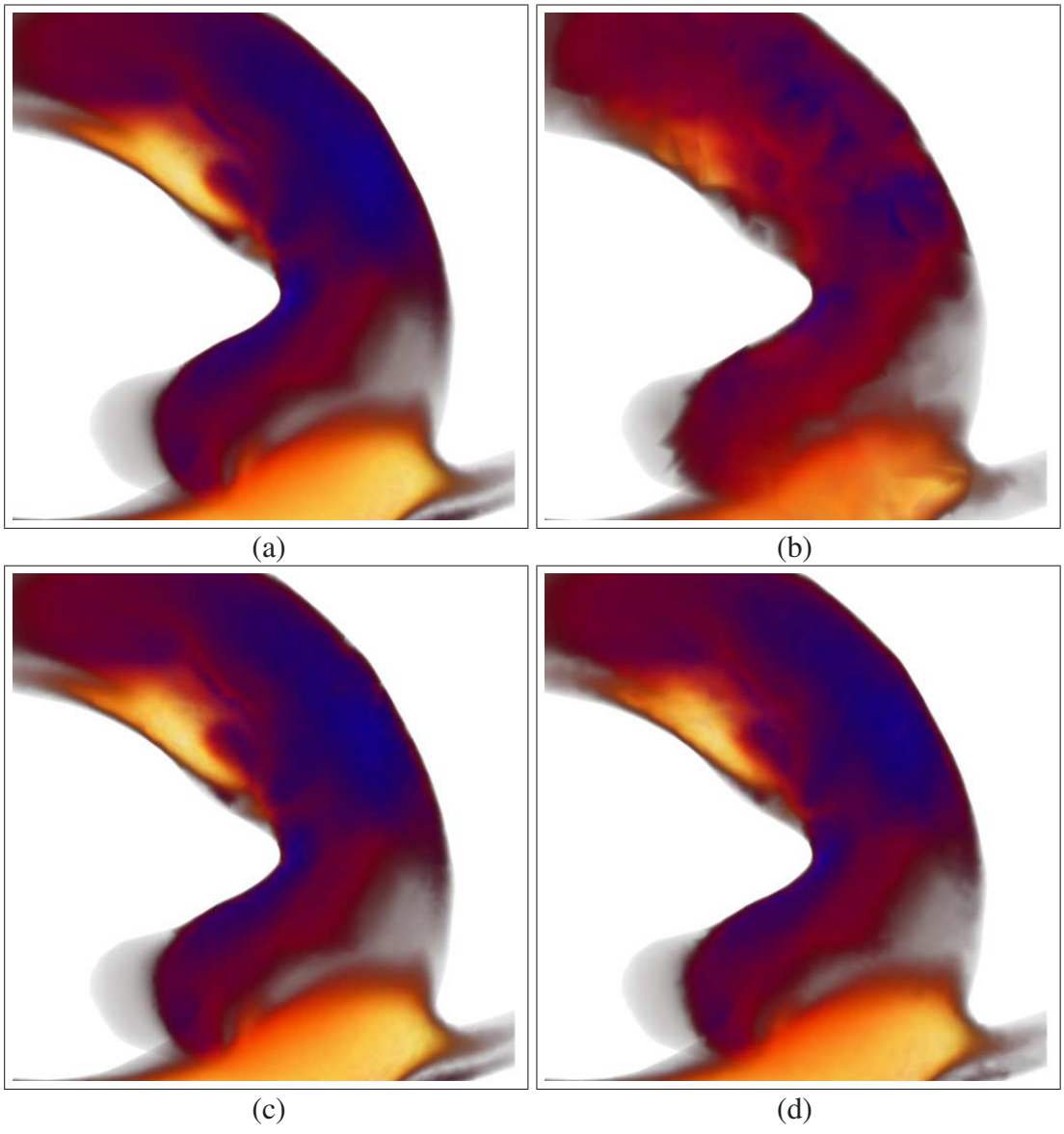



Figure 3.8: Rendered images of the Aorta dataset. a) finest, b) coarsest, c) selectively-refined, d) non-selectively refined

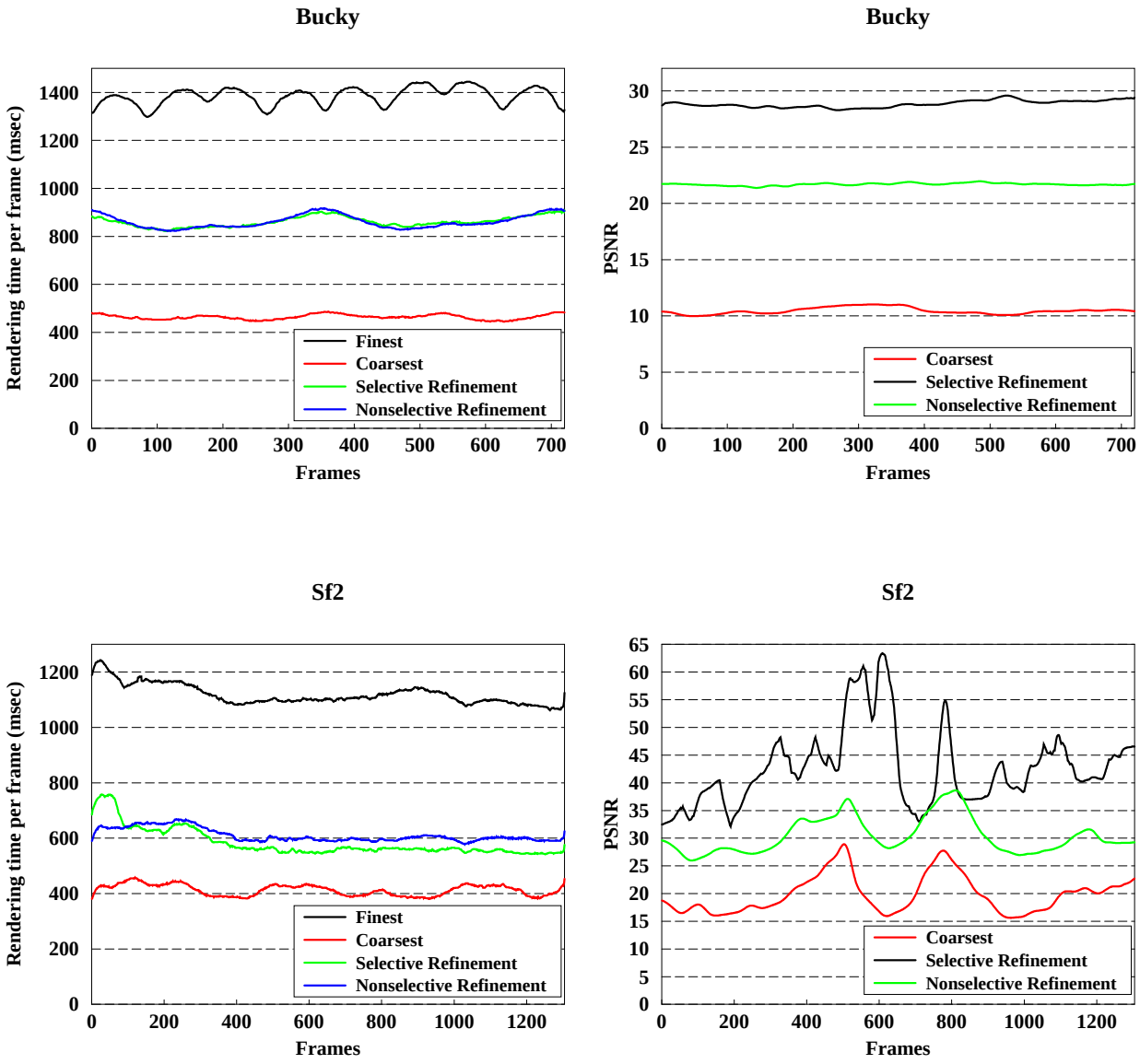



Figure 3.9: Rendering times and PSNR values for Bucky Ball and Sf2 datasets.

# Chapter 4

# Parallelization for GPU and Multi-Core CPUs

In this chapter, the parallelization of well-known cell-projection direct volume rendering algorithm is discussed. Parallel implementations for multi-core CPU systems and GPUs are explained and the results are compared to the serial algorithm running on a the single core of a CPU.

## 4.1 Cell-projection Algorithm

Direct volume rendering is a computationally-intensive process. Early algorithms focused on single-processor environments, then the trend shifted to parallel algorithms for PC clusters. With the recent developments in multi-core CPUs and GPU-based computing techniques, volume rendering algorithms for these platforms have increased in importance because single CPUs are not powerful enough to render even simple volume data into reasonable resolutions in acceptable time frames. We based our work on a well-known direct volume rendering algorithm: the cell-projection algorithm.

The cell-projection algorithm is simple yet efficient; it does not rely on tetrahedra's neighborhood information to order them. The data representation requirement is low and the data access patterns are relatively uniform, making this algorithm suitable for

GPU implementation. In this section, we present the single-core version of the cell-projection algorithm. We describe the data types used by the algorithm and then outline its steps. Then we describe the implementation of the algorithm on multi-core systems using OpenMP.

### 4.1.1 Data Structures

Volume data used in real-world application have grown, with datasets of tens of millions of tetrahedra being quite common. This growth has also increased memory requirements for direct volume rendering (DVR) implementations; to satisfy such requirements with common computer configurations, the data structures used in this work are kept as minimal as possible:

- *Vertex:* Vertex structure contains the coordinate and scalar values associated with the vertex. Float values are used to store these data, making the vertex structure 16 bytes in size.

- *Tetrahedron:* Tetrahedron structure contains indices of four vertices, which comprise the tetrahedron. Integer values are used to store index values, making the tetrahedron 16 bytes in size. The size can be reduced with encoding. Each index value is a decimal value between zero and the number of vertices in the dataset. Thus $\lceil \log_2 (NumberOfVertices) \rceil$ bits are enough to represent an index value. Usually, 24 bits are sufficient to represent a vertex index of a sizable volume dataset, making it possible to reduce the tetrahedron size 25% or more, depending on the number of vertices. Processing an unencoded index value is much faster, however, because index values are extensively used. For that reason, we decided to use the integer type to store index values to avoid such encoding overheads.

- *Intersection record:* Direct volume rendering algorithms throw a ray from each pixel. As the ray travels through the volume, it intersects with several tetrahedra. The effects of such intersections are used to determine the pixels' final color. The intersection record consists of the pixel value from

where the ray has been thrown and the tetrahedron's index that the ray intersected with. Because all the intersection records have to be created before they are consumed, their total size can be very large, and encoding the intersection record is necessary. An intersection record can be represented with $\lceil \log_2 (NumberOfTetrahedra) \rceil + \lceil \log_2 (NumberOfPixels) \rceil$ bits. To store individual records, this size should be rounded up to a multiple of eight bits. A record size of six bytes is sufficient for all the datasets and resolutions tested in this implementation.

- *Per-pixel intersection lists:* The procedure that extracts the intersection records groups them according to their pixel values, which is done via per-pixel intersection lists. This structure is simply a collection of intersection record arrays, one per each pixel value. As these arrays can increase their sizes dynamically, this structure does not introduce too much memory overhead.

- *Intersection effect:* An intersection effect structure is produced after an intersection record is processed. It includes the eye distance to the first point that the thrown ray hits on the tetrahedron. This value is used to sort the intersection effects. The color effect left by the tetrahedron on the intersecting ray is also recorded. As the color is stored as four floats, representing the *rgba* values, the size of this structure is 20 bytes. Because the structure is created and destroyed on a per-pixel basis, its effect on the memory requirement is low.

- *Color map:* A color map is the tabulated form of the transfer function. It is an array of color values. A minimum scalar value is assigned to the first entry and a maximum scalar value is assigned to the last entry. The scalar values associated with the remaining entries are calculated by interpolating the minimum and maximum scalars. Color entries are found by using the transfer function with the associated scalars.

### 4.1.2 Algorithm

The overall flow of the single-core version of the cell-projection algorithm is given in Algorithm 1. First, screen space coordinates of the vertices are calculated and the intersection records are extracted. Then calculating, sorting and compositing intersection effects occurs in a sequence for every pixel. The steps of the algorithm are described below.

```
CellProjectionAlgorithm()
begin
    IntersectionRecord *PerPixelIntersectionLists[Width][Height];
    SSC=ComputeScreenSpaceProjections(Vertices);
    ExtractIntersectionRecords(Tetrahedra, Vertices, SSC, PerPixelIntersectionLists);
    for i = 0 upto Width do
        for j = 0 upto Height do
            list=PerPixelIntersectionLists[i][j];
            IntEffctList=CalculateIntersectionEffects(list);
            SortIntersectionEffects(IntEffctList);
            Color c = ReduceIntersectionEffects(IntEffctList);
            Image[i][j]=c;
end
```

**Algorithm 1:** Cell-projection algorithm.

*Screen space projection of vertices:* In this step, vertices are projected onto the screen according to the view parameters, and their screen space coordinates are calculated. These coordinates are stored in the *SSC* array, which is then used during tetrahedron projections.

*Extracting intersection records:* This step can be considered as the screen space projection of the tetrahedra. The algorithm calculates screen space projections of each tetrahedron. Rays thrown for any pixel under the projection of a tetrahedron intersect with the tetrahedron. Thus, a tetrahedron will contribute to the colors of the pixels under its projection. The cell-projection algorithm extracts any such tetrahedron-pixel pairs (the intersection records). Intersection records are stored in pixel index-based

array, called *PerPixelIntersectionLists*.

Intersection records are found by traversing each tetrahedron; the algorithm calculates its screen space projections by projecting its four vertices. When the projection points of these vertices are connected, a triangle or a quadrilateral will be formed, and with a basic scan-line algorithm, the pixels covered by this projection area can be calculated. The tetrahedron id and the pixel values are then encoded into an *IntersectionRecord* and inserted into the *PerPixelIntersectionLists*.

*Calculating intersection effects:* When a thrown ray travels through a tetrahedron, it loses intensity and its color is affected. The effects of all the tetrahedra that a ray has travelled through can be combined to obtain the final effect on the ray. This procedure (Algorithm 2) uses the intersection record list for the current pixel and calculates the intersection effects for each record in the list individually.

Algorithm 2 calculates the intensity and color of a tetrahedron intersection on the ray. The first step calculates two intersection points ($ip_0$ and $ip_1$) of the ray and the input tetrahedron. Let $ip_0$ be the closer point to the eye. The distance between the eye point and $ip_0$ is recorded in the intersection effect record; this value will be used in the sorting phase. The ray's path, which is the line segment between $ip_0$ and $ip_1$, is divided into a pre-defined number (*NumOfSamples*) of line segments. At the starting point of these line segments, the scalar value is calculated with interpolations using the vertices of the tetrahedron. The color of this point is determined by the *ColorMap* table. As the ray travels from $ip_0$ to $ip_1$, the color and intensity contributions can be approximated. We describe the interpolation process by the following example:

Let $ip_0$ be $(10, 10)$ and $ip_1$ be $(22, 19)$. Dividing the path into three segments, the points we are interested in are $ip_{01}$ =$(14, 13)$ and $ip_{02}$ =$(18, 16)$. The algorithm approximates the path that the ray travels with three line segments of length five: [$ip_0$, $ip_{01}$], [$ip_{01}$, $ip_{02}$] and [$ip_{02}$, $ip_1$]. The ray starts traveling with full intensity and each segment is assumed to have a uniform color, which is the color at the beginning of the segment. The color contributions of the line segments to the pixel are proportional to the color attributes of the traveled region, the travel distance, the opacity coefficient of the region and the intensity of the ray itself. The ray loses most of its energy while traveling through non-transparent regions; thus later regions have a lesser effect on the

final color. After the ray travels through all regions, its final color is recorded.

```
CalculateIntersectionEffects(Tetrahedron t, Pixel p)
begin
    IntersectionEffect record;
    Ray *R=new Ray(p);
    [ip0, ip1]=RayTetrahedronIntersection(R,t);
    record.dist = |ip0−EYE|;
    d=|ip1−ip0| / NumOfSamples;
    Color c=[0, 0, 0, 0];
    record.color=[0, 0, 0, 0];
    for i = 0 to NumOfSamples do
        Point ip=ip0+d × i ;
        s=InterpolateScalar(t,ip);
        c=getColorFromScalar(s);
        for j = 0 to 4 do
            record.color[j]+=DistConst
            ×d × (1 − record.color[3]) × record.color[j];
    return record;
end
```

**Algorithm 2:** Calculating the effect of the intersection between a ray and a tetrahedron on the ray.

Accurately calculating the scalar value of a point on a tetrahedron is important, as poor interpolations may cause significant artifacts. The interpolation process is given in Algorithm 3. It starts by selecting a reference vertex, which can be any one of a tetrahedron's vertices. Then the $M$ matrix, which contains the positions of the other three vertices of the tetrahedron relative to the reference vertex, is calculated. The $N$ vector stores the relative position of the input point to the reference vertex. The scalar vector (*Scalar*) contains scalar value differences of the vertices relative to the scalar values of the reference vertices. Then the equation $M \times R = Scalar$ is solved to obtain the $R$ vector, which represents a coefficient vector that will give the relative scalar value of a point when multiplied with the relative position vector of that point. The `SolveEquation` function calculates this coefficient, dot product of the relative position vector of point ($N$) and the coefficient vector ($R$). By adding the scalar value of the reference vertex, the interpolated scalar is found.

*Sorting intersection effects:* This step is achieved using the *dist* parameter. The number of elements to sort is usually in the low hundreds. Although many sorting algorithms

```
InterpolateScalar(Tetrahedron t, point p);
begin
    float M[3][3];
    float Scalar[3];
    float N[3],R[3];
    float s;
    for i = 0 upto 3 do
        M[i]=t.V[i+1].coords - t.V[0].coords;
    N=p-t.V[0].coords;
    for i = 0 upto 3 do
        Scalar[i]=t.V[i+1].scalar - t.V[0].scalar;
    SolveEquation(M, Scalar, R);
    /* Solves the equation M × R = Scalar for R. */
    s=R · N+ t.V[0].scalar;
    return s;
end
```

**Algorithm 3:** Interpolation of scalar value within a tetrahedron.

can be used for this job, the size of the list favors some of them; we found that quicksort performs well for the size range of the intersection lists.

*Compositing intersection effects:* Sorting the intersection effects by the distance of the first intersection point to the eye orders the tetrahedra by the ray's visit order. Algorithm 4 describes the composition of the contributions along the ray. If the opacity of accumulated color exceeds a pre-defined threshold, the ray terminates because the remaining tetrahedra will have no significant effects (early ray termination).

```
ComposeIntersectionEffects(IntersectionEffect *list);
begin
    Color c=[0, 0, 0, 0];
    for i = 0 upto list.length do
        Color r=list[i].color;
        for j = 0 to 3 do
            c.color[j]+=(1 -c.color[3]) × r.color[j];
        if c.color[3]>=0.9999 then
            break;
    return c;
end
```

**Algorithm 4:** Composition of intersection effects.

### 4.1.3 Multi-Core Implementation with OpenMP

The cell-projection algorithm is highly suitable for parallelization, because, for the most part, memory accesses are structured, and race conditions are thus avoided. OpenMP provides a useful interface for parallelization on multi-core processors with a shared memory architecture. It divides the workload among threads and then executes them through different cores. Since the cores use the shared memory, there are no data transfer issues as there are with parallel clusters.

OpenMP also supports parallelization of for-loops by distributing the iterations among threads. This approach works unless one iteration requires data produced by another iteration or there are race conditions among iterations. Screen space projection of vertices can be parallelized trivially, since no race conditions exist. The for-loops that process intersection records for each pixel can also be parallelized, but as the iterations in these loops use some temporary data, that data should be replicated for each thread to avoid confusion.

Extracting intersection records necessitates travelling through the tetrahedra and inserting intersection records to the per-pixel intersection lists under a tetrahedron's projection. Since different tetrahedra can have projections on the same pixel, a race condition on that pixel's intersection list is possible. To avoid this scenario, the *Per-Pixel Intersection Lists* structure should be replicated. After the extractions are complete, the lists of each thread can be combined. While this approach is amenable to OpenMP parallelization we observed that it does not improve computation time significantly; thus, we used the serial version.

## 4.2 Cell-projection Algorithm on GPU

Although the CPU-based cell-projection algorithm's performance is reasonable for small datasets, it is not sufficient for large datasets. However, this algorithm is well suited to a GPU implementation, as it focuses on one group of data at a time and its memory accesses are structured. Further, as the algorithm is well structured in terms

of execution flow, it can be efficiently parallelized. In this section, we detail CUDA implementation and memory management issues and discuss progressive rendering.

### 4.2.1 CUDA Implementation

Although the cell-projection algorithm is highly suitable for parallel implementation, to increase efficiency, hardware restrictions and capabilities must be considered. Graphics cards contain many processing units, but as each computation unit is much slower than a CPU core, their computation power depends on a high level of parallelism. Memory is another important restriction. Current GPUs usually have fewer than 2GBs of memory, with 1GB of memory more common. Since volume data can be very large, such memory restrictions can easily result in a bottleneck. The algorithm should consider such restrictions and be able to work with limited memory.

We present our GPU implementation of the cell-projection algorithm using CUDA in Algorithm 5. The serial version processes each pixel individually, but this approach is not suitable for GPUs. First of all, each pixel's workload is too low to be efficiently parallelized; the number of CUDA threads should be at least in the tens of thousands. The number of intersection records per pixel would be much less. Assigning one pixel's process to different threads, similar to multi-core implementation, is also not suitable, as the execution flow and memory access patterns are unorganized. Further, each pixel's workload differs drastically, which would cause significant workload imbalance among threads.

For the GPU, we grouped the pixels, calling each group a hash block, and processed one group at each iteration. The *NumOfRenderIterations* variable represents the number of hash blocks, which depends on the dataset size. If the value of *NumOfRenderIterations* is low, the workload per render iteration increases, which allows more efficient parallelization but increases the memory requirement. The amount of available memory thus limits the number of iterations. In our implementation, we used 16 or 64 rendering iterations. We describe the steps of the GPU-based algorithm below.

*CPU to GPU data transfer:* The tetrahedra and vertices data are copied just once

```
GPUCellProjectionAlgorithm();
begin
    CopyToGPU(Vertices);
    CopyToGPU(Tetrahedra);
    CopyToGPU(ColorMap);
    CopyToGPU(ViewParameters);
    GPU.Run(ComputeScreenSpaceProjections(Vertices));
    SSC=CopyFromGPU(ScreenSpaceCoordinates);
    IntersectionRecord *PerHashBlockIntersectionLists[NumOfRenderIterations];
    ExtractIntersectionRecords(Tetrahedra, Vertices, SSC,
    PerHashBlockIntersectionLists);
    for i=0 upto NumOfRenderIterations do
        CopyToGPU(PerHashBlockIntersectionLists[i]);
        GPU.Run(processHashBlock);
        GPU.Run(sortHashBlock);
        GPU.Run(composeHashBlock);
    Display(OutputImage);
end
```

**Algorithm 5:** GPU-based cell-projection algorithm.

after CUDA initialization is completed. The view parameters are copied at the beginning of each rendering; i.e., whenever the volume is redrawn. The color map is copied at the beginning, but can be updated if the transfer function changes. We assume the graphics card contains enough memory to hold these data, leaving some space. The data transfer between the main memory and the GPU memory is very fast and has little effect on rendering times.

*Screen space projection of vertices:* This function runs on the GPU using 512 blocks with 256 threads each. Usually using the same computation, each thread calculates the screen space coordinates of one vertex; the execution flow differs only if exceptions are observed, which is not frequent. Vertices are assigned to threads sequentially; thus memory access patterns are uniform and the shared memory is highly utilized. As a result, this function is very efficiently parallelized.

*Extracting intersection records:* This step can be considered the screen space projection of the tetrahedra and is executed on the CPU. After the screen space coordinates of the vertices are calculated, they are copied back to the CPU. This operation takes little time, since the data size is small and the data transfer rates are high. This algorithm

works similarly to its serial version; the only difference is that instead of the intersection records being grouped according to their pixel coordinates, they are grouped according to their hash blocks. A pixel's hash block is simply computed as follows: The last two or three bits of a pixel's $x$ and $y$ coordinates are concatenated. With two bits, 16, and with three bits 64, hash blocks are addressed. The most important property of the hash function, which helps balance the workload of each hash block, is to ensure that the pixels in each block represent a sub-sampling of the whole image rather than being grouped in certain parts of the image.

We implemented this function on CPU because of irregularity and memory concerns. Each tetrahedron has a different projection area, and the execution flow of the scan-line algorithm differs for each tetrahedron. The number of pixels in each tetrahedron's projection area differs significantly, unevenly distributing the workload. Most importantly, race conditions will be observed. Many tetrahedra will have projections on the same hash blocks, thus they will try to write into identical locations. The only solution for such race conditions is to extract the intersection records first and organize them into hash blocks later. Although such an approach works, it introduces significant overhead.

*Calculating intersection effects:* Apart from GPU-specific optimizations, this algorithm is similar to its serial version. It starts by copying the intersection records for the current hash block to the GPU. Each thread is assigned an intersection record and responsible for producing the corresponding intersection effect data. Execution flows are mostly uniform. Continuous threads are assigned to process contiguous intersection records, thus, memory accesses are also uniform for some parts. However, the tetrahedra contain references to vertices, which are not contiguous, and accesses to the vertices cannot be made uniform. However, for repeated access to non-uniform data, the algorithm uses shared memory to store the data temporarily, which makes the subsequent memory accesses uniform and much faster. This function is launched with a grid of 1024 blocks; because of the size difference of the shared memory between devices, those with a computing capability of 2.0 or higher are launched with 192 threads per block and others use 64 threads per block.

*Sorting intersection effects:* The sort operation differs from its serial counterpart

significantly. In the serial version, the intersection effects for each pixel must be sorted, and thus, many small sorting jobs are done independently, which enables efficient use of sorting algorithms like *quicksort*. However, in the CUDA version, intersection effects for every pixel in a hash block are mixed; the sorting function must group an individual pixel's intersection effects and then sort these groups. Further, in the CUDA version, the sort lists are much larger. In this work, we used efficient radix sort implementation by Satish et al. [52] from the Thrust library [53]. This library includes optimized implementations of various functions for GPUs and, integrated with CUDA, it can use device memory allocated from there. Radix sort has *O(n)* time complexity, which allows us to use larger hash blocks (thus fewer rendering iterations) without negatively affecting sorting times. It can also be efficiently parallelized.

Radix sorting is not a comparison-based sorting technique. It uses standard data types, given as input arrays, as the key and takes another array of standard data types as data. Using the keys, it sorts the keys and the data. Since radix sort cannot use custom structures, we divided the data inside the intersection effect structure into different arrays: *pix*, *dist* and *clr*. The *pix* array (pixel) does not exist in the serial version, but because sorting is not done on a per-pixel basis in this version, this distinction is needed. The *pix* array elements are computed from the pixel coordinates, with the current rendering iteration revealing the last two or three bits of each pixel's $x$ and $y$ coordinates. The rest of the bits are packed and each *pix* value is computed. For example, let the resolution be $1024 \times 1024$ and the number of iterations be 16. Then, each pixel's $x$ and $y$ coordinates can be represented with eight bits, making each *pix* value 16 bits in size.

Our sorting implementation must group the pixels' intersection effect data, then sort the groups according to distance. Since radix sort is stable, sorting first according to distance and then according to the *pix* value achieves this.

Algorithm 6 shows the sort algorithm. *index* is an empty buffer, and the algorithm begins by filling that buffer with the integer sequence 0, 1, 2 ..., representing the data index. The data is sorted using *dist* as the key and *index* as the data. Then, the *pix* array must be reordered according to the new indices from the *index* array. The *gather* function from the *Thrust* library performs these reorderings.

```
SortHashBlock(uint *pix, float *dist, uint *index, color *clr);
begin
    thrust.sequence(index);
    thrust.sort(dist, index);
    thrust.gather(index, pix);
    thrust.sort(pix, index);
    thrust.gather(index, clr);
end
```

**Algorithm 6:** GPU-based sort algorithm for hash blocks.

The initial sorting step sorts the data according to eye distances. The second step, using *pix* as the key and *index* as the data, groups the data. With the final gather, the *pix* and *clr* arrays are sorted by eye distance and grouped by pixel values.

*Compositing intersection effects:* The *ComposeHashBlock* function uses the sorted *clr* and *pix* arrays as input. It combines the color values in sorted order for each pixel and computes the final pixel color. The composition process is described in Algorithms 7 and 8.

```
ComposeHashBlock(int iteration, uint *pix, color *clr);
begin
    color currFB[ Width×Height / NumOfRenderIterations ];
    Reset(currFB);
    while length>CPUSwitchThreshold do
        GPU.Run(Reduce(pix,clr,currFB);
    hostPix=CopyToCPU(pix);
    hostClr=CopyToCPU(clr);
    hostCurrFB=CopyToCPU(currFB);
    for i=1 upto hostPix.length do
        composeRecord(hostCurrFB[hostPix[i]], hostClr[i],
        hostCurrFB[hostPix[i]]);
    currFB=CopyToGPU(hostCurrFB);
    GPU.Run(updateFrameBuffer(currFB, iteration));
end
```

**Algorithm 7:** Composition algorithm for hash blocks.

Algorithm 7 uses a temporary frame buffer, *currFB*. The size of this buffer is equal to the number of pixels processed at each iteration. Algorithm 7 runs on the CPU. It repeatedly calls the *Reduce* function (Algorithm 8), which runs on the GPU and

```
Reduce(uint *pix, color *clr, color *currFB);
begin
    tid=block.id× block.size+thread.id;
    index=tid;
    while index < pix.length do
        if pix[2 × index] == pix[2 × index + 1] then
            composeRecord(clr[pix[2 × index]], clr[pix[2 × index + 1]],
            clr[pix[2 × index]]);
        else
            composeRecord(currFB[pix[2 × index + 1]], clr[pix[2 × index + 1]],
            currFB[pix[2 × index + 1]]);
        index+=grid.size× block.size;
end
```

**Algorithm 8:** Reduction algorithm for hash blocks.

reduces the size of the array by half. This function relies on the input arrays being sorted, as each thread processes a consecutive pair of entries in the input arrays. First, the *pix* values of these pairs are compared. If the values are equal, then, because the arrays have been sorted, the two color values can be combined and represented as a single color value. If the *pix* values are not equal, the later record is output and the earlier record is retained. A pair's *pix* values can be different only if the later record is the first record of the pixel that has not yet been output.

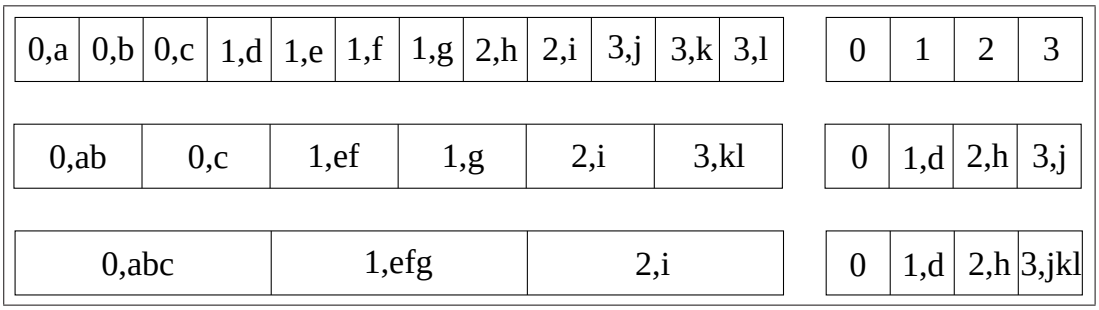



Figure 4.1: Reduction example.

Figure 4.1 illustrates the algorithm with an example. The process starts with 12 *pix*, *clr* entries. The right-hand side shows the temporary frame buffer, which is initially empty. After the first reduction, the data shrinks to six entries. (1, *d*), (2, *h*) and (3, *j*) entries are output to the frame buffer while (0, *c*), (1, *g*) and (2, *i*) entries are retained. Entry couples with the same *pix* values are combined to obtain (0, *ab*), (1, *ef*), and (3, *kl*). Further reductions are executed in the same way. To reduce two

entries to one, the algorithm uses the *composeRecord* function, which takes three colors as input. It combines its first two inputs and writes the result to its third input. This function works similarly to the serial version.

The result of the *Reduce* function is an interleaved array. Further reductions can be performed on the interleaved arrays, or the array can be compacted. We use an optimized compaction mechanism. After the first reduction, we compact the interleaved array into its first half using the gather functionality of the *Thrust* library. This operation frees the other half for temporary storage. Further reduction operations write their results to that free space. This approach eliminates the need to compact reduction operations beyond the first one and allows parallelization of memory accesses for greater efficiency.

In the *Reduce* function, each thread is responsible for combining two entries into one. With further reductions, the arrays shrink significantly; thus after a certain point, the threads cannot be utilized efficiently. We used two mechanisms to overcome this problem. (i) We run the *Reduce* function with size-dependent grid and block sizes. With large sizes, we use 512 blocks per grid and 256 threads per block. With smaller sizes, we stepwise reduce the size to 64 blocks per grid and 32 threads per block. (ii) After a certain length, we complete the reductions in the CPU. Because the data is now so small, transfers between the CPU and GPU take little time and the CPU executes the reduction operations more efficiently because parallelism is limited.

After the reductions are complete, all intersection effects are combined into the temporary frame buffer. With the *updateFrameBuffer* function, which runs on the GPU, the data in this buffer is transferred into the actual frame buffer. The naive version completes the image after all render iterations have been completed; however, this function also supports progressive rendering, which is very useful for increasing interactivity.

### 4.2.2 Progressive Rendering

Volume rendering is time consuming, particularly for high resolutions, and this adversely affects the interactivity. Progressive rendering aims to perform a low-resolution

volume rendering and progressively improve image quality while displaying the outputs to the user. As low-resolution rendering is faster, a low-quality image is displayed fairly quickly. As the process continues, the image progressively improves, while the user is observing the latest output.

The hash-block approach provides a natural framework for progressive rendering. As mentioned earlier, each hash block will produce the overall sub-sampling of the whole image. For example, if we use 16 hash blocks to render a $1024 \times 1024$ image, then each hash block will produce a $256 \times 256$ image, which is a low resolution version of the whole image. Each of those low resolution images are slightly shifted from each other, so that when combined they will constitute the high resolution image. Progressive rendering simply requires processing the hash blocks in a specific order, so that the processed hash blocks can be combined to obtain higher and higher resolutions progressively.

The pixel values are assigned to each hash block according to their last two or three bits. This approach divides the whole image into $4 \times 4$ or $8 \times 8$ sub-windows respectively. Each pixel in a sub-window is processed by a different hash block. Without progressive rendering, only the values of pixels assigned to the currently processed hash block are updated in the frame buffer. On the other hand progressive rendering requires some of the neighboring pixels being updated as well. For example, after the first hash block has been processed, every pixels values in a sub-window is updated with the computed pixel's value. Accordingly, the lower resolution version can be displayed without waiting the rendering to finish.

Figure 4.2 illustrates the progressive rendering process. In the example, we have 16 render iterations and a $4 \times 4$ sub-window. Right bottom corner of each cell displays the index value of the pixel within the sub-window. The larger value represents the index of the pixel whose value is currently set to the current pixel. The blue background indicates the currently processed pixel, and the red background indicates the pixels whose values are updated with current pixels values.

In the first iteration, the $0^{th}$ pixel is processed and the results are written to every pixels. In the second iteration, the $10^{th}$ pixel is processed and the results are written to pixels on the top half. At each iteration another pixel is processed and the results are

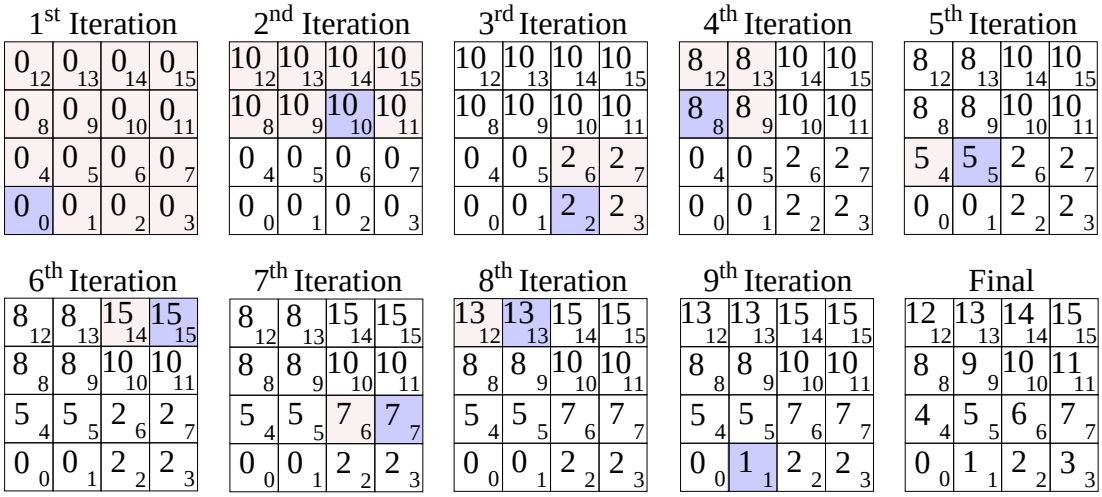



Figure 4.2: Progressive rendering.

written to some neighboring pixels, until every pixel value is computed.

The processing order of pixels is important for progressive rendering. With the given ordering, we can obtain various sub-resolutions after certain rendering iterations. For example, we can output, $1 \times 1$, $2 \times 1$, $2 \times 2$, $4 \times 2$ and $4 \times 4$ sub-resolutions of our $4 \times 4$ sub-windows after $1^{st}$,$2^{nd}$,$4^{th}$,$8^{th}$ and $16^{th}$ iterations, respectively. As a result, the low resolution versions can be quickly obtained and displayed, while the resolution progressively improves. This technique have very little overhead but improves the interactivity greatly.

### 4.2.3 Memory Management

Memory is usually the limiting factor for the size of volume data that a renderer can visualize. We have employed several methods to keep both GPU and system memory requirement low. Our motivation was to keep memory footprint as low as possible without adversely affecting the performance or accuracy.

Although system memory is large, it can become a limiting factor for high-image resolutions, especially, because the cell-projection algorithm extracts the intersection records and stores them in the memory before processing. To work with the available memory, our implementation can render images in multiple passes. For example, an

image with $4096 \times 4096$ resolution can be rendered in four passes of $2048 \times 2048$ or 16 passes of $1024 \times 1024$. This approach introduces some overhead; however, it guarantees that the application will fit in the physical memory with no swapping, and thus improves performance.

GPU memory requirement is more crucial, since it is smaller. Vertex and tetrahedron data are the main components of the volume data and should be placed in the GPU memory. Vertex structure is minimal. Tetrahedron structure can be compacted from 16 bytes to 12 bytes by encoding. However, since this would introduce noticeable performance overhead, due to decoding operations and non-uniform memory access patterns, our implementation use 16 bytes to represent the tetrahedron structure. Accordingly, $16 \times (NumberOfVertices + NumberOfTetrahedra)$ bytes are required to store these data. For the largest dataset in our test set, sf1 with 14 millions of tetrahedra, tetrahedra and vertices require about 250 Mb's of GPU memory.

The maximum possible number of intersection records in a hash block is directly proportional to the memory requirements during the hash block processing. This value depends on the dataset and view parameters greatly. On the other hand, this value can be easily reduced by using higher number of hash blocks or using multi-pass rendering. Accordingly we can change the memory requirement according to the available memory. In our implementation 34 bytes per intersection record is needed. This memory is reused as much as possible during the processing of hash blocks. When the sf1 dataset is rendered using 64 hash blocks with a $1024 \times 1024$ resolution in a single pass, 120 Mb's of memory is needed for hash block processing. Together with tetrahedra, vertices and other smaller data structures, the memory requirement falls well below 500 MB's. Accordingly, a graphics card with 2GB's of memory would be sufficient to render a volume of a hundred millions of tetrahedra with similar characteristics to sf1 dataset using 64 hash blocks with a $2048 \times 2048$ resolution in a four passes.

## 4.3 Results

We used a PC with a four-core AMD CPU with SMP architecture running at 3.2 GHz, 4GB of system memory and an nVidia GTX 560 graphics card. We tested our implementations on five different datasets (see Figure 4.3). We rendered each dataset with three different view parameters and for three different resolutions: $512 \times 512$, $1024 \times 1024$ and $2048 \times 2048$. We report the average results for each resolution.

| **Dataset** | **Data Size** | **Resolution** | **Serial** | **Multi-Core** | | **GPU** | |
|---|---|---|---|---|---|---|---|
| | | | Time | Time | Speed-up | Time | Speed-up |
| **Comb** | 215.0 | $512 \times 512$ | 10.45 | 2.96 | 3.531 | 0.53 | 19.863 |
| | | $1024 \times 1024$ | 41.64 | 11.77 | 3.537 | 1.61 | 25.877 |
| | | $2048 \times 2048$ | 168.19 | 48.72 | 3.452 | 5.97 | 28.179 |
| **Bucky** | 1250.2 | $512 \times 512$ | 71.76 | 20.07 | 3.576 | 3.16 | 22.735 |
| | | $1024 \times 1024$ | 285.94 | 82.45 | 3.468 | 10.33 | 27.671 |
| | | $2048 \times 2048$ | 1334.23 | 523.23 | 2.550 | 47.57 | 28.049 |
| **Aorta** | 1386.9 | $512 \times 512$ | 31.70 | 9.45 | 3.356 | 1.27 | 25.055 |
| | | $1024 \times 1024$ | 125.06 | 37.21 | 3.361 | 4.09 | 30.547 |
| | | $2048 \times 2048$ | 554.41 | 196.12 | 2.827 | 16.99 | 32.633 |
| **Sf2** | 2067.7 | $512 \times 512$ | 40.34 | 11.80 | 3.418 | 2.44 | 16.516 |
| | | $1024 \times 1024$ | 159.48 | 47.08 | 3.387 | 6.77 | 23.554 |
| | | $2048 \times 2048$ | 637.79 | 193.30 | 3.300 | 28.76 | 22.180 |
| **Sf1** | 13980.1 | $512 \times 512$ | 82.77 | 26.41 | 3.135 | 6.63 | 12.484 |
| | | $1024 \times 1024$ | 318.24 | 97.04 | 3.280 | 18.19 | 17.492 |
| | | $2048 \times 2048$ | 1615.24 | 732.10 | 2.206 | 100.01 | 16.152 |

Table 4.1: Rendering times and speed-ups of GPU, multi-core and serial cell-projection algorithms. Data size is given in thousands of tetrahedra. Rendering times are in seconds.

Table 4.1 show that significant speed-ups are obtained with the multi-core and GPU implementations. The multi core implementation achieves a 3.2-fold increase in speed

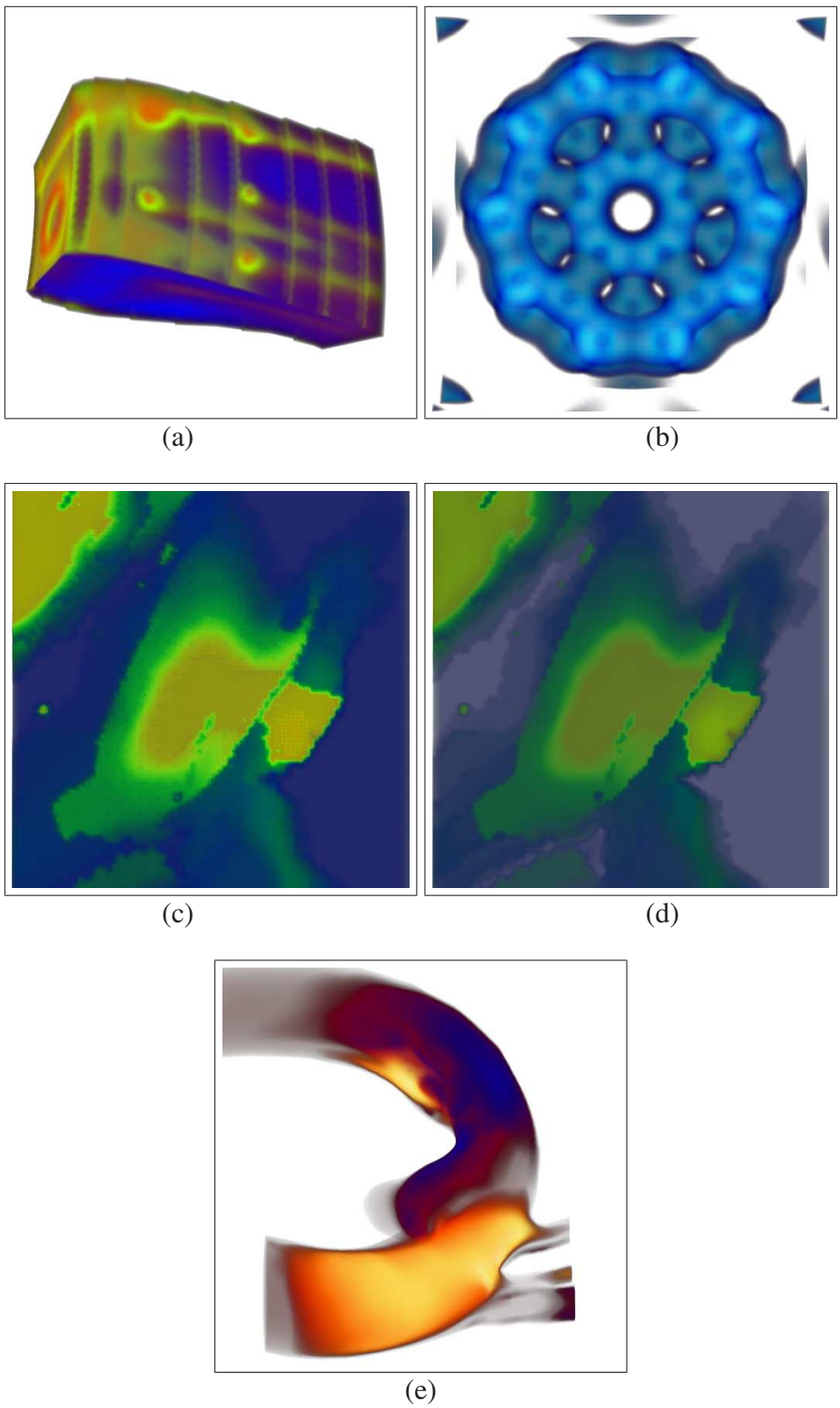
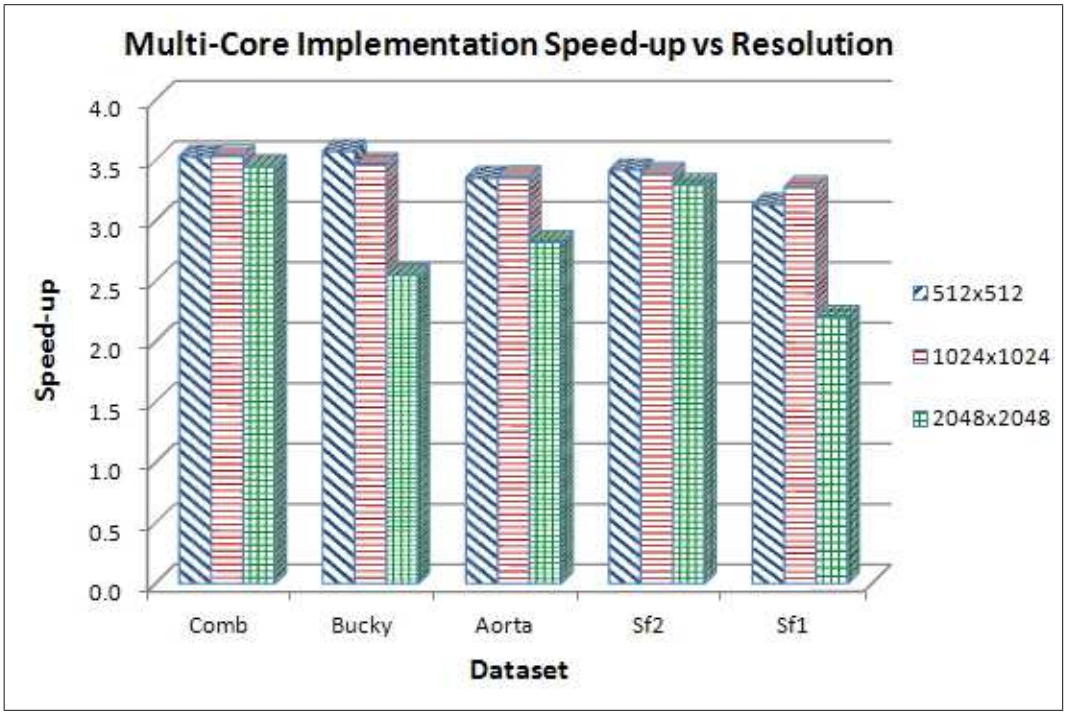



Figure 4.3: Rendered images of various datasets: (a) Comb dataset, (b) Bucky dataset, (c) Sf2 dataset and (d) Sf1 dataset, (e) Aorta dataset.

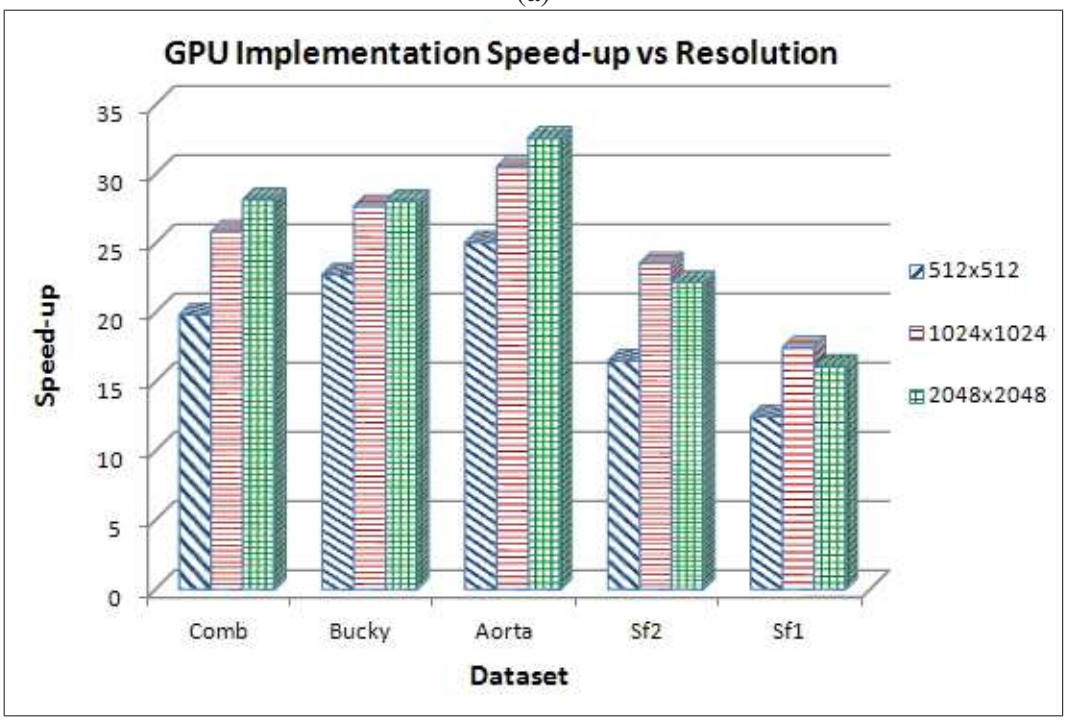
on average. Considering the rendering has a significant input-output (IO) component, these speed-ups are very promising. The GPU implementation achieves 23.3-fold increase in speed on average, with some speed-ups reaching above 32-fold. Considering we used a middle-segment graphics card in the tests, these speed-ups are also very promising.

Figure 4.4 (a) shows the speed-ups obtained for various resolutions of different datasets using the multi-core implementation. This implementation achieves almost identical speed-ups for $512 \times 512$ and $1024 \times 1024$ resolutions. For the $2048 \times 2048$ resolution, speed-ups are slightly slower because of memory limitations. Our multi-pass rendering approach solves high-memory requirement problem, but introduces some overhead, which reduces the speed. For resolutions above $2048 \times 2048$, we expect speed-ups to be higher because serial implementation will also incur multi-pass rendering overheads.

The GPU implementation produces higher speed-ups for the $1024 \times 1024$ and $2048 \times 2048$ resolutions (cf. Figure 4.4 (b)) because, larger jobs use the GPU's tens of thousands threads more efficiently. The $512 \times 512$ resolution does not utilize the graphics hardware as much, resulting in lower speedups. The multipass rendering overhead affected the speed-up of $2048 \times 2048$ resolution, but particularly for smaller datasets, the associated overhead was balanced by the higher utilization. For resolutions above $2048 \times 2048$, we expect that speed-ups would remain high.




(a)




(b)

Figure 4.4: Speed-ups for various resolutions of different datasets: (a) multi-core implementation and (b) GPU implementation.

# Chapter 5

# *MaterialVis*: Material Visualization Based on Direct Volume and Surface Rendering Techniques

In this chapter first we outline the general framework of *MaterialVis*, followed by two sections on the preprocessing and rendering steps. In these sections the main algorithms are presented in the form of pseudo-codes, leaving technical details to the accompanying Appendices. Then, some of the capabilities of the tool are demonstrated using an embedded quantum dot data set. Even though our primary emphasis in *MaterialVis* is on functionality, but not the speed, nevertheless we provide performance benchmarks for a wide range of datasets.

## 5.1 General Framework

Figure 5.1 illustrates the framework of *MaterialVis* which has two main stages: *preprocessing* and *rendering*. The preprocessing stage takes the raw input and constructs the volumetric representation. For (poly)crystalline structures the preprocessing step further continues and assigns error values to atoms representing crystal defects. The rendering stage visualizes the constructed volume representation. The input reader

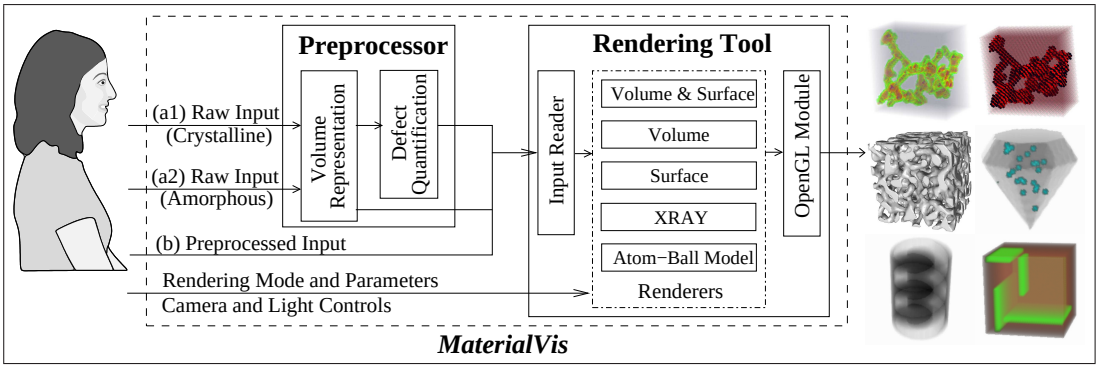



Figure 5.1: The overall framework of *MaterialVis*

module reads the volumetric representation and initializes the renderers. At any time, one of five renderers is selected by the user and the visualization is performed. These renderers use the OpenGL-based drawing module to display the volumetric data. The rendering tool is an interactive tool. The user interactively provides various inputs to renderers, such as camera and light information and several renderer-specific parameters.

## 5.2 Preprocessing

*MaterialVis* operates on a very simple input format. For amorphous materials, the types and atomic coordinates of each atom in the material is sufficient. However, for crystalline structures, the tool also requires primitive and basis vector information of the crystal structure. If this information is not readily available, our earlier work, *BilKristal* [4, 5], could be utilized to extract the unit cell information from the crystal structure.

*MaterialVis* construct a volumetric representation using the coordinates of a set of points representing atoms in the material. There are two types of volumetric representations: *regular* and *unstructured* grids. Regular grid representation is widely used in medical imaging fields where the input data is fixed in resolution. For material visualization, interest points are the atoms; crystalline defects are attributed to them and they constitute the surface structure. Because the regular grid representation is defined independent to atoms, a fairly high grid resolution must be used in order to capture

crystal defects and surface structures in high detail. On the other hand, unstructured grid representation uses atoms as vertices. Accordingly, despite using the connectivity information, the unstructured grid representation is more compact and suited better for material visualization. Because the tetrahedra are the simplest 3D primitives, we perform tetrahedralization to convert atomic coordinates into an unstructured volumetric representation.

After tetrahedralization, we extract the surface polygons of the created volume. The surface polygons are required by the surface rendering modes. *MaterialVis* focuses on visualizing crystal defects; thus, for the crystal structures the defects must be quantified for each atom in the crystal. The preprocessing stage performs these tasks and produces a data file storing the volumetric representation of the material. For crystal structures, quantified crystal defects are also included. In our experiments, we observed that the datasets with sizes up to half a million atoms could be preprocessed in less than twenty minutes. The preprocessing stage data flow is summarized in Figure 5.2.

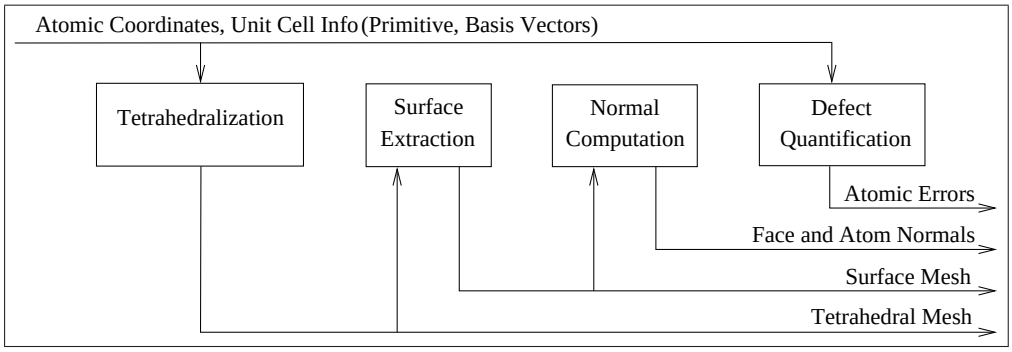



Figure 5.2: The preprocessing stage data flow

### 5.2.1 Construction of the Volumetric Representation

The construction of the volume representation starts with tetrahedralization of atoms. Each atom is represented as a point in 3D space. Tetrahedra cannot overlap with other tetrahedra and all parts of the volume must be covered by exactly one tetrahedra. The generated tetrahedra must be as close to a regular tetrahedron as possible (all sides are

equilateral triangles) because volumes containing many sliver tetrahedra do not represent the volume accurately and may cause rendering artifacts. Delaunay tetrahedralization is the approach that generates such tetrahedra and it is the default tetrahedralization scheme in *MaterialVis* because it produces superior results. We adapt *Bowyer-Watson Delaunay triangulation* [54, 55] to generate Delaunay tetrahedra. Because Delaunay tetrahedralization is not scalable for data sets containing millions of points, we devised a *pattern-based tetrahedralization* algorithm.

Our pattern-based tetrahedralization algorithm is based on the fact that the crystal structures have regular repeating patterns. The algorithm tetrahedralizes a unit cell of the crystal and searches for the occurrence of this pattern in the actual dataset containing atoms. Hence, it cannot handle arbitrarily unstructured point sets or highly deformed crystals. It does not work on amorphous materials. It can tolerate small deformations, some interstitial impurity atoms and some vacancies. It can handle cavities in the crystal structures, as long as the crystal remains as a single piece. The volumetric representation constructed by the pattern-based tetrahedralization is not as good as the one obtained by the Delaunay tetrahedralization, thus may produce inferior rendering results; but the pattern-based tetrahedralization is much faster for larger input sizes. *MaterialVis* only switches to pattern-based tetrahedralization for very large input datasets, which otherwise would take hours to pre-process. For the details of Delaunay tetrahedralization and pattern-based tetrahedralization, please refer to Appendix A.

After the tetrahedralization, the preprocessing stage continues with surface extraction. The surface extraction process simply extracts faces of tetrahedra which are not shared by another tetrahedra. For each face, the normal values are calculated. The face normals are used in flat shading. For smooth shading, the vertex normals should be computed by averaging the normals of the faces sharing the vertex.

### 5.2.2 Quantifying Crystal Defects

We classify crystal defects into three groups. The first group of defects is the positional defects, which are caused by the deviation of atoms from their perfect positions relative to their neighbors. The graphite crystal with slightly shifted layers is an example.

Atoms in these shifted layers have positional defects. The second group of defects is caused by vacant positions in crystals where some atoms should exist. The third group of defects is caused by extra (interstitial impurity) atoms where some foreign atoms could be found at off-lattice sites. The majority of crystal defects can be represented as one of these or a combination of them.

*MaterialVis* calculates defect values of atoms for each type of defect. They are calculated using the local neighborhood of atoms; any defect in the local neighborhood of an atom contributes to the atom's defect. In this way, the defects are represented and visualized properly because a large volumetric region is affected.

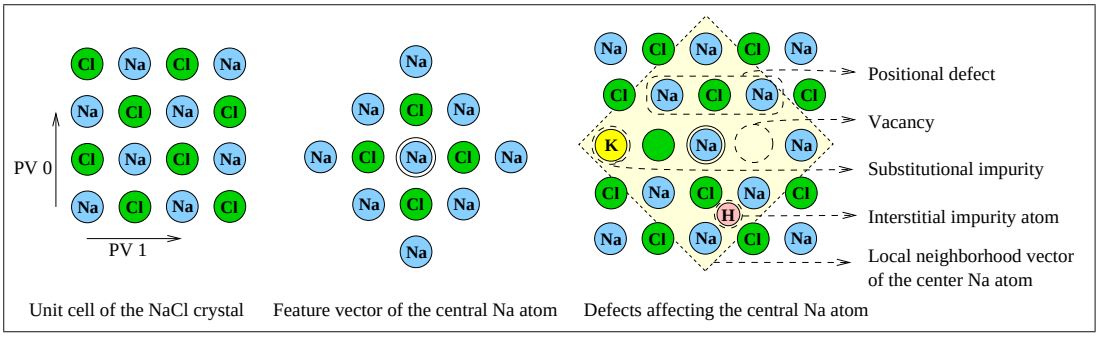



Figure 5.3: Illustration of the defect quantification for the *NaCl* crystal

Figure 5.3 illustrates a sample crystal structure with various defects. The unit cell and the primitive vectors of the *NaCl* crystal are shown on the left. Although there are simpler primitive vectors for the *NaCl* structure, we use the given primitive vectors for demonstration purposes. In the middle part, the feature vector of a *Na* atom is given. It includes every atom within the maximum primitive vector length distance to it in a perfect crystal. On the right part of the figure, a sample crystal segment demonstrates various types of crystal defects. The local neighborhood (the yellow background region) vector of the atom is compared with the feature vector of the atom and the error values that will be assigned to the atom are computed accordingly.

The defect quantification process is described in Algorithm 9. Defect quantification is performed for every atom in the crystal. First, the local neighborhood vector (*LNV*) of the atom is extracted. *LNV* includes all the atoms within a certain distance to the atom. We used the maximum primitive vector length as the distance, however this

value can be tuned by the user. Then, the feature vector, which is the local neighborhood vector of the atom in a perfect crystal, is computed.

```
DefectQuantification(Atoms A)
begin
    foreach (Atom a in A) do
        //Extract all atoms within a certain distance to
         atom a
        LNV=extractLocalNeighborhoodVector(a);
        //Extract all atoms within a certain distance to
         atom a in a perfect crystal
        FV=computeFeatureVector(a.type);
        //Assign defect upon feature comparisons
        a.defect=compareFeatures(FV, LNV);
end
```

**Algorithm 9:** Defect quantification algorithm

Lastly, the local neighborhood and the feature vectors are compared to quantify the defect value. The comparison process matches each atom in the local neighborhood vector to its corresponding atom in the feature vector. Hence, it finds any positional differences between corresponding atoms and any vacancies or interstitial impurity atoms in the local neighborhood vector. The detailed description of the defect quantification algorithm can be found in Appendix A.

### 5.2.3 Lossless Mesh Simplification

In order to capture small material features, like surface topology and crystalline defects, *MaterialVis* use highly detailed tetrahedralization where each atom is represented with a vertex. On the other hand, this representation is usually over-detailed for uniform regions in the material structures. Crystal defects constitute the volumetric features of materials for visualization purposes. *MaterialVis* aims to use volume rendering techniques to visualize such defects. Amorphous materials or perfect crystalline structures do not contain any defects; hence, their structure is mostly uniform. Moreover, many materials containing crystal defects still contain a significant portion of

uniform structure. Representing such uniform regions at a low level of detail would reduce the mesh size significantly. We propose a lossless mesh simplification algorithm that would simplify the volumetrically uniform regions in the material improving the rendering performance, without affecting the surface structure and the regions bearing some crystalline defects.

```
LosslessMeshSimplification(Atoms A, Tetrahedra T)
begin
    //Extract and sort all non-surface edges with no defect
    EdgeList=ExtractEdgeList(T);
    while EdgeList is not empty do
        e=EdgeList.getShortestEdge();
        if No tetrahedron with a vertex having non-zero defect will be affected from
        the collapse of edge e then
            //Collapse edge e into newly created vertex v'
            v'=collapse(e);
            //Delete tetrahedra that use edge e and update
             tetrahedra that use a vertex of edge e to use v'
             instead
            UpdateTetrahedra(T, e, v');
            //Update the edge list upon tetrahedral changes
            UpdateEdgeList(EdgeList, e, v');
end
```

**Algorithm 10:** Lossless mesh simplification algorithm

The lossless mesh simplification algorithm is based on edge-collapse-based reduction techniques. This algorithm was first proposed by Hoppe [28] for triangular meshes. We extended the simplification algorithm to tetrahedral meshes [1]. Edge-collapse technique works by repeatedly collapsing edges into new vertices. An edge-collapse would eliminate tetrahedra using the collapsed edge and stretch the tetrahedra using only one vertex of the collapsed edge. We specify the constraints for selecting the edges to collapse in such a way to ensure lossless compression. The details are given in Algorithm 10. In order to preserve surface details, no surface edge can be collapsed. Also, an edge with a vertex on the surface can only be collapsed onto the surface vertex. After an edge collapse, various tetrahedra are affected by either being

deleted or being stretched. If any of these affected tetrahedra contain an atom with a non-zero defect value, the edge is not collapsed because it will modify the visual output. The simplification ratio depends highly on the dataset. With the test datasets we used, we achieved simplification ratios of up to 30% of the original size. The detailed description of the lossless mesh simplification algorithm can be found in Appendix A.

## 5.3 Rendering

*MaterialVis* provides rendering functionality with various modes and display options, such as lighting and cut-planes. It utilizes graphics acceleration via *OpenGL* graphics application programming interface (API). The rendering tool supports five modes: volume and surface rendering, volume rendering, surface rendering, XRAY rendering, and atom-ball model rendering. Each rendering mode is useful for some aspect of material analysis. A user-friendly graphical interface is provided, allowing users to control the tool easily. For detailed explanation about features and functionalities of the *MaterialVis* tool, please refer to the users manual provided online.

### 5.3.1 Volume and Surface Rendering

Volume and surface rendering aims to visualize both the material topology and the crystal defects. It is the slowest but most flexible rendering mode. The user can set many properties of the visualization. The volume rendering is based on the cell-projection algorithm (see Section 4.1) that we used in our earlier work [1]. We extended the mentioned algorithm to handle surfaces. We selected the cell-projection algorithm for several reasons. First of all, cell-projection is a very robust and flexible algorithm. It can be modified to support advanced features easily. It does not require any auxiliary data such as neighboring information. Its execution flow and memory access patterns are mostly uniform, making it ideal for parallel implementations [2]. Our implementation utilizes multi-core CPU hardware. We can achieve almost linear speed-ups [2]; i.e., 3.0 to 3.5-fold speed-ups for quad-core CPUs (see Section 4.3).

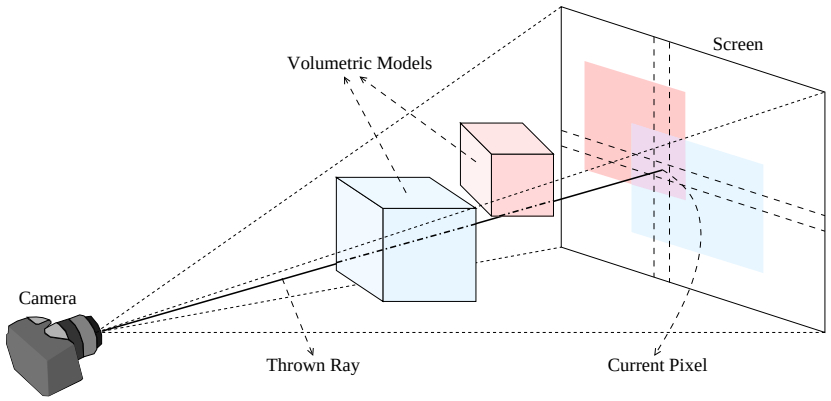



Figure 5.4: The raycasting framework

We decided not to use GPU-based implementation for two reasons. First, the conventional GPU based volume rendering algorithms, albeit being fast, cannot support features, such as surface processing, multi-variable visualization, advanced transfer functions, because they rely on limited shader programming techniques. Secondly, although the CUDA or OpenCL based GPU implementations are capable to support required features, they are not very robust and they are highly hardware dependent.

The cell projection algorithm is a ray-casting-based rendering technique. Figure 5.4 demonstrates the processing of a single pixel. The visualization parameters are the camera position, orientation and the projection angle. A ray is cast for every pixel on the screen image, traveling the volume and hitting the center of the pixel. The ray starts with full intensity. While the ray traverses the volume, its color is affected by the volume it visited and its intensity is reduced. The final color that the ray assumes after exiting the volume defines the pixel color. Algorithm 11 presents our version of the cell-projection algorithm.

The cell-projection algorithm projects each tetrahedron and face onto the image as the first step. All the pixels that lie under the projections of each face and tetrahedra are found and associated with those faces and tetrahedra. The algorithm constructs the image pixel by pixel. First, the list of tetrahedra and faces associated with the current pixel are extracted. Then intersection contributions are calculated for each face or tetrahedra in the list. While calculating the contributions, tetrahedra and face intersections are treated differently. The intersection contribution structure contains two pieces of data. The first one is the camera distance to the entry point of the tetrahedron or the

```
VolumeAndSurfaceRenderer()
begin
    //Associate the tetrahedra and the faces with the
     pixels that they are projected onto
    ProjectTetrahedraOntoImageSpace();
    ProjectFacesOntoImageSpace();
    //Process pixel by pixel
    foreach Pixel p do
        //Extract the faces and tetrahedra that are
         projected upon p
        list=getProjectedFacesAndTetrahedra(p);
        foreach Face or Tetrahedra fot in list do
            //Compute the contibution of fot on the ray cast
             from p
            CalculateIntersectionContributions(fot,p);
        SortByEyeDistance(list);
        p.color={0,0,0,0};
        //Combine the intersection contributions with alpha
         blending and alpha correction to compute p's color
        foreach Face or Tetrahedra fot in list do
            CompositeColor(p.color,fot);
end
```

**Algorithm 11:** The cell-projection algorithm

face which is used in visibility sorting of intersection records. The second piece of data is the color and intensity of a full intensity ray that travels through the tetrahedron or the face.

After the intersection contributions are computed, the results are sorted according to the camera distance. Then starting from near to far, the intersection contributions are composited into a single intensity value, which is assigned as the pixel color.

The calculation of tetrahedron intersection contributions starts by finding the entry and exit points of the ray on the tetrahedron (cf. Figure 5.5 (a)). It takes several samples on the line segment between the entry and exit points. The color and transparency of each sample is calculated by interpolation. The sampled colors are combined into a single color. While combining the colors, front-to-back alpha-blending is used and the alpha channel value is corrected for each sample. The contribution of each color is proportional to the segment length of the sample. The larger the tetrahedron, the

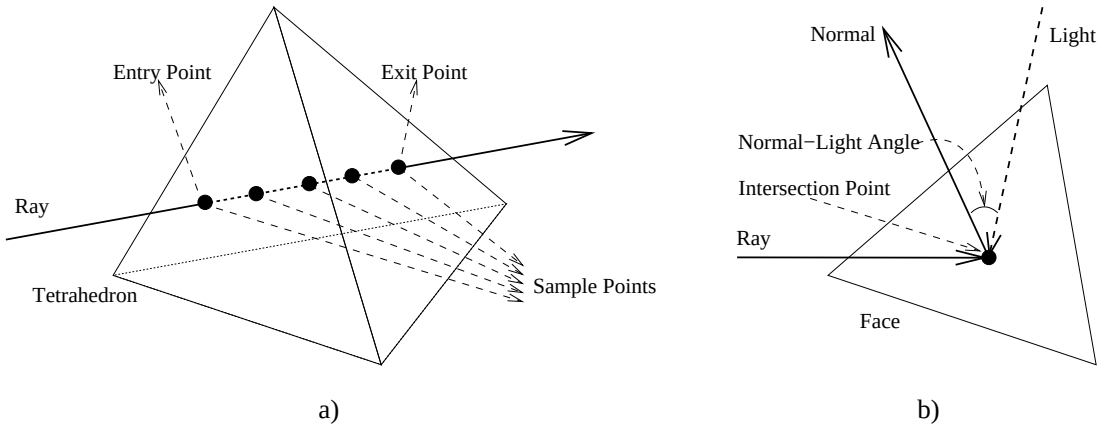



Figure 5.5: Color composition along tetrahedron-ray intersections for direct volume. a) Tetrahedron-ray intersection and sample points, and b) face-ray intersection and normal-light angle

higher its contribution will be. The remaining light intensity is directly proportional to the contribution. For example, for a fully-opaque volume, only the entry color matters because the ray will lose all of its intensity at the beginning.

Volumetric features are generally revealed by the use of appropriate transfer functions. The transfer functions are simply mapping functions that compute the color and intensity values for each set of attributes. They are very critical for the perception. The transfer function should be defined in a way to highlight the features of prime interest. Defects in crystal structures can be an example of such interested features. Usually, interesting features are present in a small fraction of the volume data. In that case, very transparent colors should be assigned to the attributes that one is not interested in and a range of relatively opaque colors should be assigned to interesting features. Thus, the interesting features can be visualized in high detail while the other parts are barely represented. Although general principles can be laid out easily, defining good transfer functions is an important research area.

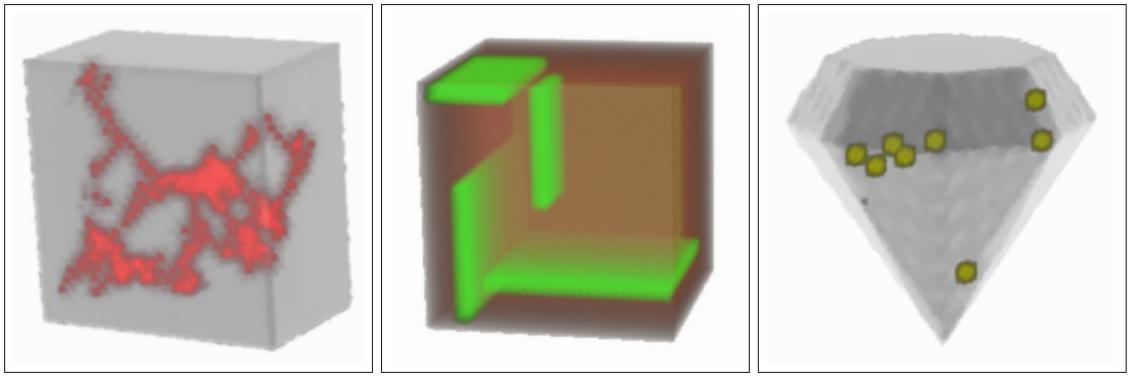
*MaterialVis* uses a simple but flexible approach for defining the transfer function. The colors of vertices are determined by the defects associated with the atom defining the vertex. The quantified defect values of an atom $a$ are converted into color values using the defect parameters of the atom as follows:

$$
\begin{aligned}
a.Color = & BaseColor + \\
& a.positionalDefect \times PositionalDefectColor \times PositionalDefectMultiplier + \\
& a.extraAtomDefect \times ExtraAtomColor \times ExtraAtomMultiplier + \\
& a.vacancyDefect \times VacancyColor \times VacancyDefectMultiplier
\end{aligned}
$$

The color and error multipliers used in the equation are tunable by the user. The face intersections are used to handle the effects of the surface. The calculation of the face intersection contributions handles the lighting effects that are missing in pure volume renderers. The color and transparency of the faces and the lighting parameters are tunable by the user.

Lighting effects underline the surface structure without hiding the volume visualization. The face intersection contribution calculation starts by finding the intersection point between the face and the ray. The distance from the camera to the intersection point is computed. The color of intersection is computed using interpolation of the colors of face vertices. The normal for the intersection point is calculated. If the shading mode is flat, than the face normal is used. If shading mode is smooth, the vertex normals are interpolated. Figure 5.5 (b) demonstrates the face ray intersection and the light-normal angle.

We use Phong illumination model for this rendering mode because the specification of an excessive number of lighting parameters used by complex illumination models puts the burden on the user. The main focus in this rendering mode is still the volume rendering part; hence, a simpler lighting model works well and is more user-friendly. More detailed explanations about volume and surface rendering algorithm can be found in Appendix A.



(a) (b) (c)

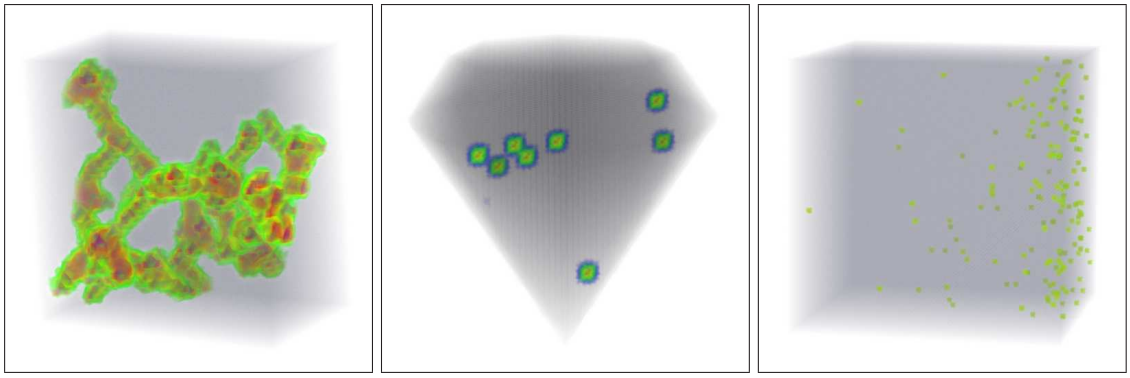
Figure 5.6: Rendered images of various dataset: (a) NaCl cracked, (b) Cu line defect, (c) A centers (substitutional nitrogen-pair defects) in diamond.

Figure 5.6 shows the visualization of some material datasets using this mode. We tuned the rendering parameters to focus on the defects in the crystal volume and the surface related parameters to give an impression of the structure itself but not overwhelm the volume visualization. Since the volume and surface rendering mode is flexible, the user can visualize the material in various ways and analyze various aspects of the data efficiently.

### 5.3.2 Volume Rendering

Volume rendering aims to visualize the defects in the crystal. Since surfaces are not represented, it gives only a very rough idea about the topology of the material. We use Hardware Assisted Visibility Sorting (*HAVS*) for volume rendering [26]. The algorithm performs some of the computations and rendering on the graphics hardware; hence, it is partially GPU accelerated. It is not as fast as surface rendering. Figure 5.7 presents the visualization of some datasets with this mode.

The high performance of the *HAVS* algorithm is due to its use of the graphics hardware. The algorithm converts the volume rendering problem into a simpler version that can be solved on the GPU. Although this approach is fast, it also has drawbacks. The first problem is in visibility sorting. *HAVS* performs a rough but fast visibility sorting



(a) (b) (c)

Figure 5.7: Volume rendering mode: (a) NaCl cracked, (b) A centers (substitutional nitrogen-pair defects) in diamond, (c) Palladium with hydrogen.

on the CPU, which may have errors. The algorithm relies on a shader program running in the GPU to correct these errors before rendering. Due to the limitations in the graphics hardware, all of the errors might not be corrected, leading to visual artifacts. This situation is very particular for irregular tetrahedralizations. Luckily, material structures have fairly regular tetrahedralization, thus *HAVS* work well with *MaterialVis*.

The second problem is the limitations on color computations. *HAVS* use a pre-integration table in terms of 3D textures to compute the contributions of tetrahedra. This brings a restriction on color computations so that the visualization attribute of the volume, the quantified defect value in our case, can only be a scalar. In the defect quantification stage, we assign three defect attributes to each vertex: *positional*, *vacancy*, and *extra (interstitial impurity) atom defects*. *HAVS* cannot handle three attributes; thus these defect values must be merged as a single defect. We compute a weighted sum using the user-specified weights: *positional defect multiplier*, *extra atom multiplier*, and *vacancy defect multiplier*. We calculate the scalar defect value of atom $a$ using the defect parameters of the atom as follows:

$$\begin{aligned} a.scalar = & a.positionalDefect \times PositionalDefectMultiplier + \\ & a.extraAtomDefect \times ExtraAtomMultiplier + \\ & a.vacancyDefect \times VacancyDefectMultiplier \end{aligned}$$

After all defect values are computed, they are normalized to the range [0, 1].

The scalar-to-color conversions are performed using a simple color map specified by the user. The color map is a set of entries mapping a certain scalar value to a certain color and intensity. The colors and intensities of intermediate scalar values are found using linear interpolation between the color map entries. Figure 5.8 shows a sample color map where five entries are defined and the whole scalar range is computed from these entries. The example map focuses on the scalar range [0.4, 0.6]; thus, it can distinguish scalar values in this range much better than the other parts.

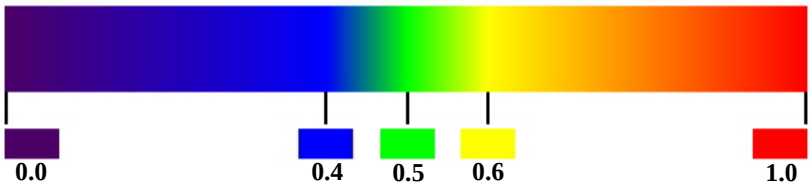
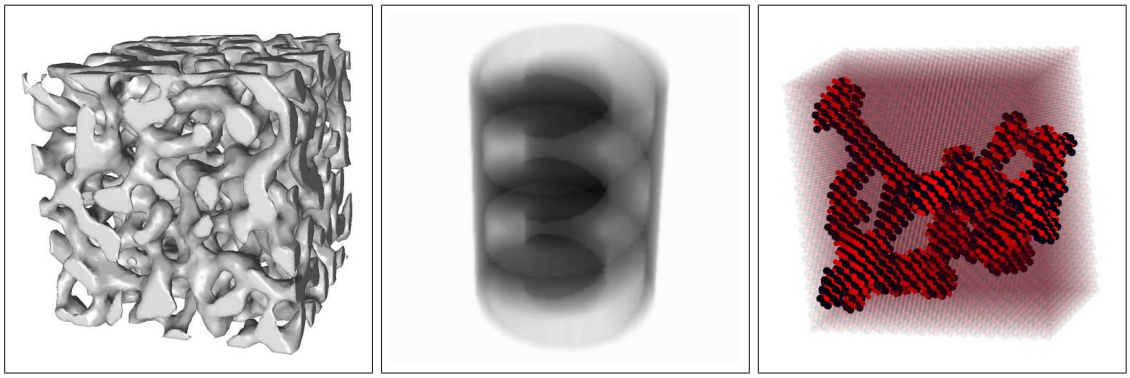



Figure 5.8: An example color map

### 5.3.3 Surface Rendering

Surface rendering aims to visualize the topological structure of the material and is suited to visualize datasets with an underlying topological structure. The sponge dataset is one example. Figure 5.9 (a) presents the rendered output of sponge dataset with this mode. For regular datasets without any specific shape, this mode cannot provide much information.



(a) (b) (c)

Figure 5.9: Sample images in different rendering modes. (a) Surface rendering mode - Sponge dataset, (b) XRAY rendering mode - $CaCuO_2$ spiral dataset, (c) Atom-ball model rendering mode - NaCl cracked dataset

We can easily render the surface of the material because the surface data is present in the volume representation. Cut-planes change the surface structure but with the surface reconstruction algorithm, the current surface data is maintained. The rendering is performed using *OpenGL* rendering functionality. The triangular mesh that represents the surface is rendered by *OpenGL* directly. Vertex or face normals are fed to the shaders, depending on the selected shading model being *smooth* or *flat*, respectively. The color and the shininess of the surface material can be specified by the user. Because surface data is directly rendered with *OpenGL*, surface rendering is GPU accelerated. The surface data is only a small portion of the volume data; hence, surface rendering is a fast rendering mode, compared to the other rendering modes.

### 5.3.4 XRAY Rendering

XRAY rendering mode can be considered as a simplified volume visualization technique. Its output resembles the XRAY images, hence it is named after it. Figure 5.9 (b) presents a material rendered in this mode. This rendering mode is particularly useful for visualizing the internal structure of crystals. It is aimed to fill a small gap that other rendering modes cannot address well. XRAY rendering mode does not visualize the errors in the structure of a crystal. Similar to the surface rendering mode, it focuses on the topology. However, unlike the surface rendering mode, it does not just visualize the outer surface but visualizes the volume.

The algorithm is a simplified version of volume and surface rendering algorithm. Basically, for each thrown ray, the faces it intersects with are found and sorted with intersection order. The odd numbered faces would be the entry faces, where ray enters inside the material and even numbered faces would be the exit faces. These faces are used to calculate the distance that the ray travels inside the material. The calculated distance is then used as the opacity coefficient for the pixel that ray is thrown for. Because the algorithm uses surface polygons to visualize the volume, the input size is much smaller than the modes that use tetrahedra. This mode is relatively fast even though the implementation is not GPU accelerated.

### 5.3.5 Atom-Ball Model Rendering

Atom-ball model rendering mode visualizes the material as a group of atoms. It does not consider the volumetric properties and the surface structure of the material. This mode is useful to understand the relations between atoms and to examine small datasets. It is the only mode that distinguishes between different types of atoms in the material because it treats the material as a set of atoms, rather than as a volume or a surface. Atoms are drawn as spheres. The user can set the colors of each atom type. The atom radii given in the input file are used as the radii of the spheres representing atoms. However, the user is allowed to set a parameter, which scales down the radii. In this way, the user can visualize the crystal with actual atom radii in a very compact form, or scale down the radii to obtain a spacious version where individual atoms can be distinguished easily.

Atom-ball model rendering can visualize the crystal defects in a restricted way. The user can set the transparency of atoms that do not contain any defects, which makes the atoms with defects distinguished easily. However, this mode cannot help to assess the magnitude of defects and differentiate different defect classes. Figure 5.9 (c) depicts the visualization of NaCl cracked dataset with this rendering mode.

The rendering is done using basic *OpenGL* functionality to draw spheres representing atoms. However, in order to handle transparency, the atoms should be sorted in visibility order. This mode is also GPU accelerated; it is a fast mode and can be used interactively.

## 5.4 Demonstration: Embedded Quantum Dot Datasets

In order to demonstrate the usage and various capabilities of *MaterialVis*, we describe the steps of how we have used the tool for the structural analysis of two real-world quantum dot datasets that we have been working on. Quantum dots are semiconductors with built-in structural irregularities. Such irregularities provide the semiconductor unique electrical properties. Quantum dots have possible uses in various areas such as

quantum computing, solar cells, medical imaging, LEDs and transistors. Biasiol and Heun [56] and Ulloa et al. [57] present in-depth information about the structure and physical properties of quantum dots.

We used two InGaAs type quantum dot datasets, one with random alloying among the cations and one without. The base semiconductor is the GaAs compound. The quantum dot is grown layer by layer. The atoms belonging to each layer are deposited onto existing layers. Deposited atoms use the existing lattice structure to bind. When the quantum dot layers are to be grown, indium atoms are deposited instead of gallium atoms at certain regions. Although the indium atoms are larger than the gallium atoms, they still fill the binding sites for gallium atoms. The resulting crystal structure becomes highly stressed. Eventually, indium atoms cause deformations in the crystal structure, relaxing to stable positions. The crystal regions with such deformations have significantly different electrical properties. By managing the deposition of indium atoms, building quantum dots with various shapes and properties is possible.

Both of the quantum dot datasets contain just under 1.5 million atoms. Due to the deformations in the crystal structure, pattern-based tetrahedralization cannot be used for quantum dot datasets. They must be treated as amorphous materials where Delaunay tetrahedralization must be used; hence, it is crucial to keep input sizes low. However, in order to simplify our task, we can mask the Arsenic atoms from the dataset. Arsenic is the common atom that is found throughout the whole material more or less homogeneously. What we are really interested in is the distribution of Gallium and Indium atoms. If Arsenic atoms are included, they will have significant effect on the volume visualization, reducing the effects of interested properties of the material. Secondly, masking the Arsenic atoms reduces the size of the datasets significantly. This helps to keep pre-processing times low.

We can also employ another input simplification technique. Volume rendering techniques mainly visualize the gallium and indium distributions in the material. It does not depend on the density of atoms in a certain region. For example, in InGaAs quantum dots, certain parts of the material will be made of just regular GaAs alloy and certain parts will be made of just InAs alloy. Because we masked the Arsenic

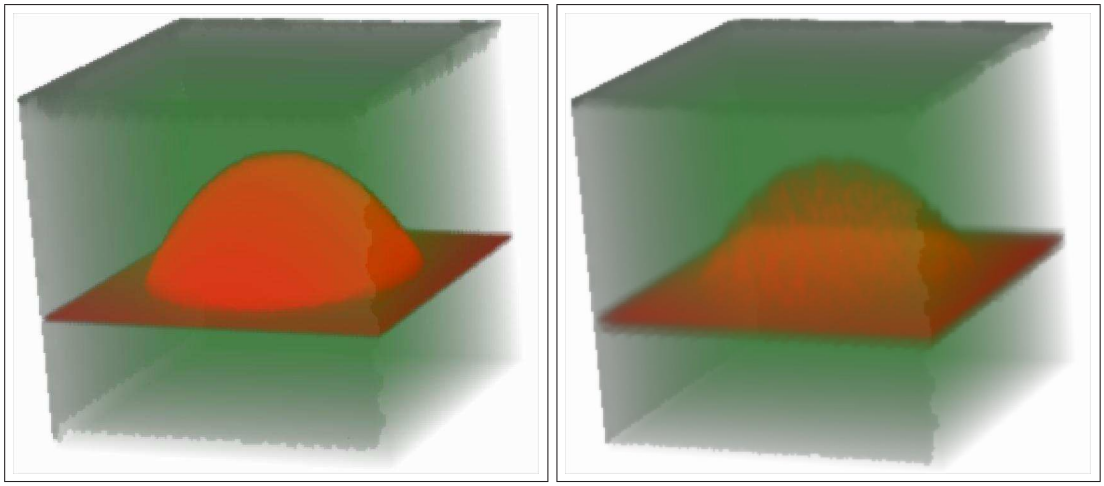
atoms, those parts will be composed of just single type of atoms. For volume rendering purposes, it does not matter if we represent such regions with all the atoms or just a fraction of them; hence, we can reduce the input size significantly.

We employed a simple data size reduction technique. First, we included the atoms belonging to the surface of the material. Because our datasets have rectangular prism shape, determining the boundary atoms was straightforward. Secondly, we uniformly sampled the whole material and included the sampled atoms, which helps to keep the tetrahedralization regular. Finally, we included every atom that has another atom of different type within a certain distance. With this technique, we can capture the regions with Gallium-Indium transitions with high detail. We also reduced the sizes of our two datasets to 5.8% and 8.5% to their original sizes, without losing any information regarding the visualization.

The next step is scalar assignment. Because we are only interested in Gallium-Indium transitions, we assigned *0.0* to Gallium atoms and *1.0* to Indium atoms. However, users can assign any scalar values depending on the properties they want to visualize. After scalar assignments, the datasets are ready to be pre-processed. Because the data sizes are kept low, pre-processing takes just a few minutes. After pre-processing, we tuned the rendering parameters. We used volume and surface rendering. We set the surface lighting parameters so that the material surfaces are just identifiable. We assigned a green, high transparency color as the base color. This color represent the Gallium atoms bearing *0.0* scalar value. The scalar values are used as the positional defect. We used a high opacity red color to positional defect. Accordingly, we observed the Indium atoms in red. Figure 5.10 depicts the rendered images of our sample datasets.

## 5.5 Benchmarks

Minimum hardware requirements of *MaterialVis* are rather modest. We tested the tool without any problems on various low end computers. On the other hand, the rendering times heavily depend on available computational power. The performance of the *volume and surface rendering* and the *XRAY rendering* modes depend on the CPU



(a) (b)

Figure 5.10: *InGaAs* quantum dots: (a) without random alloying, (b) with random alloying.

power. They can also benefit from multi-core CPUs. Other rendering modes are GPU bound modes; high-end graphics cards will increase the performance significantly. The minimal configuration should have a graphics card with *OpenGL 1.5* support. Stand-alone graphics cards with private memory is recommended. Memory requirements heavily depend on the input size. In our tests, we barely reached 1GB of memory usage. A standard personal computer with a stand-alone graphics card could run the tool without any significant latency.

We tested the tool with various datasets. In the sponge dataset [10], which was already mentioned in the introduction section, we tackled the volumetric imaging of a highly complicated structure. In the dataset we used, the stoichiometry of $SiO_x$ was fixed to $x = 1$, i.e., SiO by setting the silicon excess to 30 vol. %. There are more than half a million atoms in total.

The quantum dot represents a self-assembled InGaAs quantum dot embedded in a GaAs matrix. It contains a lens-shaped quantum dot placed on an InAs half-monolayer-thick wetting layer. The random alloy variant has 20% indium and 80% gallium compositional alloying between the cation atoms. Both structures are first prepared in the zinc blende sites of the GaAs crystal, followed by strain relaxation using molecular statics as implemented in the LAMMPS code [58]. Here, the interatomic force fields are described by the Abell-Tersoff potentials [59, 60]. The sponge and

| | Number of atoms | Pre-processing | Volume and surface rendering | Volume rendering | Surface rendering | XRAY rendering | Atom-ball rendering |
|---|---|---|---|---|---|---|---|
| NaCl Cracked | 25,725 | 9,254 | 707 | 66 | 11 | 275 | 28 |
| Cu Line Defect | 173,677 | 171,559 | 1,329 | 269 | 15 | 302 | 187 |
| Diamond Vacancy Defect | 44,982 | 17,824 | 891 | 109 | 14 | 266 | 49 |
| A Centers (Substitutional Nitrogen-pair Defects) in Diamond | 45,005 | 18,072 | 879 | 112 | 15 | 265 | 49 |
| Graphite Slided | 66,576 | 140,563 | 1,405 | 138 | 13 | 284 | 72 |
| Palladium with Hydrogen | 137,549 | 103,399 | 1,471 | 254 | 14 | 298 | 148 |
| CaCuO2 Spiral | 199,764 | 114,221 | 1,484 | 305 | 16 | 337 | 216 |
| Sponge | 534,841 | 602,869 | 2,748 | 1,015 | 22 | 471 | 578 |
| Quantum Dot without Random Alloying | 86,338 | 84,376 | 611 | 145 | 16 | 175 | 114 |
| Quantum Dot with Random Alloying | 125,595 | 1,161,757 | 730 | 196 | 16 | 182 | 181 |

Table 5.1: Preprocessing and rendering times of each dataset (in milli-seconds).

quantum dot datasets are real-world datasets that are researched actively.

The *NaCl Cracked* dataset represent a *NaCl* crystal with some positional defects. The atoms with defects represent a crack. The datasets *Cu Line Defect*, *Diamond Vacancy Defect*, *A Centers (Substitutional Nitrogen-pair Defects) in Diamond* [61], and *Graphite Slided* represent crystals with some well-known defects. The *Palladium with Hydrogen* dataset represents a block of palladium metal absorbing hydrogen from one of its faces. The $CaCuO_2$ *Spiral* dataset presents a cylinder-shaped crystal with a spiral sculptured from inside. These datasets are synthetic datasets and they are specifically designed to showcase various crystal defects and interesting topological structures using the features and capabilities of our rendering tool.

Table 5.1 presents the preprocessing and rendering times of each dataset on a middle-end PC with 3.2 GHz quad-core CPU and nVidia GTX560 GPU. The longest preprocessing time is less than 20 minutes. Despite the high computational cost of volume and surface rendering mode, the highest rendering time is 2.7 seconds for tested datasets. With other rendering modes, interactive rendering rates were achieved for all tested datasets.

# Chapter 6

# Conclusion

We propose a low overhead dynamic selective refinement scheme for unstructured tetrahedral meshes. We use a progressive mesh representation that support selective refinement. In addition to static selective refinement queries, such as spatial refinements or field value based refinements, we incorporate dynamic refinement capabilities. We propose a heuristic selective refinement algorithm, which automatically refines the mesh according to the camera parameters. We propose an importance metric that estimates the effect of the vertices on the rendered image.

We test the proposed scheme on several datasets with different characteristics and viewing parameters. The results are quite satisfactory. We achieve up to 60% speed-ups and up to 88% reduction in GPU memory requirements for the same image quality with selective refinement as compared to the non-selective refinement. We also show that different direct volume renderers can be modified to work with the our framework with little extra effort.

We propose multi-core and GPU implementations of the cell-projection algorithm for direct volume rendering. The proposed algorithms are designed to be highly memory efficient, using available system and GPU memories. With the multi-pass rendering approach, our implementation is able to render high resolution images. We tested our implementations with several large datasets with different characteristics and under various view parameters. The multi-core implementation produced up to 3.5-fold speed-ups, while the GPU implementation reached 32-fold speed-ups. Together with

the progressive rendering mechanism, we achieved interactive rates for many datasets.

*MaterialVis* is a functional visualization tool, which can easily process million-atom datasets. It supports many rendering modes to accentuate both the topology and the defects within the nanostructures. What distinguishes *MaterialVis* from other visualization tools is that it can handle the materials as a volume or a surface manifold, as well as a set of atoms.

*MaterialVis* provides many features for material visualization. It includes a user-friendly visualization environment. The user can analyze the material structure in a flexible way. *MaterialVis* supports many rendering modes, each one is useful for visualizing different aspects of the material. It is a powerful visualization tool for rendering both the topology and the defects of crystal structures. While *MaterialVis* implementation is based on various important works, many improvements are also made. Several features are added in order to improve the capabilities and usability of the tool.

*MaterialVis* combines our knowledge on direct volume visualization on a real-world application. The volume rendering framework is based on our earlier works on selective refinement for unstructured tetrahedral meshes, and multi-core parallelization of cell-projection algorithms. *MaterialVis*, to the best of our knowledge, is the only material visualization tool, that treats the materials as volumes and surface manifolds in addition to a simple set of atoms. This approach allows to visualize various aspects of the materials, such as crystalline defects and surface topology, that have been very difficult to visualize before. Together with our experience on volume visualization, the *MaterialVis* tool can reach interactive speeds, despite its high computational workload.

We believe that *MaterialVis* will be an instrumental software for crystallographers, polymer and macromolecule researchers, solid state physicists, or more generally material scientists in need to analyze nanostructures embedded within a matrix of atoms. Although only a small part of its visualization capabilities could be demonstrated throughout this work, the user can easily tune the rendering parameters with the user-friendly interface to obtain custom visual representations of materials.

There are several possible improvements to our work. The importance metric

weight computations can be automated. Optimal weights significantly differ depending on the dataset and transfer function characteristics. We determined the exponential weights experimentally. The reason for this approach was keeping the research focused. However, an algorithm that analyzes the dataset and the transfer function to compute optimal weights would be an improvement on our view-dependent refinement framework.

In our GPU implementation of Cell Projection algorithm, we proposed several solutions to improve scalability, such as multi-pass rendering and high hash block sizes. Although these approaches make rendering images with very high resolutions feasible, they are not optimized for speed. Our current implementation relies on the vertices and tetrahedra information being able to fit in the GPU memory with some extra GPU memory to spare for temporary data. This approach limits the size of the dataset that can be rendered with our implementation. These issues, which could enable rendering large datasets, would be investigated. These techniques are also applicable to other GPU-based algorithms.

*MaterialVis* tool provides a unique perspective on visualization of materials. However, it can be converted into a much powerful material analysis tool. Because the tool represents the material as a volume, a surface manifold and a set of atoms; it contains the infrastructure to perform several topological, atomic level and quantum level analysis. Thus, in collaboration with domain experts, such analysis functionalities could be added to the tool, converting it into a much capable material analysis tool.

# Bibliography


[1] E. Okuyan, U. Güdükbay, and V. İşler, "Dynamic view-dependent visualization of unstructured tetrahedral volumetric meshes," *Journal of Visualization*, vol. 15, pp. 167–178, 2012. 10.1007/s12650-011-0122-x.

[2] E. Okuyan and U. Güdükbay, "Direct volume rendering of unstructured tetrahedral meshes using CUDA and OpenMP," *Journal of Supercomputing*, vol. 67, no. 2, pp. 324–344, 2014.

[3] E. Okuyan, U. Güdükbay, C. Bulutay, and K. H. Heinig, "Materialvis: Material visualization tool using direct volume and surface rendering techniques," *Journal of Molecullar Graphics and Modelling*, vol. 50, no. 1, pp. 50 – 60, 2014.

[4] E. Okuyan, U. Güdükbay, and O. Gülseren, "Pattern information extraction from crystal structures," *Computer Physics Communications*, vol. 176, no. 7, pp. 486–506, 2007.

[5] E. Okuyan and U. Güdükbay, "Bilkristal 2.0: A tool for pattern information extraction from crystal structures," *Computer Physics Communications*, vol. 185, no. 1, pp. 442–443, 2014.

[6] P. Cignoni, L. De Floriani, P. Magillo, E. Puppo, and R. Scopigno, "Selective refinement queries for volume visualization of unstructured tetrahedral meshes," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 10, no. 1, pp. 29–45, 2004.

[7] S. P. Callahan, J. L. D. Comba, P. Shirley, and C. T. Silva, "Interactive rendering of large unstructured grids using dynamic level-of-detail," in *Proc. of IEEE Visualization*, pp. 199–206, 2005.

[8] H. Berk, C. Aykanat, and U. Güdükbay, "Direct volume rendering of unstructured grids," *Computer & Graphics*, vol. 27, no. 3, pp. 387–406, 2003.

[9] B. Gault, M. P. Moody, J. M. Cairney, and S. P. Ringer, "Atom probe crystallography," *Materials Today*, vol. 15, no. 9, pp. 378–386, 2012.

[10] J. Kelling, G. Ódor, M. F. Nagy, H. Schulz, and K.-H. Heinig, "Comparison of different parallel implementations of the 2+1-dimensional KPZ model and the 3-dimensional KMC model," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 175–187, 2012.

[11] D. Friedrich, B. Schmidt, K. H. Heinig, B. Liedke, A. Mcklich, R. Hbner, D. Wolf, S. Klling, and T. Mikolajick, "Sponge-like Si-$SiO_2$ nanocompositemorphology studies of spinodally decomposed silicon-rich oxide," *Applied Physics Letters*, vol. 103, no. 13, 2013.

[12] B. Liedke, K.-H. Heinig, A. Mcklich, and B. Schmidt, "Formation and coarsening of sponge-like Si-$SiO_2$ nanocomposites," *Applied Physics Letters*, vol. 103, no. 13, 2013.

[13] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker, "A streaming narrow-band algorithm: Interactive computation and visualization of level sets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, pp. 422–433, 2004.

[14] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware*, HWWS '00, (New York, NY, USA), pp. 109–118, ACM, 2000.

[15] K. Engel, M. Kraus, and T. Ertl, "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware*, HWWS '01, (New York, NY, USA), pp. 9–16, ACM, 2001.

[16] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser, "Smart hardware-accelerated volume rendering," in *Proceedings of the Symposium on Data Visualisation*, VISSYM '03, (Aire-la-Ville, Switzerland, Switzerland), pp. 231–238, Eurographics Association, 2003.

[17] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM Computer Graphics (Proceedings of SIGGRAPH)*, vol. 21, pp. 163–169, July 1987.

[18] M. Kraus and T. Ertl, "Cell-projection of cyclic meshes," in *Proceedings of IEEE Visualization*, pp. 215–559, 2001.

[19] R. Cook, N. L. Max, C. T. Silva, and P. L. Williams, "Image-space visibility ordering for cell projection volume rendering of unstructured data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 6, pp. 695–707, 2004.

[20] P. Shirley and A. Tuchman, "A polygonal approximation to direct scalar volume rendering," *Proceedings of the 1990 Workshop on Volume Visualization*, vol. 24, pp. 63–70, November 1990.

[21] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno, "Tetrahedral projection using vertex shaders," in *Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, VVS '02, (Piscataway, NJ, USA), pp. 7–12, IEEE Press, 2002.

[22] A. Maximo, R. Marroquim, and R. Farias, "Hardware-assisted projected tetrahedra," *Comp. Graph. Forum*, vol. 29, pp. 903–912, 2010.

[23] M. P. Garrity, "Raytracing irregular volume data," in *Proceedings of the IEEE Workshop on Volume Visualization*, (New York, NY, USA), pp. 35–40, ACM, November 1990.

[24] K. Koyamada, "Fast traversal of irregular volumes.," in *Visual Computing - Integrating Computer Graphics with Computer Vision*, (Berlin), pp. 295–312, Springer, 1992.

[25] M. Weiler, M. Kraus, M. Merz, and T. Ertl, "Hardware-based ray casting for tetrahedral meshes," in *Proceedings of the 14th IEEE Visualization*, VIS '03, pp. 44–, IEEE Computer Society, 2003.

[26] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva, "Hardware-assisted visibility sorting for unstructured volume rendering," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 11, no. 3, pp. 285–295, 2005.

[27] C. T. Silva, J. L. D. Comba, S. P. Callahan, and F. F. Bernardon, "A survey of GPU-based volume rendering of unstructured grids," *RITA*, vol. 12, no. 2, pp. 9–30, 2005.

[28] H. Hoppe, "View-dependent refinement of progressive meshes," in *Proc. of ACM SIGGRAPH*, pp. 189–198, 1997.

[29] H. Hoppe, "Progressive meshes," in *Proc. of ACM SIGGRAPH*, pp. 99–108, 1996.

[30] M. Garland and P. Heckbert, "Surface simplification using quadric error metrics," in *Proc. of ACM SIGGRAPH*, pp. 209–216, 1997.

[31] I. Trotts, B. Hamann, K. Joy, and D. Wiley, "Simplification of tetrahedral meshes," *IEEE Visualization*, pp. 287–295, 1998.

[32] P. Chopra and J. Meyer, "Tetfusion: An algorithm for rapid tetrahedral mesh simplification," in *Proc. of IEEE Visualization*, pp. 133–140, 2002.

[33] O. Staadt and M. Gross, "Progressive tetrahedralizations," in *Proc. of IEEE Visualization*, pp. 397–402, 1998.

[34] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno, "Tetrahedral projection using vertex shaders," in *Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, VVS '02, pp. 7–12, 2002.

[35] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno, "Multiresolution representation and visualization of volume data," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 3, no. 4, pp. 352–369, 1997.

[36] Z. Du and Y.-J. Chiang, "Out-of-core simplification and crack-free LOD volume rendering for irregular grids," *Comp. Graph. Forum*, vol. 29, pp. 873–882, 2010.

[37] R. Sondershaus and W. Straßer, "View-dependent tetrahedral meshing and rendering," in *Proc. of the 3rd Int. Conf. on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, pp. 23–30, 2005.

[38] CrystalMaker Software Ltd., "CrystalMaker," 2013. Available at `http://www.crystalmaker.com/crystalmaker/index.html`.

[39] Shape Software, "Shape Software," 2012. Available at `http://www.shapesoftware.com/00\_Website\_Homepage`.

[40] Robert T. Downs et al., "XtalDraw," 2004. Available at `http://www.geo.arizona.edu/xtal/group/software.htm`.

[41] K. Momma, "VESTA – JP-Minerals," 2011. Available at `http://jp-minerals.org/vesta/en`.

[42] Crystal Impact, "Diamond crystal and molecular structure visualization," 2012. Available at `http://www.crystalimpact.com/diamond`.

[43] C. F. Macrae, P. R. Edgington, P. McCabe, E. Pidcock, G. P. Shields, R. Taylor, M. Towler, and J. van de Streek, "*Mercury*: Visualization and analysis of crystal structures," *Journal of Applied Crystallography*, vol. 39, pp. 453–457, Jun 2006.

[44] D. Ushizima, D. Morozov, G. H. Weber, A. G. Bianchi, J. A. Sethian, and E. W. Bethel, "Augmented topological descriptors of pore networks for material science," *IEEE Trans. on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2041–2050, 2012.

[45] J. Li, "AtomEye: An efficient atomistic configuration viewer," *Modelling and Simulation in Materials Science and Engineering*, vol. 11, no. 2, p. 173, 2003.

[46] Dept. of Energy and Advanced Simulation and Computing Initiative, "VisIt," 2013. Available at `https://wci.llnl.gov/codes/visit/home.html`.

[47] A. Kokalj, "XCrySDen - A new program for displaying crystalline structures and electron densities," *Journal of Molecular Graphics and Modelling*, vol. 17, no. 3 - 4, pp. 176–179, 1999.

[48] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," in *Proceedings of ACM SIGGRAPH'87*, (New York, NY, USA), pp. 163–169, ACM, 1987.

[49] P. Cignoni, C. Costanza, C. Montani, C. Rocchini, and R. Scopigno, "Simplification of tetrahedral meshes with accurate error evaluation," in *Proc. of IEEE Visualization*, pp. 85–92, Oct. 2000.

[50] J. El-sana and A. Varshney, "Generalized view-dependent simplification," *Comp. Graph. Forum*, vol. 18, pp. 83–94, 1999.

[51] C. Correa and K.-L. Ma, "Visibility histograms and visibility-driven transfer function," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 17, no. 2, pp. 192–204, 2011.

[52] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," Tech. Rep. NVR-2008-001, NVIDIA Corporation, September 2008.

[53] J. Hoberock and N. Bell, "Thrust: A parallel template library, `http://www.meganewtons.com`," 2012. Version 1.3.0.

[54] A. Bowyer, "Computing Dirichlet tessellations," *The Computer Journal*, vol. 24, no. 2, pp. 162–166, 1981.

[55] D. F. Watson, "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes," *The Computer Journal*, vol. 24, no. 2, pp. 167–172, 1981.

[56] G. Biasiol and S. Heun, *Compositional Mapping of Semiconductor Quantum Dots and Rings*. Physics reports, Elsevier, 2011.

[57] J. Ulloa, P. Offermans, and P. Koenraad, "InAs quantum dot formation studied at the atomic scale by cross-sectional scanning tunnelling microscopy," *Handbook*

*of Self Assembled Semiconductor Nanostructures for Novel Devices in Photonics and Electronics*, p. 165, 2011.

[58] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995.

[59] D. Powell, M. Migliorato, and A. Cullis, "Optimized tersoff potential parameters for tetrahedrally bonded iii-v semiconductors," *Physical Review B*, vol. 75, no. 11, p. 115202, 2007.

[60] C. Bulutay, "Quadrupolar spectra of nuclear spins in strained $In_xGa_{1-x}$ as quantum dots," *Phys. Rev. B*, vol. 85, p. 115313, 2012.

[61] G. Davies, "The a nitrogen aggregate in diamond-its symmetry and possible structure," *Journal of Physics C: Solid State Physics*, vol. 9, pp. L537–L542, 1976.

# Appendix A

# *MaterialVis* Algorithms

## A.1 Delaunay Tetrahedralization Algorithm

Algorithm 12 describes the Bowyer-Watson Delaunay tetrahedralization. The algorithm starts by constructing an initial tetrahedron with four newly introduced points: *a*, *b*, *c*, and *d*. This initial tetrahedron is selected large enough to contain every point in the point set inside. Then each point in the input data set are iteratively inserted into the existing tetrahedralization constructed so far. Each tetrahedra define a sphere because four points in space defines a sphere.

The *findCollidingTetrahedra* finds the tetrahedra whose spheres contain the currently inserted point. These tetrahedra violate the Delaunay property and they need to be deleted. The *findCollidingTetrahedra* function uses an octree structure to search for such tetrahedra in an efficient way. The parameters of the enclosing sphere of each tetrahedron are computed and stored into the octree structure. Insertions into the octree structure are performed according to the center coordinates of these spheres as in a regular octree. The range information, the minimum bounding box containing all the spheres inserted into an octree node, is also stored in the corresponding node. This range information is used by the *findCollidingTetrahedra* function to decide whether to continue the search on an octree branch or not.

After the tetrahedra whose enclosing spheres contain the point to be inserted are

```
Tetrahedralization(PointSet P)
begin
    //Build the initial tetrahedron with artificial points.
     It is defined large enough so that no point in P lies
     outside of it
    Tetrahedra=new tetrahedron(new points(a,b,c,d));
    //Iterative insertion of each point
    while P is not empty do
        v=P.nextPoint;
        //Find the tetrahedra whose spheres contain v
        TSet=findCollidingTetrahedra(v);
        //Find boundary faces of the volume defined by
         colliding tetrahedra
        FSet=Set of All Faces from TSet;
        FSet=eliminateDuplicates(FSet);
        //Delete each tetrahedra in the colliding tetrahedra
         list
        foreach (tetrahedron t from TSet) do
            Tetrahedra.delete(t);
        //Create a tetrahedron for each face in the boundary
         faces
        foreach (face f from FSet) do
            Tetrahedra.add(new tetrahedron(f,v));
    //Eliminating any tetrahedra with artificial points
    foreach (tetrahedron t from Tetrahedra) do
        if (t contain points a, b,c or d) then
            Tetrahedra.delete(t);
end
```

**Algorithm 12:** Delaunay tetrahedralization algorithm

found, the faces of these tetrahedra are extracted as a sub-volume and the duplicate faces are eliminated. This duplicate elimination step eliminates both copies of the duplicate faces, thus only the faces on the surface of the sub-volume (boundaries of the cavity) remain. The extracted faces of the boundary of the cavity is used to re-tetrahedralize the void volume. The algorithm creates a new tetrahedron for each boundary face by combining the face with the newly inserted point. As the final step, any tetrahedra containing the initially introduced points, $a$, $b$, $c$ or $d$, are deleted.

## A.2 Pattern-based Tetrahedralization Algorithm

Algorithm 13 describes the pattern-based tetrahedralization. The algorithm tetrahedralizes a unit cell of the crystal and searches for the occurrence of this pattern in the actual dataset containing atoms. The process starts with the tetrahedralization of unit cells in the crystal. This part only uses the primitive and basis vectors of the crystal. One of the basis vectors are translated into the origin and the atoms of the unit cell are extracted. The unit cells do not just include the basis vectors; they also contain any atom whose coordinates in terms of primitive vectors relative to the basis vector translated to the origin is in the unit range at all dimensions. In other words, unit cell atoms include atoms from neighboring unit cells whose atoms lie on the shared faces. This set of unit cell atoms are tetrahedralized using Delaunay tetrahedralization. However, some restrictions are applied. Mainly, the tetrahedralization of the unit cell must be constrained so that the corresponding faces, namely top-bottom, left-right, far-near, must match. This constraint is crucial because these faces will be shared when unit cells are stacked in the crystal and the constraint ensures the created tetrahedra from neighboring unit cells fit together perfectly.

Figure A.1 demonstrates how pattern-based tetrahedralization works. For illustration purposes, we use a simple two-dimensional cubic lattice with just nine atoms, instead of a three-dimensional lattice. The unit cell of the lattice contains a single basis vector; thus, it has four corner atoms. The unit cell triangulation for the 2D case contains two triangles (tetrahedra for the 3D case). The calculation of new triangles starts with a random vertex, the third vertex in the example. To determine the unit cell of this vertex, the algorithm checks the existence of three other unit cell vertices with correct relative coordinates to the third vertex. The $0^{th}$, $1^{st}$ and $4^{th}$ vertices are found. The first and second triangles (tetrahedra) are defined using the unit cell triangulation (tetrahedralization) template. The atoms in the immediate neighborhood of the third vertex, $0^{th}$, $4^{th}$ and $6^{th}$ vertices, are inserted into a processing queue. The vertices are processed in the order they are inserted into the queue. The green vertex represents the currently processed atom, the turquoise vertices represent the atoms in the processing queue, and the brown vertices are the processed atoms.

The algorithm uses an octree structure to speed up the search of atoms in a specific

```
PBT(PointSet P, UnitCellInfo uci)
begin
    //Construct the unit-cell tetrahedralization template
    UCT=UnitCellTetrahedralization(uci);
    //Construct the Octree and insert every point into it
    Octree pointTree = createOctree(P);
    //Initialize the breath-first queue
    Queue BFQueue=new Queue();
    point p=selectSeed(P,uci);
    BFQueue.enqueue(p);
    //Unit-cell discoveries with breath-first approach
    while BFQueue is not empty do
        p=BFQueue.dequeue();
        //Search for template points centering p
        foreach (point q in UCT) do
            pointTree.search(p + q);
        if (no point in UCT could be found) then continue;
        //Tetrahedra discoveries
        foreach (tetrahedron t in UCT) do
            if (every point in t has been found) then
                Tetrahedra+=new tetrahedron(p, t);
        //Enqueue the seeds of neighboring unit-cells into
         the BFQueue
        for (i = -1 to 1) do
            for (j = -1 to 1) do
                for (k = -1 to 1) do
                    if (i, j, k) = (0, 0, 0) then continue;
                    point q=p+i × uci.PV[0] + j × uci.PV[1] + k × uci.PV[2];
                    if (q has not been enqueued before) then
                        BFQueue.enqueue(q);
    ReconstructSurface();
end
```

**Algorithm 13:** Pattern-based tetrahedralization algorithm

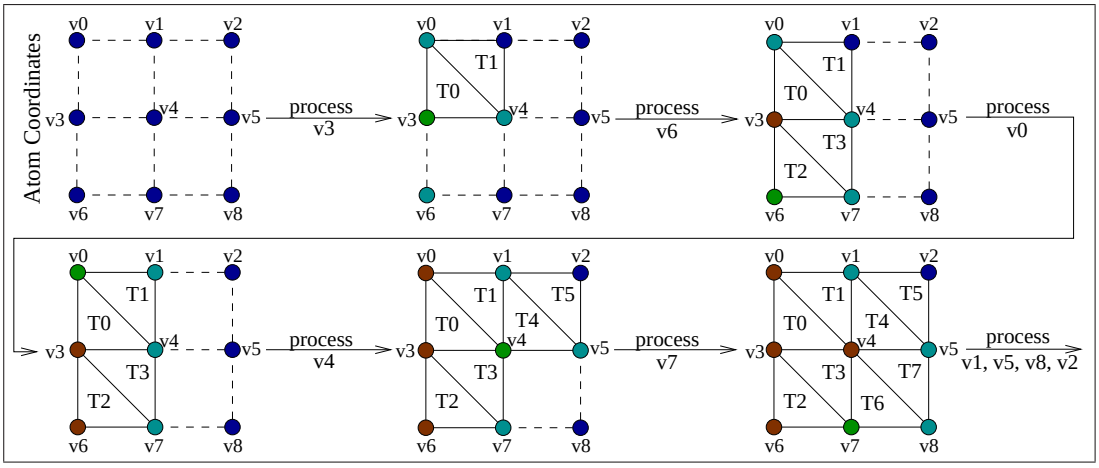



Figure A.1: The illustration of the pattern-based tetrahedralization

region. It selects a seed point to start the search process. The seed can be any atom from the crystal with the same type of the basis vector at the origin. The algorithm searches for unit cells with the same type of basis vector using a breadth-first strategy because this strategy works better to locate the crystal defects than other search strategies, such as depth-first search. After the seed is selected, it is inserted into the *BFQueue* structure and the search starts. At each iteration of the breadth-first search, the algorithm selects the first atom, $p$, from the *BFQueue*. Then it tries to match the neighbors of $p$ to the unit cell tetrahedralization (*UCT*) template. The algorithm checks the existence of each point in the *UCT* template relative to $p$ in the actual crystal. Each *UCT* tetrahedra whose all points relative to $p$ have been identified are inserted into *Tetrahedra* list.

After the search of the unit cell tetrahedra is completed for this seed point, the algorithm finds new seed points and the search continues in the same manner. The breadth-first search continues by checking the immediate neighbors of $p$'s unit cell. If the base points of these unit cells have not been enqueued before, they are enqueued. It should be noted that the algorithm does not require the existence of such points. In fact, if such points do not exist in the crystal, virtual points are introduced and inserted into the *BFQueue*. This is required in cases such as the seed point could be a missing atom or an atom near the surface of crystal because ignoring such atoms will cause missing the remaining unit cell. The algorithm ensures the termination by stopping the breath first search on current seed when no atom could be found from its unit cell.

Because the pattern-based tetrahedralization utilizes unit cells to tetrahedralize the

crystal, it cannot handle surfaces that do not align with unit cell faces. In such cases, very rough and jagged surfaces are created. In order to smooth such surfaces, the algorithm simply adds some new tetrahedra in rough parts to fill the cavities on the surface.

## A.3 Defect Quantification Algorithm

*MaterialVis* calculates defect values of atoms for each type of defect. They are calculated using the local neighborhood of atoms; any defect in the local neighborhood of an atom contributes to the atom's defect. In this way, the defects are represented and visualized properly because a large volumetric region is affected. The defect quantification is described in Algorithm 14.

Defect quantification starts by determining the boundaries of the local neighborhood. We define the local neighborhood as the sphere around the atom whose radius is the length of largest primitive vector, called local neighborhood radius (*LNR*). Using small *LNR* values will reduce the size of the local neighborhood, thus any defect will be represented in a small part of the crystal. Using large *LNR* values will make any defect to be represented in a large part of the crystal, which increases the computational cost. The trade-off between the computational cost and the visualization quality is controlled with a user-specified parameter.

The next step is to construct the feature vectors for each basis vector. The feature vector is a sample that represents the appearance of the perfect crystal. The *createFeatureVector* function takes a basis vector, translates it to the origin and identifies all the atoms around it within the distance *LNR* or closer from the perfect crystal. The algorithm continues by determining the local neighborhood of each atom from the actual crystal and comparing it to the feature vector. However, determining the local neighborhood of atoms is not a trivial task. The brute-force approach is to scan all the atoms and extract the ones that lie in the local neighborhood. This leads to quadratic computational cost and it is very time consuming for large datasets. Octrees storing range data could be utilized to speed up the process. Because of the overhead of constructing and maintaining the octree, we use regular grids to speed up the process, whose

construction and maintenance is simple.

```
DefectQuantification(Atoms A, UnitCellInfo uci)
begin
    //Define the local neighborhood radius; the distance
     that an atom is affected from defects within
    LNR=LengthOfLargestPrimitiveVector;
    //Compute the feature vectors; the type and relative
     position of atoms to each basis vector within LNR
     distance in the perfect crystal
    for (i = 0 upto NumOfBasisVectors) do
        FV[i]=createFeatureVector(i, uci, LNR);
    //Construct the grids
    atom QueryGrid[32][32][32];
    atom RefGrid[32][32][32];
    CreateGrid(A, LNR, QueryGrid, RefGrid);
    //Compare the feature vector and local neighborhood
     vector for each atom using grid approach
    for (i = 0 upto 32) do
        for ( j = 0 upto 32) do
            for (k = 0 upto 32) do
                foreach (Atom atomR in RefGrid[i][j][k]) do
                    LNV=NULL;
                    //Extract the local neighborhood vector
                    foreach (Atom atomQ in QueryGrid[i][j][k]) do
                        if (distance(atomR, atomQ) ≤ LNR) then
                            LNV+=atomQ;
                    //Assign defect upon feature comparisons
                    atomR.defect=compareFeatures(FV[atomR.type], LNV);
end
```

**Algorithm 14:** Defect quantification algorithm

Figure A.2 illustrates a simple 2D cubic lattice with 64 atoms mapped by a $2 \times 2$ grid. Query grids have the exact dimensions of $4 \times 4$. Thus the lattice is divided equally into four query grids. For each query grid, there is a corresponding reference

grid. The reference grid contains every atom that is within maximum feature vector length distance to any atom in the query grid for any dimension. To find the feature vector of any atom in a query grid, only searching the atoms in the corresponding reference grid is sufficient.

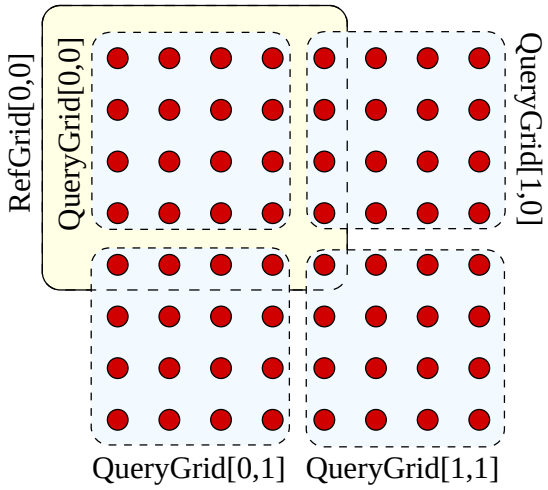



Figure A.2: The illustration of the query and reference grids

The algorithm constructs regular grids to speed-up the local neighborhood extraction process. The regular grid divides the volume into $32 \times 32 \times 32 = 32,768$ cells (sub-volumes). Then atoms are inserted into one of these cells of the reference grid (*RefGrid*) according to their coordinates. The query grid (*QueryGrid*) is constructed in a different way. Any atom whose coordinate values differ by at most *LNR* at each axis from the boundaries of a cell of the *QueryGrid* is inserted into that cell. In other words, for any atom $a$ in the cell $RefGrid[x][y][z]$, we identify all the atoms within *LNR* distance to $a$ in $QueryGrid[x][y][z]$. This process is repeated for each cell in the regular grid. The local neighborhood vectors, *LNV*, for each atom in $RefGrid[x][y][z]$, are found using the query grid cell $QueryGrid[x][y][z]$. Although the algorithm still have quadratic time complexity, the number of comparisons are significantly reduced with the regular grid approach.

The *compareFeatures* function compares the feature and local neighborhood vectors of an atom and assigns the defect value. The first step is to match the elements of the feature and local neighborhood vectors. Two atoms are matched only if their atom types are identical and their positions relative to their origin differ less than a

pre-specified threshold. After matchings are completed, the squares of the relative distance differences between matched atoms are added. This value multiplied with a constant depending on the unit cell size and assigned to the atom as the positional defect, or simply defect. The number of atoms left unmatched in the feature vector is assigned as the missing atom defect, and the number of atoms left unmatched in the local neighborhood vector is assigned as the extra atom defect.

There are also some special cases that *compareFeatures* function should handle. First of all, near the boundaries of vectors, the positional defects can cause false missing and extra atom defects. For example, an atom that matches to an atom near the boundaries of the feature vector (FV) might be left out of the local neighborhood vector (LNV) due to some positional defect, leading to a false missing atom defect. Similarly, some atoms that must have been left out of the local neighborhood vector (LNV) could be included, leading to a false extra atom defect. In order to avoid these cases, unmatched atoms near the boundaries of the feature vector and the local neighborhood vectors are not counted as defects. Secondly, atoms very close to the surface will cause significant missing atom defects. Because such missing atoms are caused by the topology of the crystal, they should not be treated as missing atoms. The algorithm simply ignores missing atom defects very close to the surface.

## A.4 Lossless Mesh Simplification Algorithm

The lossless mesh simplification algorithm is based on edge-collapse based reduction techniques. Edge-collapse technique works by repeatedly collapsing edges into new vertices. An edge-collapse would eliminate tetrahedra using the collapsed edge and stretch the tetrahedra using only one vertex of the collapsed edge. We specify the constraints for selecting the edges to collapse in such a way to ensure lossless compression. The details are given in Algorithm 15. In order to preserve surface details, no surface edge can be collapsed. Also, an edge with a vertex on the surface can only be collapsed onto the surface vertex. After an edge collapse, many tetrahedra are affected by being deleted or being stretched. If any of affected tetrahedra contain an atom with non-zero defect value, the edge is not collapsed since it will modify the visual output.

```
LosslessMeshSimplification(Atoms A, Tetrahedra T)
begin
    //Construct the edge list that stores collapse
     candidates
    foreach Edge e of T do
        if e is not a surface edge AND both vertices of e have zero defect then
        EdgeList.Insert(e);
    Sort(EdgeList);
    //According to edge length from shortest to longest
    while EdgeList is not empty do
        e=EdgeList.getFirst();
        //Get the first edge and remove it from the EdgeList
        AffectedTetrahedra=getAffectedTetrahedra(e);
        //Find and return any tetrahedra in T that contain
         one or both vertices of e; these would be affected
         from the collapse of e
        //Continue to collapse e only if the collapse will
         not affect any part with non-zero defect values
        if If no tetrahedron in AffectedTetrahedra contain a vertex with non-zero
        defect value then
            v'=collapse(e);
            //If e contains no surface vertices, v' is set to
             the center of two vertices of e.  If e contains
             one surface vertices, v' is set to that surface
             vertex
            foreach tetrahedron t in AffectedTetrahedra do
                if t contain both vertices of e then T.delete(t);
                //Delete any tetrahedra that use e as an edge
                else Update(t,e,v');
                //Update any reference to vertices of e in t to
                 the new vertex v'
            UpdateEdgeList(EdgeList,e,v');
            //Delete any edge in EdgeList which use one of
             the vertices of e and insert every newly created
             edge that use v' to the EdgeList
end
```

**Algorithm 15:** Lossless mesh simplification algorithm

The simplification ratio highly depends on the dataset. With the test datasets we used, we achieved simplification ratios up to 30% of the original size.

## A.5 Volume and Surface Rendering Algorithm

Algorithm 16 presents our version of the cell-projection algorithm. The cell-projection algorithm projects each tetrahedron and face onto the image as the first step. The projections are stored in terms of intersection records, which represent a certain primitive is projected upon a certain pixel. For each pixel in the image, a list of intersection records are maintained. Screen space projections of vertices are computed and stored in the *SSC* array. Then the projections of tetrahedra and faces are computed and corresponding intersection records are inserted into the *PerPixelIntersectionLists* array. Because the pixel coordinates are implicitly stored in array indices of *PerPixelIntersectionLists* array, the inserted records only store a pointer to the face or tetrahedron. After the projections of all tetrahedra and faces are processed, the *PerPixelIntersectionLists* array contain a list for each pixel, which stores all tetrahedra and faces that project onto that pixel.

The algorithm constructs the image pixel by pixel by computing the intensity contributions of each intersection record. Tetrahedra and face intersection records are treated differently while calculating the intensity, but the output intensity structure is identical. The intensity contribution structure contains two pieces of data. The first one is the camera distance to the entry point of the tetrahedron or the face. This data is used in visibility sorting of intersection records. The second piece of data is the intensity contribution of the ray that travels through the tetrahedron or the face.

After the intensities are computed, the results are sorted according to the camera distance. Then starting from near to far, the intensity contributions are composed into a single intensity value, which is assigned as the pixel intensity.

The calculation of tetrahedron intensity contributions is described in Algorithm 17. The algorithm starts by finding the entry and exit points of the ray on the tetrahedron (cf. Figure 5.5 (a)). It samples points along the line segment between the entry and

```
SurfaceAndVolumeRenderer()
begin
    //Calculate the screen-space coordinates of each vertex
    SSC=ComputeScreenSpaceProjections(Vertices);
    //Group the lists storing every ray-tetrahedron and
     ray-face intersections for each pixel
    IntersectionRecord PerPixelIntersectionLists[Width][Height];
    //Fill out the PerPixelIntersectionLists list via the
     projection of tetrahedra and faces onto the screen
    foreach (tetrahedron t in volume data) do
        ProjectionPixels=ExtractProjectionPixels(t,SSC);
        foreach (pixel p in ProjectionPixels) do
            PerPixelIntersectionLists[p.x][p.y]+=t;
    foreach face f in surface data do
        ProjectionPixels=ExtractProjectionPixels(f,SSC);
        foreach (pixel p in ProjectionPixels) do
            PerPixelIntersectionLists[p.x][p.y]+=f;
    foreach Pixel p with indices i,j do
        list=PerPixelIntersectionLists[i][j];
        IntensityContrib IntensityContribList=NULL;
        //Cast the ray for the current pixel
        Ray R=new Ray(i,j);
        //Compute the intensity contributions of each
         intersection with R
        foreach (IntersectionRecord ir in list) do
            if (ir is tetrahedron intersection) then
                IntensityContribList+=CalculateTetrahedronIntensityContrib(ir, R);
            else if (ir is surface intersection) then
                IntensityContribList+=CalculateSurfaceIntensityContrib(ir, R);
        //Sort the intensity contributions according to eye
         distances
        SortIntensityContributions(IntensityContribList);
        //Compose the intensity contributions in sorted
         order
        Color c = ComposeIntensityContributions(IntensityContribList);
        Image[i][j]=c;
end
```

**Algorithm 16:** The cell-projection algorithm

exit points. The intensity of each sample is calculated by interpolating the intensities of tetrahedron vertices. The interpolated intensity also contains the alpha channel value representing the transparency. The sampled intensities are combined into a single intensity. While combining the intensities, front-to-back alpha-blending is used and the alpha channel value is corrected for each sample. The contribution of each intensity value is proportional to the segment length of the sample. For a fully-opaque volume, only the entry intensity matters because the ray will lose all of its intensity at the beginning.

The intensity of vertices are determined by the defects associated with the atom defining the vertex. The quantified defect values of an atom $a$ are converted into the intensity values using the following equation:

$$
\begin{aligned}
a.Color = & BaseColor + \\
& a.PositionalDefect \times PositionalDefectColor \times PositionalDefectMultiplier + \\
& a.ExtraAtomDefect \times ExtraAtomColor \times ExtraAtomMultiplier + \\
& a.VacancyDefect \times VacancyColor \times VacancyDefectMultiplier
\end{aligned}
$$

The color and error multipliers used in the equation are tunable by the user. Algorithm 18 calculates the surface intensity contributions. The color and transparency of the faces and the lighting parameters are tunable by the user.

The algorithm for computing the surface intensities starts by finding the intersection point between the face and the ray. The distance from the camera to the intersection point is computed. The intensity of intersection is computed using interpolation of the intensities of face vertices. The normal for the intersection point is calculated. If the shading mode is flat, the face normal is used. If shading mode is smooth, the vertex normals are interpolated. Figure 5.5 (b) demonstrates the face ray intersection and the light-normal angle. We use Phong illumination model for this rendering mode. The main focus in this rendering mode is still the volume rendering part; hence, a simpler lighting model is user-friendly and works well.

```
CalculateTetrahedronIntensityContrib(ir, R)
begin
    IntensityContrib ic;
    tetrahedron t=ir.tetrahedron;
    //Calculate the entry and exit points of the ray on the
     tetrahedron
    [EntryPoint, ExitPoint]=findIntersectionPoints(t, R);
    ic.distance=|Camera − EntryPoint| ;
    //Calculate the sample length
    d=|ExitPoint − EntryPoint| /NumOfSamples ;
    ic.int=0, 0, 0, 0;
    //Sample points along the line segment in the
     tetrahedron
    for (i=0 upto NumOfSamples) do
        //Find the position and the intensity of the sample
         point via linear interpolation
        point p=(i×EntryPoint+(NumOfSamples−i)×ExitPoint)/NumOfSamples;
        Intensity pInt=InterpolateIntensity(p, t);
        //Add the contribution of current sample on ic.int
        ic.int=CombineIntensity(ic.int, pInt, d);
    return ie;
end
```

**Algorithm 17:** Calculation of tetrahedron intensity contributions.

```
CalculateSurfaceIntensityContrib(ir, R)
begin
    IntensityContrib ic;
    face f=ir.face;
    //Compute the ray-face intersection point
    IntersectionPoint=findIntersectionPoint(f, R);
    ic.distance=|Camera − IntersectionPoint| ;
    ic.int=InterpolateIntensity(IntersectionPoint, f);
    //Use interpolated vertex formals or face normal
     depending on the shading model
    N=getNormal(IntersectionPoint, f);
    foreach (Light l) do
        //Calculate the intensity contribution for each
         light source using Phong illumination model and
         add to the intensity
        ic.int += Calculated intensity for light source l;
    return ic;
end
```

**Algorithm 18:** The calculation of surface intensity contributions.

## A.6 XRAY Rendering Algorithm

XRAY rendering mode can be considered as a simplified volume visualization technique. Algorithm 19 describes XRAY rendering. The algorithm starts similar to the cell-projection algorithm. The first difference is that this algorithm do not extract intersection records from tetrahedra. The second difference comes from the intensity calculations. This mode do not calculate the exact intensities; hence, the *CalculateXRAYIntensityContrib* function just computes the distance from the camera to the intersection point. The *ReduceDistance* function calculates the distance that the current ray travels inside the material by using the intensity contribution list. The odd numbered records indicate the faces that the ray enters into the material and the even numbered records indicate the faces that the ray exited the material. Thus the total distance the ray travels through can be computed. Finally, the intensity of the pixel can be computed by multiplying *XRAYAlphaLength*, *XRAYBaseColor*, and the distance that the ray travels through the material.

```
XRAYRenderer()
begin
    //Calculate the screen-space coordinates of each vertex
    SSC=ComputeScreenSpaceProjections(SurfaceVertices);
    //The list storing every ray-face intersections grouped
     into pixels
    IntersectionRecord PerPixelIntersectionLists[Width][Height];
    //Fill out the PerPixelIntersectionLists list via the
     projection of faces onto the screen
    foreach (face f in surface data) do
        ProjectionPixels=ExtractProjectionPixels(f, SSC);
        foreach (pixel p in ProjectionPixels) do
            PerPixelIntersectionLists[p.x][p.y]+=f;
    foreach Pixel p with indices i,j do
        list=PerPixelIntersectionLists[i][j];
        IntensityContrib IntensityContribList=NULL;
        //Cast the ray for the current pixel
        Ray R=new Ray(i, j);
        //Calculate the intersection point to eye distances
         for each face intersection with R
        foreach (IntersectionRecord ir in list) do
            IntensityContribList += CalculateXRAYIntensityContrib(ir, R);
        //Sort the intersections with R according to their
         eye distance
        SortIntensityContribs(IntensityContribList);
        //Extract the distance that the ray travels inside
         the volume
        float d=ReduceDistance(IntensityContribList);
        Image[i][j]=d × XRAYAlphaLength × XRAYBaseColor;
end
```

**Algorithm 19:** XRAY rendering algorithm

# Appendix B

# *MaterialVis* User Manual

## B.1 Installation Notes

*MaterialVis* is a stand-alone program. Simply, the required files must be placed in the same directory. The *MaterialVis.exe*, *MaterialVisUI.exe* and *Tetrahedralization.exe* files are the main executable files. Required dll files, *glut32.dll*, *glew32.dll* and *AntTweakBar.dll*, must either be copied into a system directory, or be placed alongside the executables. The shaders directory and its contents (**.vp* and **.fp* files) must also be placed alongside the executables. By default, all these files are presented together, thus no manual work is necessary.

The executables can also be built from the source code. The project is developed using *Microsoft Visual Studio 2005*. However, it can be converted to a higher Visual Studio version and compiled. Apart from the standard libraries coming together with the Visual Studio, the project requires the *GLUT* libraries. *GLUT* is a freeware graphics library and the latest version of the *GLUT* libraries can be obtained from the Internet. For the installation and the Visual Studio integration, please refer to the *GLUT* documentation.

## B.2 File Formats

*MaterialVis* operates on a very simple input format. The input file contains a line of text for each atom in the material specifying the type of the atom and its 3D coordinates. For amorphous materials, this data is sufficient. However, if the input material is a crystal structure, the data also must include primitive and basis vector data of the crystal structure. The orientations of these vectors must match the crystal structure; when a crystal structure is analyzed, exactly the same unit cell data must be extracted from the crystal. The file extension is important. The .dat extension must be used, for the *MaterialVis* to determine the type. Please refer to *.dat* files, from the sample datasets as examples.

The input datasets with *.dat* extension are used for pre-processing. The preprocessing stage outputs the volume representation into a file, which can be rendered by the rendering module. The output file can either be in binary or in text format, depending on the user's preference. The binary format is more compact and faster to load, whereas the text format allows users to take the data generated by the preprocessing stage and use it with other tools. The binary format uses the *.crb* extension and the text format uses *.crt* extension. *MaterialVis* runs the rendering tool directly when a *.crb* or *.crt* files are given as input.

Users can save the display options and the rendering parameters that they tuned in the rendering tool into a view parameter file. The view parameter file has extension *.crtf*. These files are self explanatory text files and can also be edited manually.

## B.3 Hardware and Software Requirements

The hardware requirements of *MaterialVis* are modest. We tested the tool without any problems on various low end computers. On the other hand, the rendering times heavily depend on available computational power. The performance of the *Surface and volume rendering* and the *XRAY rendering* modes depend on the CPU power. They can also benefit from multi-core CPUs. Other rendering modes are GPU bound modes;

high-end graphics cards will increase the performance significantly. The minimal configuration must have a graphics card with *OpenGL 1.5* support. Stand-alone graphics cards with private memory is recommended. Memory requirements heavily depend on the input size. In our tests, we barely reached 1GB of memory usage. In general, any standard personal computer with a stand-alone graphics card could run the tool without any significant delays.

*MaterialVis* requires *Microsoft .NET Framework 2.0* or higher. The graphics card drivers must be installed properly. The graphics cards 3D acceleration and OpenGL support must also be enabled.

## B.4 Usage

*MaterialVisUI.exe* must be executed to start *MaterialVis Loader*. Other executables are designed to be called by the loader and must not be executed directly. *MaterialVis Loader* presents a very simple user interface. Initially, the dataset file must be selected. If the selected dataset file extension is *.dat*, i.e., the raw input file is selected, the loader asks for the output format. The user can select, binary, text or both. Then the pre-processing stage starts. Figure B.1 displays the *MaterialVis Loader* when a raw input is selected.

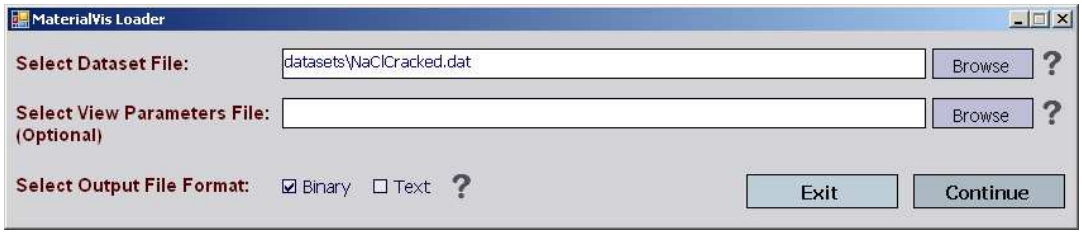



Figure B.1: *MaterialVis* Loader with raw input selected.

If a pre-processed input file (a file with *.crt* or *.crb* extension) is selected, the user can go directly to the rendering tool. If a previously saved view parameter file exists, it can be selected so that the saved configuration can be loaded. If no view parameter file is selected, then the default view parameters will be loaded. Figure B.2 displays the *MaterialVis Loader* when a pre-processed input is selected. The usage of the rendering

tool will be explained in Section B.6.

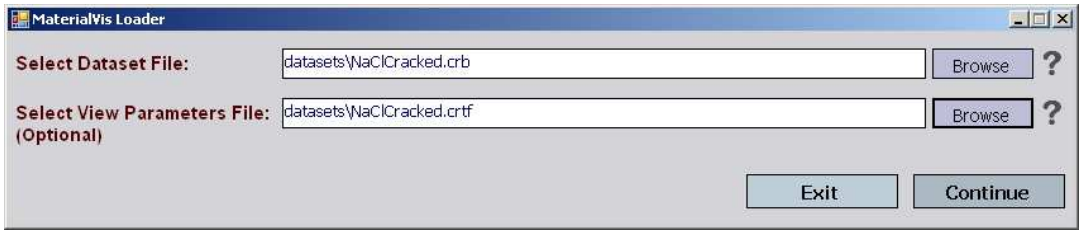



Figure B.2: *MaterialVis* Loader with pre-processed input selected.

## B.5 Pre-processing

The pre-processing stage is automated and no user intervention is required. The pre-processor window exist for informative purposes. The steps and the operations performed are logged in the window, so the user can observe the progress of the pre-processing. The user is allowed to cancel anytime. After the operation is completed, the user can return to the *MaterialVis* Loader, to continue with rendering. Figure B.3 displays the pre-processing interface.

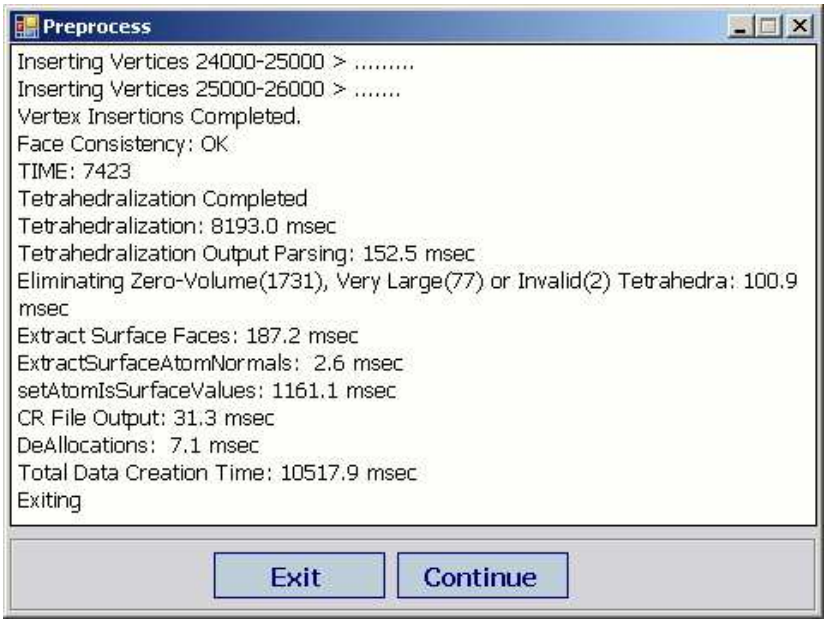



Figure B.3: *MaterialVis* pre-processing interface

## B.6 Rendering

The rendering tool is responsible for the visualization of the material in different rendering modes. Figure B.4 displays the rendering tool interface. The usage is explained in the following subsections in detail.

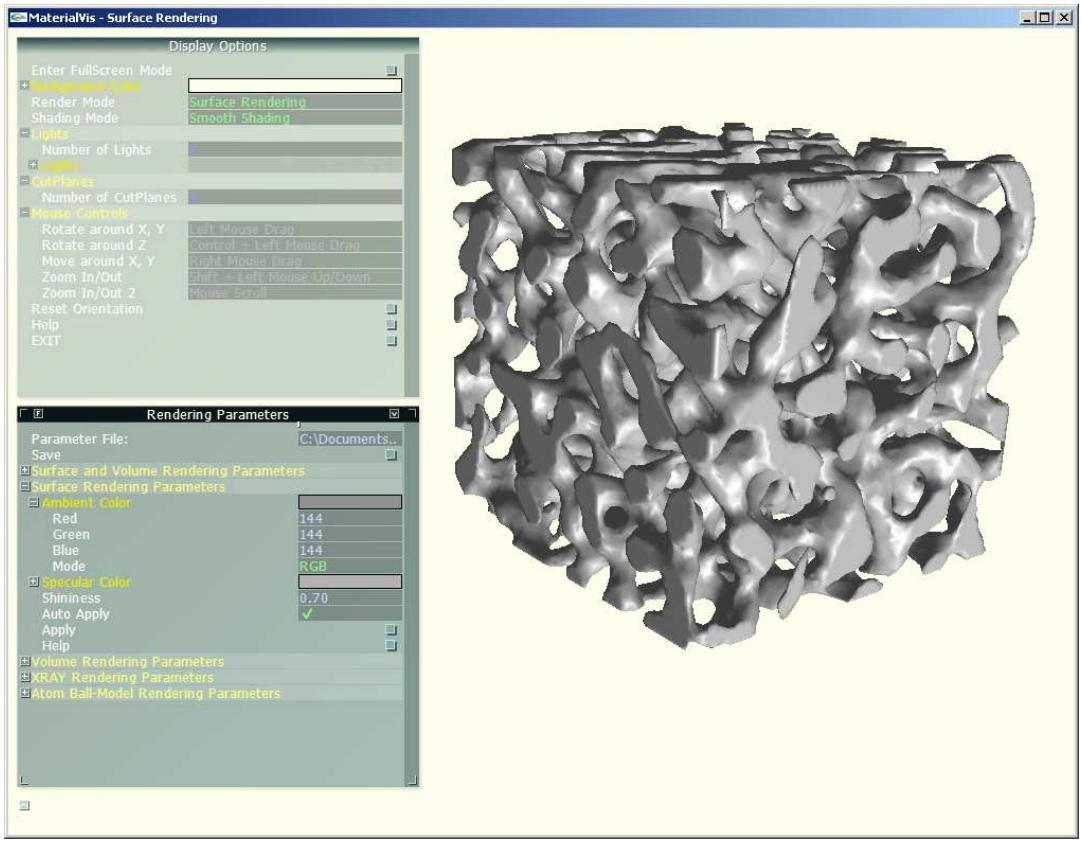



Figure B.4: *MaterialVis* rendering tool interface

### B.6.1 Controls

The orientation of the material can be controlled with the mouse and keyboard inputs. Rotations around the X, Y and Z axes, zooming in or out, and translations on the X and Y axes are supported. The list of controls and the corresponding actions are listed as follows:

- **Left Mouse Drag:** Rotate around the X and Y axes
- **Control + Left Mouse Drag:** Rotate around the Z axis
- **Right Mouse Drag:** Translate on the X and Y axes
- **Shift + Left Mouse Up/Down:** Zoom in or out
- **Mouse Scroll:** Zoom in or out

### B.6.2 Display Options


Figure B.5: The Display Options Menu

The display options menu contains renderer-mode-independent parameters, such as the background color, lighting and cut-plane parameters. Figure B.5 displays the general

layout of the menu. The following options are available.

- **Enter Full Screen Mode:** Switches between full screen and windowed display *(Shortcut: 'f')*.
- **Background Color:** Selects the background color.
- **Menu Color:** Selects the menu color.
- **Render Mode:** Selects the rendering method using the combo box *(Shortcut: 'd')*.
- **Shading Mode:** Selects the shading model *(Shortcut: 's')*.
- **Number of Lights:** Minimum one light source must be defined. Maximum eight light sources are allowed.
- **Lights:** Sets light position, color and intensity values and enable or disable the light.
- **Number of CutPlanes:** Maximum eight cut planes can be defined.
- **Cutplanes:** Sets the cut plane equation, enable or disable it. Cut plane equation is defined as $ax+by+cz\ op\ d$. Only atoms whose coordinates satisfy all enabled cut plane equations are displayed.
- **Mouse Controls:** Displays a quick help on mouse controls regarding zoom, rotations and translations.
- **Reset Orientation:** Resets the model orientation to the initial state *(Shortcut: 'r')*.
- **EXIT:** Closes the application *(Shortcut: 'q')*.

### B.6.3 Rendering Parameters

The rendering parameters menu contains renderer specific parameters. Figure B.6 displays the general layout of the menu.

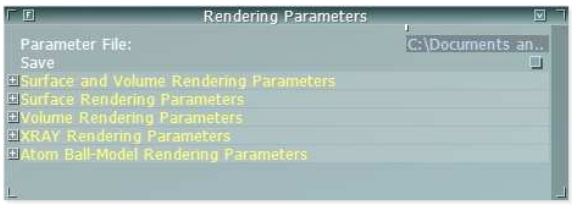



Figure B.6: The Rendering Parameters Menu - Overview

Two options regarding saving the rendering parameters are as follows:

- **Parameter File:** The file name for saving the parameter file. Enter "default" to save parameter file with the same name as the dataset.
- **Save:** Saves currently set parameters to the specified file.

#### B.6.3.1 Volume and Surface Rendering

Figure B.7 displays the volume and surface rendering parameters in the rendering parameters menu. The options presented are as follows:

- **Volume Alpha Length Constant:** Represents the opacity of the volume. Increasing this constant makes the volume more opaque.
- **Face Alpha Constant:** Represents the amount of accentuation of surface polygons. The front facing surface polygons contribute the rendered image proportional to this constant. Color values are determined by interpolating surface vertex colors.
- **Ambient Reflection Coefficient:** Represents the amount of ambient reflection for surface polygons.
- **Diffuse Reflection Coefficient:** Represents the amount of diffuse reflection for surface polygons. The contributions from all light sources are accumulated. The contribution of each light source to the diffuse color is proportional to the cosine of the angle between the surface normal and the light direction vector.

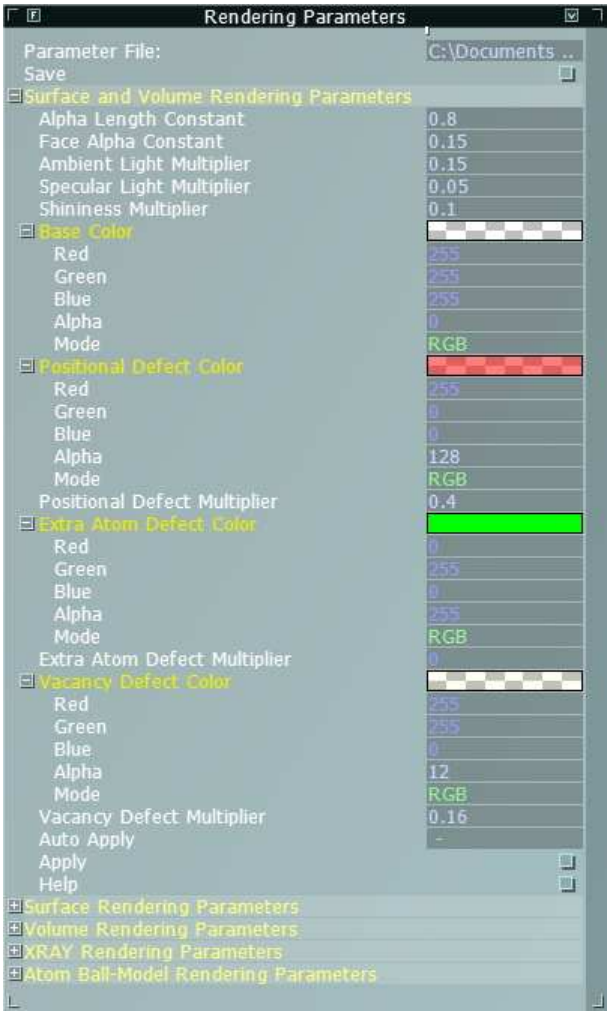



Figure B.7: The Rendering Parameters Menu - Volume and Surface Parameters

- **Specular Reflection Coefficient:** Represents the amount of specular reflection for surface polygons. The contributions from all light sources are accumulated. The contribution of each light source to the specular color is proportional to the cosine of the angle between the view vector and the reflection vector raised to the power shininess coefficient.
- **Shininess Coefficient:** Represents the shininess coefficient for surface polygons.
- **Base Color:** Represents the color assigned to each atom in the material.

- **Positional Defect Color:** Represents the color assigned to atoms with positional crystal defects in their neighborhood.
- **Positional Defect Multiplier:** Represents the weight of the positional crystal defect color on atoms final colors.
- **Extra Atom Defect Color:** Represents the color assigned to atoms with extra atom defects in their neighborhood.
- **Extra Atom Defect Multiplier:** Represents the weight of the extra atom defect color on atoms final colors.
- **Vacancy Defect Color:** Represents the color assigned to atoms with vacancy defects in their neighborhood.
- **Vacancy Defect Multiplier:** Represents the weight of the vacancy defect color on atoms final colors.
- **Auto Apply:** Enables to update rendering parameters upon any value change. This trigger re-rendering of the dataset. It is recommended to leave this option disabled because rendering is time consuming.
- **Apply:** Applies the rendering parameters and triggers re-rendering. The color of an atom 'a' is computed as

$$\begin{aligned} a.color = \; & BaseColor + \\ & a.PositionalDefect \times PositionalDefectMultiplier \times PositionalDefectColor + \\ & a.extraAtom \times ExtraAtomDefectColorMultiplier \times ExtraAtomDefectColor + \\ & a.vacancyAtom \times VacancyDefectColorMultiplier \times VacancyDefectColor \end{aligned}$$

#### B.6.3.2 Surface Rendering

Figure B.8 displays the surface rendering parameters in the rendering parameters menu. These parameters are used to define surface material. The options presented are as follows:

- **Ambient and Diffuse Color:** Represents the ambient and diffuse colors of the surface material.
- **Emission Color:** Represents the emission color of the surface material.
- **Specular Color:** Represents the amount of specular color of the surface material.
- **Shininess Coefficient:** Represents the shininess coefficient for the surface material.
- **Auto Apply:** Enables to update rendering parameters upon any value change. Rendering in this mode is fast, so auto applying is recommended.
- **Apply:** Applies the rendering parameters and triggers re-rendering.

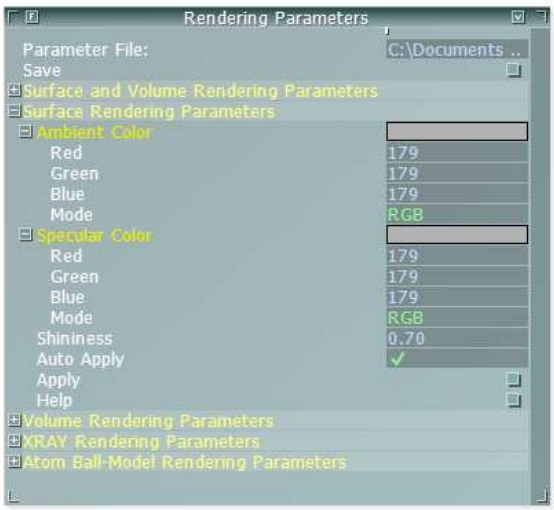



Figure B.8: The Rendering Parameters Menu - Surface Parameters

#### B.6.3.3 Volume Rendering

Figure B.9 displays the volume rendering parameters in the rendering parameters menu. The options presented are as follows:

- **Volume Alpha Length Constant:** Represents the opacity constant of the volume. Increasing this constant will make the volume more opaque.
- **Positional Defect Multiplier:** Represents the weight of positional defects on the scalar value of an atom.
- **Extra Atom Defect Multiplier:** Represents the weight of extra atom defects on the scalar value of an atom.
- **Vacancy Defect Multiplier:** Represents the weight of vacancy defects on the scalar value of an atom.
- **Number of ColorMap Entries:** Minimum two entries must be defined. Maximum 32 entries are allowed.
- **Color Map Entry:** Scalar field represents the normalized scalar value. The valid range is [0,1]. Color field represents the corresponding color.
- **Auto Apply:** Enables to update the rendering parameters upon any value change, which triggers re-rendering of the dataset. It is recommended to leave this option disabled because rendering is time consuming.
- **Apply:** Applies the rendering parameters and triggers re-rendering. The scalar value of an atom 'a' is calculated as

$$\begin{aligned} a.scalar = \; & a.PositionalDefect \times PositionalDefectMultiplier + \\ & a.extraAtom \times ExtraAtomDefectColorMultiplier + \\ & a.vacancyAtom \times VacancyDefectColorMultiplier \end{aligned}$$

  After scalars are computed for every atom, the values are normalized to [0,1] range. These scalar values and color map are used to determine the color of an atom using linear interpolation.
- **Create Scalar Histogram:** Creates a histogram file, "histogram.csv" for scalar values of the atoms. This file can be quite useful while creating the color map.


Figure B.9: The Rendering Parameters Menu - Volume Parameters

#### B.6.3.4 XRAY Rendering

Figure B.10 displays the XRAY parameters in the rendering parameters menu. The options presented are as follows:

- **XRAY Alpha Length Constant:** Represents the opacity constant of the volume. Increasing this constant makes the volume more opaque.
- **XRAY Base Color:** Represents the base color that the XRAY Renderer uses.
- **Auto Apply:** Enables to update rendering parameters upon any value change. This trigger re-rendering of the dataset. Because rendering is time consuming, It is recommended that this option is disabled.
- **Apply:** Applies the rendering parameters and triggers re-rendering.

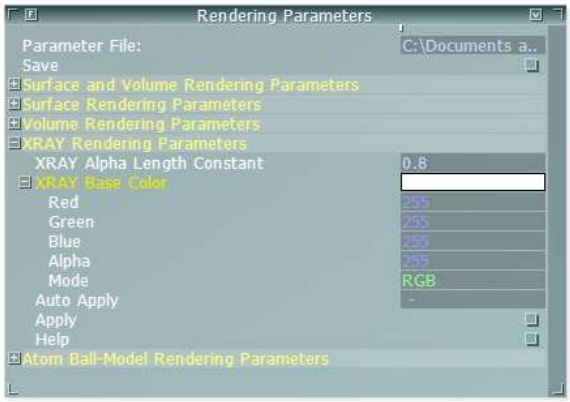



Figure B.10: The Rendering Parameters Menu - XRAY Parameters

#### B.6.3.5 Atom-Ball Model Rendering

Figure B.11 displays the atom-ball model parameters in the rendering parameters menu. The options presented are as follows:

- **Atom-ball Scale:** Represents the scaling factor for drawing an atom-ball model. Value 1.0 indicates drawing each atom with its actual radius.
- **Regular Atom Opacity:** Represents the opacity coefficient for atoms without any errors in their neighborhood. This way regular parts of the crystal can be made semi-transparent revealing the erroneous parts.
- **Atom-ball Colors:** Sets the color for each atom type.
- **Atom-ball Visibilities:** Shows or hides each atom type individually.
- **Auto Apply:** Enables to update rendering parameters upon any value change. Rendering in this mode is fast, so auto applying is recommended.
- **Apply:** Applies the rendering parameters and triggers re-rendering.

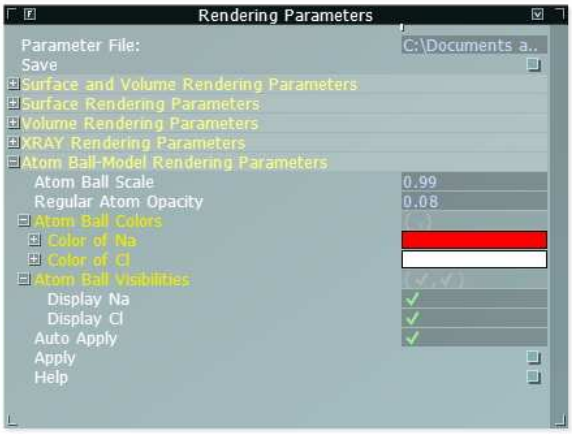



Figure B.11: The Rendering Parameters Menu - Atom-Ball Model Parameters

#### B.6.3.6 Help

*MaterialVis* contains an extensive embedded help describing the parameters the user can modify. The user can access the help in two ways. The help icon at the bottom left of the screen opens the top-level help menu. There are also several help buttons at the configuration menus. Figure B.12 displays a small part of the help menu.

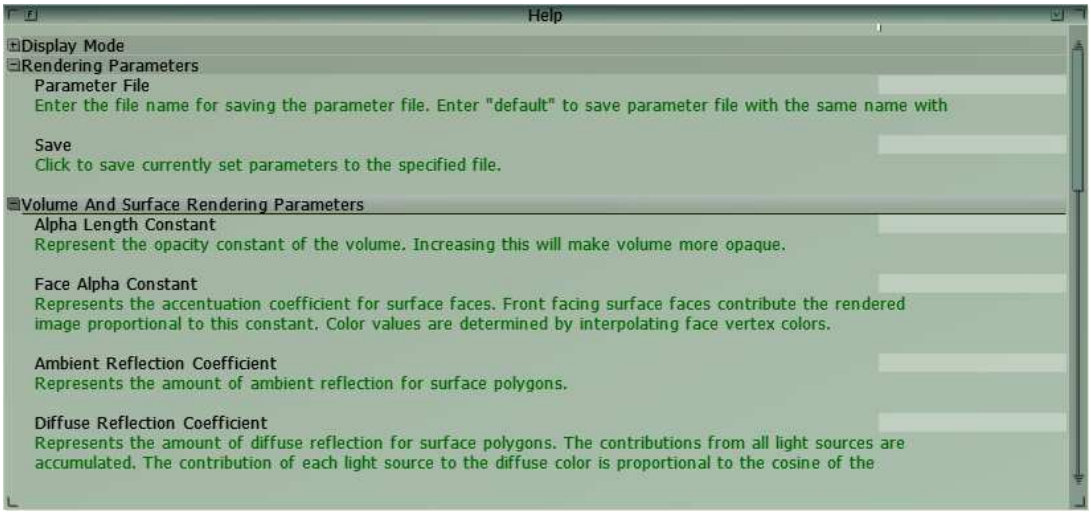



Figure B.12: The Help Menu