REAL-TIME VISUALIZATION OF 3D MEDICAL SCAN DATA

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Konstantin Igor Shkurko

August 2010

ABSTRACT

The immense progress of imaging technologies has radically changed the practice of medicine both in terms of diagnosis and intravascular surgery. Using technologies such as magnetic resonance imaging (MRI), and computed axial tomography (CAT) scans, doctors are now able to "see" internal organs and structures in high-resolution detail. Today using expensive specialized hardware, one can generate three-dimensional visualizations providing accurate interpretations and revolutionizing the medical field.

This thesis presents a substantially different method to visualize volume datasets by treating them as a scattering volume and rendering the images on a small cluster of parallel computers. With sufficient computing power, the data can be explored interactively without any loss of information.

We utilize a basic raycasting algorithm with several acceleration techniques, such as global empty space skipping, early ray termination, a local gradient cache and increased data access coherency. By selecting efficient data subdivisions, we eliminate the memory and bus-bandwidth latencies and maximize the computing power of each core. The cache coherence of the data access due to the bricking scheme produced almost real-time rendering speeds that are independent of the viewing direction. We tested these algorithms on three different datasets at varying output image resolutions.

In the near future, with increased computing power and sufficient bandwidth, it will be possible to use a cluster of machines to render time-dependent datasets in real time and to deliver these images directly into an operating room.

BIOGRAPHICAL SKETCH

Konstantin Igor Shkurko was born in the small research town of Akademgorodok, near Novosibirsk, Russia. He spent his childhood mesmerized in his grandfather's chemistry lab, exploring personal computers and other tools of the hard sciences. Shortly after moving to the U.S., Konstantin focused on studying mathematics and physics in WWP High School North near Princeton, NJ. All the hard work paid off when Konstantin, starting as a transfer student, earned his B.A. in Mathematics and Physics from Cornell University in 2007. After discovering 3D Studio Max during his freshman year, Konstantin enrolled in Professor Steve Marschner's "Introduction to Computer Graphics" with the hope of understanding what all the buttons meant. To his surprise, he learned more than enough to understand the software and found his passion for Computer Graphics. This motivated Konstantin to fit graphics classes into his schedule beside a full mathematics and physics workload. After graduating, Konstantin joined the Cornell Program of Computer Graphics to earn his Master's degree outlined in this thesis. A decade ago, little did a young Siberian boy know that his hobby of creating colorful animations with QBasic would culminate with this work.

Dedicated to my family

ACKNOWLEDGEMENTS

Thank you, Don, for this tremendous opportunity, your support and wisdom. Your drive and your vision are truly inspirational. I am grateful for the wonderful experience and the chance to contribute to the Program of Computer Graphics.

Thank you, Alex, KB, Steve, and Doug for your support and advice over the years. Without it, I would not have started pursuing computer graphics and applied mathematics.

Thank you, Edgar and Brendan, for being good friends and tolerating all the late night discussions about hardware, coding, animation and trips to CTB for triple-shot espressos.

Thank you, Linda, Hurf, Peggy, and Lars, for your administrative support and copious help with everything that needed to get done on the daily basis.

Thank you, Todd, David, and Cari, for being great office mates and offering your support in grading, teaching and flushing out Probe.

Thank you, Adam, Miloš, and Jaroslav, for your graduate school advice, research discussions and awesome lunches.

Thank you, Bruce, for your software engineering expertise, openness and availability.

Thank you, Peter and Matt, for joining me in random technobabble.

Thank you, Roy and Vikash, for providing the volume datasets and the help to get started with this project.

Thank you, Paris, for your immense emotional support, your patience and day-to-day excitement. It would have been a much longer journey without your help.

Thank you, Sarah, Alonso, Tom, Kalpana, Peter, Michael, Carolyn, Willa, and everyone else I missed, for being my closest friends, keeping me out of trouble and being willing to spend an hour of your time at 3 AM to help me with my work.

Finally, I would like to thank my parents and my sister for their love, support and encouragement these past few years.

This research was funded by the Program of Computer Graphics, the Department of Architecture, the Department of Computer Science, and the National Science Foundation ITR / AP: CCF-0205438.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The immense progress and development of imaging technologies has radically changed the practice of medicine both in terms of diagnosis and intravascular surgery. Using technologies such as X-rays, positron emission tomography (PET) scans, magnetic resonance imaging (MRI), and computed axial tomography (CAT) scans, doctors are now able to "see" internal organs and structures in high-resolution detail. The visualization of this data has revolutionized the medical field.

In the past, standard methods for imaging technology involved looking at two-dimensional images with skilled experts, usually radiologists, interpreting the results. However today volumetric imaging software can generate three-dimensional visualizations providing much more accurate interpretations. Powerful software and hardware implementations have evolved so that several commercially available workstations allow the physician to interactively navigate and explore the data. Algorithms using segmentation techniques, which create surface models from three-dimensional datasets, are frequently used but this requires costly pre-computation time and is usually conducted within a physician's office.

This thesis proposes a substantially different method to visualize datasets by treating them as a scattering volume without the creation of additional artifacts. With sufficient computing power, the data can be explored interactively without any loss of information. As we enter into a new realm of computer environments with "cloud computing" and parallel architectures, it will now be possible to bring these visual results directly into the operating room provid-

ing yet another beneficial tool for the intravascular surgeons. Without the use of any specialized hardware, we have attempted to provide real-time, three-dimensional visualization of the internal organs of a patient using a small cluster of parallel machines. Although with the size of our cluster we cannot quite achieve real-time visualization (30 frames per second, fps), we can effectively and efficiently render complex datasets at ten frames per second. With the advances in hardware technology, we anticipate that providing this capability combined with image-processing techniques that can position X-ray or fluoroscopic images obtained during surgery, we can provide better visualization tools in an economic manner.

The rest of this chapter is organized in the following way. We state the precise problem this thesis addresses in Section 1.1 and describe the data parameters in Section 1.2. We define several important terms in Section 1.3 and outline the organization for the rest of this work in Section 1.4.

1.1 Problem Statement

As scanned medical datasets are growing in size and being used more frequently, visualization techniques that are accurate and interactive are becoming essential. This work focuses on creating and evaluating a system capable of real-time visualization of three-dimensional (3D) medical scan data. A typical output of our system is shown in Fig. 1.1. We require that each high resolution output image is generated fast enough to allow real-time viewing (refresh rate of at least 30 Hertz); that the algorithm must not degrade output image quality and that all relevant data is shown. Because our interest is in the medical field,

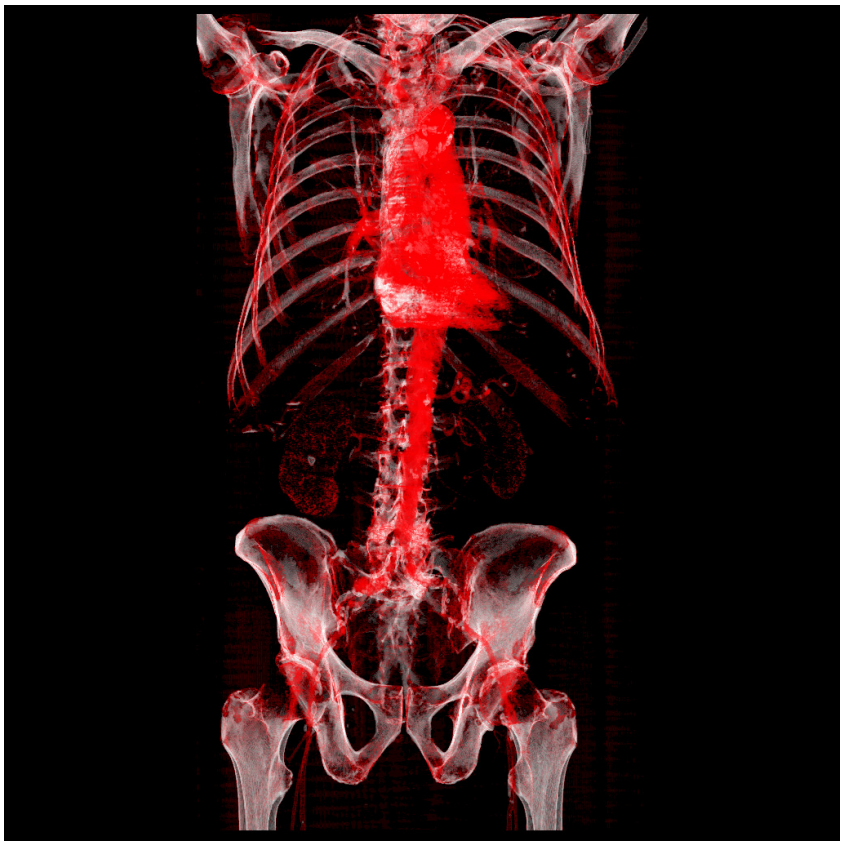


Figure 1.1: An example of a volume rendering output.

we focus on visualizing computed tomography (CT) datasets, introduced in the following section.

1.2 Data Description

The CT volume datasets store a density representation, which is measured in Hounsfield units (HU). These units place air at -1000 HU, water at 0 HU and bone above 400 HU. The data is quantized for storage as 12-bit integers, which may or may not be signed. Signed data has the range of $[-1024, 3071]$ and unsigned has the range of $[0, 4095]$, [NEM08]. Other quantizations are applicable but are not used in this work.

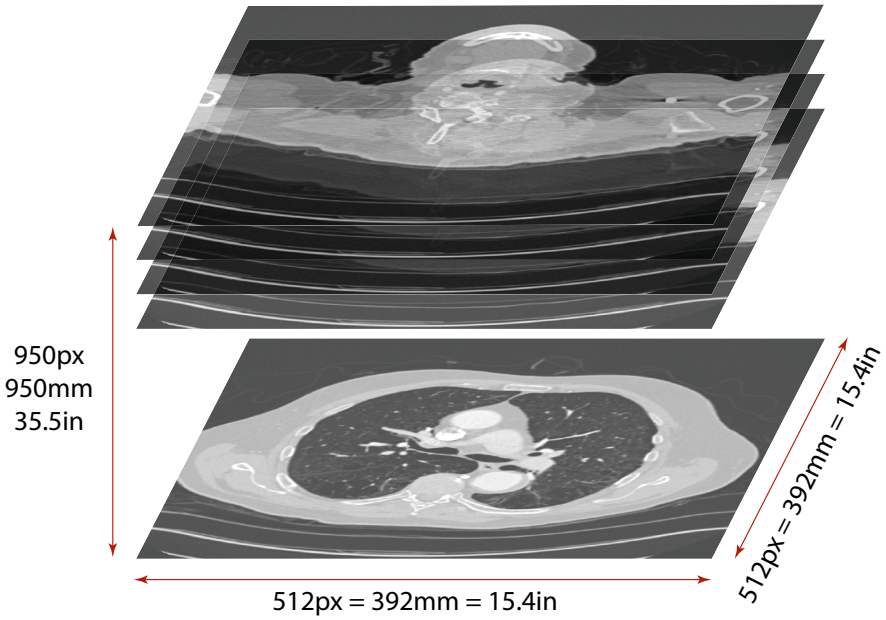
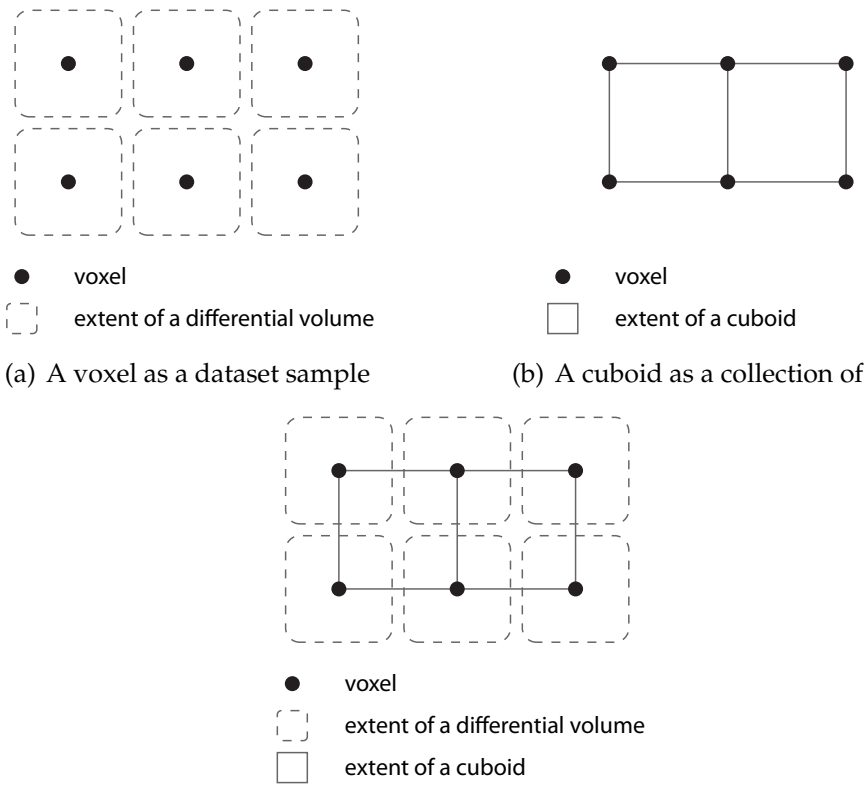


Figure 1.2: CT volume dataset as a stack of images.

The dataset can be thought of as a stack of images, shown in Fig. 1.2. Typical resolutions include several hundred images of 512×512, although they can exceed $1,000$ slices at 1024×1024. The typical distance between each data sample in a slice with the resolution of 512×512 is 0.075 cm (0.0295 in) and 0.077 cm (0.0303 in) between slices. As the scanning technology improves, datasets with higher resolutions will need to be rendered.

1.3 Voxel and Cuboid Definitions

We need to consider the CT dataset as a collection of data samples, where each sample represents a density value of a differential volume of material. Each data value is assumed to lie at the center of its differential volume. We refer to these dataset samples as voxels, which can be thought of as volumetric pixels. Pictorially, this is shown in two dimensions (2D) in Fig. 1.3(a).



(a) A voxel as a dataset sample

(b) A cuboid as a collection of voxels

(c) Comparison of a voxel volume and a cuboid

Figure 1.3: A definition of a voxel and a cuboid in 2D for a small volume.

Unfortunately, differential volumes around voxels do not help with defining interpolating functions (discussed in Section 2.3). As a result, we need a different construction - a cuboid. Like a cube, which generalizes a square to three dimensions, a cuboid generalizes a rectangle. In order to see this construct, we need to think about the dataset as a collection of voxels organized in a 3D grid. Connecting eight nearest voxels surrounding a point within the 3D grid produces the cuboid, shown in Fig. 1.3(b)[1]. Because we consider data samples within a volume, we refer to the cuboid as eight neighboring voxels. Fig. 1.3(c) compares the physical extends of a voxel volume to a cuboid in 2D.

These definitions formalize the discussions of algorithms in Chapter 2. We

[1]Note that some of the literature uses voxel to refer to what we defined as a cuboid.

use voxel to mean dataset samples and cuboid to describe a construct of eight neighboring voxels connected together.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 summarises the body of previous work. We discuss the modules shared by all of the algorithms before providing their overview and selecting raycasting as the basis for our work. Chapter 3 presents acceleration methods for raycasting and then improvement results. We focus Chapter 4 on parallelizing the accelerated raycasting algorithm. We discuss several techniques and evaluate their performance. The discussion of the overall results is in Chapter 5. We conclude and present directions for future work in Chapter 6.

CHAPTER 2

VOLUME RENDERING ALGORITHMS

There has been a large volume of work towards visualizing scalar volume data since the 1980s. A scalar physical quantity, like density, is sampled to represent the object to be studied. Earlier algorithms visualize the data by approximating surfaces of a constant value, called iso-surfaces. Marching Cubes, [LC87], is a well known method of this type. It uses a mesh of triangles located inside cuboids. First, cuboid vertices are thresholded to determine those which lie outside the iso-surface. A voxel is considered outside (inside) if its density is higher (lower) than the density value of the iso-surface. If one vertex is outside and another inside, then the edge connecting them intersects the iso-surface and contains a vertex of a triangle approximating the iso-surface. There are $2^8 = 256$ possible triangle configurations, which Lorensen and Cline reduced to 14 by accounting for symmetries. After classifying the cuboid to hold one of these cases, the algorithm uses linear interpolation to compute triangle vertex locations along cuboid edges. Generating the triangle mesh in Marching Cubes can be parallelized by exploiting the independence of data. Once the surface has been generated, a modern graphics board can produce the output images quickly. However, there is a major drawback to this method, when the user decides to view an iso-surface of a different value. Each time the value changes its iso-surface must be regenerated before being displayed, which can take a large amount of time. Another drawback of the original method is low accuracy of the iso-surface approximation via the triangle mesh with vertex locations estimated by linear interpolation along the cuboid's edges. Because the scalar data is used to generate triangle meshes to be displayed, the Marching Cubes algorithm belongs to a class of indirect volume rendering methods [MHB+00].

On the other hand, direct volume renderers would use the scalar data directly to compute the exact intersection of an iso-surface with each viewing ray instead of rasterizing triangle meshes generated from the data. Levoy introduced this set of methods by formulating the direct volume rendering pipeline[1] for iso-surface generation in [Lev88]. Without changing the pipeline, Levoy's framework can be extended to create a completely different way to visualize 3D scalar data. Instead of rasterizing[2] surfaces of constant value, these methods integrate opacity-weighted colors derived from data along a viewing ray through the volume. Because direct volume renderers display the data directly without modifying it, they are preferred by medical professionals, [NT01].

Volume visualization algorithms can be classified in a manner shown in Fig. 2.1. A natural classification comes from which projective transformation is used: either orthographic or perspective. All image rays in an orthographic projection have a constant (vector) direction equal to the normalized viewing direction, which allows for a few computational simplifications. However, because all viewing rays are parallel, the output image lacks perspective fidelity. As a result, one can measure geometric distances directly, but may have trouble correctly judging spatial relationships between the displayed objects.

Perspective projection methods use rays emanating from a virtual camera (eye) through the image plane to create very realistic images. As a result of diverging viewing rays, methods using this projection can be prone to errors from undersampling data far away from the eye. Because each ray has its own direction, this projection is computationally more expensive. Furthermore, the non-

[1]We consider the volume rendering pipeline as a sequence of computing steps that takes a volume dataset as input and produces an image.

[2]Triangle rasterization is a process of converting (projecting) a triangle in three-space onto the output image pixels.

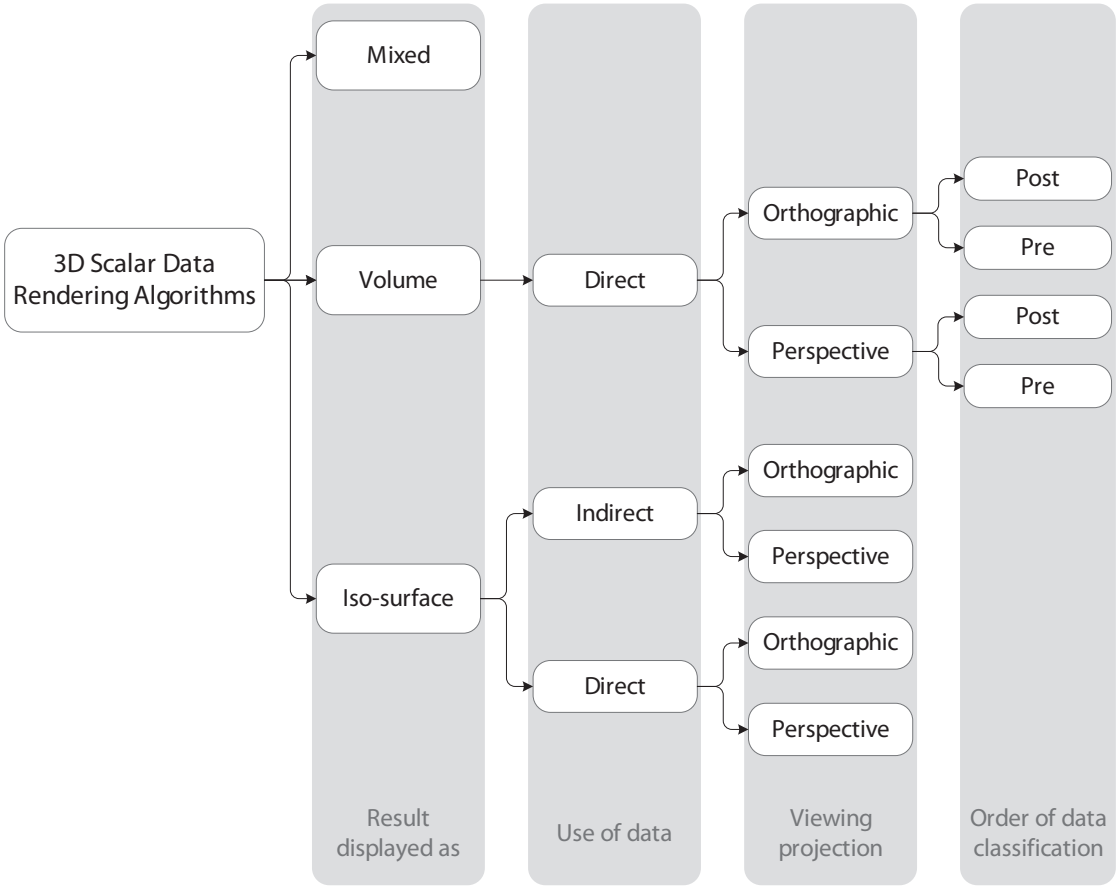


Figure 2.1: Classification of volume rendering algorithms.

linearity of the depth makes distance computations between objects non-trivial. An important characteristic to note is that orthographic transformation is mathematically equivalent to the perspective transformation with the eye located at infinity.

All volume rendering algorithms produce a visualization of the data, given a user-defined transfer function which assigns colors and opacities to data samples. These samples are usually located along viewing rays cast into the volume, but the data may be sampled in a variety of other ways.

Volume renderers can be divided further by the application order of data classification and interpolation. Classification of a data sample is the process of

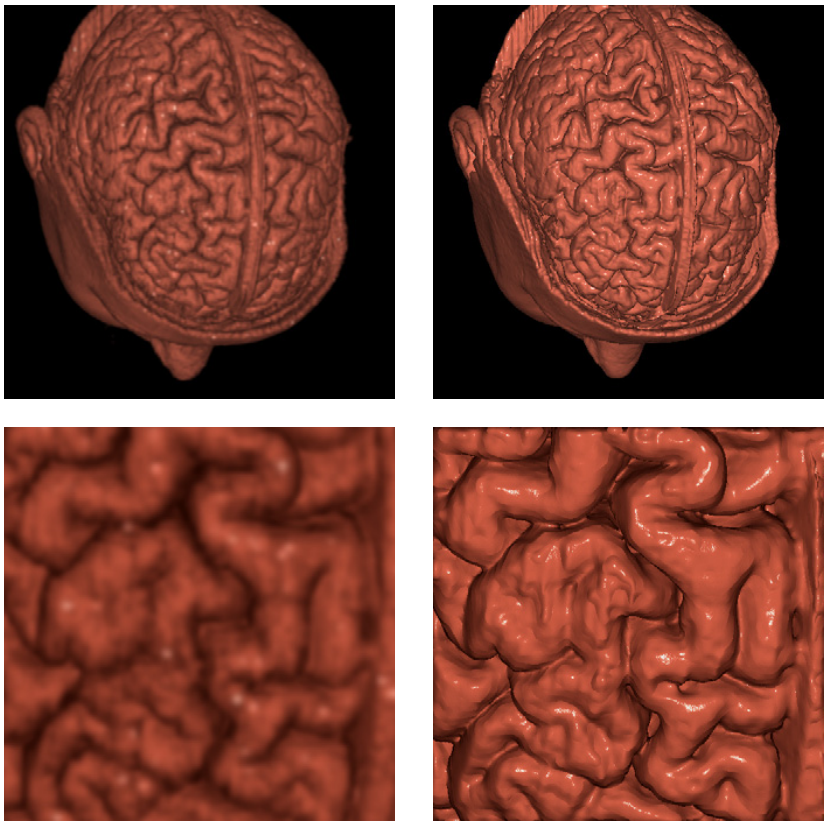


Figure 2.2: Pre-classified (left column) and post-classified rendering (right column). The latter yields sharper images since the opacity and color classification is performed after interpolation. This method eliminates the blurry edges introduced by the interpolation filter, [KM05].

assigning to it a material based on the data value. Material properties, like color and opacity, are stored in the transfer function. Pre-classification methods first use the transfer function to generate colors and opacities for each voxel near the sample location. Then, these colors and opacities are interpolated to generate the final sample color and opacity [MHB+00]. If, on the other hand, the data is interpolated to approximate the value of the sample and then the transfer function is used to classify it, the method is considered post-classifying. Post-classification tends to produce sharper images as shown in Fig. 2.2.

In addition to interpolation and classification of data, all volume rendering algorithms share two additional modules necessary to produce the final image: shading and compositing. Shading a sample produces a color based on its classification and, if desired, approximating the shape of the iso-surface the sample lies on by adding a specular highlight that depends on the surface curvature. Compositing combines multiple samples together while incorporating their occlusion due to the accumulated opacity along the viewing ray.

To discuss the differences between various algorithms, we must elaborate on the shared modules between them. We start with the model for the interaction of light with the volumetric data, which is considered to be a collection of small particles with specific physical properties of light transport. The remainder of this chapter is organized as follows. We derive the commonly used low-albedo rendering integral via the Emission-Absorption Illumination Model in Section 2.1. The transfer function is described in detail in Section 2.2. Section 2.3 focuses on the interpolation kernels, and Section 2.4 discusses the details of the most prominent direct volume rendering algorithms. We conclude in Section 2.5.

2.1 Emission-Absorption Illumination Model

A model for the interaction of light with volume data needs to be established before being able to generate images. We start with the physical description of the model, derive the general rendering equation and then formulate the widely-used low-albedo rendering integral. This was introduced in [Bli82, KH84] and formally derived in [Max95]. Max's derivation is briefly described below.

The optical behavior of the volumetric data can be modeled as a cloud of

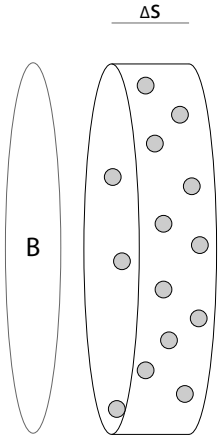


Figure 2.3: A cylindrical slab of volumetric material used to derive the Emission-Absorption Illumination Model. Each small circle represents a perfectly spherical volumetric particle that absorbs and emits light. Here, B is the surface area of the base and $\triangle s$ is the thickness of the cylinder. Figure modified from [Max95].

small particles that are homogeneously distributed within a small volume and can absorb, scatter and emit light. For the sake of simplicity, we consider a medium with negligible contribution from multiple (particle-particle) scattering. In other words, it is composed of low-albedo particles which have low reflectivity. One physical interpretation of this situation is that the particles are very small resulting in a little chance of being hit by light. An alternative interpretation is that the particles are highly absorbing, and hence have little chance to reflect light.

For the sake of simplicity, assume that these particles are identical perfect spheres of radius r and projected area $A = \pi r^2$. Let ρ be the volume density of the particles in a cylinder of volume $B\triangle s$, shown in Fig. 2.3. This cylinder contains $N = \rho B\triangle s$ number of particles.

Using this physical model, we derive the simple absorption-only case, which we later extend to include emission. For the discussion on including multiple

scattering into the model, see [Max95, Bli82]. If we assume $\triangle s$, the width of the cylinder, is small allowing few overlaps[3] between projected volumes of particles, then the total occlusion area due to the particles becomes $NA = \rho B \triangle s A$. In the perfect absorption-only case, each particle absorbs all of the light that hits it. The fraction of light flowing through the base of the cylinder that is occluded can be derived as ${}^{\rho B \triangle s A}/_{B} = \rho \triangle s A$. In the limit of the cylinder's thickness approaching 0, $\triangle s \to 0$, we get the following differential equation for the light intensity along a ray:

$$\frac{dI}{ds} = -\tau(s) I(s), \tag{2.1}$$

where s is the location along a ray in the direction of light and $\tau(s) = \rho(s)A$ is the extinction coefficient defining the rate of occlusion of light. The negative sign is necessary because, as the light travels deeper into the participating media, a larger portion of it is absorbed by the particles.

Let's consider a completely opposite scenario where particles are perfectly emissive. In this case, the model is analogous to a very hot and almost transparent gas that glows. Once again, consider the situation shown in Fig. 2.3 where a cylinder of volume $B \triangle s$ contains $\rho B \triangle s A$ number of perfectly spherical particles. Let these particles glow with intensity C per unit projected area, producing a glow flux per unit area of $C \rho \triangle s A$ through the base of the cylinder, [Max95]. Taking a limit of the cylinder's width, $\triangle s \to 0$, yields the following differential equation of light intensity for the emission-only case:

$$\frac{dI}{ds} = C(s)\rho(s)A = C(s)\tau(s) = g(s), \tag{2.2}$$

where $g(s)$ is the source term that accounts for emission.

[3]The low-albedo assumption implies that the homogeneously distributed absorbing particles are tiny (or that there are few of them in the volume), thus the likelihood that the projected volumes of particles overlap is small.

Combining the differential equations describing absorption in Equation (2.2) and emission in Equation (2.2), we arrive at

$$\frac{dI}{ds} = g(s) - \tau(s)I(s). \tag{2.3}$$

This differential equation has the following solution, derived in [Max95]:

$$I(L) = I_0 T(0, L) + \int_0^L g(s)T(s, L)\, ds \qquad T(s_1, s_2) = \exp\left(-\int_{s_1}^{s_2} \tau(t)\, dt\right), \tag{2.4}$$

where I_0 is the light intensity along the ray prior to entering the medium and $T(s_1, s_2)$ calculates the light attenuation between locations s_1 and s_2 along a ray. This formulation is in back-to-front order along the viewing ray, so the far edge of the volume is located at $s = 0$, while $s = L$ is a location within the volume or the point where the ray exits near the eye. Equation (2.4) is the general rendering equation. In practice, most algorithms use a slightly different form where the source term defined in Equation (2.2) is kept as $g(s) = C(s)\tau(s)$ with $C(s)$ as the intensity (color) of the sample. This form of the rendering equation accounts for the higher reflectivity of particles with larger volume densities, [MHB+00]. The end result of this manipulation is the widely-used low-albedo formulation:

$$I(L) = I_0 T(0, L) + \int_0^L C(s)\tau(s)T(s, L)\, ds \qquad T(s_1, s_2) = \exp\left(-\int_{s_1}^{s_2} \tau(t)\, dt\right). \tag{2.5}$$

In order to compute the light intensity $I(L)$, we need to discretize Equation (2.5) converting the integrals into Riemann sums. By dividing the interval from 0 to L into n segments of equal length, we define $\triangle s = {}^{L}/_{n}$. For the sake of simplicity, let $s_i = i\triangle s$, $i = 1 \ldots n$, be a sample corresponding to i$^{\text{th}}$ segment of the viewing ray. To discretize the transparency function $T(s_k, s_m)$ defined in Equation (2.5), let $s_k = k\triangle s$ and $s_m = m\triangle s$ and assume that the extinction coefficient

$\tau(i\triangle s)$ is constant along each segment s_i. Hence

$$\begin{aligned} T(s_k, s_m) = \exp\left(- \int_{s_k}^{s_m} \tau(s)\, ds \right) &= \exp\left(- \sum_{i=k}^{m} \tau(i\triangle s)\, \triangle s \right) \\ &= \prod_{i=k}^{m} \exp\Big(- \tau(i\triangle s)\, \triangle s \Big) = \prod_{i=k}^{m} t_i. \end{aligned} \tag{2.6}$$

To simplify further, use the Taylor series of the exponential function:

$$t_i = \exp\Big(- \tau(i\triangle s)\triangle s \Big) = 1 - \tau(i\triangle s)\triangle s + \frac{\Big(\tau(i\triangle s)\triangle s\Big)^2}{2} - O(\triangle s^3), \tag{2.7}$$

where the term t_i defines the transparency of the i$^{\text{th}}$ segment and is opposite to the opacity

$$\alpha_i = \alpha(i\triangle s) = 1 - t(i\triangle s) = 1 - t_i = \tau(i\triangle s)\triangle s + O(\triangle s^2). \tag{2.8}$$

As shown in Section 2.4, volume rendering algorithms use opacities instead of transparencies along ray segments. Converting Equation (2.6) to use opacities gives:

$$T(s_k, s_m) = \prod_{i=k}^{m} \exp\Big(- \tau(i\triangle s)\, \triangle s \Big) = \prod_{i=k}^{m} t_i = \prod_{i=k}^{m} (1 - \alpha_i). \tag{2.9}$$

Substituting the above into Equation (2.5), and using $n = {}^{L}/_{\triangle s}$:

$$\begin{aligned} I(L) &= I_0 T(0, L) + \int_0^L C(s)\tau(s)T(s, L)\, ds \\ &\approx I_0 \prod_{i=1}^{n} (1 - \alpha(i\triangle s)) + \sum_{i=1}^{n} C(i\triangle s)\alpha(i\triangle s) \cdot \prod_{j=i+1}^{n} (1 - \alpha(j\triangle s)) \qquad (2.10) \\ &= I_0 \prod_{i=1}^{n} (1 - \alpha_i) + \sum_{i=1}^{n} C_i \alpha_i \prod_{j=i+1}^{n} (1 - \alpha_j). \qquad (2.11) \end{aligned}$$

The formula for the light intensity, Equation (2.11), is recursive in $(1 - \alpha)$ and leads to the famous recursive front-to-back compositing formula, [PD84, Lev88]:

$$\begin{aligned} c_{new} &= C(i\triangle s)\alpha(i\triangle s)(1 - \alpha_{old}) + c_{old} \\ \alpha_{new} &= \alpha(i\triangle s)(1 - \alpha_{old}) + \alpha_{old}, \end{aligned} \tag{2.12}$$

```
input  : Opacities α_i=a[i] and emission colors C_i=C[i] for samples along
         the ray. Background light intensity I_0=Io
output: Composited output color for ray
1  rayColor ← 0;
2  T ← 1.0 ;                          // accumulated ray transparency
3  i ← n;
4  while  T > small_threshold && i > 0 do
5      rayColor ← rayColor + T * C[i] * a[i];
6      T ← T * (1 - a[i]);
7      i ← i  - 1;
8  end
9  rayColor ← rayColor + Io * T;
10 return resulting pixel color rayColor;
```

Figure 2.4: Algorithm for compositing a ray in the front-to-back order, resulting from Equation (2.13), [Max95]. For a ray shot from the eye into the volume, the n^{th} sample occurs at the entrance closest to the eye and the first sample occurs at the exit. The *small_threshold* parameter on line 4 stops sample compositing early if the contribution from the upcoming samples becomes negligible.

where c_{new} and α_{new} are the resulting color and opacity, while c_{old} and α_{old} are the previously composited color and opacity. $C(i\triangle s)$ and $\alpha(i\triangle s)$ are the color and opacity of the sample located at $s_i = i\triangle s$. If we unroll Equation (2.11) and use $1 - \alpha_i = t_i$, we can derive an alternative formula to compute the output color:

$$\begin{aligned} I(L) &= I_0 \prod_{i=1}^{n}(1-\alpha_i) + \sum_{i=1}^{n} C_i\alpha_i \prod_{j=i+1}^{n}(1-\alpha_j) \\ &= C_n\alpha_n + t_n\Big(C_{n-1}\alpha_{n-1} + t_{n-1}(\ldots(C_1\alpha_1 + t_1 I_0)\ldots)\Big). \end{aligned} \tag{2.13}$$

This leads to a front-to-back compositing algorithm, shown in Fig. 2.4.

Both of the compositing algorithms require colors and opacities of samples located at s_i, for $i = 1 \ldots n$. These quantities are obtained by applying a user-defined transfer function to each of the samples. The next section discusses transfer functions.

2.2 Sample Color and Opacity via a Transfer Function

In the previous section, we have established the low-albedo model for volume-light interaction. We also introduced two schemes for compositing samples along a ray within the volume. However, the volume data represents a scalar physical quantity like density and does not automatically provide a mapping to colors and opacities. As a result, volume renderers rely on a user to provide a transfer function for this necessary correspondence.

There are two ways to define a transfer function precisely depending on whether the data is mapped to colors and opacities separately. Noting that both data and opacity are scalars, while color is a three-vector, one can define a combined transfer function as: $\Psi_{co} : \mathbb{R} \rightarrow \mathbb{R}^4$. The secondary definition, used in this work for simplicity, splits the transfer function into two components: color, $\Psi_c : \mathbb{R} \rightarrow \mathbb{R}^3$, and opacity, $\Psi_o : \mathbb{R} \rightarrow \mathbb{R}$. One can also include a gradient (three-vector) as an input into the transfer function in a manner similar to the direct volume renderer Levoy introduced in [Lev88] and extended in [Lev90].

Transfer functions Ψ have several properties that should be mentioned. Clearly, their domain depends on the input data. For example, the density data from a Computed Tomography (CT) scan lies in the range of $[0, 4095]$ or $[-1024, 3071]$, [NEM08]. The range of Ψ_c ultimately depends on the output display device. A typical monitor can display 24-bit color values at eight bits per color channel, thus limiting the range of the color transfer function Ψ_c to $[0, 255]^3$. It is possible to allow Ψ_c to produce floating point values; however, with the current displays constrained to eight bits per color channel, there is no way to display this directly without applying a windowing transformation,

which remaps the dynamic range into a displayable color. This is a limitation of the current technology that will eventually be removed. The opacity transfer function Ψ_o has a floating point range of $[0, 1]$ with zero representing full transparency and one representing full occlusion of the sample.

An important property of transfer functions is that they map every piece of input data to a color and opacity. Transfer functions do not have to require that for every piece of input data there is only one color and opacity. As a result, one piece of data can map only to one color and opacity; otherwise, the algorithm has to pick the correct classification from multiple possibilities.

There are several ambiguities in the formulation of the rendering integral in Equation (2.5). The first is that the sample color can be defined in several ways depending on the desired application. For example, the color can include light reflections to add a sense of shape of the iso-surface the sample lies on. Thus the color term $C(s)$ can be broken apart into $C(s) = E(s) + T(s) + R(s)$, where $E(s)$ is the emission, $T(s)$ is the transmission and $R(s)$ is the reflection of light at a sample location s.

The emission $E(s)$ can be considered to be of the same color as the material the sample is classified as, $E(s) = \Psi_c(s)$. In the case that the iso-surface is semi-transparent, $T(s)$ can be computed by compositing the contributions of all samples located on this ray behind the surface. Because this is the same as continuing to integrate along the ray, this term can be omitted.

The Blinn-Phong bidirectional reflectance distribution function, [DBB06], can be used to compute the reflective part of $C(s)$:

$$R(s) = k_a C_a + k_d C_l \Psi_c(s) \Big(\mathbf{N}(s) \cdot \mathbf{L}(s) \Big) + k_s C_l \Big(\mathbf{N}(s) \cdot \mathbf{H}(s) \Big)^{n(s)}, \tag{2.14}$$

Variable	Description
k_a	ambient light coefficient
k_d	diffuse reflection coefficient
k_s	specular reflection coefficient
C_a	ambient color
C_l	color of the light source
$\Psi_c(s)$	reflective color of the material
$\mathbf{N}(s)$	normal vector of iso-surface at sample location
$\mathbf{L}(s)$	light direction vector
$\mathbf{H}(s)$	normalized half-vector between the light and the viewing vector
$n(s)$	specular Phong exponent

Table 2.1: Variables in the Blinn-Phong reflection model, [DBB06].

where s is the location of the sample. Table 2.1 defines the rest of the variables. To produce physically-correct lighting, the intensity of the light C_l should be attenuated by the transparency of the volumetric media along the light's path, $C_l'(s) = C_l T(l, s)$. This introduces self-shadowing effects, shown in Fig. 2.5. They might be desirable, [KM05], but come at a high computational price because for every sample, the algorithm must integrate opacities through the volume along the path between the sample location and the light. As a result, most algorithms omit this attenuation factor.

Another ambiguity in Equation (2.5) comes from the application of the transfer function to 3D locations given by s, the scalar location along a ray. Letting O be the origin of the viewing ray and $\mathbf{d}$ its direction, defines 3D location $\mathbf{s}(s) = O + s \cdot \mathbf{d}$. An implicit sampling function $S(s) = S(\mathbf{s}(s))$ maps a 3D location to a scalar data value, $S : \mathbb{R}^3 \rightarrow \mathbb{R}$. The relationship between emission E, opacity τ, color transfer function Ψ_c, and opacity transfer function Ψ_o can be stated precisely as:

$$\begin{aligned} E(s) &= \Psi_c\Big(S(O + s \cdot \mathbf{d})\Big) = \Psi_c(S(s)) \\ \tau(s) &= \Psi_0\Big(S(O + s \cdot \mathbf{d})\Big) = \Psi_o(S(s)). \end{aligned} \tag{2.15}$$

Rewriting Equation (2.5) using the above relationships and including the sur-

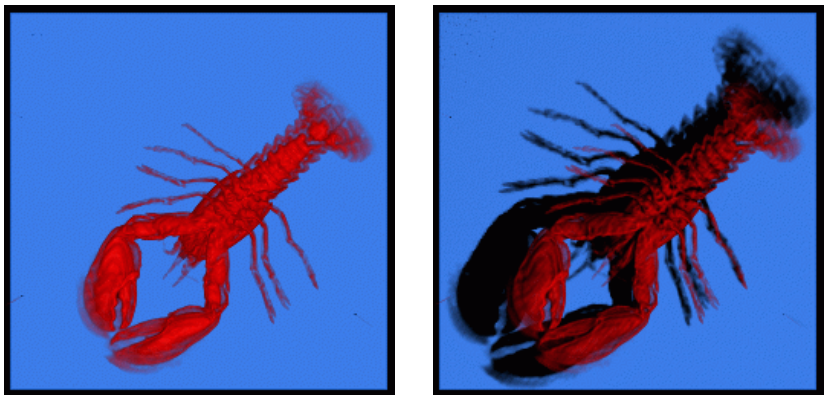


Figure 2.5: CT lobster dataset rendered without shadows (left) and with shadows (right). The shadows on the wall behind the lobster as well as self-shadowing of the legs creates greater realism, [KM05].

face shading decomposition of $C(s) = E(s) + R(s)$, we get

$$T(s_1, s_2) = \exp\left(-\int_{s_1}^{s_2} \Psi_o(S(t))\, dt\right)$$
$$I(L) = I_0 T(0, L) + \int_0^L \Big(\Psi_c(S(s)) + R(S(s))\Big)\, \Psi_o(S(s))\, T(s, L)\, ds. \tag{2.16}$$

Before describing various direct volume rendering approaches, sampling functions $S(s)$ have to be discussed to remove the last technical ambiguity. This is presented in the next section.

2.3 Interpolation Function

Volume rendering algorithms generate an intensity of a ray through the volumetric data by compositing samples along it. Because samples almost never overlap with any of the data locations, one needs a way to interpolate data values. Pre-classification algorithms use the interpolation function to combine colors and opacities of the data near the sample. Post-classification algorithms

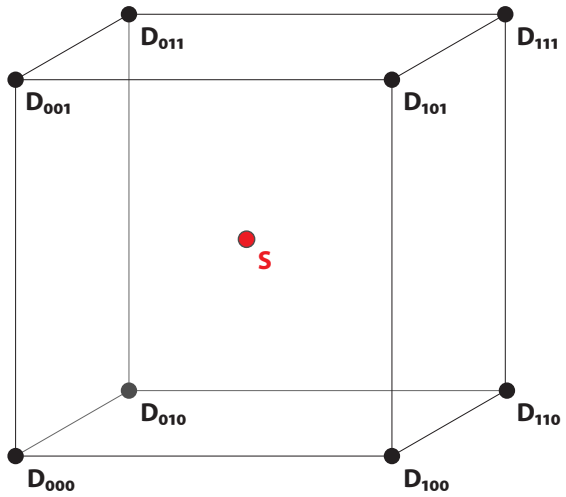


Figure 2.6: A cuboid formed by connecting eight nearest voxels surrounding the location of the sample S. The data naming convention $D_{i,j,k}$ is relative to that sample.

use the interpolation function to combine the data near the sample. The most important properties of interpolation functions are the amount of introduced smoothing and the speed of calculation. In this section, we consider a few interpolating kernels: nearest neighbor, bi-linear, tri-linear and Gaussian-weighted.

To simplify the discussion in this chapter, we need several definitions. Let the 3D location of a sample along a ray be $\mathbf{s}(s) = [\mathbf{s}_x, \mathbf{s}_y, \mathbf{s}_z]^T$. This is labeled as S in Fig. 2.6, which shows a cuboid formed by connecting eight voxels neighboring the location $\mathbf{s}(s)$. We use the following naming convention for these voxels: $D_{i,j,k} = D(i, j, k)$ with $\{i, j, k\} = \{0, 1\}$. Each subscript of $D_{i,j,k}$ refers to the corresponding dimension (x, y, or z respectively) and encodes the location of a voxel relative to the sample S. Using this notation, we can compute the dataset location for a voxel D_{101} as $(\lceil \mathbf{s}_x \rceil, \lfloor \mathbf{s}_y \rfloor, \lceil \mathbf{s}_z \rceil)$.

The nearest neighbor kernel is the easiest to formulate mathematically be-

cause it returns the value of the data that is closest to the sample location

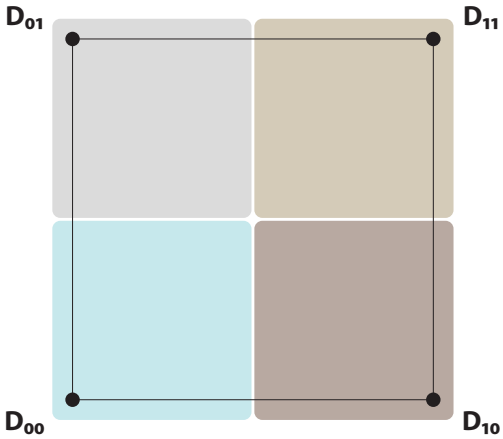
$$S_{nn}^{3D} = D(\,\text{round}(\mathbf{s}_x), \text{round}(\mathbf{s}_y), \text{round}(\mathbf{s}_z)), \tag{2.17}$$

where the function round(x) rounds x to the nearest integer. In 2D, the regions that interpolate to the same data are shown in Fig. 2.7(a). For data equally spaced with a distance of one in 3D, these regions become cubes of width one centered around the data values. The human eye is very sensitive to the jagged edges and unpleasant staircasing that result from a zero-order interpolation, and therefore nearest neighbor interpolation generally gives the poorest visual results, [KM05].

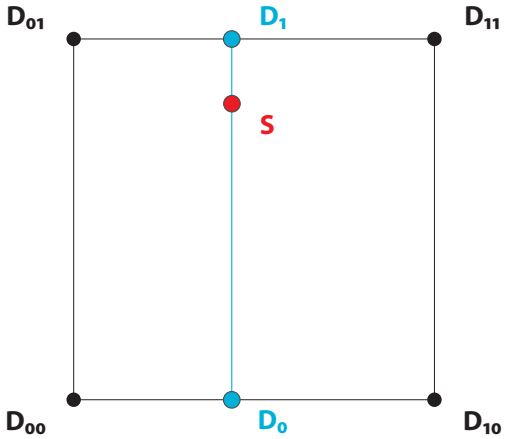
Linear interpolation is widely used because it gives good results with low computational cost and is easily extended to two (bi-linear interpolation) and three (tri-linear interpolation) dimensions. Consider a simple 1D case, where the sample is located at ε in between data points D_0 and D_1 located at x_0 and x_1 respectively. The linear interpolation yields

$$\begin{aligned} S_{lin}^{1D} &= D_0 + (\varepsilon - x_0)\frac{D_1 - D_0}{x_1 - x_0} \\ &= D_1(\varepsilon - x_0) - D_0(1 - (\varepsilon - x_0)), \text{ for } x_1 - x_0 = 1. \end{aligned} \tag{2.18}$$

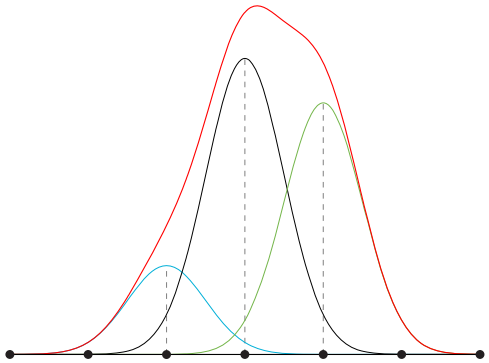
To extend this scheme to higher dimensions, the filter is consecutively applied in each dimension. The bi-linear interpolation can be computed by first interpolating in the x direction and then interpolating the results in the y direction as shown in Fig. 2.7(b). The tri-linear interpolation kernel is shown in Fig. 2.8. It produces reasonably sharp results at the cost of seven linear interpolations, which totals to 14 multiplications and 14 additions per sample for the data spaced one unit apart. A function interpolated with a linear filter no longer suffers from staircase artifacts. However, it has discontinuous derivatives at the



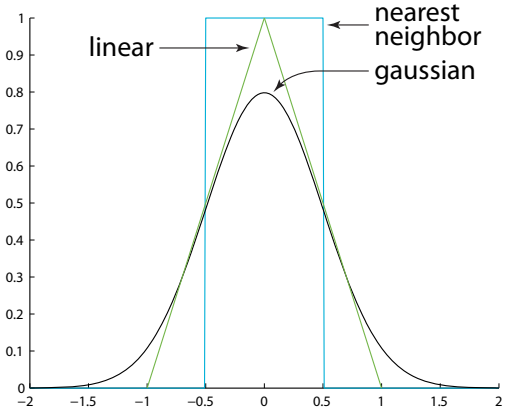
(a) Nearest neighbor interpolation with the areas of nearest data within 1×1 square



(b) Bi-linear interpolation kernel applied to 1×1 square of data



(c) Gaussian interpolation kernel with $\sigma = 0.5$ applied to data points in 1D



(d) Normalized filters in the spatial domain: nearest neighbor, linear, and Gaussian

Figure 2.7: Comparison between nearest neighbor, bi-linear and Gaussian interpolating kernels. (a)-(c) show application of filters in 2D, while (d) compares their profiles in 1D.

boundaries between voxel cuboids which can lead to noticeable banding when the visual qualities change rapidly from one cuboid to the next, [KM05].

Gaussian kernels are typically used to resample images because they act as a low-pass filter by removing high frequency noise and smoothing out edges. However, this kernel has infinite support and potentially all data can affect the interpolated value. To mediate this, the function is clamped at a certain radius,

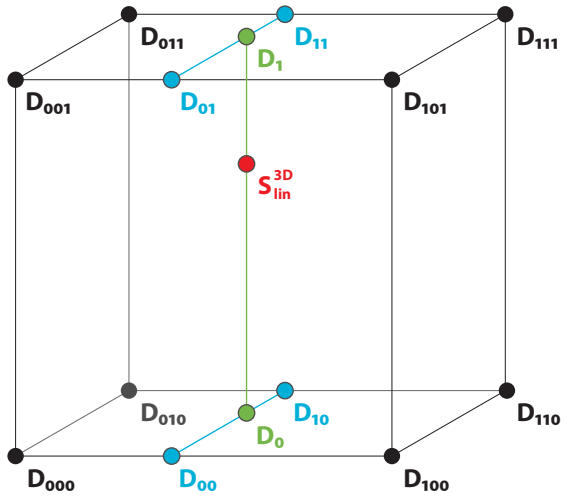


Figure 2.8: Tri-linear interpolation kernel applied to a cuboid of data. The kernel decomposes into seven linear interpolations, given by Equation (2.18). First, interpolate four pairs of data in the x direction to obtain D_{00}, D_{10}, D_{01}, and D_{11}. Then interpolate these two pairs in the y direction to obtain D_0 and D_1. Final interpolation in the z direction gives the final result S^{3D}_{lin}.

such that data beyond it does not contribute to the result. An un-clamped normalized Gaussian kernel centered at a and with a width of σ has the following form:

$$G^{1D}(\varepsilon; a) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\varepsilon - a)^2}{2\sigma^2}\right) \tag{2.19}$$

$$G^{3D}(\varepsilon; \mathbf{a}) = \frac{1}{(2\pi)^{\frac{3}{2}}\sigma_x\sigma_y\sigma_z} \exp\left(-\frac{(\varepsilon_x - \mathbf{a}_x)^2}{2\sigma_x^2} - \frac{(\varepsilon_y - \mathbf{a}_y)^2}{2\sigma_y^2} - \frac{(\varepsilon_z - \mathbf{a}_z)^2}{2\sigma_z^2}\right), \tag{2.20}$$

where the 3D form of the Gaussian, $G^{3D}(\varepsilon; \mathbf{a})$, is obtained by multiplying $G^{1D}(\varepsilon_{\{x,y,z\}}; \mathbf{a}_{\{x,y,z\}})$ for each major direction.

Consider a set representing data locations, $A_\varepsilon = \{\mathbf{a}_i\}$, $i = 1 \ldots n_\varepsilon$, such that it is the set of all data points within the support of the 3D Gaussian with the variances $\sigma_{\{x,y,z\}}$ centered at the sample location ε. Let D_i be the value of the voxel located at $\mathbf{a}_i \in A_\varepsilon$. Thus, the interpolated value located at ε is

$$S^{3D}_{gaus} = \sum_{i=1}^{n_\varepsilon} D_i G^{3D}(\varepsilon; \mathbf{a}_i). \tag{2.21}$$

Similar formalism generates the rule for Gaussian interpolation of data in 1D, which is shown in Fig. 2.7(c). One must carefully select contributing data for interpolation since more than eight nearest neighbors may be within the support of a Gaussian kernel. The low-pass filtering property of this kernel may not be desirable if the data contains necessary high frequency details which may be lost in the process.

There are many other interpolating functions that may be useful. It is important to consider their computational cost with other properties, like smoothing. This section concludes the discussion of the assumptions and computations necessary to understand the details of rendering algorithms discussed in the next section.

2.4 Algorithms

Over the years, there have been many proposed algorithms for volumetric visualization of 3D scalar datasets. Previous sections provided the necessary groundwork that most of the popular approaches use. In this section, we explore the work relevant to raycasting, shear-warp, splatting and hardware-based methods. For a good overview of these methods, see [MHB+00, MFS06, KM05].

As described in the beginning of this chapter, a typical volume rendering algorithm involves three essential steps: classification, interpolation and shading and compositing. The data is interpolated and classified in the order depending on whether pre- or post-classification is used. This results in a color and opacity at a sample, located on the viewing rays through the output image. These

Property	Raycasting	Shear-Warp	Splatting
Sampling Rate	freely selectable	fixed $[1.0, 0.58]$	freely selectable
Sample Evaluation	point sampled	point sampled	averaged across $\triangle s$
Interpolation Kernel	tri-linear	bi-linear	Gaussian
Data Classification	post	pre, opacity-weighted colors	post
Voxels Considered	all	relevant	relevant

Table 2.2: Distinguishing features of the selected direct volume rendering algorithms. Modified from [MHB+00].

samples get shaded and composited together to provide the final output color.

Pre-classification methods can be accelerated by pre-computing the colors and opacities of all voxels prior to rendering. Only the visible voxels with non-zero opacities must be stored. However, this produces a problem similar to Marching Cubes: the pre-computation has to be redone every time the transfer function changes. Another issue with pre-classification renderers is the tendency for excessive blurring when the output image resolution exceeds the resolution of the volume data. This occurs in zoomed viewing or with wide perspective frustums, [MHB+00]. Because of these drawbacks, most volume rendering algorithms use post-classification methodology.

Unfortunately, post-classifying the data has a problem as well. Due to the partial volume effect, interpolated data can be classified as a material not present at the sample location, which can lead to false colors in the final image. This can be avoided by prior segmentation, which can add severe staircasing artifacts due to the introduced high frequency, [MHB+00].

There are several different direct volume rendering algorithms discussed in this section: raycasting, shear-warp, splatting, special hardware and texture mapping. Table 2.2 shows the distinguishing features of several algorithms,

[MHB+00]. Meißner et al. [MHB+00] used several data sets to compare raycasting, shear-warp and splatting. Fig. 2.9 shows the rendered images for the skull dataset, which was obtained from rotational biplane X-ray scan with the resolution of 256^3. Fig. 2.10 shows the rendered images for the blood vessel in the head dataset, which was obtained from rotational biplane X-ray with the resolution of 256^3. Lastly, Fig. 2.11 shows the rendered images for the Marschner-Lobb dataset to display how these algorithms handle high frequency data. This was generated from [ML94] with the resolution of 41^3.

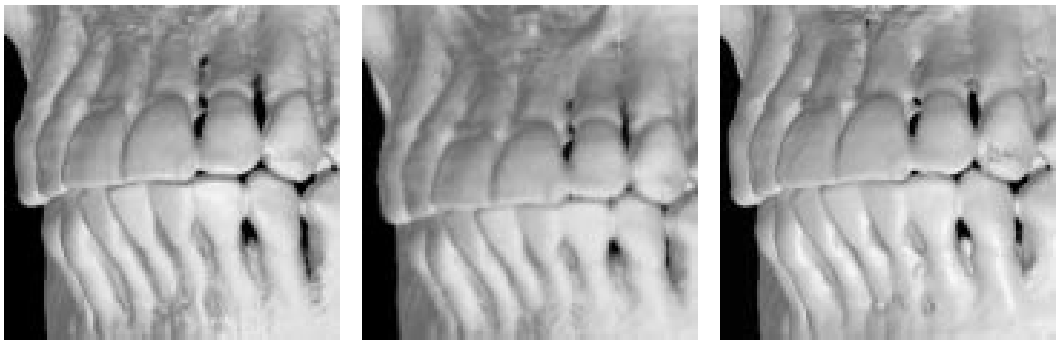
The remainder of this section is organized as follows. First in Section 2.4.1, we discuss the method of raycasting because it provides the simplest introduction to building a direct volume renderer. Section 2.4.2 discusses shear-warp method, which is considered to be the fastest software renderer to date. In Section 2.4.3, we present another fast and very popular direct volume renderer called splatting. The final section describes the methods based on special hardware or utilizing graphics boards.

2.4.1 Raycasting

Raycasting, introduced in [TT84] and [Lev88], provides a basis for the direct volume renderers that is conceptually straightforward. The algorithm is very versatile and uses either perspective or orthographic projection to generate an image from the data. It also supports both pre-classification, [Lev88, Lev90], and post-classification, [TSH98, GBKGl04a].

The raycasting algorithm gets its name from casting viewing rays through the output image pixels into the volume. Each ray is subdivided into samples lo-

(a) raycasting, $m = 2$ (b) shear-warp, $m = 2$ (c) splatting, $m = 2$

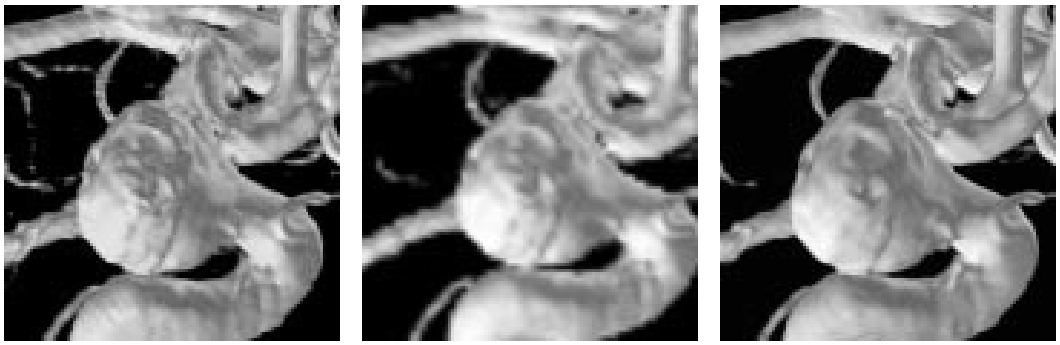


(d) raycasting, $m = 6$ (e) shear-warp, $m = 6$ (f) splatting, $m = 6$

(g) raycasting, $m = 8$ (h) shear-warp, $m = 8$ (i) splatting, $m = 8$

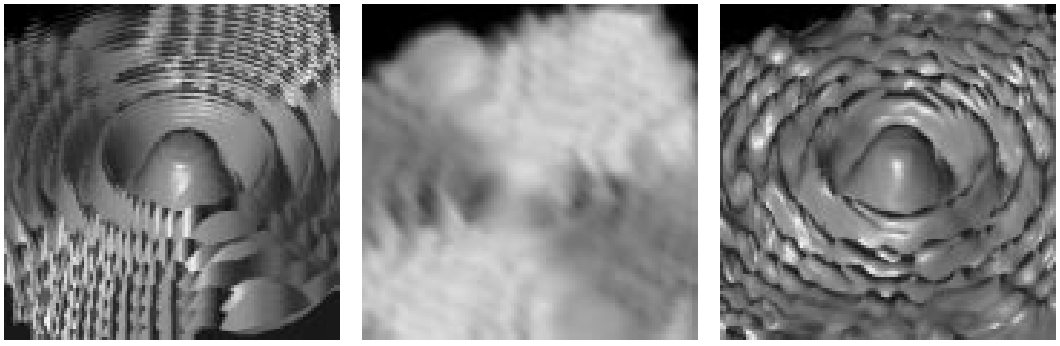
Figure 2.9: CT Head (skull) dataset, 256^3, rendered with raycasting, shear-warp and splatting to compare the rendering quality at different magnification levels, m. Images from [MHB+00].

(a) raycasting, $m = 1$ (b) shear-warp, $m = 1$ (c) splatting, $m = 1$



(d) raycasting, $m = 2$ (e) shear-warp, $m = 2$ (f) splatting, $m = 2$

Figure 2.10: MRI Head (blood vessel) dataset, 256^3, rendered with raycasting, shear-warp and splatting to compare the rendering quality at different magnification levels, m. Images from [MHB+00].



(a) raycasting (b) shear-warp (c) splatting

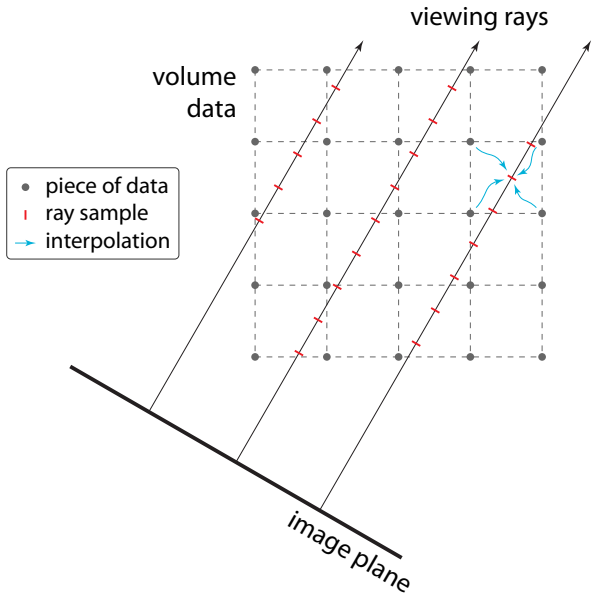
Figure 2.11: Marschner-Lobb function, 41^3, rendered with raycasting, shear-warp and splatting to compare the rendering quality at magnification level of six. Images from [MHB+00].

cated a specified distance apart, $\triangle s$. Then, for pre-classification, the data around the sample is classified via the transfer function to obtain the colors and opacities. These colors and opacities are interpolated onto the sample. However, computing them independently leads to an incorrect integration of color, which is corrected by interpolating their product instead, [MHB+00].

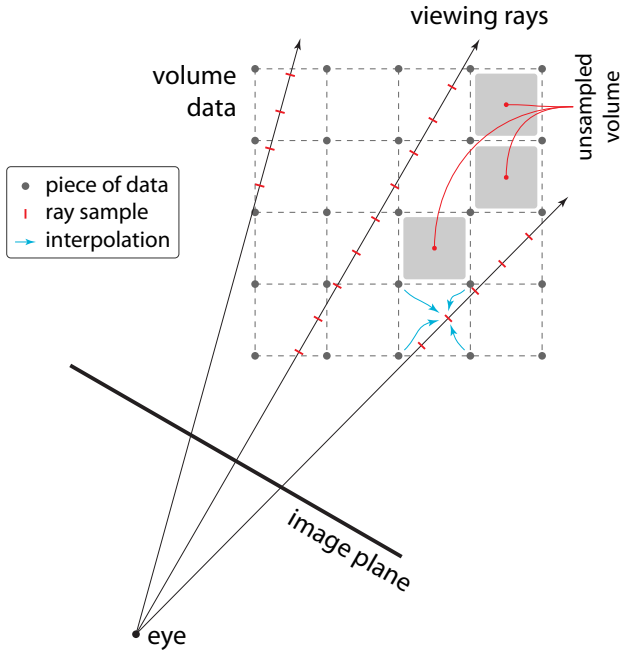
Post-classification raycasting interpolates the data onto the sample first and then classifies that sample via the transfer function. Tri-linear interpolation is typically used but the framework allows for other filters. (See Section 2.2 for a formal discussion of the interpolation kernels.) The final step in raycasting is shading and compositing of samples along each viewing ray. This can be done in back-to-front or front-to-back order, but the latter is preferred because it allows the use of early ray termination to accelerate the raycasting time. Fig. 2.12 illustrates a raycaster using orthographic (a) and perspective (b) projections.

With the equal sampling distance, pattern sampling artifacts may arise in the final image. In order to remedy this, the positions of the samples can be jittered or samples from neighboring rays can be interleaved, [KH01]. Another sampling-related issue emerges with strictly iso-surface rendering, e.g. the Marching Cubes algorithm [LC87], if the vertices making up the triangle mesh approximating an iso-surface are constrained to lie on cuboid edges. The inaccuracy of this approximation is fixed by analytically computing the location of the iso-surface within each, cuboid [MKW+04].

As simple as brute-force raycasting is, it has a very high computational cost. One of the popular accelerations is early ray termination [Lev90]. It stops all computations for a ray that has reached an opacity threshold. After a certain sample on a terminated ray, the contributions from the following samples are



(a) Orthographic projection



(b) Perspective projection

Figure 2.12: Raycasting illustrated in 2D. The ray samples get composited front-to-back to allow early ray termination. To obtain a value at a sample, the data (post-classification) or colors (pre-classification) are interpolated. When using a perspective projection (b), the rays diverge from each other making it possible to miss some part of the volume entirely (shaded in grey).

negligible and their computation can be omitted.

Another important acceleration is empty space skipping, which has several forms. On the smallest scale, shading and compositing computations can be skipped for an invisible sample, which still needs to be classified. Light-weight space leaping [LK04] skips empty space up to the first visible voxel. Other methods build hierarchical min-max trees which allow the determination of whether a node contains any visible data, [Lev90]. Another method reduces the memory storage requirement of the min-max tree by rearranging the data into bricks, [GBKGl04a, GBKGl04b]. This strategy improves the coherency of data access for the raycaster and allows skipping empty space one brick at a time. Raycasting accelerations are discussed in more detail in Chapter 3.

Raycasting can exploit a great deal of parallelism derived from the fact that each viewing ray can be computed independently, hence image subregions can be computed simultaneously. One can also parallelize this method by subdividing the data into regions and computing a separate output image for each. However, a major difficulty is compositing these output images correctly. Parallelization schemes are discussed in Chapter 4.

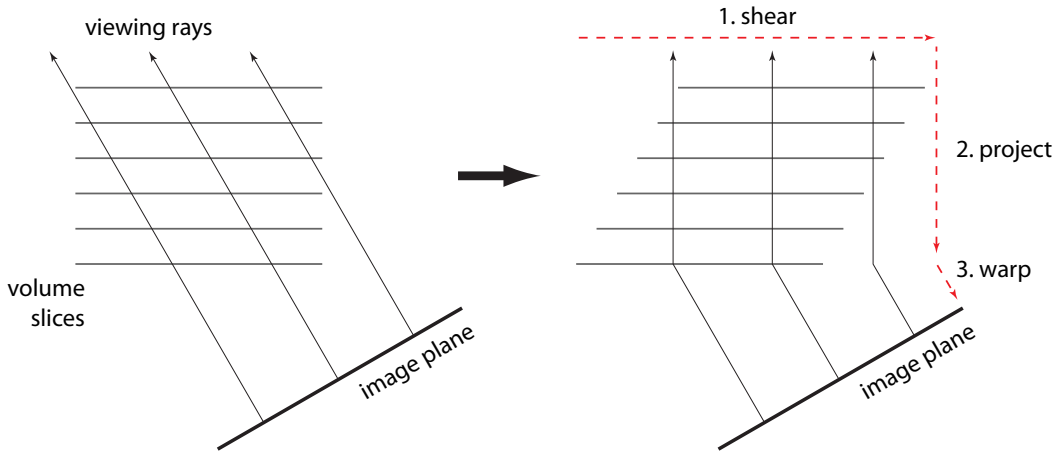
Such schemes come naturally because raycasting provides a very flexible software framework for volume and iso-surface image generation. It is easily modified to use perspective or orthographic views, pre- or post-classification of data, and a multitude of interpolating functions. Raycasting is considered to produce images of the best quality because it uses tri-linear interpolation. Because of the computational expense, this method requires clever acceleration structures and parallel implementations to achieve real-time computation speeds.

2.4.2 Shear-Warp

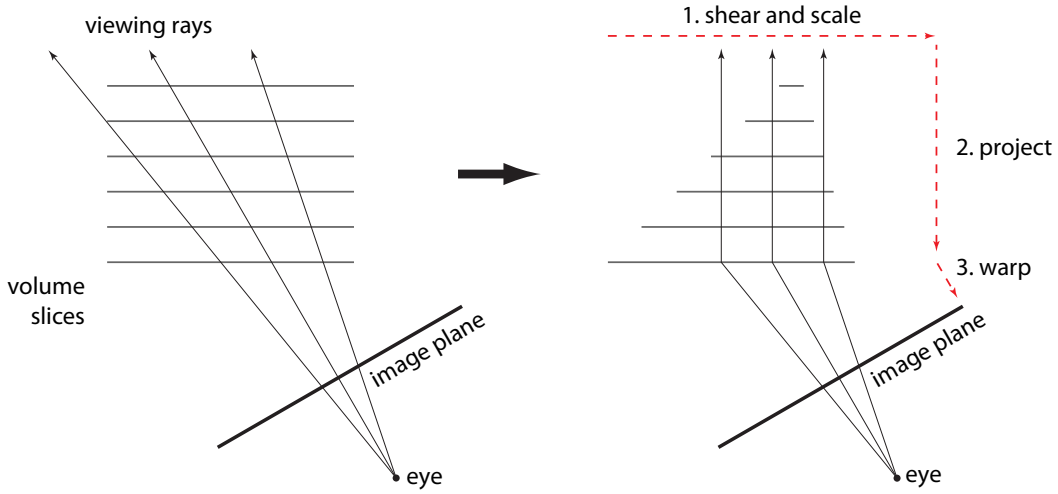
Shear-warp was proposed by Lacroute and Levoy in [LL94] and extended by Sweeney and Mueller in [SM02]. This algorithm is considered to be the fastest software volume renderer, [MHB+00, SM02]. Its speed is achieved by employing a clever encoding scheme and simultaneously traversing the data and the output image. During the traversal, opaque image regions and transparent voxels are skipped. Most of the speedup comes from the single traversal of the volume data in memory order, which is extremely cache-friendly.

The insight of the shear-warp algorithm comes from splitting the projection matrix into a shearing component and a warping component. To allow the most cache-friendly data access, the authors also used a permutation matrix that flips the order of major axes to make the z-axis most parallel to the viewing direction.

In a pre-processing step, the voxels are run-length encoded (RLE) based on pre-classified opacities. This results in creating three separately encoded volumes, one for each of the major viewing directions. The rendering is performed by shearing the appropriate encoded volume such that the rays are perpendicular to the volume slices. The rays obtain their sample values via bi-linear interpolation within the traversed volume slices. A final warping step transforms the volume-parallel baseplane image into the screen image. Because bi-linear interpolation is used, the accuracy of the interpolated sample is limited. Fig. 2.13(a) shows shear-warp with an orthographic projection. Perspective projection can also be used but requires the sheared slices to be scaled based on their distance from the eye, as shown in Fig. 2.13(b). The scaling is necessary because perspective rays diverge as they traverse the volume.



(a) Orthographic projection



(b) Perspective projection

Figure 2.13: Shear-warp projection can be decomposed into three steps: shearing of volume slices, projecting them onto an intermediate image and then warping it to produce the correct output. Because rays diverge in the perspective transformation, the volume slices have to be scaled based on their distance from the eye. Figure modified from [LL94].

As a result, the distance between the sheared slices $\triangle s$ depends on the viewing direction as:

$$\triangle s = \sqrt{\left(\frac{dx}{dz}\right)^2 + \left(\frac{dy}{dz}\right)^2 + 1}, \tag{2.22}$$

where $[dx, dy, dz]^T$ is the normalized viewing vector, reordered such that dz is the major viewing direction. The value for $\triangle s$ ranges between 1.0 for axis-aligned views, $\sqrt{2} \approx 1.41$ for edge-on views and $\sqrt{3} \approx 1.73$ for the corner-on views. One cannot change $\triangle s$ to super-sample the data; thus the Nyquist-Shannon sampling theorem[4] is potentially violated for all but the axis-aligned views. Another drawback of the method is that the resolution of the output image is obtained by a bi-linear interpolation after the warping step, which becomes an issue for viewport resolutions much larger than the data resolution, [MHB+00, SM02].

Sweeney and Mueller improved on the original shear-warp algorithm in [SM02] by addressing most of its shortcomings: pre-classification of data, output image blurriness at close-up viewing, artifacts from lack of sampling between sheared slices, and the large memory consumption due to the need for three copies of the data. The authors noted that the RLE-encoded volumes are identical for y and z major viewing directions and store only one, reducing the memory footprint of the shear-warp algorithm by a third.

To fix blurring from pre-classifying the data, the order of interpolation and classification has been switched. Now, one can approximate the iso-surface the sample lies on by shading that sample, which requires a data gradient. Instead of computing it directly, the authors used a look-up table into pre-computed

[4]The Nyquist-Shannon sampling theorem states that a sampled bandlimited signal with the highest frequency B can be perfectly reconstructed from an infinite sequence of samples if the sampling rate exceeds $2B$, [SAG+05]. Because we are reconstructing a signal from the given data samples, the reconstruction frequency must not drop below the original sampling frequency. This means that to satisfy the theorem, the maximum sampling distance allowed is $\triangle s = 1$.

gradient directions.

To mediate the blurring of the final image after warping due to smaller data resolution, the authors upsampled the sheared images. This allows for the post-warp image resolution to be almost the same as the resolution of the output image. Compared to the original method, this increases the required computation proportionally to the increase in sampling rate of the sheared images.

Final contribution of [SM02] is the ability to obey the Nyquist-Shannon sampling theorem by using an extra sheared slice half-way in between the original sheared slices. This adds quite a lot of overhead but guarantees the sampling distance to be appropriate, $\triangle s < 1.0$, at all times.

Because the shear transformation of data slices uses bi-linear interpolation in memory-order, the original algorithm is very fast but does not produce images as sharp as raycasting. The work by Sweeney and Mueller [SM02] attempts to remedy the quality issue while retaining the cache-coherent data access of the method. However, their proposed solution can significantly degrade performance without promising results qualitatively similar to raycasting, [SM02]. Although considered to be the fastest software renderer to date, shear-warp is not recommended for visualization where sharp features within the data must be preserved until image quality issues are resolved.

2.4.3 Splatting

Splatting is a projective algorithm introduced by Westover in [Wes90]. The volumetric data is thought of being comprised of overlapping radially symmetric

basis functions scaled by the voxel values. The algorithm projects the centers of these kernels onto the output image and then rasterizes and composes the kernel footprints. The rasterization is achieved efficiently via using a pre-computed footprint look-up table, which stores the contribution of data relative to the radius from the center of the kernel. A major advantage of this method is that it projects and rasterizes only the data relevant to the output image.

The traditional splatting approach [Wes90] sums the voxel kernels within the volume slices that are most parallel to the image plane. This approach is prone to popping, or severe brightness variations, when the plane that is most parallel to the image plane changes. Another drawback of this method is its inability to change the sampling distance along each ray. Mueller and Crawfis proposed splatting with image aligned slabs in [MC98] to mitigate both of these issues. In their method, all voxels are projected and clipped onto slabs, which are then composited together in front-to-back order. The slab thickness can be changed allowing for various sampling distances along a ray. Fig. 2.14 shows the 2D version of splatting with image aligned slabs, [MC98].

We observe that splatting replaces a point sample of raycasting by a sample average across the sampling distance. This introduces an additional low-pass filtering that helps to reduce aliasing when $\triangle s > 1.0$. Splatting typically uses radially symmetric Gaussian kernels, which have better anti-aliasing characteristics than linear filters, with the side effect that they perform some signal smoothing, [MHB+00].

The original splatting algorithm in [Wes90] has another contribution to blurring due to pre-classifying the data. Mueller et. al. in [MMC99] rewrote the algorithm to post-classify the data: splat footprints are rasterized onto the cur-

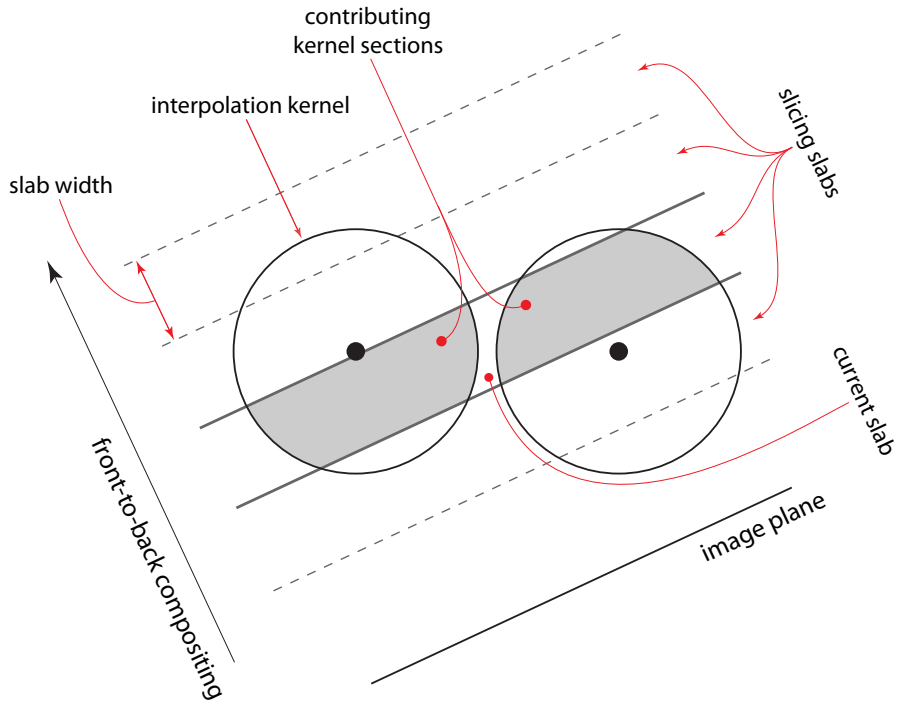


Figure 2.14: Overview of splatting with image aligned slabs. Circles represent the extent of radially symmetric interpolation kernels centered at the data locations represented by black dots. These kernels are clipped to the current slab, which gets composited to the output image in front-to-back order. Figure modified from [MC98].

rent slab and then samples within are classified via the transfer function. Finally, these samples are shaded to include light reflection off the iso-surface in which the sample is embedded. This requires vector gradients, which Mueller et. al. computed by generating gradient splats.

Huang et. al. in [HMSC00] improved the speed of the footprint rasterization by using a 1D representation of the kernel, called FastSplat, instead of the original 2D representation. To implement an acceleration similar to early ray termination, one can use a dynamically computed screen occlusion map to cull invisible splats early from the rendering pipeline, [MSHC99]. The main operations are the transformation of each relevant data location into screen space, followed by an index into the occlusion map to test for visibility. If the splat

is visible, then its footprint is rasterized into the sheetbuffer which is an image of the current slab. It should be noted that although early splat elimination saves the cost of footprint rasterization for invisible voxels, their transformation must still be performed to determine their occlusion. Early ray termination, on the other hand, causes the ray to stop and subsequent samples to be skipped, [MHB+00].

Updating splatting to produce perspective images is difficult because the perspective transformation changes the shapes of spherical splats into ellipses. Hence, the kernels are no longer radially symmetric and their footprint has to be recomputed for every ray. Mueller and Yagel accelerated perspective splatting by combining it with raycasting in [MY96]. The perspective rays are traced through the volume to intersect the radially symmetric splats. In this way, the perspective projection is computed by raycasting, which allows the use of certain accelerations like early ray termination or bounding volumes.

Another method to produce perspective volumetric images with splatting uses a transformation between spherical and elliptic splats. Zwicker et. al. tackled this unnecessary overhead in [ZPvBG01] by using elliptical Gaussian kernels to splat footprints directly. The authors approximated the perspective projection by a local affine transformation which, they claimed, does not introduce significant error. The output images tend to be less blurry because the elliptic splats have reduced footprints due to non-uniform scaling. For more details, see [ZPvBG01].

Because splatting is an inherently projective algorithm, one can leverage the processing power of graphics boards for rasterization and projection. Zwicker et. al. improved their earlier algorithm by using perspective-correct splats and

implemented their method using shaders in [ZRB+04]. Another improvement they introduced is clipping intersecting splats, which allowed for sharp features to be displayed.

Besides the ability to leverage the computational power of a graphics processor as a projective algorithm, splatting guarantees that every contributing voxel is included in the output image. Some averaging artifacts can appear due to the use of radially symmetric interpolation functions and pre-classification of data. In addition, producing correct perspective projection of splats introduces much complexity and requires high computational power.

2.4.4 Custom Hardware and Texture Mapping

Using special hardware for volume rendering is appealing because it allows for hardware implementation of the time-critical computations. There have been several proposed approaches that achieve real-time volume rendering speeds for medium-sized regular data sets: Cube [KK99], Vizard [KS97, MKW+02], and VolumePro™ [PHK+99]. Fig. 2.15 shows the VolumePro™ board manufactured by TerraRecon, Inc. Changing or extending custom hardware is time-consuming and costly; hence, once general processors increased their computational speed, these hardware solutions started to phase out of use, [MFS06]. Besides the lack of easy programmability and extendibility, custom hardware solutions are limited to displaying datasets that fit into the on-board memory. The discussion below follows the overview of the custom hardware solutions presented in [MFS06].

The shear-warp algorithm, discussed in Section 2.4.2, was used to create

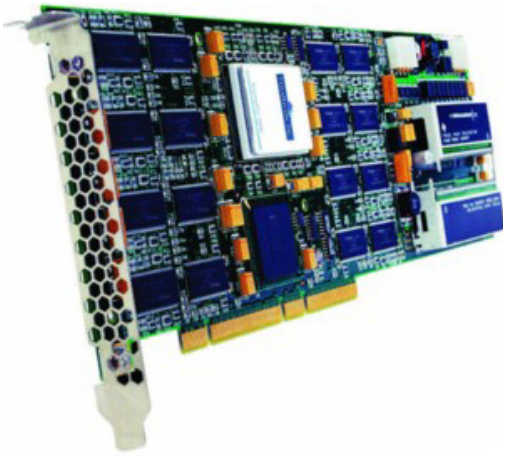


Figure 2.15: The VolumePro™ board manufactured by TerraRecon, Inc. is an example of special hardware designed for fast direct volume rendering. The design of VolumePro™ is outlined in [PHK+99] and [WBLS03]. Image from [Ter09].

the Cube system, [KK99]. This system uses a hybrid approach: the custom board shears the volume data while the graphics board warps and composites the sheared slices. This design led to a well-known VolumePro™ board [PHK+99], which has programmable sample and voxel processing pipelines. Voxel processors traverse data slice-by-slice in memory order and store slices in on-chip buffers. These buffers are traversed by sample processors responsible for illumination, filtering and compositing, [MFS06]. Even though the original VolumePro™ board used orthographic projection, the next generation uses perspective projection [WBLS03]. The authors also improved the rendering quality by using tri-linear interpolation instead of the bi-linear interpolation used in shear-warp algorithms.

Vizard [KS97] and Vizard II [MKW+02] were based on an image-order ray casting including early-ray termination. The performance is not comparable to VolumePro™ because it was implemented via field-programmable gate arrays. This makes the system more extendable but at the cost of image computation speed, [MFS06].

All of these methods offer viable solutions for visualizing datasets, but are becoming less favorable due to the improvements of the graphics processing units (GPUs). As the technology progresses, the issues of very limited available on-board memory, slow floating-point computation speed and lack of programmability are being removed from the GPUs.

Cabral et al. [CCF94] was one of the first to use texture capabilities of GPUs for rendering volumetric datasets. The volume is loaded into texture memory and the hardware rasterizes polygonal slices parallel to the output image plane. The slices are then blended in back-to-front order because then current hardware lacked the accumulation buffer for opacity. The method allows the free choice of the sampling distance between slices and uses tri-linear interpolation to compute the values at sample locations, [MHB+00].

Engel et al. [EKE01] improved the rendering quality by proposing pre-integrated volume rendering. Röttger improved on this approach with volumetric clipping and advanced lighting in [RGW+03]. These methods do not account for culling samples based on visibility. To achieve this, Li proposed to use an opacity map which stores the minimum opacity of the pixels in the output image, [LMK03]. The data is partitioned into subvolumes with similar properties that depend on the transfer function, e.g. data values within a certain range are grouped together. Each subvolume is a node in a kd-tree which is used to render the subvolumes in the correct visibility order. The subvolumes are culled and clipped against the opacity map. Krüger et al. [KW03] improved the texture mapping approach further by implementing early ray termination and empty space skipping via an early depth test to terminate fragment processing. Later approaches exploited the programmability of fragment shaders to implement

raycasting on a GPU. See [MFS06] for details of several approaches.

With the release of Compute Unified Device Architecture (CUDA) it has become possible to exploit the computational power of a GPU directly without the need to utilize texture mapping or programmable fragment shading functionality. It is now possible to implement many of the software renderers discussed previously directly on a GPU and benefit from the massive parallelism and the computational speed that the hardware provides. This seems to be the trend in the current research.

2.5 Summary

In this chapter, we described the basic concepts embedded in a volume renderer. We started with the model for the interaction of light with the volume of data and derived the low-albedo rendering integral used by all algorithms. We discussed in detail the modules shared by all direct volume renderers, like data interpolation, classification via transfer functions and compositing samples. We used this preliminary information to describe several classes of algorithms that are most popular: raycasting, shear-warp, splatting and hardware-based methods.

We have commented on accuracy and calculation speed of each of these methods as these properties are the most important to the medical field applications. Visualizing scanned scalar data for diagnostic purposes imposes the following constraints on accuracy of a rendering method: an algorithm should not produce severe artifacts and must render all of the data to produce a volumetric image. For best diagnosis, the medical field requires each output image

be calculated fast enough to allow real-time viewing.

Because of the constraint on image quality in medical applications, we chose the raycasting algorithm as the basis for the direct volume renderer. To reach this decision, we followed the comparison between algorithms discussed in Section 2.4 and based on reviews presented in [MHB+00, MFS06, KM05]. The superior image quality of raycasting comes at a high computational cost, which was prohibitive for the real-time applications in the past. However, current emergence of massively parallelized computing, referred to as "cloud computing," provides easy access to enough computational power making real-time visualization possible without the need to purchase and maintain a similar system locally. Using this technology can bring visualization tools capable of producing high quality output into operating rooms but it requires parallelizing the raycasting algorithm to utilize such systems. The rest of this thesis focuses on possible approaches and evaluates them.

CHAPTER 3

RAYCASTING ACCELERATION METHODS

The previous chapter described several volume rendering algorithms in detail and classified them based on [MHB+00, MFS06, KM05]. Because the quality of output images is essential for medical applications, we selected raycasting as the basis method for this work. Its largest drawback is the high computational cost which necessitates the use of acceleration techniques described in this chapter. Unfortunately, acceleration techniques alone are not sufficient enough to achieve real-time rendering, hence the algorithm needs to be parallelized. Several parallelization approaches are discussed in Chapter 4.

The basic raycasting algorithm, described in Section 2.4.1, consists of casting viewing rays through the output image pixels into the volume. The rays can be parallel or diverging, depending on whether orthographic or perspective projection is used. The final color of a ray results from integrating color contributions of volume data along its path. Since the volume is discrete, we approximate this integral as a Riemann sum over a finite number of samples along each ray. Applying a user-specified transfer function to interpolated data at sample locations provides their color and opacity.

To gauge the complexity of an unaccelerated raycaster, we can calculate the number of samples needed to generate an image. The cost of classifying and compositing samples is minimal compared to shading and interpolating data for each sample, hence we only account for the latter.

Consider a cubic data set with the 1024^3 voxels and a sampling distance of one. For an orthographic view, the number of samples per ray is 1024. To pro-

duce an output image with resolution of 1024^2, we need $1024^2 \cdot 1024 \approx 1,074$ million samples.

Section 2.3 discussed several interpolating functions with different properties and computational costs. The cheapest is the tri-linear interpolation because it requires 14 additions and 14 multiplications per sample. Generating all samples for the above example requires 28 operations for each of the $1,074$ million samples resulting in approximately 30 billion operations. Assuming these are floating point computations taking five clock cycles each, a single 3.33 GHz processor requires about 50 seconds to generate all of the samples. This time estimate is rather crude and does not take into account any effects of fetching data, compiler optimizations, or sample shading. However, it shows the importance of acceleration methods for the simple raycaster.

Clearly, eliminating as many samples as possible will result in a substantial speed-up. There are several techniques based on this idea, the most important being early ray termination and empty space skipping. Similarly, rendering images at smaller resolutions improves the speed as well. Another approach decreases the amount of necessary calculations per sample via pre-computed data. Gradients are a popular choice but their storage requires several times more memory than the data itself. This emphasizes the problem of cache coherent data access, which can be addressed by rearranging the data into bricks.

We start with a general discussion of acceleration techniques, then specify details of the brick hierarchy and outline our accelerated raycaster. The remainder of this chapter is organized as follows. Section 3.1 describes the multiple rendering resolutions and Section 3.2 describes gradient pre-computation. We discuss early ray termination in Section 3.3 and empty space skipping in Sec-

tion 3.4. Details behind the brick hierarchy are outlined in Section 3.5. We outline the accelerated raycaster in Section 3.6 and evaluate the performance of the mentioned accelerations in Section 3.7. We conclude in Section 3.8.

3.1 Multiple Rendering Resolutions

Using several rendering resolutions is the simplest acceleration technique to implement. It applies in a rapid exploration of the data, when output image resolution is not important, such as while changing the viewing location and direction via rotation, zooming in/out, and panning around the volume. When the area of interest is found, the resolution is increased to produce high quality output. This technique is also useful for exploring the data via changing the transfer function.

3.2 Gradient Pre-Computation

The technique of gradient pre-computation builds directly upon the principle of trading memory for speed in compute-heavy applications. In volume visualization, pre-computation is applicable to gradients, which represent a vector normal to an iso-surface at the sample location and are needed for shading samples. First we discuss why numerical gradients correctly approximate iso-surface normals and the methods for their computation. We finalize by describing the scheme and its memory cost.

By definition, an iso-surface at a sample location is the surface of a constant

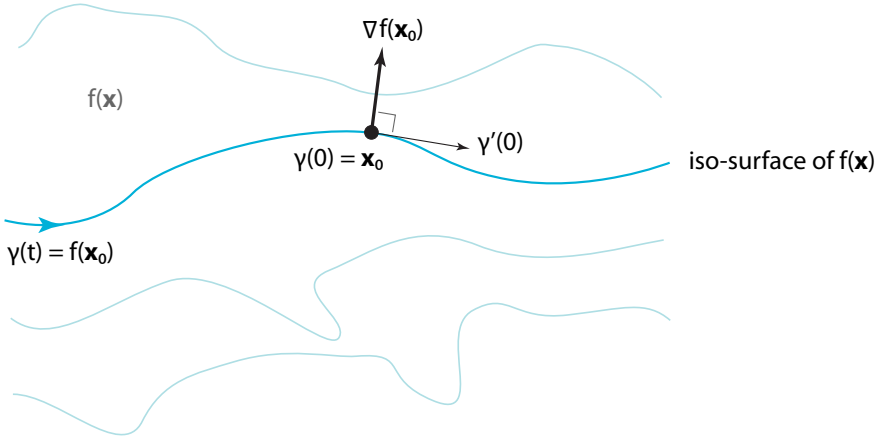


Figure 3.1: Computing a normal vector to an iso-surface as a gradient of the volume data.

value equal to the interpolated data value at that location. Mathematically, an iso-surface through $\mathbf{x}_0$ is a level set of a function $f : \mathbb{R}^3 \to \mathbb{R}$ representing the volume data: $\{\mathbf{x} \mid f(\mathbf{x}) = f(\mathbf{x}_0)\}$, shown in Fig. 3.1. If we parametrize a curve on this level set as $\gamma(t) = \mathbf{x}$ with $\gamma(0) = \mathbf{x}_0$, then the above can be rewritten as $f(\gamma(t)) = f(\mathbf{x}_0)$. Taking a derivative yields $\nabla f(\mathbf{x}_0) \cdot \gamma'(0) = 0$. Since $\gamma'(t)$ is tangential to the curve $\gamma(t)$, the gradient $\nabla f(\mathbf{x}_0)$ must be normal to $\gamma'(0)$, and, hence, the curve $\gamma(t)$ at $\mathbf{x}_0$. Because the curve $\gamma(t)$ is chosen arbitrarily, the gradient $\nabla f(\mathbf{x}_0)$ is normal to the iso-surface.

A gradient at a sample location can be computed via a tri-linear interpolation of eight gradients located at the neighboring voxel locations. Central finite differences, Equation (3.1), are typically used to compute the gradients located inside the volume. At the edges, one can use one-sided differences, Equation (3.2).

$$\frac{\partial f(\mathbf{x}_0)}{\partial x_i} \approx \frac{f(\mathbf{x}_0 + \mathbf{h}_i) - f(\mathbf{x}_0 - \mathbf{h}_i)}{2\|\mathbf{h}_i\|} \tag{3.1}$$

$$\frac{\partial f^-(\mathbf{x}_0)}{\partial x_i} \approx \frac{f(\mathbf{x}_0) - f(\mathbf{x}_0 - \mathbf{h}_i)}{\|\mathbf{h}_i\|} \qquad \frac{\partial f^+(\mathbf{x}_0)}{\partial x_i} \approx \frac{f(\mathbf{x}_0 + \mathbf{h}_i) - f(\mathbf{x}_0)}{\|\mathbf{h}_i\|}, \tag{3.2}$$

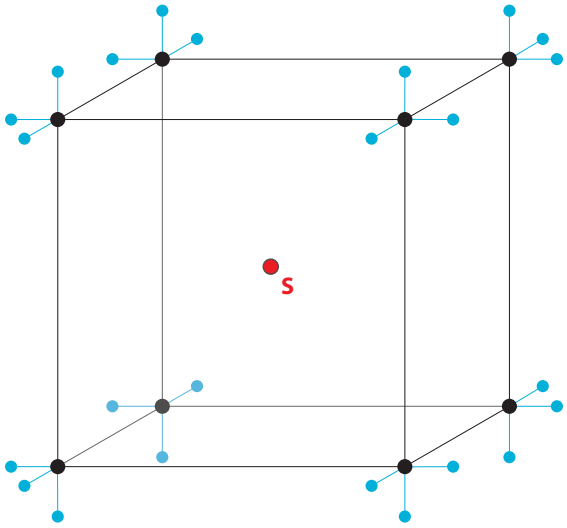


Figure 3.2: Voxel neighborhood to compute a gradient at a sample location S. The line segments connecting outer voxels to the nearest neighbors are shorter for pictorial purposes only.

where $i = \{1, 2, 3\}$, $\mathbf{h}_i$ is the vector distance between voxels in the i^{th} direction, and $\mathbf{x}_0$ is the sample location aligned to the voxel grid. The superscript minus and plus signs in Equation (3.2) indicate left and right one-sided differences respectively. The gradient vector combines the results from applying finite differences in each dimension. Computing all of the necessary gradients requires 32 voxels surrounding the sample location, shown in Fig. 3.2. This is considerably larger than the eight nearest neighbors needed for interpolating data.

Using central differencing with the grid spacing of $||\mathbf{h}_i|| = 1$, for $i = \{1, 2, 3\}$, each of the eight vector gradients requires three additions and three multiplications. Interpolating the gradients requires seven tri-linear interpolations per dimension for a total of 21 interpolations per gradient. Hence the net cost of one gradient estimation per sample is $2 \cdot 21 + 3 \cdot 8 = 66$ additions and multiplications each.

It should be noted that the gradients at voxel locations are shared by close

samples of the neighboring rays, hence reusing gradients saves computation. These values can be stored on either a local or a global level. The memory footprint of the global approach is several times larger than the data itself. Consider a 16-bit dataset of 1024^3 resolution, which requires two giga-bytes (GBs) of storage. Storing pre-computed gradients as 3D vectors of 32-bit floating point numbers requires an extra 12 GB of storage, a six-fold increase over the original data size. This scalability severely limits the size of the data one can display. Local gradient storage remedies this problem and is discussed in Section 3.5.3.

3.3 Early Ray Termination

Early ray termination is based on the way accumulated transmissivity of a viewing ray diminishes as more samples are added. Each sample contributes multiplicatively, described by Equation (2.9):

$$T_k = \prod_{i=1}^{k}(1 - \alpha_i), \tag{3.3}$$

where T_k is the ray's transmissivity after the first k samples and α_i is the opacity of the i^{th} sample. If T_k reaches a small enough value at some sample, the contribution from samples beyond it is nearly zero. This threshold can be a control parameter provided by the user. We refer to it as an opacity threshold to signify that, once the threshold is reached, the ray becomes opaque and blocks almost all of the light after a certain sample.

Early ray termination was originally introduced by Levoy in [Lev88]. This acceleration is excellent for culling distant samples that do not contribute to the final color value. Consider a ray with the first five samples of opacity equal to

0.9. Then the sixth sample contributes only

$$\alpha_6 \prod_{i=1}^{5}(1-\alpha_i) = 0.9 \cdot (1-0.9)^5 = 0.000009 \tag{3.4}$$

of its color. If we change the opacities of the first five samples to 0.1, the sixth sample contributes 0.059049 of its color. As can be seen, this method works well only if a viewing ray reaches the opacity threshold before leaving the volume. Hence, acceleration results vary depending on the data set and the transfer function used for visualization.

3.4 Empty Space Skipping

Early ray termination, described in the previous section, culls ray samples occurring beyond the point where the ray reaches an opacity threshold. Empty space skipping uses the transfer function to skip any samples that are totally transparent and hence contribute nothing. There are two types of empty space skipping, local (samples) and global (collections of samples).

On the local per sample level, an algorithm can skip samples with opacity near zero. Even though this saves sample shading computations, data interpolation is still necessary and may be expensive. To save this cost as well, one can skip empty space on a global level by considering transparency of collections of voxels. Most visualizations isolate small features within the volume by making the surrounding space invisible. All samples within this space can be skipped safely.

Light weight space leaping, introduced in [LK04], marches a small subset of image rays, called detector rays, through the volume. Each detector ray records

a number of transparent samples before reaching the first opaque one and stores this into a leap buffer. The minimal distance is then spread from detector rays to all other image rays. A major benefit of this method is its small memory footprint; however, it may miss small details if the detector ray sampling is too sparse relative to the data.

Min-max octrees [WG92] are a popular acceleration structure for empty space skipping. Each tree node stores the minimum and the maximum values of the data inside. A node is transparent if the transfer function classifies the interval $[min, max]$ as transparent. Computing this classification quickly utilizes a summed area table, which encodes the opacities in the following manner:

$$C_{sat}(0) = \alpha_0 = \Psi_o(0) \qquad C_{sat}(k) = C_{sat}(k-1) + \alpha_k, \tag{3.5}$$

where C_{sat} is the summed area table and $\alpha_k = \Psi_o(k)$ is the opacity of the data value equal to k for $k \geq 1$. A node is transparent if the following holds:

$$C_{sat}(max) - C_{sat}(min) = \sum_{i=min}^{max} \Psi_o(i) = \sum_{i=min}^{max} \alpha_i = 0. \tag{3.6}$$

The computational expense of checking for transparency consists of two look-ups into the table, which, for CT data sets, has 4096 entries. The min-max octree is a key acceleration for iso-surface rendering because it helps culling irrelevant tree nodes that do not contain the value of the iso-surface to be rendered.

These octrees work efficiently if the range of data within each node is small. However, if the data occupies a large range, the node may be inaccurately classified as opaque and rendered every frame. This misclassification is caused by the crude data approximation that min-max intervals provide and can be improved by using quantized binary histograms. We discuss applying this approach within a data bricking hierarchy in Section 3.5.4.

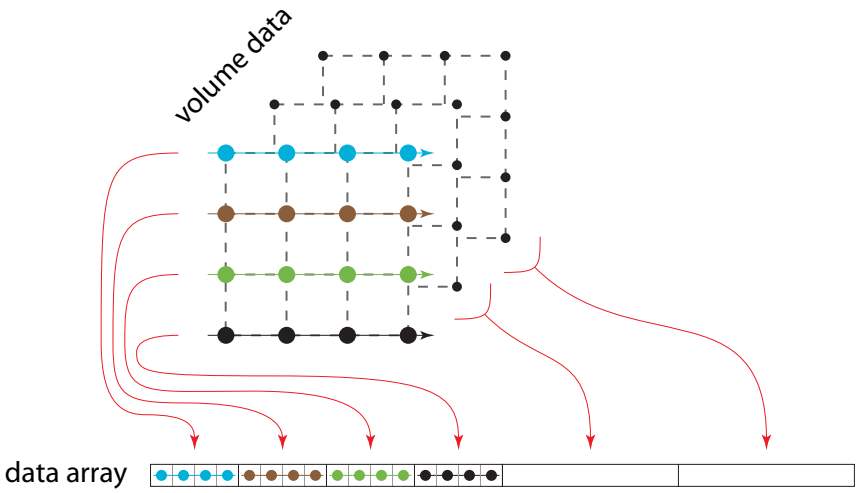


Figure 3.3: Data stored in the linear order in memory: front-to-back one slice at a time, each stored in row-major order.

3.5 Data Bricking Hierarchy

Bricking schemes have existed for some time and come in several varieties. One can consider them an extension of min-max octrees that improves coherency of accessing data by rearranging it. Besides empty space skipping, the hierarchies improve rendering speed by integrating rays through one brick at a time.

Normally, the volume data is stored in memory in XYZ order, shown in Fig. 3.3: one data slice at a time each in row-major order. We refer to this order as linear and define computing the memory index from a location within the volume in the following way:

$$I^{lin}_{(i,j,k)} = i + D_x(j + D_y \cdot k), \tag{3.7}$$

where (i, j, k) is the integer voxel location, D_x and D_y are the dimensions of each data slice. A sequence of images in Fig. 3.5 shows the data access pattern of integrating through a 2D volume one ray at a time. It can be seen that the data access is more or less random throughout the whole linear array.

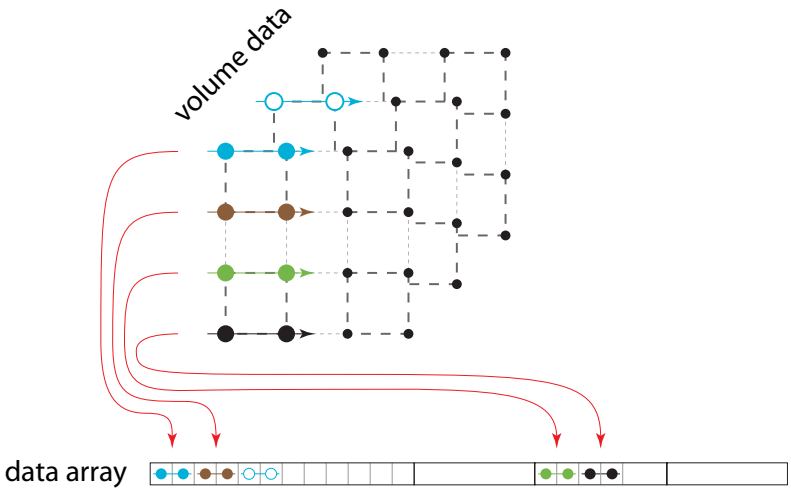


Figure 3.4: Data with resolution 4×4×3 is stored in the bricked order in memory: bricks stored linearly with linear storage of data within each brick. Each brick has dimensions of $2 \times 2 \times 3$.

To improve this access pattern, the data can be rearranged into bricks. Each brick stores its data linearly, while it lies in a linear order with other bricks. This is shown in Fig. 3.4. Computing the memory index into this data arrangement is done in the following way:

$$\begin{aligned} B_{(i,j,k)} &= i'' + BVD_x(j'' + k'' \cdot BVD_y) \\ C_{(i,j,k)} &= i' + BD_x(j' + k' \cdot BD_z) \\ I^{b,lin}_{(i,j,k)} &= (BD_x \cdot BD_y \cdot BD_z)B_{(i,j,k)} + C_{(i,j,k)}, \end{aligned} \tag{3.8}$$

with the variables defined in Table 3.1. Integrating along all rays one brick at a time improves the data access pattern, shown as a sequence of images for a 2D case in Fig. 3.6. Once a brick is rendered, none of its interior data is needed again. However, the data access is still random on the local per brick level, as well as which brick is rendered next.

The data rearrangement faces a difficulty for samples near brick edges because they depend on data in the neighboring bricks. This can be remedied by padding each brick with the edge data from its neighbors, which keeps data

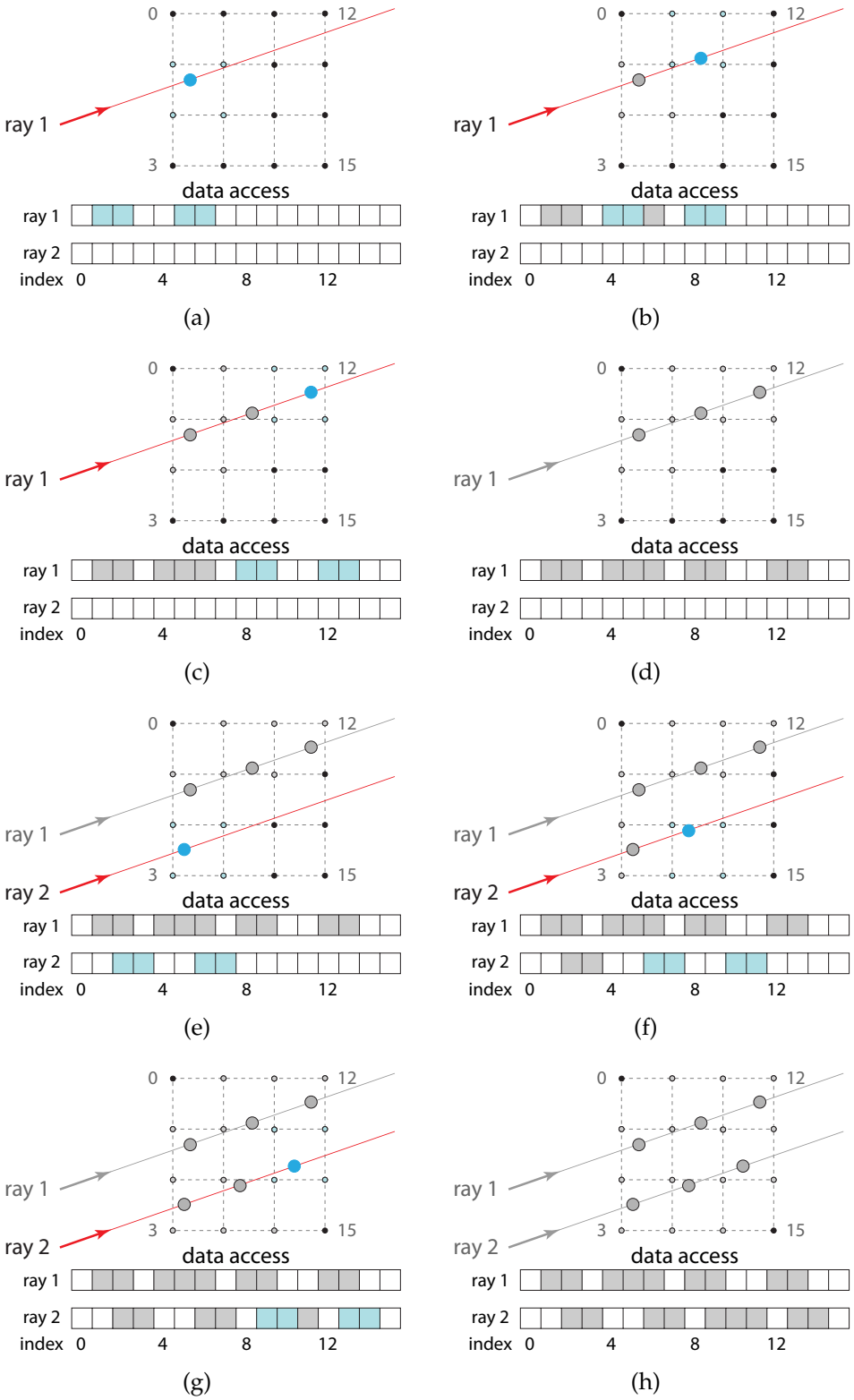


Figure 3.5: A sequence showing memory access pattern of integrating through a 2D volume one ray at a time. The data is stored in a linear order.

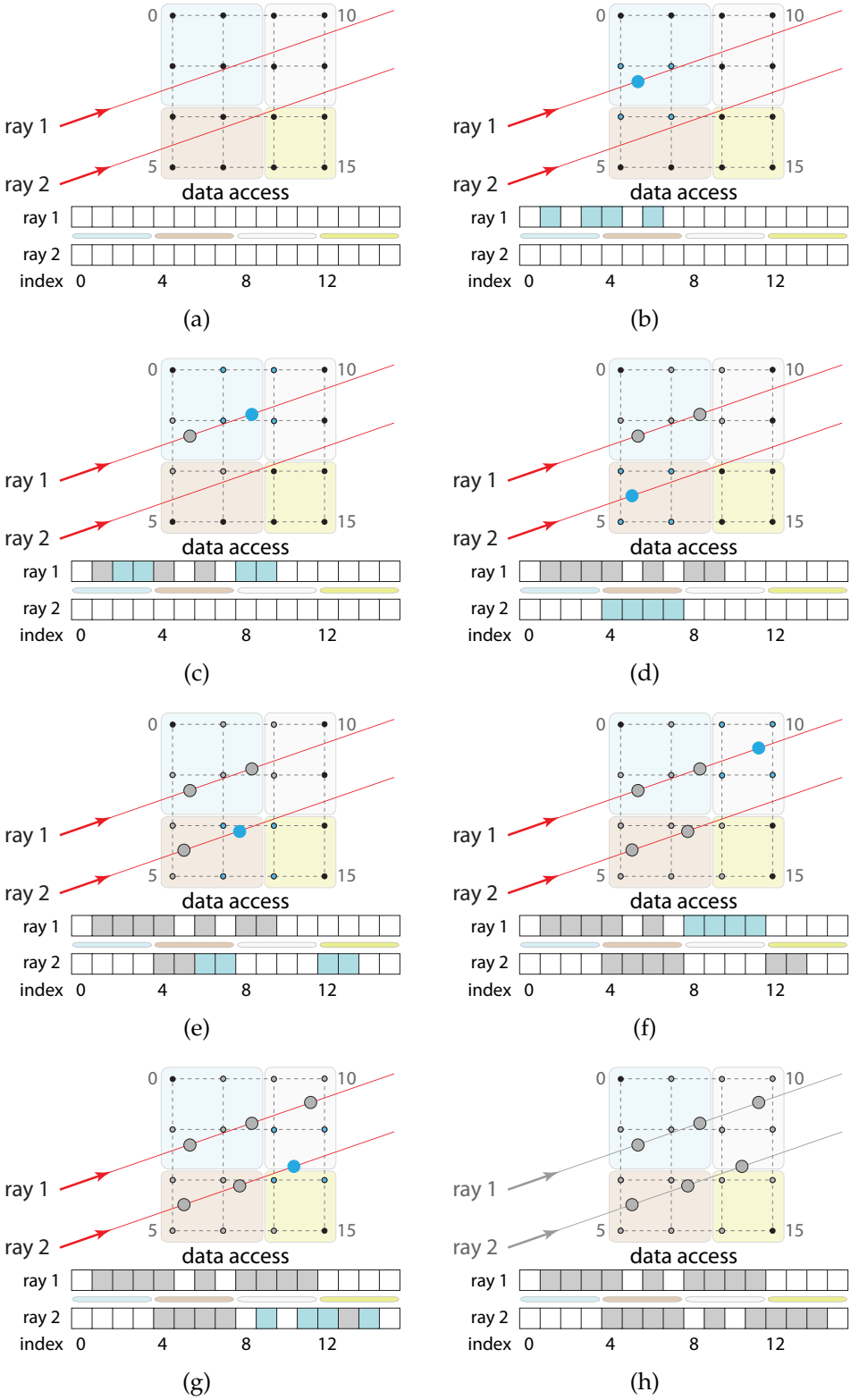


Figure 3.6: A sequence showing memory access pattern of integrating through a 2D volume stored brickwise: linearly stored bricks with data stored linearly.

Variable	Description
(i, j, k)	3D integer location of a voxel
$D_{\{x,y,z\}}$	volume dimensions
$BD_{\{x,y,z\}}$	brick dimensions
$BVD_{\{x,y,z\}} = \frac{D_{\{x,y,z\}}}{BD_{\{x,y,z\}}}$	number of bricks in each dimension
$\{i, j, k\}' = \{i, j, k\} \bmod BD_{\{x,y,z\}}$	voxel location within a brick
$\{i, j, k\}'' = \left\lfloor \frac{\{i,j,k\}}{BD_{\{x,y,z\}}} \right\rfloor$	location of a brick containing the voxel

Table 3.1: Variables in the bricked index computations.

access coherency but increases storage requirements which may not be permissible. Alternatively, looking up the necessary data has no effect on memory storage but affects access coherency. The latter method is discussed in Section 3.5.2.

To improve coherency further, we can store the data inside each brick in a non-linear order. It decreases the distance in memory between spatially neighboring data. We consider Hilbert and Morton orders in Section 3.5.1.

The hierarchy can be constructed from the bottom up to consist of two or three levels. An example of a brick hierarchy with two levels is shown in Fig. 3.7. This benefits datasets with large expanses of transparent space spanning multiple bricks. Each brick node holds references to its children nodes, rays passing through it and a binary histogram for space skipping, discussed in Section 3.5.4. During rendering, the rays are cast into the volume and are stored inside brick nodes they intersect. These rays are integrated front-to-back one node at a time. If a node is deemed transparent, all of the rays inside are advanced to the nodes they enter next. If a node is opaque, the algorithm recurses through the node's children. The algorithm integrates all rays residing in the brick at

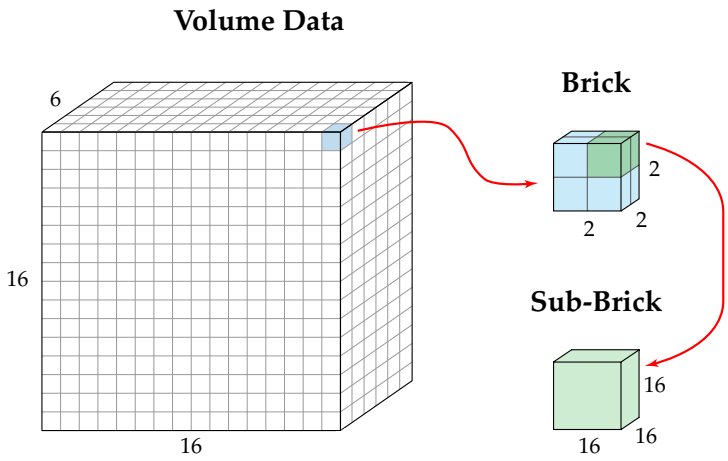


Figure 3.7: Brick hierarchy with two levels for a dataset of $512^2 \times 192$ voxels. At the lowest level lies a sub-brick with dimension of 16^3 voxels. At the second level lies a brick which consists of two sub-bricks in each dimension. The whole volume uses $16 \times 16 \times 6$ bricks.

the lowest level of the hierarchy. Because the rays can only be integrated at the deepest level, it is beneficial to keep the hierarchy shallow. The brick hierarchy was introduced in [GBKGl04b, GBKGl04a] and forms the basis for accelerating our raycaster. The details are discussed in the rest of this section.

3.5.1 Non-Linear Data Storage Order

Typically, the data is stored linearly within each brick which is excellent for quickly fetching neighbors in the x direction. One can improve the cache coherency of accessing data in the other directions by storing it non-linearly. There are two methods we consider: Hilbert order, [LS97], and Morton order, [Sam90]. Both are based on space-filling curves that place spatially neighboring data closer together in memory than the linear order.

Fig. 3.8 shows both of the space-filling curves in 2D for several dimensions.

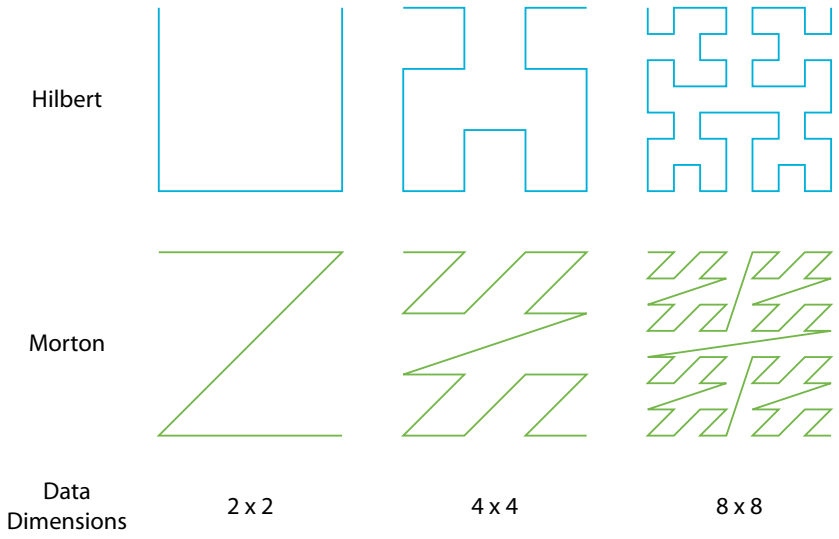


Figure 3.8: Morton and Hilbert space-filling curves for several data sizes in 2D. Notice the recursive nature of the curves.

The index computation for the Morton order, also known as Z-order, is illustrated in Fig. 3.9 and has the following form:

$$
\begin{aligned}
B_{(i,j,k)} &= i'' + BVD_x(j'' + k'' \cdot BVD_y) \\
I^{b,mor}_{(i,j,k)} &= (BD_x \cdot BD_y \cdot BD_z)B_{(i,j,k)} \\
&\quad + \Big(swzl(i') + [swzl(j') \ll 1] + [swzl(k') \ll 2]\Big),
\end{aligned}
\tag{3.9}
$$

where (i, j, k) is the integer voxel location, $B(i, j, k)$ is the 1D integer index of the brick containing the voxel, $\ll$ is a leftwise bit shift[1], and $swzl(m)$ is the function that puts two zeros in between every bit of a binary form of m. The rest of the variables are defined in Table 3.1.

The $swzl(m)$ function in Equation (3.9) is rather expensive because it is made up of several multiplications, bit shifts and additions. It can be accelerated via a look-up table connecting indices to swizzled bits derived from them. For example, consider a brick of 32^3 voxels. The swizzle look-up table would hold 32

[1]Consider a binary number 101101 = 45. Applying a single left bit shift results in 1011010 = 90 and is equivalent to multiplying the original number by 2.

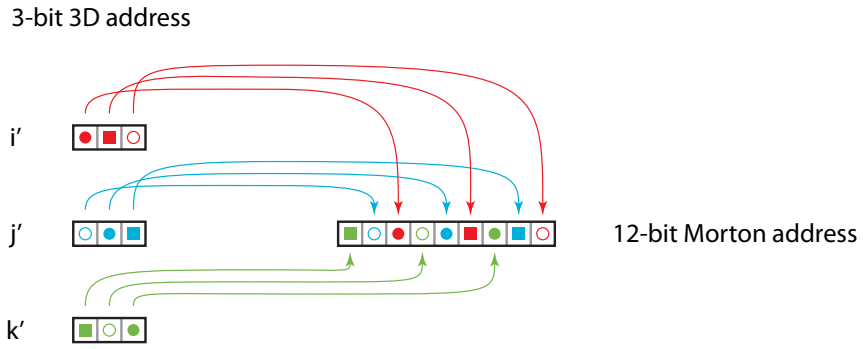


Figure 3.9: Morton address computation in 3D. The three-bit integer memory addresses get swizzled for a 12-bit address into the data array. In essence, this illustrates the operation $\Big(swzl(i') + [swzl(j') \ll 1] + [swzl(k') \ll 2]\Big)$.

entries of 16-bit integers requiring a total of 64 bytes of storage. In the current processor architectures, this look-up table fits in exactly one cache line providing the best memory performance possible. Computing the Morton index can be accelerated further by looking at $B_{(i,j,k)}$ in Equation (3.9), which represents the memory location of the first voxel in the brick containing the voxel at (i, j, k). Clearly, this memory location is the same for all of the voxels inside that brick. Extrapolating further, this memory location is the same for all of the samples that lie within that brick. Hence, during rendering, this quantity must be computed only once per brick.

One issue with using a non-linear order is that brick dimensions must be equal to each other and be a power of two due to the recursive nature of this kind of non-linear indexing. If non-square dimensions are desired, the bricks should be divided into cubic sub-bricks with dimensions that are powers of two. Padding the original dataset with zeros is necessary if the data dimensions are not divisible by brick dimensions.

3.5.2 Fast Data Index Computation

When the volume data is stored in a linear order, the voxel location can be converted into a memory index via Equation (3.7). However, this formula applies only to one data location at a time. To get a value of a sample via interpolation, one needs to access eight voxels neighboring the sample. For a non-bricked dataset, all of the required indices can be computed in the following way, [GBKGl04b]:

$$\begin{aligned}
I^{lin}_{(i,j,k)} &= i + D_x(j + k \cdot D_y) & I^{lin}_{(i,j,k+1)} &= I^{lin}_{(i,j,k)} + D_x \cdot D_y \\
I^{lin}_{(i+1,j,k)} &= I^{lin}_{(i,j,k)} + 1 & I^{lin}_{(i+1,j,k+1)} &= I^{lin}_{(i,j,k)} + D_x \cdot D_y + 1 \\
I^{lin}_{(i,j+1,k)} &= I^{lin}_{(i,j,k)} + D_x & I^{lin}_{(i,j+1,k+1)} &= I^{lin}_{(i,j,k)} + D_x(D_y + 1) \\
I^{lin}_{(i+1,j+1,k)} &= I^{lin}_{(i,j,k)} + D_x + 1 & I^{lin}_{(i+1,j+1,k+1)} &= I^{lin}_{(i,j,k)} + D_x(D_y + 1) + 1,
\end{aligned} \tag{3.10}$$

where $D_{\{x,y,z\}}$ are the dimensions of the volume. The basic approach is to compute the index of a voxel near the sample at $\mathbf{x}$, which in three dimensions is $(i = \lfloor \mathbf{x}_x \rfloor, j = \lfloor \mathbf{x}_y \rfloor, k = \lfloor \mathbf{x}_z \rfloor)$. This voxel is located at the D_{000} relative to the sample, shown in Fig. 2.6, which is the front lower left corner of the cuboid formed by eight neighboring voxels. The indices for the rest of the neighbors can be calculated by adding offsets, which are the same regardless of sample location.

In the case where data is bricked, this technique works well within the interior of a brick but needs to be updated to incorporate the rearrangement of data via Equation (3.8). First, one computes the index of the first voxel of a brick containing the current sample. Then the method of adding offsets is applied to compute the memory locations of voxels within the brick. Unfortunately, the offsets differ when accessing neighbors for voxels located at the brick edges. Hence these memory indices must be recomputed every time, which introduces a branch in the memory index computation to test whether offsets can be used.

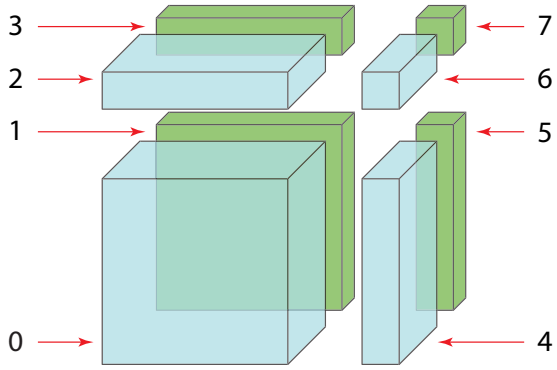


Figure 3.10: Case classification of brick voxels used in the interpolation look-up table. The first sample of the brick is in the front lower left corner and the last is in the back top right corner. The separation distinguishes edge cases.

The execution of this branch stalls the processor and slows down the rendering.

Grimm et. al. remedied the issue of processor stalls by introducing an interpolation look-up table, [GBKGl04b]. Because the sample location directly provides the front lower left corner of the voxel cuboid, only seven offsets need to be stored. The authors replaced the branch by reading appropriate offsets from the table. These offsets are categorized based on whether a sample lies near an edge of a brick or not. Because there are eight cases to consider, shown in Fig. 3.10, the table holds 56 integer offsets.

The cases are determined by classifying each voxel as on the inside or on the positive edge of the brick. Assuming the indices reside within the interval $[0, BD_{\{x,y,z\}} - 1]$ and letting $D^{-}_{\{x,y,z\}} = BD_{\{x,y,z\}} - 1$, Table 3.2 illustrates the computation of cases for all voxels inside a brick of dimension 32^3.

The computation of memory indices for neighboring voxels is updated to

Case	$i\&D_x^-$	$j\&D_y^-$	$k\&D_z^-$	i+1	j+1	k+1	$\frac{i\&\sim D_x^-}{BD_x}$	$\frac{j\&\sim D_y^-}{BD_y}$	$\frac{k\&\sim D_z^-}{BD_z}$	i+2j+4k
0	0-30	0-30	0-30	1-31	1-31	1-31	0	0	0	0
1	0-30	0-30	31	1-31	1-31	32	0	0	1	1
2	0-30	31	0-30	1-31	32	1-31	0	1	0	2
3	0-30	31	31	1-31	32	32	0	1	1	3
4	31	0-30	0-30	32	1-31	1-31	1	0	0	4
5	31	0-30	31	32	1-31	32	1	0	1	5
6	31	31	0-30	32	32	1-31	1	1	0	6
7	31	31	31	32	32	32	1	1	1	7

Table 3.2: Illustration of computing cases via $GetLUTIndex(i,j,k)$ used in the interpolation look-up table. The brick has 32 voxels in each dimension, $BD_{\{x,y,z\}}$= 32 and $D^-_{\{x,y,z\}}$=$BD_{\{x,y,z\}}$−1=31. Each successive i,j, or k is the result of the previous operation involving it. Method from [GBKGl04b].

use the look-up table in the following way:

$$\begin{aligned} I^{lin}_{(i,j,k)} &= i + D_x(j + k \cdot D_y) & J^{lut} &= GetLUTIndex(i,j,k) \\ I^{lin}_{(i+1,j,k)} &= I^{lin}_{(i,j,k)} + LUT[J^{lut}][0] & I^{lin}_{(i,j+1,k)} &= I^{lin}_{(i,j,k)} + LUT[J^{lut}][1] \\ &\vdots & &\vdots \\ I^{lin}_{(i,j+1,k+1)} &= I^{lin}_{(i,j,k)} + LUT[J^{lut}][5] & I^{lin}_{(i+1,j+1,k+1)} &= I^{lin}_{(i,j,k)} + LUT[J^{lut}][6], \end{aligned} \tag{3.11}$$

where $LUT[J^{lut}]$ is the interpolation look-up table and $GetLUTIndex(i,j,k)$ is a function to compute the case J^{lut} that the current sample falls under. The rest of the variables are defined in Table 3.1. The results of using $GetLUTIndex(i,j,k)$ for different locations within a brick are shown in Table 3.2. An important advantage of linear memory order is that all 56 offsets stay constant for all voxels in all bricks.

In the case when a non-linear brick order is used, the whole scheme needs to be updated. Offsets are used to skip to the beginning of the brick holding the appropriate neighbor voxel. The memory index of the voxel must be recomputed relative to the beginning of its containing brick. The updated computation is

shown below:

$$
\begin{aligned}
J^{b,lut} &= GetLUTIndex(i', j', k') \\
B'_{(i,j,k)} &= (BD_x \cdot BD_y \cdot BD_z) \cdot (i'' + BVD_x(j'' + k'' \cdot BVD_y)) \\
swzl3_{(i,j,k)} &= swzl(i') + [swzl(j') \ll 1] + [swzl(k') \ll 2] \\
I^{b,mor}_{(i,j,k)} &= B_{(i,j,k)} + swzl3_{(i,j,k)} \\
I^{b,mor}_{(i+1,j,k)} &= B_{(i,j,k)} + LUT[J^{b,lut}][0] + swzl3_{(i+1,j,k)} \\
&\vdots \\
I^{b,mor}_{(i+1,j+1,k+1)} &= B_{(i,j,k)} + LUT[J^{b,lut}][6] + swzl3_{(i+1,j+1,k+1)},
\end{aligned}
\tag{3.12}
$$

where the variables are defined in Table 3.1. Once again, the quantity $B'(i, j, k)$ is constant for all voxels inside a brick and the interpolation look-up table keeps the same 56 offsets for all bricks. The swizzling computation must be repeated for each neighbor, but can be accelerated via an additional look-up table, discussed in Section 3.5.1.

The principles discussed in this section can be applied to generate an offset look-up table to aid gradient computation. Because each gradient estimate needs voxels to the left and the right of the sample location, the necessary neighborhood increases to 32 voxels. Each voxel in a brick falls into one of three categories: left edge, inside, and right edge, which requires 27 cases for the look-up table. Out of 32 needed neighbor offsets, only 26 must be stored, since the eight nearest offsets are available through the interpolation look-up table. In the end, this new table stores 702 entries in addition to the previous 56. For more details, see [GBKGl04b].

3.5.3 Local Gradient Storage

As mentioned previously, gradient pre-computation is a viable acceleration technique. However, storing gradients on a global level requires extremely high memory storage, which limits the size of the dataset one can visualize. Also accessing gradients puts a strain on memory bandwidth.

Grimm et. al. introduced a per brick gradient cache in [GBKGl04a]. Their technique requires two data structures, a brick-sized gradient cache and a gradient validity bit cache. Instead of computing all gradients within a brick prior to rendering it, the authors compute only the necessary ones corresponding to the opaque portions of the brick. When a gradient is needed, the validity bit cache is checked for availability. If available, the gradient is fetched from the gradient cache; otherwise, it is computed and the corresponding validity bit is set to true. Once the rendering of the current brick is complete, both caches are reset.

3.5.4 Empty Space Skipping via Binary Histograms

Min-max octree methods use intervals to classify nodes as transparent. Consider a transfer function as a combination of intervals, each describing a separate element. A node is transparent if the intersection of its $[min, max]$ interval with all of the transfer function intervals is empty. The summed area table approach, briefly described in Section 3.5, may also be used to accelerate this process, but is not very effective as it can lead to misclassification of transparent nodes with wide data ranges.

This issue is fixed by improving the accuracy of data representation. Grimm et. al. replaced intervals with quantized binary histograms in [GBKGl04a, GBKGl04b], which are defined as:

$$\sigma_A(i) = \begin{cases} 1, & i \in A \\ 0, & otherwise \end{cases}, \tag{3.13}$$

where i is the data value to test and A is the set of all data values within a brick. To represent all data inside a brick perfectly, the width of this histogram would be equal to the range of volume data values. This is not memory effecient, so the authors quantize the histograms. For CT scans, we use a bin size of 64, which subdivides the data range of $[0, 4095]$ into 64 bins. These histograms fit nicely within 64-bit integers where each bit represents a boolean bin. This can be written as:

$$\sigma_A(i) = \begin{cases} 1, & \exists\, x, \text{s.t. } x \in A,\ x \in [64 \cdot i, 64 \cdot (i+1)) \\ 0, & otherwise \end{cases}. \tag{3.14}$$

In this case, $i \in [0, 63]$ selects the bin for consideration of the histogram representing data in a brick A.

The transfer function must also be encoded with a quantized binary histogram which is based on the opacity of data:

$$\lambda(i) = \begin{cases} 1, & \exists\, x, \text{s.t. } \alpha_x \neq 0,\ x \in [64 \cdot i, 64 \cdot (i+1)) \\ 0, & otherwise \end{cases}, \tag{3.15}$$

where i is the data value to test and α_x is the opacity of the data value x. A brick is transparent if:

$$\forall\, i \in [0, 63],\ \sigma(i) \wedge \lambda(i) = 0. \tag{3.16}$$

This method is more sensitive to largely varying data values within a brick and allows for a quick brick transparency test requiring one 64-bit integer AND operation.

input : Volume data, brick dimensions $BD_{\{x,y,z\}}$, and output image resolution $imgRes_{\{x,y\}}$
output: Initialized bricks

```
1 loadVolume();
2 precompGradients();
3 initBricks(BD, imgRes);
4 createInterpLUT(BD);
```

Figure 3.11: Algorithm for the initialization of the bricked raycaster.

3.6 Bricked Raycaster Algorithm

In this section, we describe our single-core accelerated raycaster and provide the pseudocode for it. The algorithm outlined here uses only one level in the brick hierarchy. It can be extended to use more than one level in a fairly straightforward way, described in Section 3.5.

The initialization step, shown in Fig. 3.11, must be completed every time a new volume is loaded for visualization. The brick dimensions must be set prior to loading in case padding with zeros is necessary. The function `loadVolume()` rearranges the data into Morton order within bricks that are stored linearly. Next, if desired, `precompGradients()` can be used to compute gradients stored in the global cache. The function `initBricks()` initializes data structures used within the brick hierarchy, like ray lists and quantized binary histograms. Ray lists are cleared and the binary histograms are updated to represent the data within. The function `createInterpLUT()` fills the interpolation look-up table with the appropriate offsets recalculated based on the updated brick and data dimensions.

Now the bricked raycaster is set to produce output images, sometimes re-

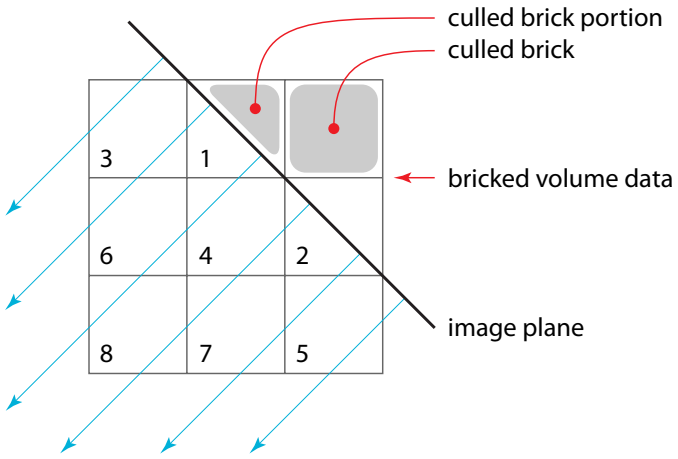


Figure 3.12: Correct brick order ensuring front-to-back integration of rays through the volume. The function `orderBricks()` from Fig. 3.11 creates this order and culls away bricks behind the image plane.

ferred to as frames. An image is re-rendered every time there is a change in the transfer function, viewing location or viewing direction. The inner core of the bricked raycaster is outlined in Fig. 3.13. The first step is correctly setting up the order in which to render bricks. This is achieved by the `orderBricks()` function. It casts viewing rays into the volume and sets their starting location just past the first intersection with a brick. Each brick also stores the references to the rays that pass through it. While rays are cast, the bricks are culled to the image plane and ordered based on the viewing direction, as shown in Fig. 3.12. This ensures that rays are integrated in the front-to-back order.

Next, the algorithm runs through the bricks in the list to be rendered. If a brick is deemed transparent via the quantized binary histogram, then all of its rays are propagated into the next brick they intersect. If any of these rays exit the volume, they are terminated. If the brick is opaque, then each of its rays is integrated through the brick based on two stopping criteria: either the ray reaches the opacity threshold and is terminated or it exits the brick.

```
input  : Output image resolution imgRes_{x,y} and viewing direction
         viewVec
output: Output image
1  outImg ← black;
2  brickOrderedList ← orderBricks(viewVec);
3  foreach curBrick in brickOrderedList do
4      if curBrick is opaque via binary histogram test then
5          foreach ray inside curBrick do
6              rayLoc ← location of ray;
7              rayColor ← color and opacity of ray;
8              while rayLoc inside curBrick and rayColor < opacity threshold do
9                  interpDens ← getDensityAt(rayLoc);
10                 interpGradient ← getGradientAt(rayLoc);
11                 rayColor ← shadeSample(interpDens) and
                   compsiteSample(interpGradient);
12                 increment rayLoc;
13             end
14             if rayLoc inside volume and rayColor < opacity threshold then
15                 advance ray to next brick;
16                 add ray to next brick;
17             end
18         end
19     else
20         foreach ray inside curBrick do
21             advance ray to next brick;
22             add ray to next brick if ray is inside volume;
23         end
24     end
25     remove all rays from curBrick;
26 end
27 return resulting output image outImg;
```

Figure 3.13: Single-core bricked raycaster algorithm that produces an output image given viewing direction, transfer function and opacity threshold.

Integration along a ray is approximated by adding color contributions from samples. At each sample location, `getDensityAt()` interpolates data and `getGradientAt()` approximates gradients via tri-linear interpolation. The resulting sample is classified via the transfer function and shaded using the Blinn-

Phong reflection model. This color, in the end, is composited to the ray's color and opacity before incrementing ray's location to the next sample.

Before moving onto the next ray, the algorithm checks whether the ray should be terminated. If not, it is advanced into the next brick it intersects. The final step before moving onto the next brick removes all ray references from the current brick. Once all bricks have been processed, the algorithm returns the computed output image.

In order to use the local gradient storage, the cache must be cleared prior to integrating the rays through the current brick (line 5 in Fig. 3.13). This cache would be used by the function `getGradientAt()` on line 10. Once again, the function `precompGradients()` in Fig. 3.11 is omitted in this case.

We have outlined all of the accelerations and showed how they fit into the raycaster framework. We evaluate their performance in the following section.

3.7 Intermediate Results

In order to evaluate the performance of the accelerations described in this chapter, we utilize a single processing core to render several datasets using different types of the basic bricked raycaster with an orthographic projection. A detailed description of the rendering cluster and further results can be found in Chapter 5.

We start by investigating the effect of the different brick sizes on rendering times in Section 3.7.1. In Section 3.7.2, we compare the effects of storing data linearly and in the Morton order. Then, we evaluate the performance of com-

puting gradients as they are needed, using a local gradient cache and a global gradient cache in Section 3.7.3. Finally, we evaluate the effect of rendering to different image resolutions in Section 3.7.4.

3.7.1 Brick Sizes

In this test, we compare how the brick sizes affect rendering times. Each of the tests in this section used one of the cores in the dual quad-core 2.66 GHz Intel® Xeon® *X*5355 machine with 16 GB of RAM. The test volume consisted of the first 128 slices of the Pre-Operation Dataset (Section 5.2) with a data resolution of $512^2 \times 128$ voxels. Note that only the tests in this section used a portion of the dataset, while all other tests used full datasets. To allow the asymmetric bricks to store their data in the Morton order, each is subdivided into smaller bricks of 8^3 voxels. These sub-bricks are arranged linearly within the bricks, which reside linearly in the main memory. The rays cast into the volume are integrated through one sub-brick at a time. We use the global gradient cache in these tests (the performance of the other schemes is compared in Section 3.7.3).

We consider the brick sizes as combinations of 16, 32, and 64 voxels per dimension. Fig. 3.14 shows the primary axes relative to the dataset to help visualize non-cubic brick sizes. For each brick size we render each of the four views (Fig. 3.15) ten times at the output image resolution of 1024^2. We plot the average rendering times in Fig. 3.16. The plot shows that using cubic bricks with dimensions of 32 or 64 voxels[2] without sub-bricks requires the least amount of time. The $32 \times 32 \times 32$ bricks with sub-bricks take 9.3 sec per image on average

[2]These are labeled as "solid 32^3" and "solid 64^3" in Fig. 3.16. They are cubic without sub-bricks, hence lacking the overhead associated with using sub-bricks.

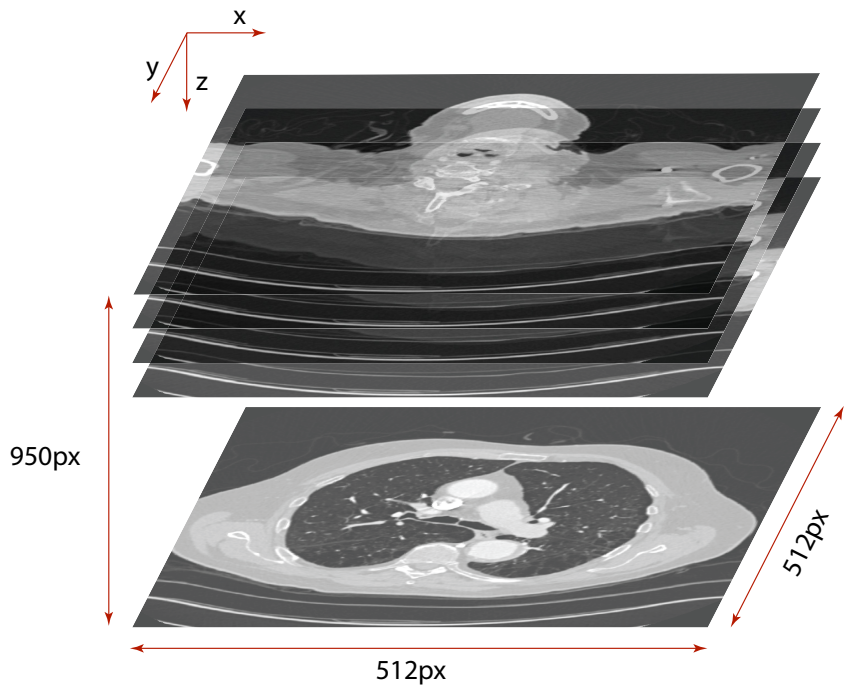
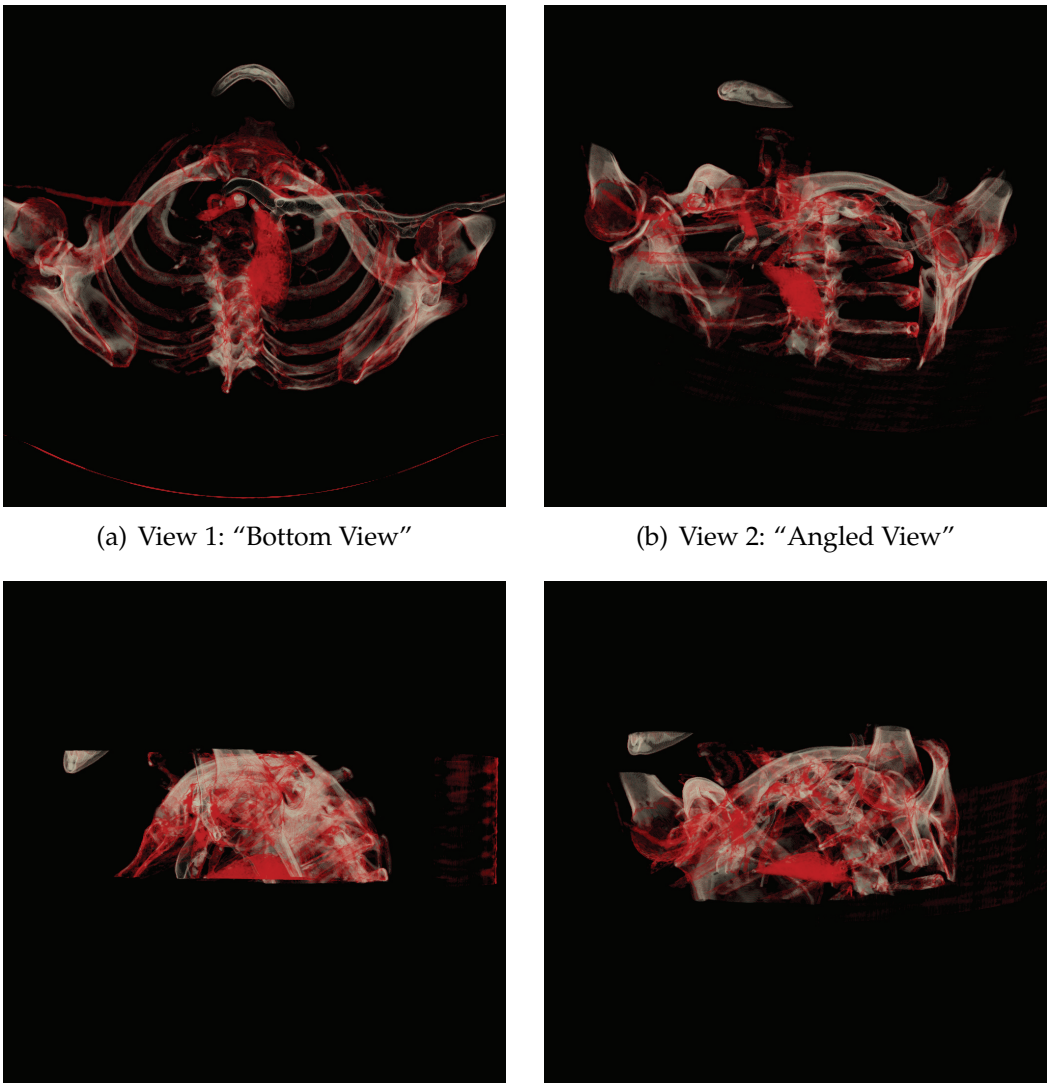


Figure 3.14: Primary axes relative to the volume dataset represented as a stack of images.

for all views, while the 32^3 bricks without sub-bricks average 4.9 sec per image, which is 1.9 times faster. Similarly, the $64 \times 64 \times 64$ bricks with sub-bricks take 17 sec on average per image and the 64^3 bricks without sub-bricks take 5.4 sec, which is 3.1 times faster.

Asymmetric bricks can benefit empty space skipping by exploiting directional similarity within the data encoded with narrower binary histograms. However, our test concludes that the overhead of using asymmetric bricks is higher than the benefits of improved space skipping. For example, bricks sized $16 \times 64 \times 64$, $16 \times 32 \times 64$ and $32 \times 16 \times 64$ exhibit high variation in average rendering times between the views. On the other hand, cubic bricks have no preferred rendering direction making the rendering times more consistent, an advantage for seamless interactive viewing.



(a) View 1: "Bottom View"

(b) View 2: "Angled View"

(c) View 3: "Side View"

(d) View 4: "Narrow Side Angle View"

Figure 3.15: Four views of the data to test the effect of brick sizes on rendering time. Each of these images has the resolution of 1024^2 and shows the first 128 slices of the Pre-Operation Dataset (Section 5.2) for a total volume of $512^2 \times 128$ voxels. The "Angled Side View" shows the dataset rotated 45° around its vertical axis, while the "Narrow Side Angle View" is the 45°-45°-45° corner view of the dataset.

We compute the memory requirements to store a brick of 16-bit voxel data and vector gradients in Table 3.3. This factor helps with identifying the brick size that is optimal to fit into the L2 cache of a processor core. Table 3.3 also shows the extra memory requirement associated with the entire single-level

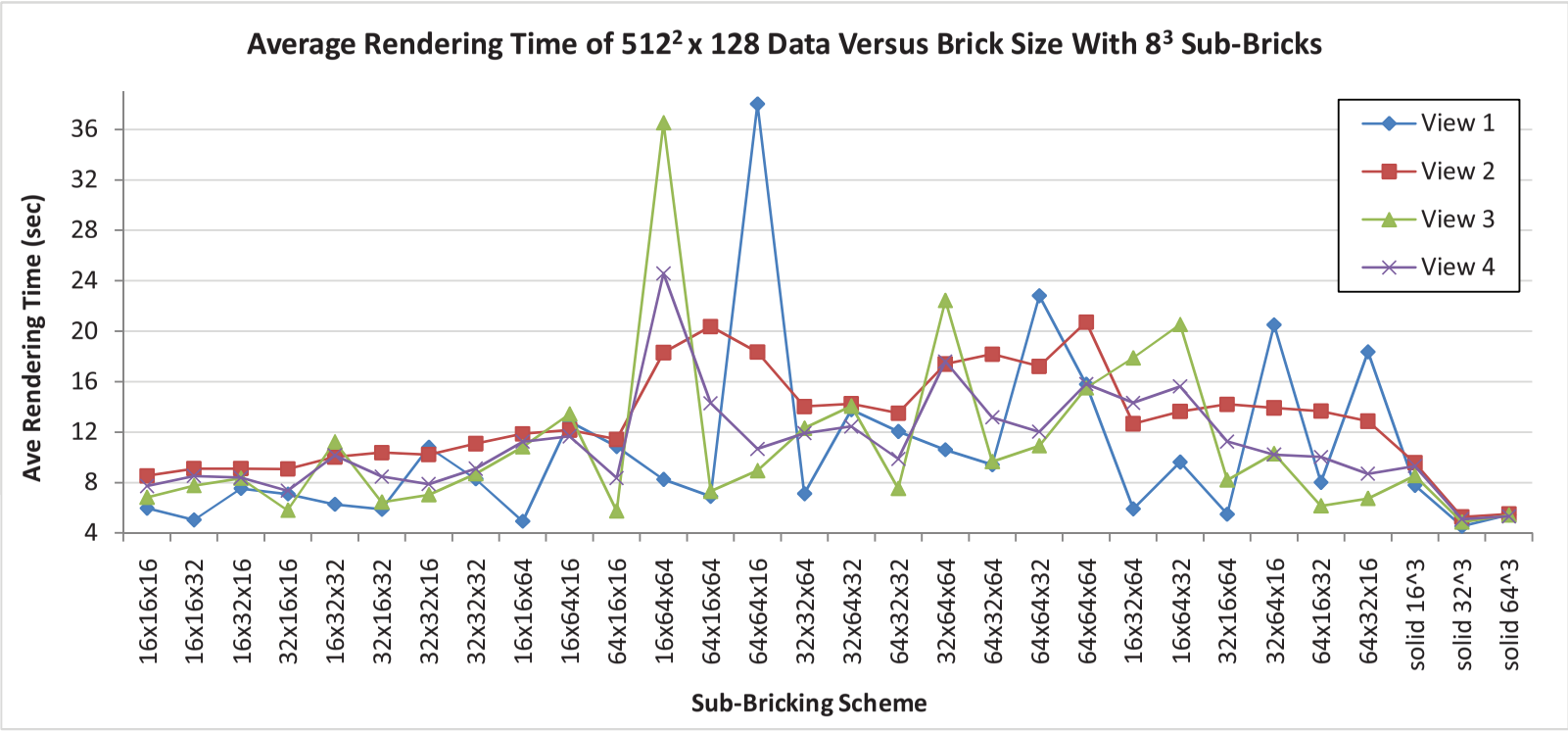


Figure 3.16: Rendering time comparison between various sizes of bricks divided into 8^3 voxel sub-bricks. There is a difference between "$32 \times 32 \times 32$" and "solid 32^3" because the latter uses no sub-bricks. Bricks of size 32^3 ("solid 32^3") and 64^3 ("solid 64^3") without subdivision perform better overall than any other scheme. Fig. 3.14 shows the axes relative to the dataset to help visualize the non-cubic brick sizes.

brick hierarchy, which includes the storage of 64-bit binary histograms, pointers to lists of rays inside each brick and the density lookup table with 56 integer entries.

For the best performance, the chosen brick size must balance between generating a low number of bricks (larger brick dimensions) and fitting brick data within the processor's L2 cache (smaller brick dimensions). In our test, bricks with 64^3 voxels each divide the volume into 128 pieces. However, storing the data within each brick requires 3.5 MB, which is more than the size of the L2 cache for a single core in our system (3 MB). As the result, poor cache coherency negatively impacts rendering times, which are on average 5.5 sec for a 1024^2 image per view.

Bricks containing 32^3 voxels require 448 KB of memory for storage and easily fit within the L2 cache of a core on our system (3 MB). Using bricks of this size subdivides our test volume into $1,024$ parts. This scheme produces the lowest rendering times of about 4.9 sec for a 1024^2 image per view. This is nine percent faster than using 64^3 bricks without sub-bricks. As the result, bricks with 32^3 voxels and no sub-bricks are used in the remainder of this work.

3.7.2 Data Storage Order

This test focuses on the impact that the data storage order has on the rendering times. We consider storing the data within the bricks linearly and in Morton order, while the bricks themselves are arranged in the linear order. We render three entire datasets 30 times from each of the five viewing directions to produce an image at the 1024^2 resolution. This test used one core of the 2.83 GHz Intel®

Dim	# Voxels	Voxel Mem	Voxel & Gradient Mem	Bricks in $512^2 \times 128$	Mem Overhead	% of Dataset
8	512	1,024 B (1 KB)	7,168 B (7 KB)	65,536 $64 \times 64 \times 16$	1,048,800 B	1.56283%
16	4,096	8,192 B (8 KB)	57,344 B (56 KB)	8,192 $32 \times 32 \times 8$	131,296 B	0.19565%
32	32,768	65,536 B (64 KB)	458,752 B (448 KB)	1,024 $16 \times 16 \times 4$	16,608 B	0.02475%
64	262,144	524,288 B (512 KB)	3,670,016 B (3.5 MB)	128 $8 \times 8 \times 2$	2,272 B	0.00339%
128	2,097,152	4,194,304 B (4 MB)	29,360,128 B (28 MB)	16 $4 \times 4 \times 1$	480 B	0.00072%

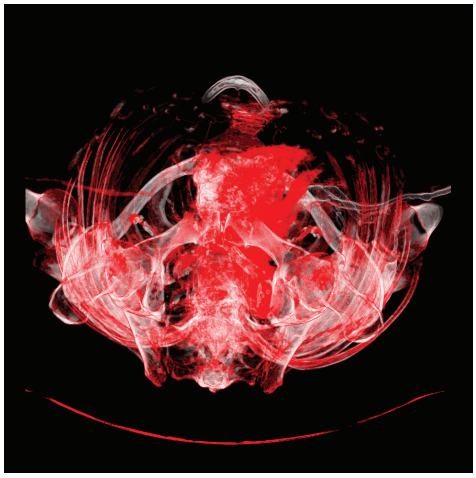
Table 3.3: Memory costs associated with cubic bricks of different sizes. "Dim" is the cubic brick dimension and "# Voxels" is the number of voxels that fit within the brick. "Voxel Mem" refers to the memory required to store 16-bit voxels within a brick, while "Voxel & Gradient Mem" considers the storage of gradient vectors as well. There are helpful in identifying the brick size that optimally fits into the L2 cache of a processing core. "Bricks in $512^2 \times 128$" shows the total number of bricks inside the $512^2 \times 128$ volume. "Mem Overhead" is the total memory requirement for the entire single-level brick hierarchy and includes binary histograms, ray pointers and a density lookup table. The final column, "% of Dataset," shows the size of the hierarchy relative to the dataset with $512^2 \times 128$ voxels.

Xeon® E5440 processor with 16 GB of RAM. For more details on the rendering system, see Section 5.1, and for the descriptions of the datasets, see Section 5.2.

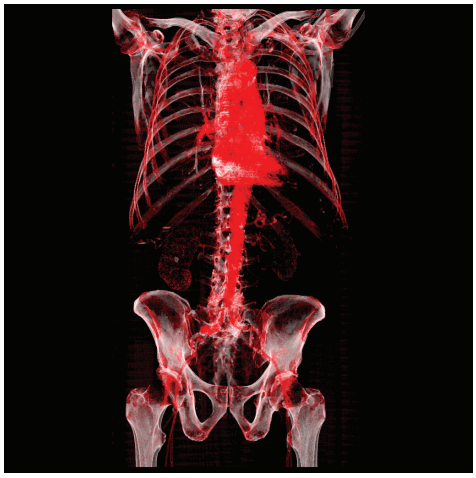
The five rendering views of the Pre-Operation Dataset are shown in Fig. 3.17. The "45° Side View," Fig. 3.17(d), shows the dataset rotated 45° around its vertical axis so that only the vertical edges of the volume are parallel to the image plane. The "45° Corner View," Fig. 3.17(e), shows the dataset rotated 45° around both its vertical and horizontal axes, so that all of its edges are at an angle to the image plane. Similar renderings of the other datasets can be found in Section 5.2.

The averages of the rendering times are shown in Fig. 3.18. Relative differences in the rendering times between using linear and Morton data storage orders are around five percent. Based on this experimental data, one cannot draw firm conclusions about which storage order is most beneficial. However, the work by Mishchenko conclusively shows that the cache coherency of accessing data stored in Morton order is better than linear order, [Mis06]. We return to this question later when we consider parallelization results in Section 4.3.

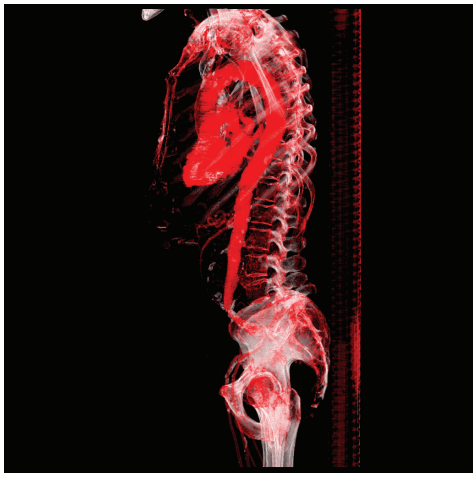
Most importantly, the average rendering times of the bricked raycaster are much smaller than the brute-force method (within 20 – 50% depending on the view). Fig. 3.18 refers to the brute-force algorithm as "Unbricked." It uses no bricking scheme and stores the entire dataset linearly in memory. The test algorithm integrates each ray through the volume entirely prior to continuing to the next. The algorithm still utilizes empty space skipping, global gradient cache and early ray termination for each ray. As a result, we can claim that using a bricking scheme improves rendering speeds.



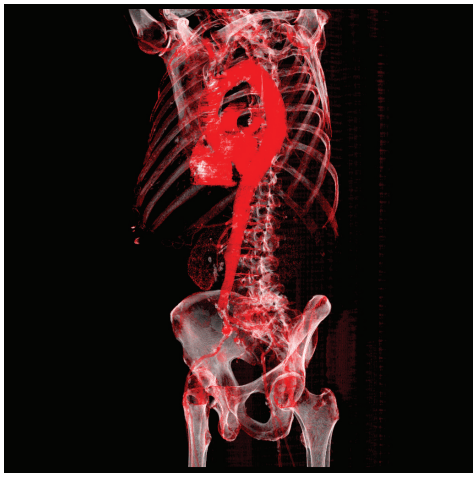
(a) “Bottom View”



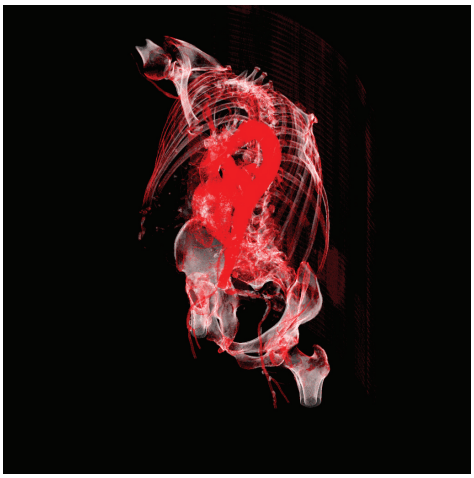
(b) “Front View”



(c) “Side View”



(d) “45° Side View”



(e) “45° Corner View”

Figure 3.17: Test views of the Pre-Operation Dataset (Section 5.2).

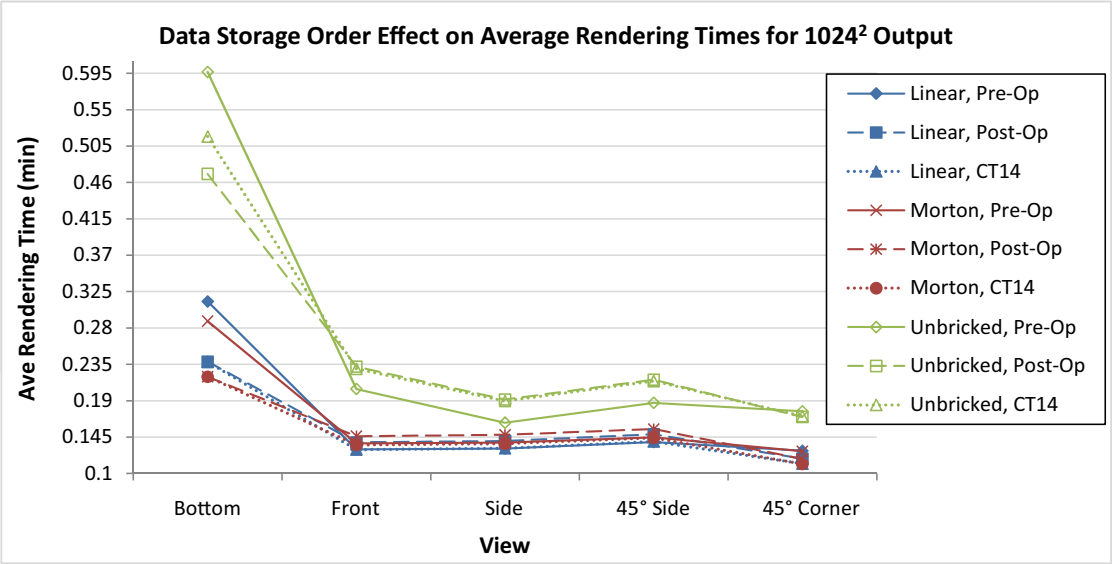


Figure 3.18: The effect on the rendering times due to data storage order for several datasets. The bricked raycaster used 32^3 bricks and global gradient cache to generate the output images at 1024^2 resolution. The "Unbricked" label refers to the brute-force raycaster without the brick hierarchy. The effect of using the gradient cache is evaluated in Section 3.7.3. The tests utilized a single core of the Intel® Xeon® E5440 processor at 2.83 GHz.

3.7.3 Gradient Pre-Computation

In this test we compare three ways of computing gradients during rendering: global pre-computed gradient cache (Section 3.2), local (brick) pre-computed gradient cache (Section 3.5.3) and per sample computation. Each of these methods is implemented within the bricked raycaster using 32^3 bricks and one ray sample per cuboid of voxels. The data was stored in the Morton order within each brick. We render three datasets 30 times from each of the five viewing directions to produce an image at the 1024^2 resolution. For computation, we use one core of the quad-core Intel® Xeon® E5440 processor at 2.83 GHz.

The averages of rendering times are shown in Fig. 3.19. Based on the experimental data, the global pre-computed gradient cache method provides render-

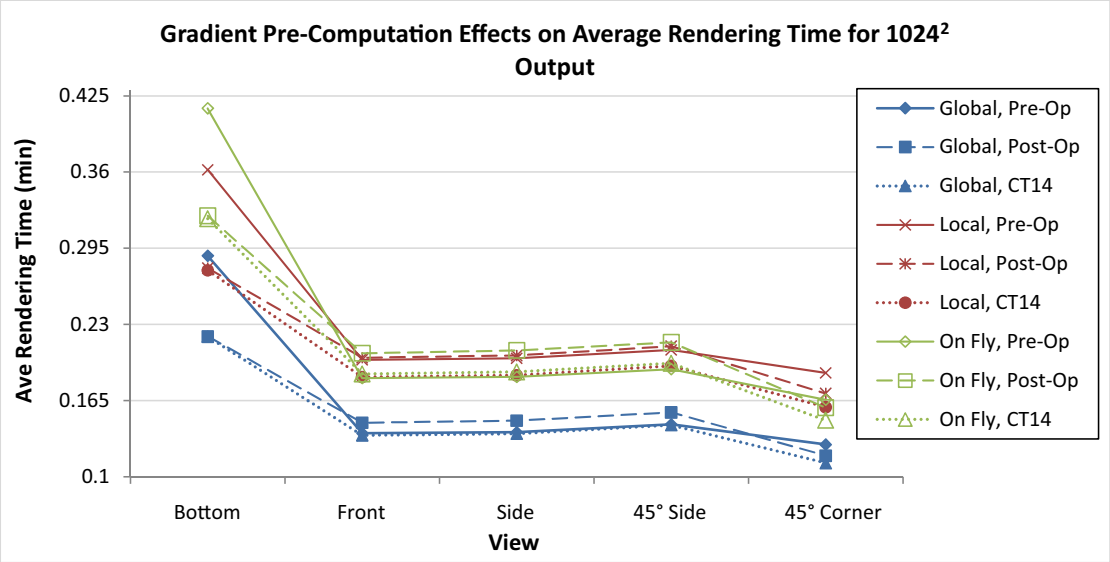


Figure 3.19: The effect on rendering times due to gradient pre-computation schemes for several datasets. To generate the output images at 1024^2 resolution, the bricked raycaster uses 32^3 bricks and one ray sample per cuboid of voxels. The data is stored in Morton order. The tests utilized a single core of the Intel® Xeon® E5440 processor at 2.83 GHz.

ing speeds that are 35 – 40% faster than other methods. However, as was mentioned earlier, this scheme has extremely high memory requirements. Hence, local gradient cache is best when memory is a limited resource preventing the use of a global gradient cache. The local gradient cache performed 0 – 5% faster than computing gradients as needed. As the result of this test, global gradient cache is used in the remainder of the algorithms we test.

3.7.4 Multiple Resolutions

In this test, we assess the image scalability of the single-core bricked raycaster with a global gradient cache and data stored in the Morton order. The brick size is set to 32^3 and we use one ray sample per cuboid of voxels.

In order to evaluate the scalability regarding output image size, we consider a special rendering time ratio r instead of the average rendering time. This ratio measures an increase in the average rendering time in terms of an increase in the resolution of the output image. First, consider ratio $r'(x, 256)$, as a fraction between the average time to render an image, $t(x)$, and the average time to render a 256^2 image, $t(256)$:

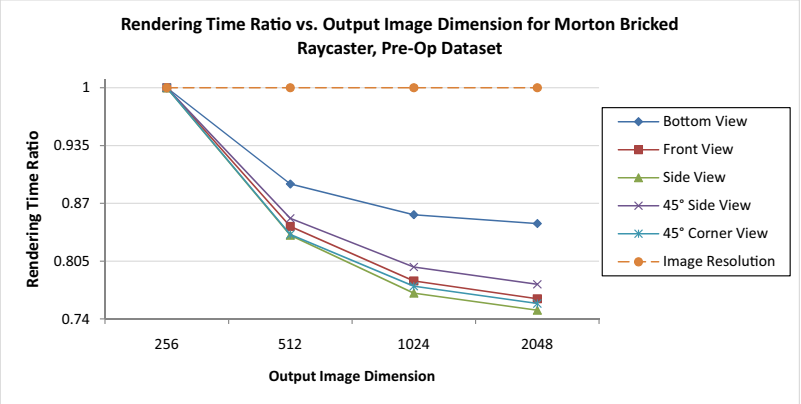
$$r'(x, 256) = \frac{t(x)}{t(256)}. \tag{3.17}$$

Define r_i to be an increase in the resolution of the output image from 256^2. For example, this ratio is $r_i(1024, 256) = 16$ when the image resolution increases from 256^2 to 1024^2. Using both of the ratios defined above, we can define the special rendering time ratio r as

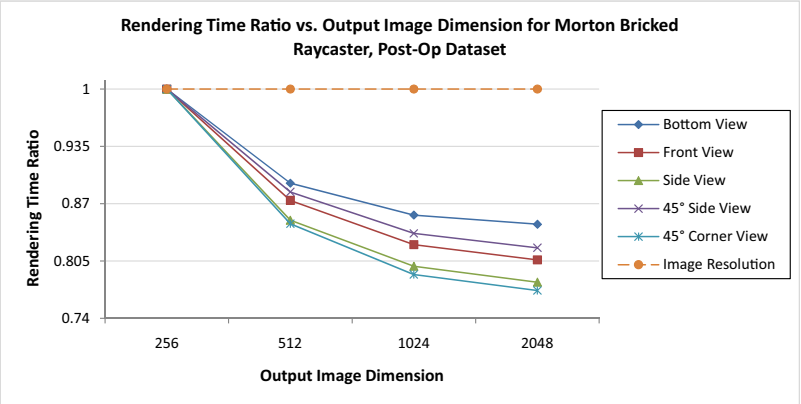
$$r(x, 256) = \frac{r'(x, 256)}{r_i(x, 256)}. \tag{3.18}$$

This ratio measures the rendering cost associated with increasing the output image resolution in terms of the increase in that resolution. In other words, if the ratio r is constant, then the cost of rendering a larger image is linearly proportional to the increase in the resolution of the output image (as a square of the single dimension of the image).

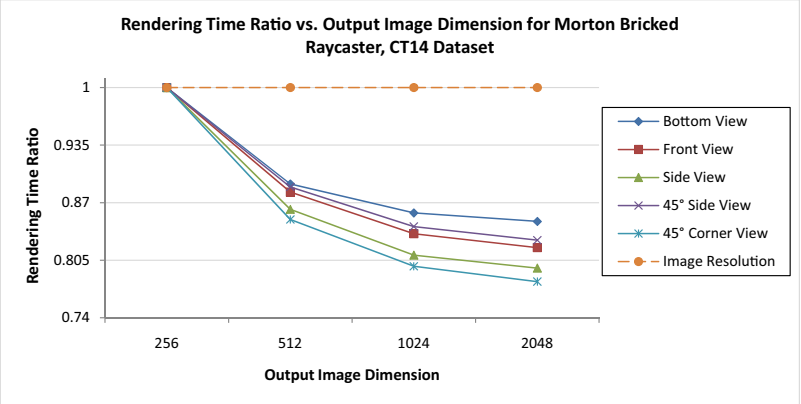
The averages of the rendering times are shown in Fig. 3.20. The trends between different views and output resolutions are similar across the test datasets. The rendering times for our single-core bricked raycaster do not increase linearly with the increase in output image resolution. This behavior suggests desired scalability in terms of output resolutions. We evaluate this premise again for the parallelized raycaster in Chapter 5. Most imporantly, this test outlines the benefits of using multiple resolutions for visualization. Because rendering at a lower resolution can be created an order of magnitude faster, it allows the user to quickly select the area of interest in the dataset before rendering it at a



(a) Pre-Operation Dataset



(b) Post-Operation Dataset



(c) CT14 Dataset

Figure 3.20: Rendering time ratio due to changes in the output image resolution for several datasets. This ratio measures an increase in the average rendering time in terms of an increase in the resolution of the output image, Equation (3.18). We used the bricked raycaster with 32^3 bricks and one ray sample per cuboid of voxels. The tests utilized a single core of the Intel® Xeon® E5440 processor at 2.83 GHz.

higher resolution. Such interactivity is essential to a productive exploration of data.

3.8 Summary

This chapter has focused on acceleration techniques for direct volume rendering algorithms and, in particular, raycasting. We started with important concepts, such as early ray termination, empty space skipping and gradient precomputation. We combined them with the brick hierarchy to create the single-core accelerated raycaster. We quantified the benefits of the approaches in order to select which of the acceleration techniques are best.

Based on the results in Section 3.7, we can conclude that a bricked raycaster with Morton data storage order, brick size of 32^3, early ray termination and global gradient cache produces the lowest rendering times. To provide rapid data exploration, multiple rendering resolutions have been used as well. However, because the rendering times are on the order of minutes per image, we consider parallelization techniques in Chapter 4.

CHAPTER 4

PARALLELIZING OUR BRICKED RAYCASTER

The previous chapter showed the need to lower the computational complexity of the raycaster and described several acceleration techniques. However, their use alone is not enough to achieve rendering times within the real-time domain. As the result, our bricked raycaster must be parallelized. We consider this parallelization in two tiers. At the lowest level, the parallelization is based on a shared-memory architecture made up of several processing cores. The higher level tier considers a distributed memory platform made up of shared-memory multi-core nodes. The ability to parallelize on both levels allows the algorithm to run on the current high performance clusters.

This chapter discusses the parallelization of our bricked raycasting algorithm to utilize a shared-memory processor with several computational cores. In this thesis we refer to such a machine as a node and consider its architecture to be schematically similar to the one shown in Fig. 4.1. This model is essential because the next generation of general purpose processors will utilize tens of processing cores, like Intel® Larrabee [SCS+08]. To parallelize the algorithm further, we discuss the utilization of several nodes comprising a distributed memory architecture.

The overall parallelization brings out two issues: data coherency and the balance between memory bandwidth and computational power. Data coherency includes cache coherency within a multi-core processor, as described in Chapter 3. (The lower level tier). Parallelization for a cluster of nodes requires similar coherency on the level of node's main memory (RAM) instead of the multi-core processor caches. (The higher level tier). In other words, if a node needs cer-

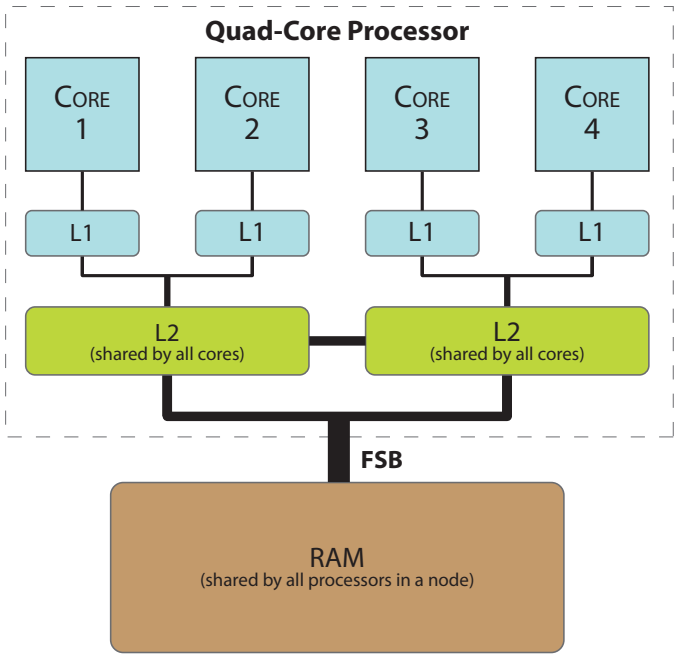


Figure 4.1: A typical quad-core processor architecture. Newer architectures add an L3 cache in between the L2 cache and RAM.

tain data that it doesn't have, a time penalty is incurred by retrieving data from another node. For the bricked raycaster, this means that the brick size must be chosen such that bricks fit into processor caches and the bricks higher in the hierarchy fit into the main memory as well.

The balance between memory bandwidth and computational power can be demonstrated best with an example that considers gradients for volume visualization. There are two ways of computing the gradients: as needed during rendering or as a pre-process that stores them to be accessed later. Fast processors with insufficient memory bandwidth may perform poorly with the precomputed approach because the memory bandwidth becomes the bottleneck for the system. Computing gradients as they are needed may improve the overall system performance since it might take less time to compute a gradient than to fetch it from the main memory. Therefore, the best system optimally balances

between requiring immense processing power and high memory bandwidth.

Section 4.1 discusses several methods that distribute the computational workload of the bricked raycaster between different cores of one node. The initial attempt, discussed in Section 4.1.1, distributes rays to different cores while rendering a particular brick. Another method, discussed in Section 4.1.2, assigns rendering of different bricks to different cores. Output image subdivision is discussed in Section 4.1.3. Then, we outline several parallelizations for multiple nodes: subdividing in image space, Section 4.2.1, and a hybrid subdivision in both image and data spaces, Section 4.2.2. We evaluate the performance of the proposed techniques in Section 4.3 and conclude in Section 4.4.

4.1 Single Multi-Core Node With Shared Memory

In this section, we consider parallelizing the algorithm to use all of the processing cores available to a single node. This is necessary prior to discussing changes to the algorithm necessary to utilize a cluster of nodes.

4.1.1 Subdividing Rays in a Brick

The heart of our bricked raycaster consists of integrating rays through each brick while maintaining a specific brick order. This order is essential to ensure the correct front-to-back integration of rays. If the order was arbitrary, compositing ray samples becomes non-trivial. The simplest parallelization avoids this issue by distributing the rays within a brick currently being rendered, as illustrated in Fig. 4.2. The algorithm is updated such that each core fetches a new ray to

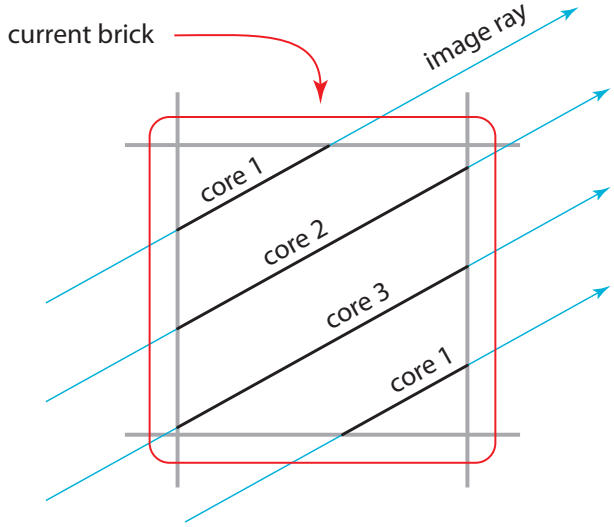


Figure 4.2: Integrating in parallel through rays within a brick. In this 2D case, the four rays in a brick are integrated by three different cores.

integrate after completing the integration of the previous one. In essence, this method parallelizes the `foreach` loop of the bricked raycaster, line 5 of Fig. 3.13.

Simple as it may be, this approach has poor overall scalability resulting from several issues. On the lowest level, there are typical load balancing difficulties originating from the number of samples varying between rays. At the higher level, the major contributor to the scalability issues is Amdahl's Law, [HP94]. In essence, the speedup of the execution time for a program using multiple processors in parallel is limited by the time used by the sequential fraction[1]. Even though the integration through each brick is accelerated via parallelism, the necessary preliminary computations are still sequential. The majority of this computation involves the function `orderBricks()` which is responsible for ordering the bricks and recording the first image ray-brick intersections.

[1]Consider a program that requires five minutes to complete, out of which one minute is spent in the sequential fraction of that program. Due to Amdahl's Law, the lowest possible execution time is one minute. Hence the parallelization speedup is limited to $\frac{5\,\text{min}}{1\,\text{min}} = 5$.

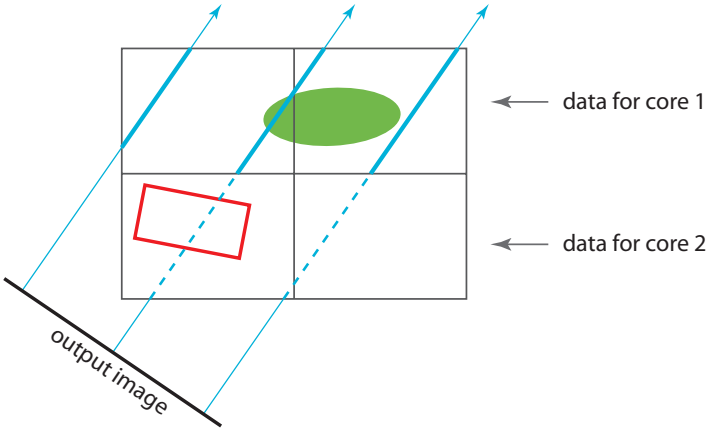
4.1.2 Volume Data Subdivision

Because volume data is independent, subvolumes can be rendered in parallel. This requires a volume subdivision scheme, like our brick hierarchy, to be leveraged for assigning subvolumes to processing cores. There are two paradigms to consider: static and dynamic data assignment. The static method, used in this work, assigns pre-determined subvolumes to each processing core for the duration of the dataset visualization. The dynamic method feeds bricks yet to be rendered to free cores.

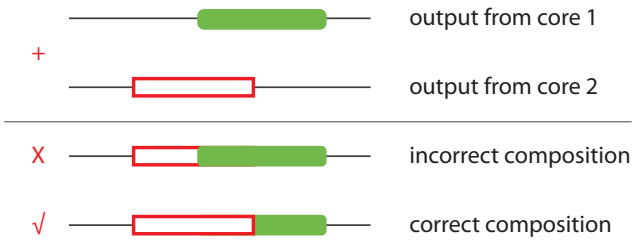
However, the bricks are not completely independent from each other because samples near their edges require data from the neighboring bricks during interpolation and gradient estimation. This can be resolved by "padding" each brick with neighboring data or by fetching data from the neighboring bricks as needed. We use the latter approach for the multi-core parallelization because all of the necessary data lies in memory within the same node and can be accessed fully by any of its cores.

During the rendering, each processing core uses the full bricked raycaster to visualize its own part of the volume. As the result, there are multiple output images that must be composited correctly to form the final image. Fig. 4.3 illustrates the application of this technique to a 2D volume rendered with two cores. Fig. 4.3(a) shows the volume embedding a rectangle and an oval. The top portion of the volume is rendered by core one and the bottom is rendered by core two. Separate output from each core, shown in Fig. 4.3(b), must be composited in the appropriate order to produce the correct final image.

Correct composition depends on several factors. First, rays must pass



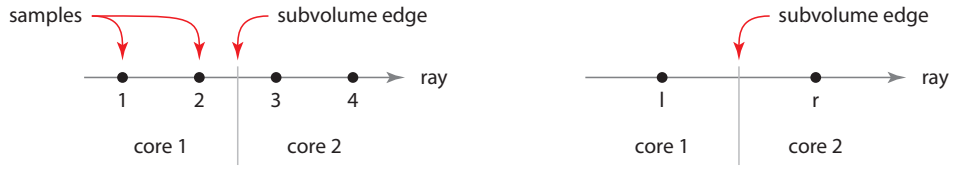
(a) The 2D volume to be integrated



(b) Integration output and comparison of the possible compositions

Figure 4.3: Parallelization by dividing the volume between different processing cores, each computing an output image. These must be combined in the appropriate order to produce the final image.

through the volume contiguously, so that subdividing the data merely splits them into several sections at the subvolume boundaries, as shown in Fig. 4.3(a). This implies that the distance between ray samples must remain constant across subvolume boundaries. Also, the ray's opacity must be accumulated correctly, which necessitates a specific compositing order. Splitting a ray into two parts rendered separately is equivalent to a ray with two samples, which can be derived mathematically. Samples along a ray are composited in the manner shown



(a) A ray is split into two parts (b) Equivalent ray with two samples

Figure 4.4: A ray split into two parts is equivalent to a ray with two samples resulting from integrating the parts separately. The correct compositing ensures that the final ray color and transmissivity are equal to the computation without splitting the ray apart.

in Equation 2.13:

$$\begin{aligned} I(L) &= C_n\alpha_n + t_n\Big(C_{n-1}\alpha_{n-1} + t_{n-1}(\ldots(C_1\alpha_1 + t_1 I_0)\ldots)\Big) \\ t_i &= 1 - \alpha_i, \end{aligned} \tag{4.1}$$

where α_i and C_i are the opacity and the color of the i^{th} sample respectively, and I_0 is the background color. Consider a ray with four samples and $I_0 = 0$, Fig. 4.4(a). By integrating along the ray in the front-to-back order, we get the following net color and transmissivity:

$$\begin{aligned} C_{net} &= C_1\alpha_1 + C_2\alpha_2(1-\alpha_1) + C_3\alpha_3(1-\alpha_2)(1-\alpha_1) + \\ &\quad + C_4\alpha_4(1-\alpha_3)(1-\alpha_2)(1-\alpha_1) \\ t_{net} &= (1-\alpha_4)(1-\alpha_3)(1-\alpha_2)(1-\alpha_1). \end{aligned} \tag{4.2}$$

We can also integrate each part of the ray separately, Fig. 4.4(b):

$$\begin{aligned} C_l &= C_1\alpha_1 + C_2\alpha_2(1-\alpha_1) \quad t_l = (1-\alpha_2)(1-\alpha_1) \\ C_r &= C_3\alpha_3 + C_4\alpha_4(1-\alpha_3) \quad t_r = (1-\alpha_4)(1-\alpha_3). \end{aligned} \tag{4.3}$$

We can composite the left color-transmissivity pair (C_l, t_l) with the right one (C_r, t_r) in two ways, depending on the order:

$$\begin{aligned} C_{lr} &= C_l + t_l C_r = C_{net} \quad t_{lr} = t_l t_r = t_{net} \\ C_{rl} &= C_r + t_r C_l \neq C_{net} \quad t_{rl} = t_l t_r = t_{net}. \end{aligned} \tag{4.4}$$

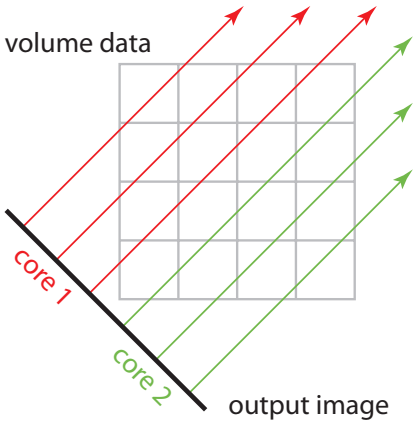


Figure 4.5: Subdividing the output image into parts to be rendered separately by each processing core. This is a 2D example using two cores that integrate their rays through the full volume independently.

The result C_{lr} is correct because it keeps the compositing order the same as the ray integration order, which in our case is front-to-back.

In the end, the bricked raycasting algorithm is updated such that each core renders its subvolume completely as if it were the full volume. However, the algorithm must allow ray samples near the subvolume edges to have access to the voxels which lie in a neighboring subvolume. Finally, the processing cores composite their results into the final image in parallel before returning it to the user.

4.1.3 Output Image Subdivision

Output image rays are inherently independent from each other and can be computed in any order allowing good scalability. Thus, each processing core renders a separate subimage as shown in Fig. 4.5.

Once again, there are two ways to assign output image portions to cores. The

static method does not scale well when the volume to be visualized occupies a small portion of the output image. A bottleneck is created when only a few cores perform most of the work.

A more efficient dynamic method subdivides the output image into small portions resulting in more pieces than there are processing cores. In our implementation, the image is divided into twice as many pieces as there are cores which allows for good load balancing. Processing cores automatically request the next subimage after they have finished rendering their previous one.

A way to balance the workload between processing cores further is to record the rendering time for each subimage and recursively divide them to equalize the rendering time. A quadtree or a KD-tree can be utilized to achieve this, but this approach is not taken in this work.

Updating the bricked raycaster to work with this parallelization requires minimal changes. Each processing core uses the original algorithm to render the whole volume into its subimage. Because of the pixel independence, all cores can simultaneously write into the output image.

4.2 Multiple Nodes Forming a Cluster

The previous section focused on parallelizing the bricked raycaster for multi-core processor architectures with shared memory. This section discusses the parallelization necessary to utilize a cluster comprised of many nodes with multi-core processors. This two-tier structure eliminates the need to manage shared memory within a system with many processors. Using a cluster of ma-

chines can also suggest algorithm performance on the future processors with tens of computational cores.

4.2.1 Output Image Subdivision

This technique is a natural extension of the image subdivision used by multi-core parallelization in Section 4.1.3. The output subimages are rendered in a dynamic order with each assigned to the next available node. Subimage block size is chosen such that it is divisible by the maximum number of cores each node can utilize. Because each node uses an output image subdivision, the workload of the resulting two-tier system can be shown by a fairly simple diagram (Fig. 4.6). A benefit of this method is that it is impervious to node crashes, since all of the work is completed regardless of the number of servers available at any time.

4.2.2 Volume Data Subdivision

In this hybrid method, instead of subdividing the data in the two tiers similar to the previous approach, each node renders the full output image of its subvolume. These images must then be combined in the correct order to form the final output image by either the visualization workstation (client) or the rendering nodes in parallel. The data subdivision provides excellent scalability in terms of the dataset size because each node can store a piece as large as its main memory allows. Such spatial subdivision extends easily to render time-dependent 3D datasets. The overview of the process is shown in Fig. 4.7.

One problem with such a division method surfaces when visualizing a small

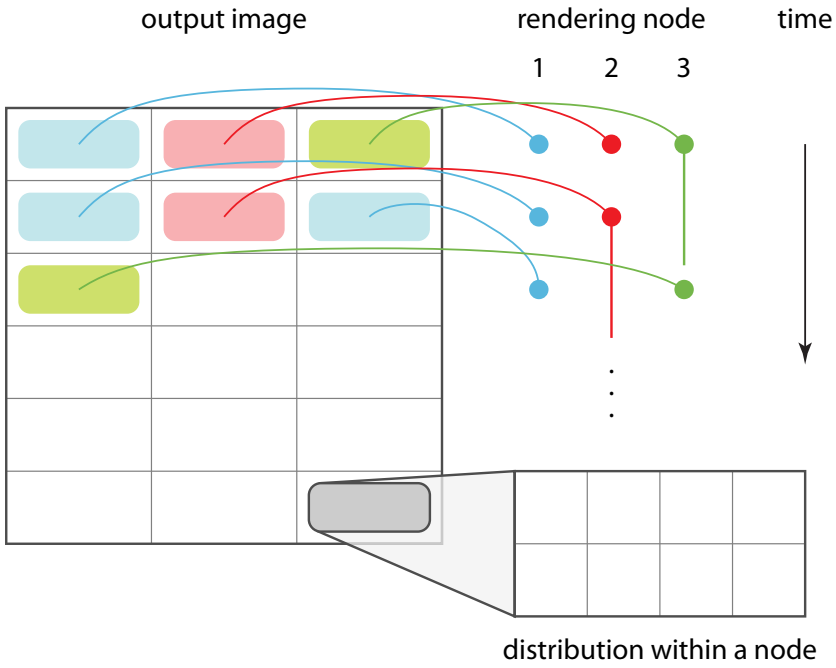


Figure 4.6: Subdividing the output image to be rendered separately by three rendering nodes. Each node is assigned a subimage shown on the right by circles. Rendering each subimage can take a different amount of time, signified by their vertical spacing. Each node treats its subimage as a full output image and renders it using the image subdivision technique of Section 4.1.3.

part of the dataset which resides within a subvolume for a specific node. One node becomes entirely responsible for rendering this subvolume and no acceleration by parallel computation can be expected. This issue may be resolved by dynamically rearranging the bricks between nodes.

4.3 Intermediate Results

We evaluate the performance of the techniques proposed in this chapter by considering the scalability of the rendering times in relation to the number of cores (or nodes) and the output image size. We render the test datasets from five separate views using an orthographic projection. During testing of multi-core al-

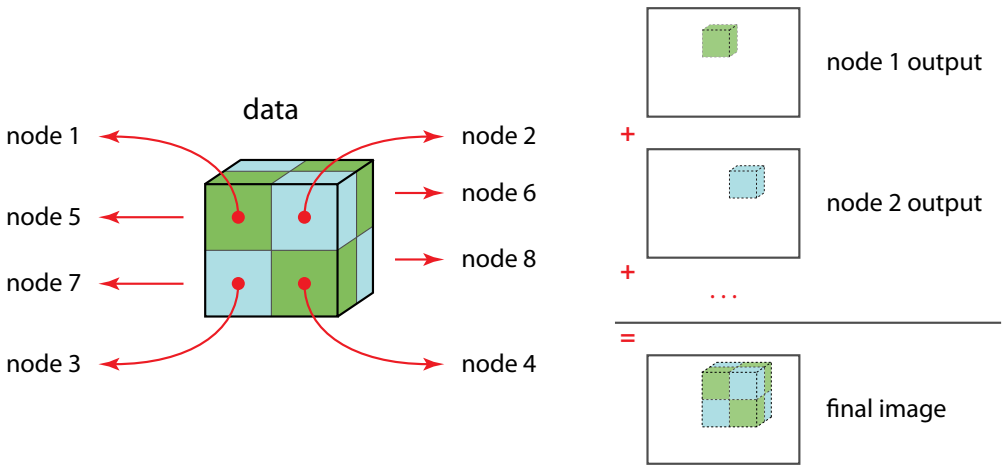


Figure 4.7: Subdividing the volume data between eight rendering nodes. Each node integrates through its own subvolume and returns the full output image. These are then composited to form the final image as shown on the right side.

gorithms, we use one node of our system to render the full output image while altering the number of cores that node uses for computation. Each rendering node consists of a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz with 16 GB of RAM. To test the parallelization to many nodes, each node uses all of its eight cores for rendering. Table 4.1 lists the naming conventions and the classifications of the rendering algorithms that we evaluate in this chapter. A detailed description of the test cluster and further results can be found in Chapter 5.

First, in Section 4.3.1, we investigate the effect on the average rendering time due to increasing the number of cores used for computation. In Section 4.3.2, we consider the change in the average rendering times due to the increase in the output image resolution. We repeat both of these tests to evaluate the multi-node algorithms in Sections 4.3.3 and 4.3.4.

Renderer Title	Short Description	Bricked	Data Storage
Unbricked, Linear	brute-force renderer with each core integrating one ray at a time through the entire volume	no	linear
By Image, Linear	each core renders own part of the image	yes	linear
By Image, Morton		yes	Morton
By Data	each core renders own subvolume to a separate image. these are then composited in parallel to form the final image	yes	Morton
By Brick Rays	each core renders some of the rays through a brick. bricks are processed sequentially	yes	Morton

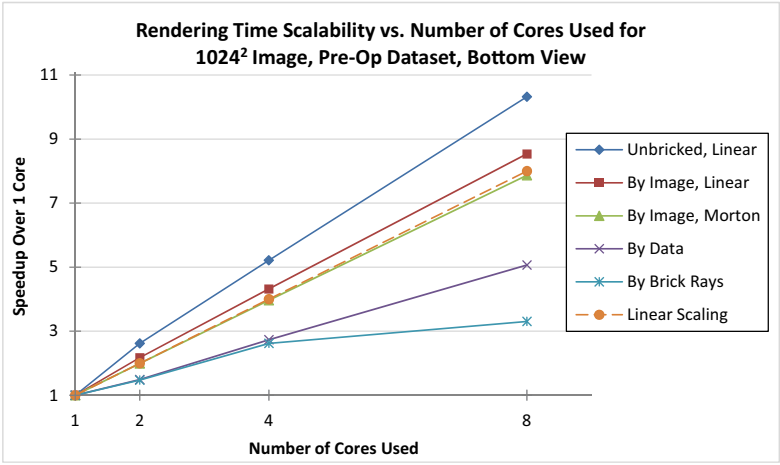
Table 4.1: Brief classification of test algorithms evaluated in both multi-core and multi-node setups. The figures throughout this section refer to the algorithms by their "Renderer Title."

4.3.1 Multi-Core: Scalability in the Number of Cores

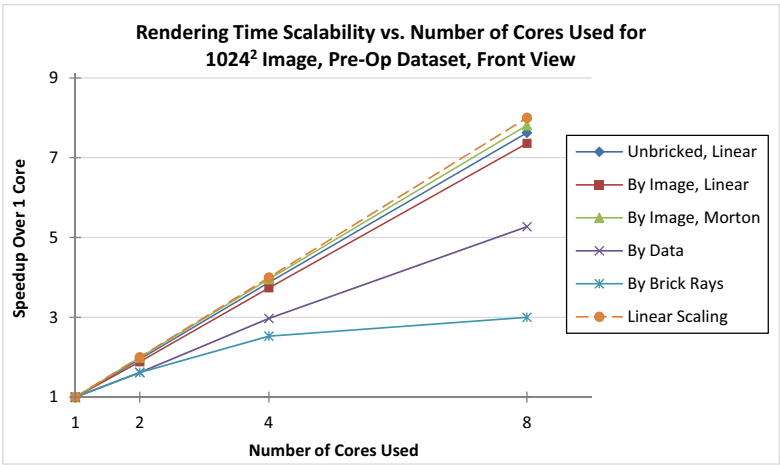
In this test, we consider the speedup of the rendering times due to the increase in the number of cores utilized for computation. We use a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM to render a dataset view 30 times at the output image resolution of 1024^2. The analysis averages the rendering times for the scalability computations.

We compute the scalability by calculating the speedup in the average rendering time achieved by utilizing several cores. If an image takes t_1 seconds to render using one core and t_8 seconds using eight cores, then the speedup is calculated as a fraction t_1/t_8. Fig. 4.8 shows this speedup for the "Bottom View" and the "Front View" of the Pre-Operation Dataset. We include only these two figures because they are similar to the ones concerning other datasets and views.

Fig. 4.8(a) shows that the speedup for both the brute-force renderer ("Un-



(a) Pre-Operation Dataset from the "Bottom View"



(b) Pre-Operation Dataset from the "Front View"

Figure 4.8: Rendering time speedup in the number of cores to generate an output image with the resolution of 1024^2. The scalability plots for the other views of the Pre-Operation Dataset are similar to (b), while the plots for the other datasets are similar to both (a) and (b). The tests utilized a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM.

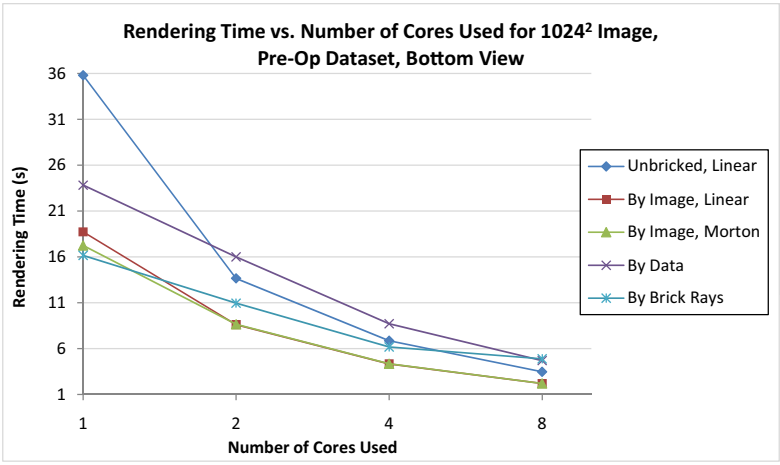
bricked, Linear") and the bricked raycaster with linear data storage ("By Image, Linear") is higher than the linear scaling in the number of cores. For eight cores, the brute-force renderer achieves a speedup of 10.32, the bricked raycaster with linear data order achieves about 8.54, while the bricked raycaster with Morton data order ("By Image, Morton") manages 7.87. The rendering time using only one core for the brute-force algorithm is 35.8 seconds while eight cores produce

the same image in 3.47 seconds. As more cores are used to produce the images, the data they are accessing is closer together in memory when compared to using one core. Several cores render through the dataset simultaneously, thus accessing the data array with some interval. In other words, two accessed pieces of data are not neighbors. In the case of the "Bottom View," this interval may have been favorable for pre-caching hence allowing for scalability higher than eight. Even though these numbers are high, only the bricked raycaster with Morton data order keeps this scalability independent of the viewing direction, as can be seen in Fig. 4.8(b). For all of the datasets and views, this algorithm achieves scalability in the range of $[7.80, 7.90]$ for utilizing eight cores.

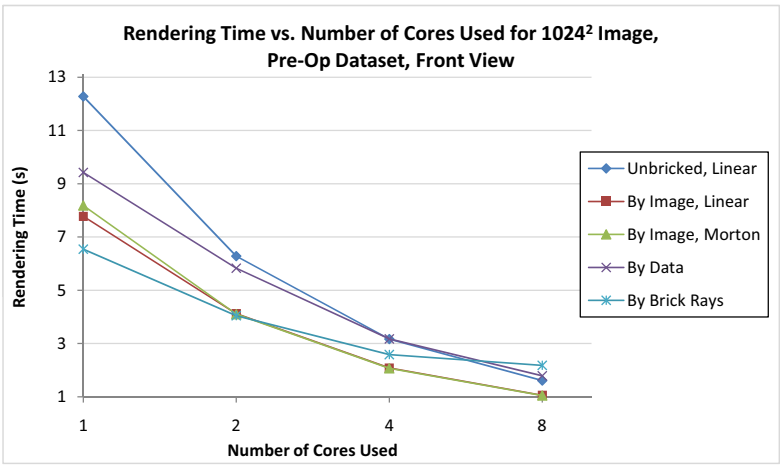
The scalability with respect to the number of cores used for rendering is not a complete measure of performance. There are two issues that need to be discussed further. First, as the image size changes, the speedup can also change. To gauge how an algorithm reacts to changes in the output image resolution, we use eight cores to render our datasets from different views using a variety of resolutions. The results are shown in Section 4.3.2.

Second, even though one algorithm may scale better than another as the number of cores increases, it may still generate the images slower. Hence we must also consider the absolute rendering time of an image. We render 30 images at the resolution of 1024^2 using different numbers of cores of a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM. The average rendering times are shown in Fig. 4.9. General trends are very similar between datasets and test views, so we only consider the "Bottom View" and the "Front View" of the Pre-Operation Dataset.

One can see that the rendering times of the proposed bricked raycaster are



(a) Pre-Operation Dataset from the "Bottom View"



(b) Pre-Operation Dataset from the "Front View"

Figure 4.9: Average rendering times for different numbers of cores generating an output image with the resolution of 1024^2. The plots for the other views of the Pre-Operation Dataset are similar to (b), while the plots for the other datasets are similar to both (a) and (b). The tests utilized a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM.

lower than any other algorithm when using two or more cores (25 – 37% faster). In addition, the proposed algorithm is able to render the test datasets into an image with the resolution of 1024^2 at a rate close to one frame per second using eight cores[2].

[2]Rendering the "Bottom View" requires about two seconds per image, while all other views are closer to one second per image. There is such a difference in rendering times because the dataset takes up more of the output image in the "Bottom View" compared to any other view.

Almost perfectly linear scalability in the number of cores and faster rendering times make the proposed bricked raycaster an excellent choice as an underlying renderer for a visualization system. An interesting observation is that the linear data order within the bricks performs similarly to the Morton order, less than two percent slower in all instances. The key idea is that rearranging the data into a brick hierarchy improves the rendering speed over the brute-force renderer without a significant dependence on the data storage order.

4.3.2 Multi-Core: Scalability in the Output Image Size

This test considers the effect that increasing the output image resolution has on rendering times. The effect is measured by the rendering time ratio r, defined in Equation (3.18). It measures the cost of rendering a larger image as a fraction of the increase in its resolution. To obtain the rendering time, we average the computation times that a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM took for a particular view or resolution. Then, we consider the maximum of these average times to provide the worst average case. Fig. 4.10 shows the rendering time ratio for the "Bottom View" of the Pre-Operation Dataset. The rest of the views for all of the datasets are similar.

An interesting trend emerges: rendering an image with the resolution of 2048^2 is about 48 times slower than computing an image with the resolution of 256^2, which is about 75% of the cost associated with increasing the image resolution 64 times. One can arrive at the conclusion that as the output image size grows, the rendering time ratio decreases until it reaches a plateau. However, we must note that this behavior for a 2048^2 image can be attributed to severely

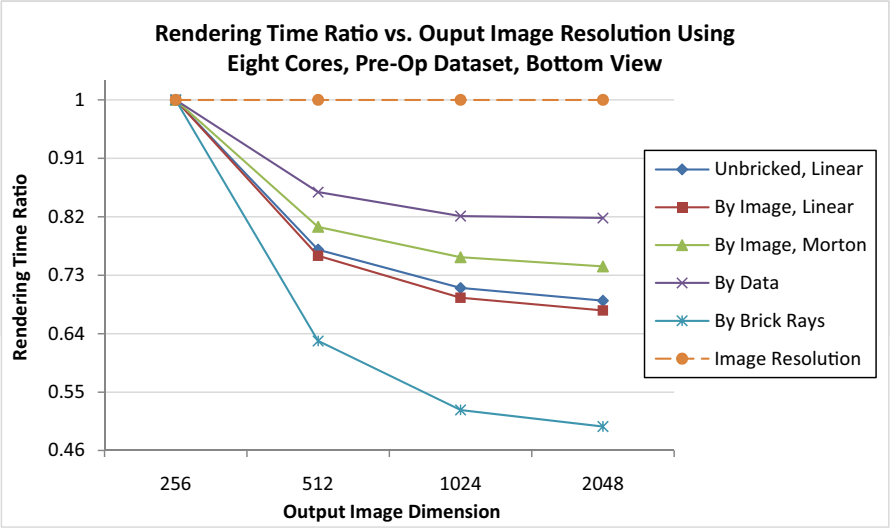


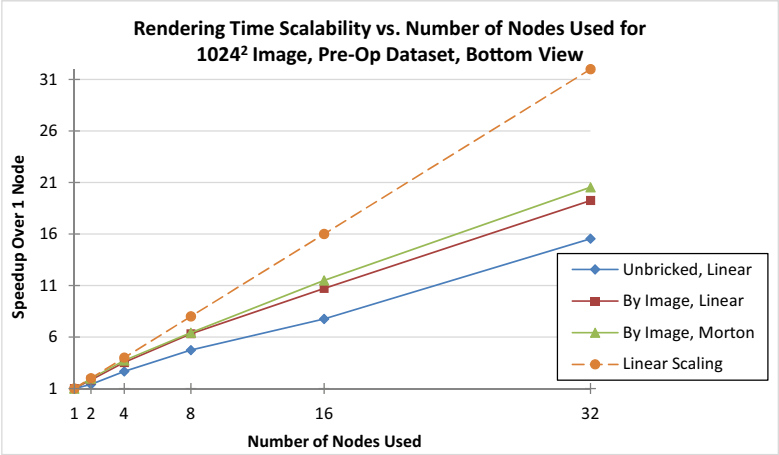
Figure 4.10: Rendering time ratios for different image resolutions of the Pre-Operation Dataset from the "Bottom View". Each were computed with eight cores on one node made up of a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz. The plots for the other views and datasets are very similar to this one.

oversampling of the dataset, since each slice has the resolution of 512^2. To test this conclusion appropriately, one must use a dataset that has the slices with the resolution of at least 1024^2.

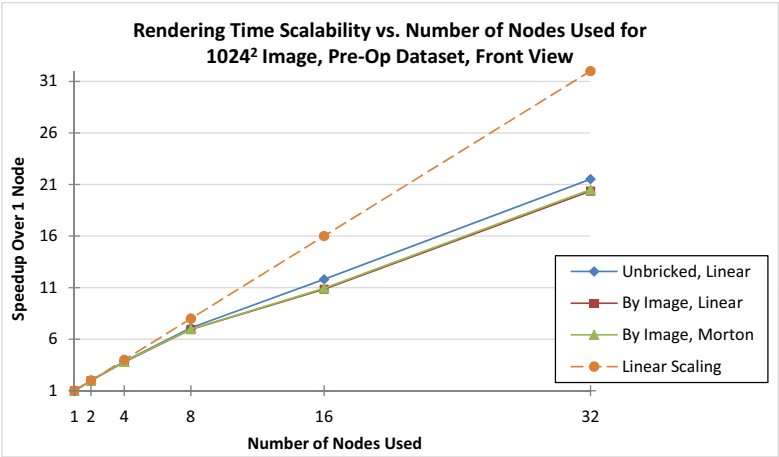
4.3.3 Multi-Node: Scalability in the Number of Nodes

To extend the analysis in Section 4.3.1 from the multi-core to the many nodes paradigm, we run similar tests using eight cores on each of a number of nodes. Each node is made up of a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz with 16 GB of RAM. This allows us to gauge the algorithm potential in utilizing future processors consisting of tens of computational cores.

We consider the rendering time speedup with the increase in the number of nodes while rendering an output image at the resolution of 1024^2. Each node



(a) Pre-Operation Dataset from the "Bottom View"



(b) Pre-Operation Dataset from the "Front View"

Figure 4.11: Rendering time speedup for different numbers of nodes to generate an output image with the resolution of 1024^2. The plots for the other views and datasets are similar to these. The tests utilized a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM.

uses eight cores to compute its part of the output image. Fig. 4.11 shows the rendering time scalability for the "Bottom View" and the "Front View" of the Pre-Operation Dataset. The other views and datasets exhibit similar behavior.

One can notice that the scalability of the selected algorithms is significant but not perfectly linear. Our proposed bricked raycaster achieves a speedup of 20.6 when using 32 nodes in parallel (256 cores). Another important factor is

that the proposed bricked raycaster is consistent in terms of scalability between different views.

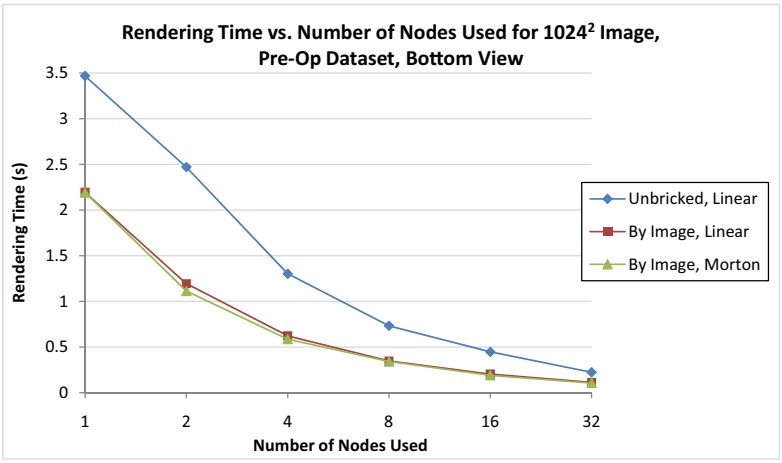
Since looking at the scalability alone is not sufficient to gauge the performance of an algorithm, we also consider the absolute rendering times to produce an image at the resolution of 1024^2. The rendering time we measure is computed in two steps. First, we record the time per node per computation. Then, we average these times for all test runs for a specific view of a dataset. Finally, we consider the worst time computed as the maximum of all average rendering times. Fig. 4.12 shows the absolute rendering times for a number of nodes using eight cores for computation each. We show only the "Bottom View" and the "Front View" of the Pre-Operation Dataset but the rest are similar to these.

On average, the proposed bricked raycaster computes the images 33.7% faster than the brute-force method, reaffirming the conclusion that bricking the volume data improves rendering speed. Once again, the difference between storing the data linearly or in Morton order is insignificant (Morton order is two percent faster than linear order).

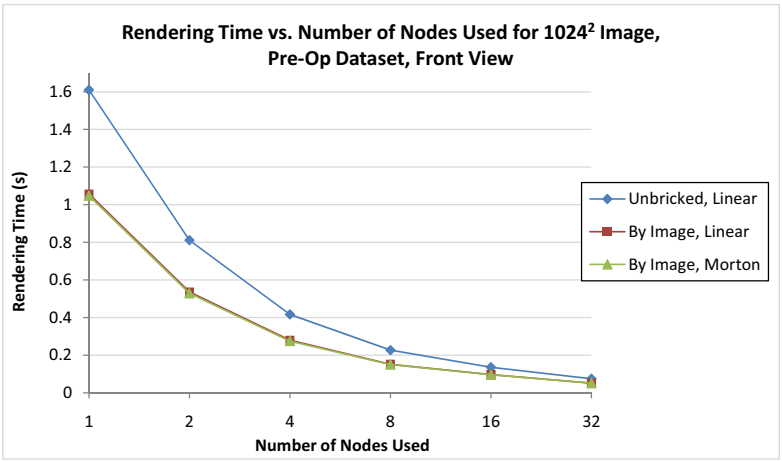
4.3.4 Multi-Node: Scalability in the Output Image Size

This test considers the effect that increasing the output image resolution has on the rendering times. Instead of considering the image scalability in terms of the rendering time ratio[3], we look at the increase in cost associated with increasing the image size. Fig. 4.13 shows the rendering time scalability for the "Bottom

[3]Rendering time ratio is defined in Equation (3.18) and measures the cost of rendering a larger image as a fraction of the increase in its resolution.



(a) Pre-Operation Dataset from the "Bottom View"

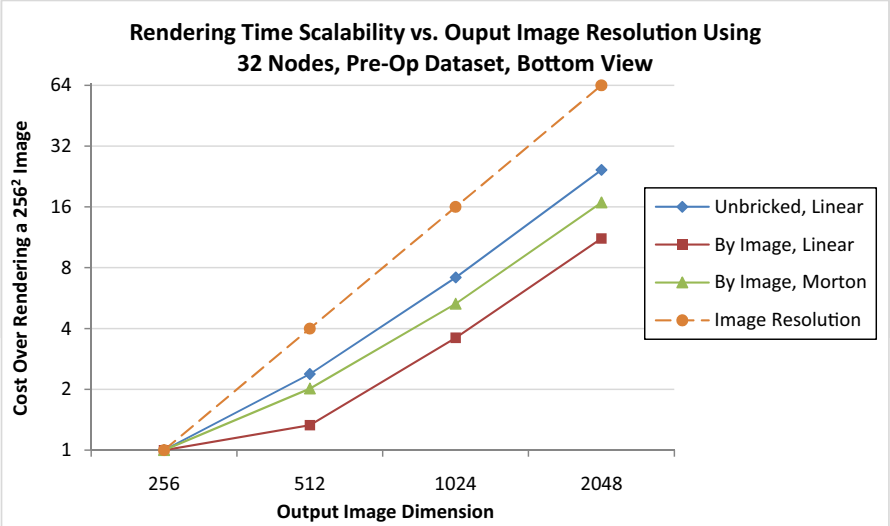


(b) Pre-Operation Dataset from the "Front View"

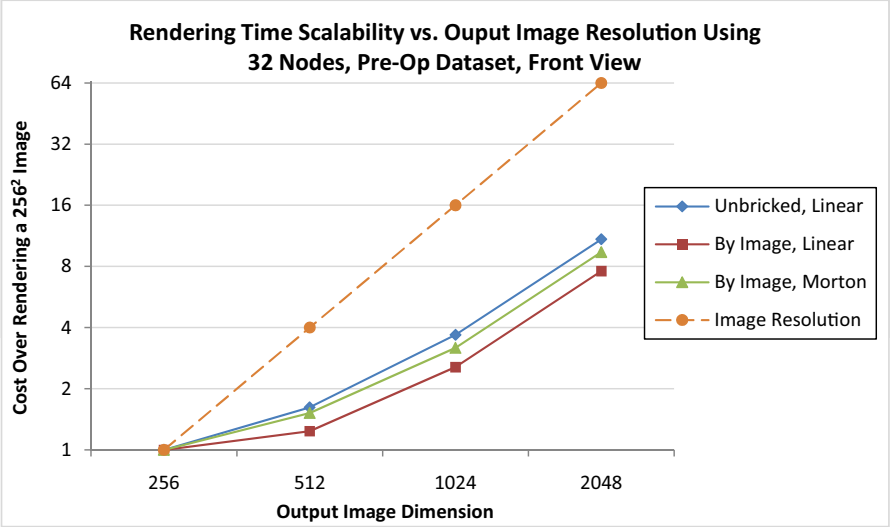
Figure 4.12: Average rendering times for several number of nodes to generate an output image with the resolution of 1024^2. The plots for the other views and datasets are similar to these. The tests utilized a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM.

View" and the "Front View" of the Pre-Operation Dataset. The plots for the other views and datasets are similar to the ones in Fig. 4.13.

As can be seen from Table 4.2, the proposed bricked raycaster with Morton order can render an image with the resolution of 2048^2 about 9.7 – 16.4 times slower than an image at the resolution with 256^2 depending on the view. One can arrive at the conclusion from this observation that such scaling is propor-



(a) Pre-Operation Dataset from the "Bottom View"



(b) Pre-Operation Dataset from the Front View"

Figure 4.13: Rendering time for several image resolutions computed with 32 nodes using eight cores for computation each. The plots for the other views and datasets are very similar to these. The tests utilized a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM.

tional to the increase in the single dimension of an image (8) rather than its area (64). However, we must note that this behavior for a 2048^2 image can be attributed to severely oversampling of the dataset, since each slice has the resolution of 512^2. To test this conclusion appropriately, one must use a dataset that

	By Image, Morton	By Image, Linear	Unbricked, Linear
Ave Bottom View	16.41	15.91	22.08
Ave Other Views	9.74	9.11	11.36

Table 4.2: Average time cost for 32 nodes computing a 2048^2 image over a 256^2 image. The average cost is calculated over all datasets. Because the scaling to produce the "Bottom View" is different from all other views, we separate the two averages. The tests utilized a dual quad-core Intel® Xeon® E5440 processor at 2.83 GHz node with 16 GB of RAM.

has the slices with the resolution of at least 1024^2.

4.4 Summary

In this chapter we have introduced several parallelization techniques for to the bricked raycaster. We discuss the parallelization in two tiers: on the multi-core level and the level of many nodes. Parallelization on both levels can build upon the independence of pixels in the output image or the independence of data within the volume dataset. We evaluate the scalability of the rendering times in terms of the output image resolution, number of cores and the number of nodes. We also consider absolute rendering times.

For the examples tested, we have found that bricking the data produces nearly linear scalability (average of 7.85 for eight cores) in terms of the number of cores used for computation and the absolute rendering times 25 – 37% faster than the brute-force raycaster. The difference in the rendering times between using linear and Morton data storage orders is two percent, which may be considered negligible. As the result, our bricked raycaster is able to render 1024^2 output images of the test datasets at one second per image using eight cores.

The scalability of the rendering times in terms of the number of nodes is not as linear as in the multi-core case, but the proposed algorithm achieves an average speedup of 20.6 for 32 nodes rendering an image with the resolution of 1024^2. The bricked renderer is able to render a 1024^2 image of the datasets at 10.9 and 20.2 images per second for the "Bottom View" and the "Front View" respectively (91.5 ms and 49.6 ms per image respectively). Such rendering speed can be considered interactive but not real-time which we defined to be at least 30 Hz in Chapter 1.1.

All of the tests show that the proposed bricked raycaster has the potential for the real-time volume rendering and we investigate several other performance characteristics in Chapter 5.

CHAPTER 5

RESULTS

The previous chapter describes the parallelization of the accelerated bricked raycasting algorithm. Tests have shown that the proposed algorithm has excellent scalability qualities. The aim of this chapter is to evaluate and comment on the overall success of the system as well as two additional tests.

We start with a description of the system and its hardware in Section 5.1. The details of the test datasets are located in Section 5.2 and the views of these data sets in Section 5.3. The descriptions of several transfer functions are in Section 5.4. Section 5.5 evaluates the algorithm performance in relation to two transfer functions and Section 5.6 analyzes the dependence of rendering times on the distance between ray samples. We conclude in Section 5.8.

5.1 Technical Description of the System

The culmination of the parallelization techniques discussed in Chapter 4 is the ability of the proposed bricked raycaster to utilize a cluster of multi-core processors. The test cluster consists of 32 nodes connected by a gigabit network. Each node has two quad-core Intel® Xeon® E5440 processors running at 2.83 GHz. A schematic for this processor is shown in Fig. 5.1. Each individual server node has 16 GB of Random Access Memory (RAM), which can hold any of our datasets and their pre-computed gradients.

The visualization system is organized such that the client computer controls all of the render nodes directly, as shown in Fig. 5.2(a). Fig. 5.2(b) shows the

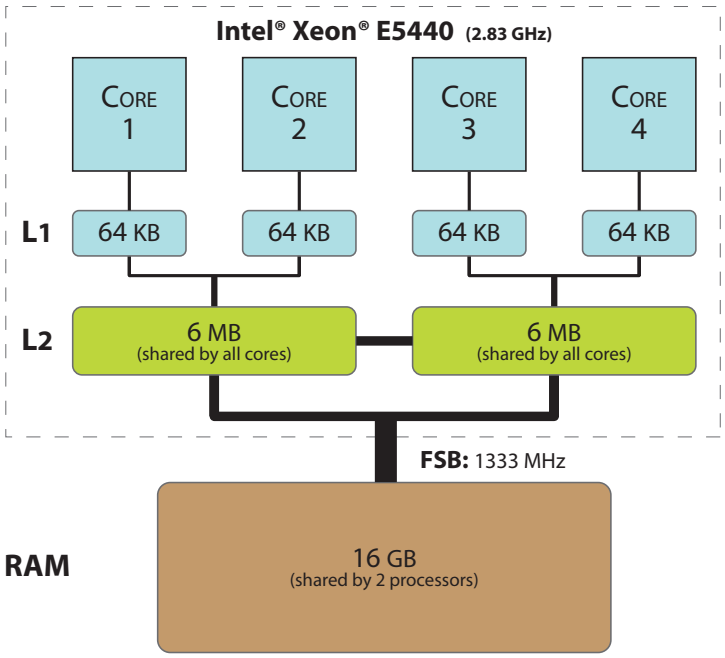
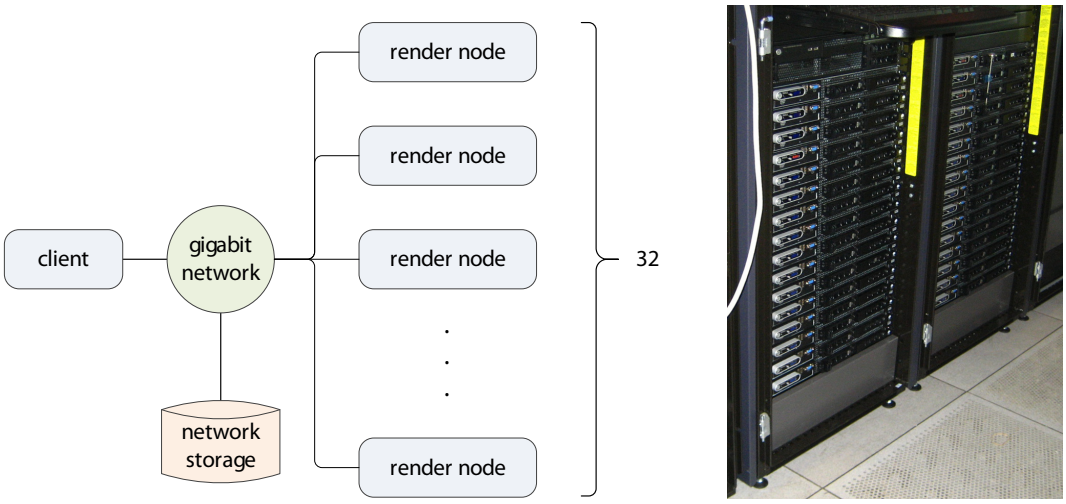


Figure 5.1: A schematic of the quad-core processor within each node.



(a) A schematic of our visualization system

(b) Our rendering cluster

Figure 5.2: A schematic and a picture of our visualization system.

picture of our compute cluster. The rendering results are returned to the client for final compositing and display. The resulting system is robust in terms of the number of nodes one can use and the network type. Such structure is necessary for utilizing massively distributed clusters.

5.2 Datasets

To test the system performance, we use the three datasets described in this section. Each of these is a CT scan of a patient diagnosed with an aortic aneurysm. The data is stored as 12-bit integers following the DICOM standard, [NEM08]. Table 5.1 describes each of the three datasets in great detail.

Dataset Title	Volume Resolution	Voxel Spacing	Slices	Test Views
Pre-Operation	$512^2 \times 928$	0.075^2 cm $\times$ 0.077 cm 0.0295^2 in $\times$ 0.0303 in	Fig. 5.3	Fig. 5.6
Post-Operation	$512^2 \times 768$	0.0709^2 cm $\times$ 0.1 cm 0.0279^2 in $\times$ 0.0394 in	Fig. 5.4	Fig. 5.7
CT14	$512^2 \times 768$	0.0762^2 cm $\times$ 0.0805 cm 0.03^2 in $\times$ 0.0317 in	Fig. 5.5	Fig. 5.8

Table 5.1: Details of test datasets. "Slices" column refers to the figure showing two dataset slices and dataset visualization from the "Front View." "Test Views" column refers to the figure showing dataset visualization from each of the five test viewing directions.

5.3 Dataset Views

As was mentioned in Section 3.7, there are five viewing directions we utilize to test the algorithm performance: "Bottom View," "Front View," "Side View," "45° Side View" and "45° Corner View."

The "Bottom View" sets the virtual camera below the dataset directed upwards. This ensures that the rays progress through the volume against the memory order of bricks slabs (or slices in the case of unbricked renderers). The "Front View" sets the virtual camera in front of the dataset. The "Side View" is similar but sets the virtual camera to the left of the dataset. The "45° Side View"

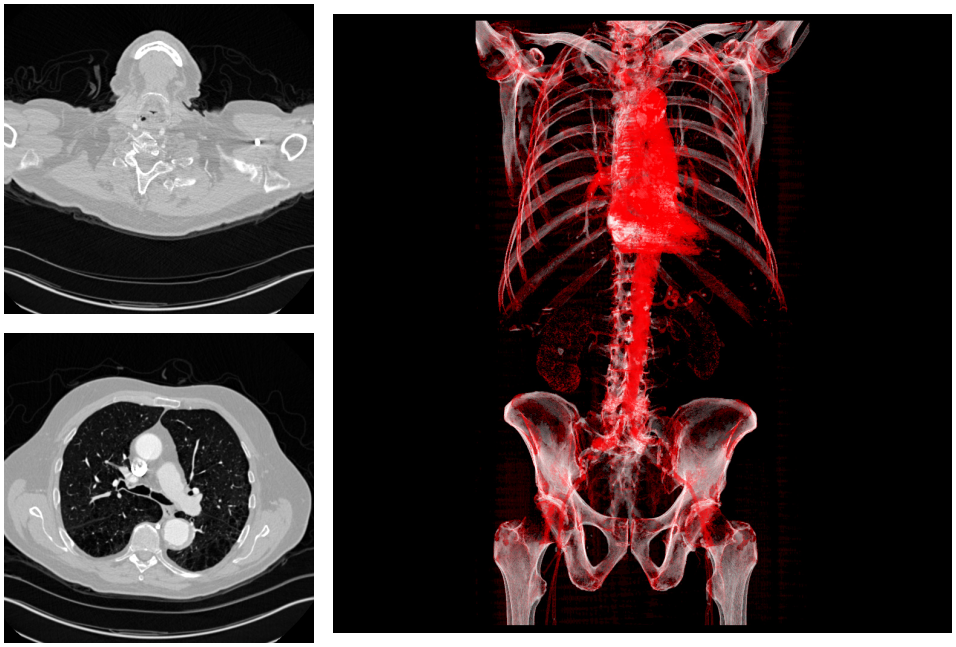


Figure 5.3: Two image slices of the Pre-Operation Dataset are on the left while its visualization from the "Front View" is on the right.

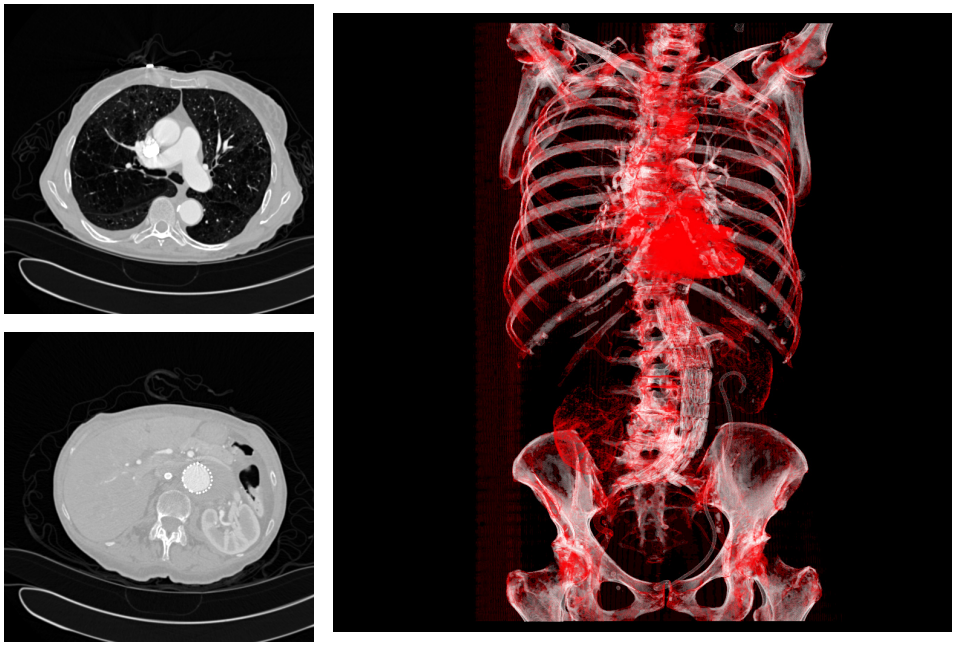


Figure 5.4: Two image slices of the Post-Operation Dataset are on the left while its visualization from the "Front View" is on the right.

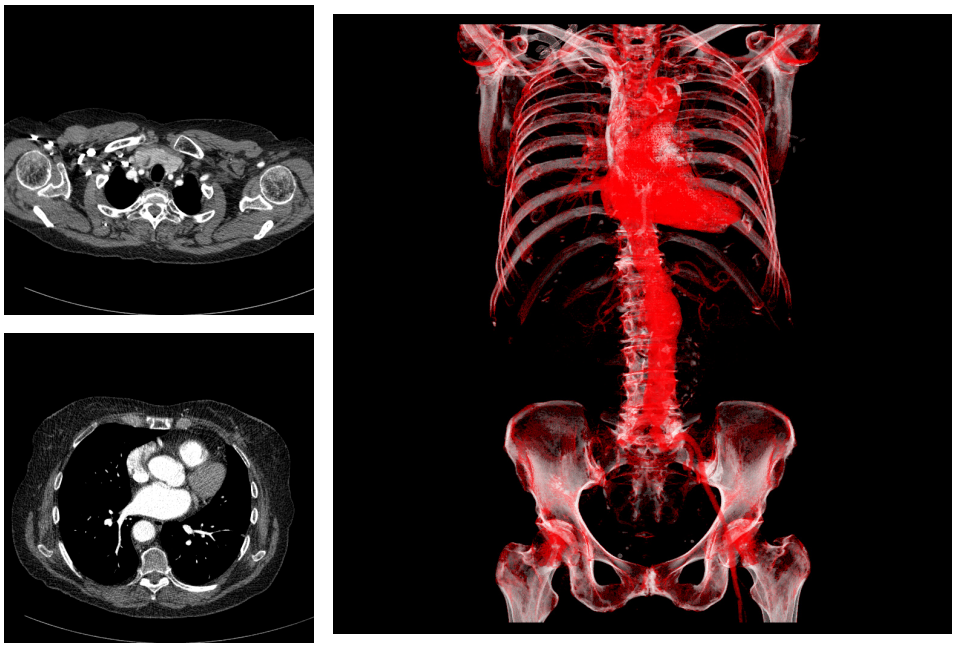
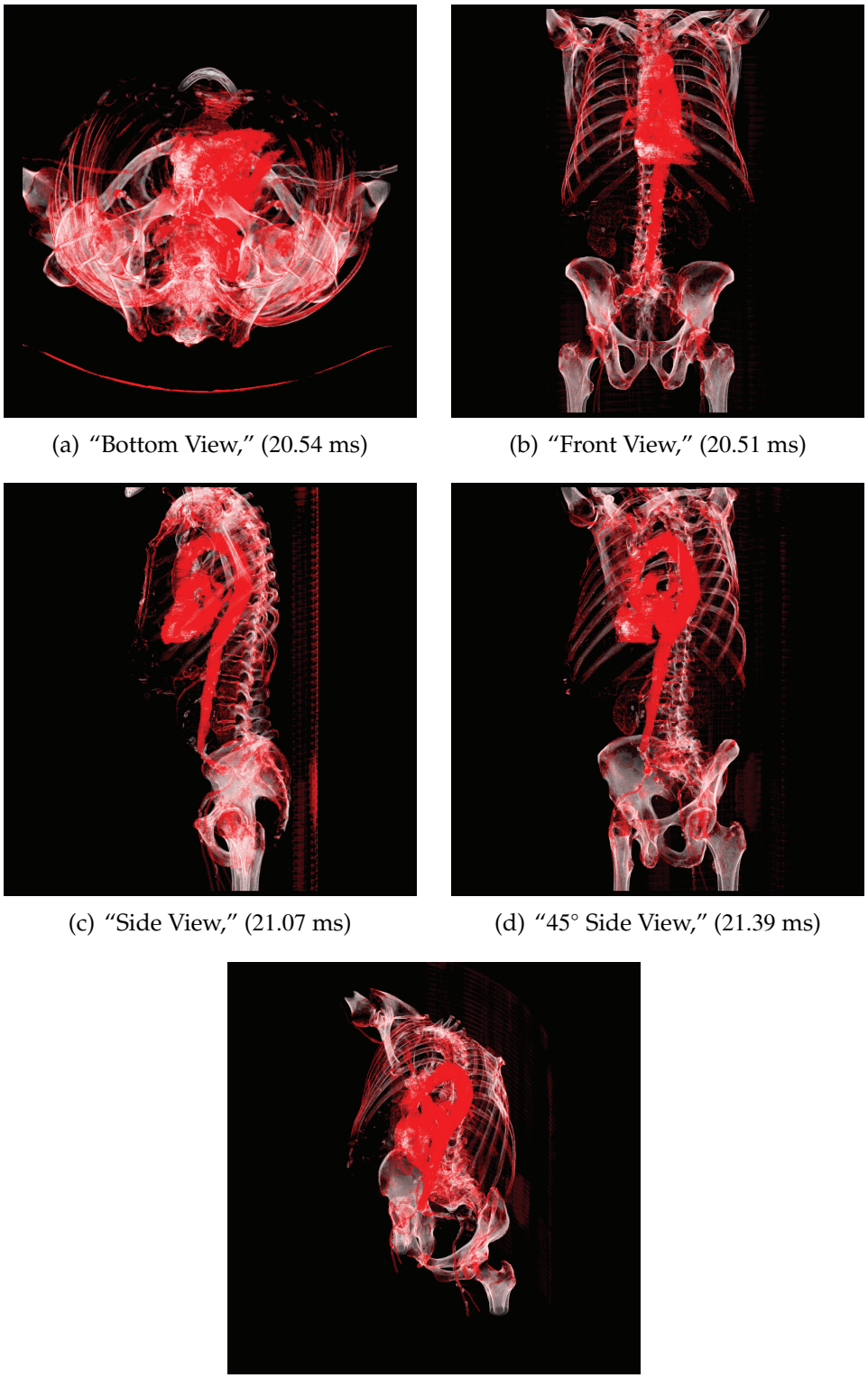


Figure 5.5: Two image slices of the CT14 Dataset are on the left while its visualization from the "Front View" is on the right.

shows the dataset rotated 45° around its vertical axis so that only the vertical edges of the volume are parallel to the image plane. The "45° Corner View" shows the dataset rotated 45° around both its vertical and horizontal axes, so that all of the volume edges are at an angle to the image plane.

Fig. 5.6, Fig. 5.7 and Fig. 5.8 show visualizations of the test views for the Pre-Operation, Post-Operation and CT14 Datasets respectively. Finally, Fig. 5.9 shows the data histograms of the Pre-Operation Dataset. It can be noticed that the density value of zero holds the largest amount of voxels because it represents the empty air around the patient and in their lungs. Large portions of the data have densities representing soft tissues and bone, which are rendered using the Partial Transfer Function in Fig. 5.9(c).



(a) "Bottom View," (20.54 ms)

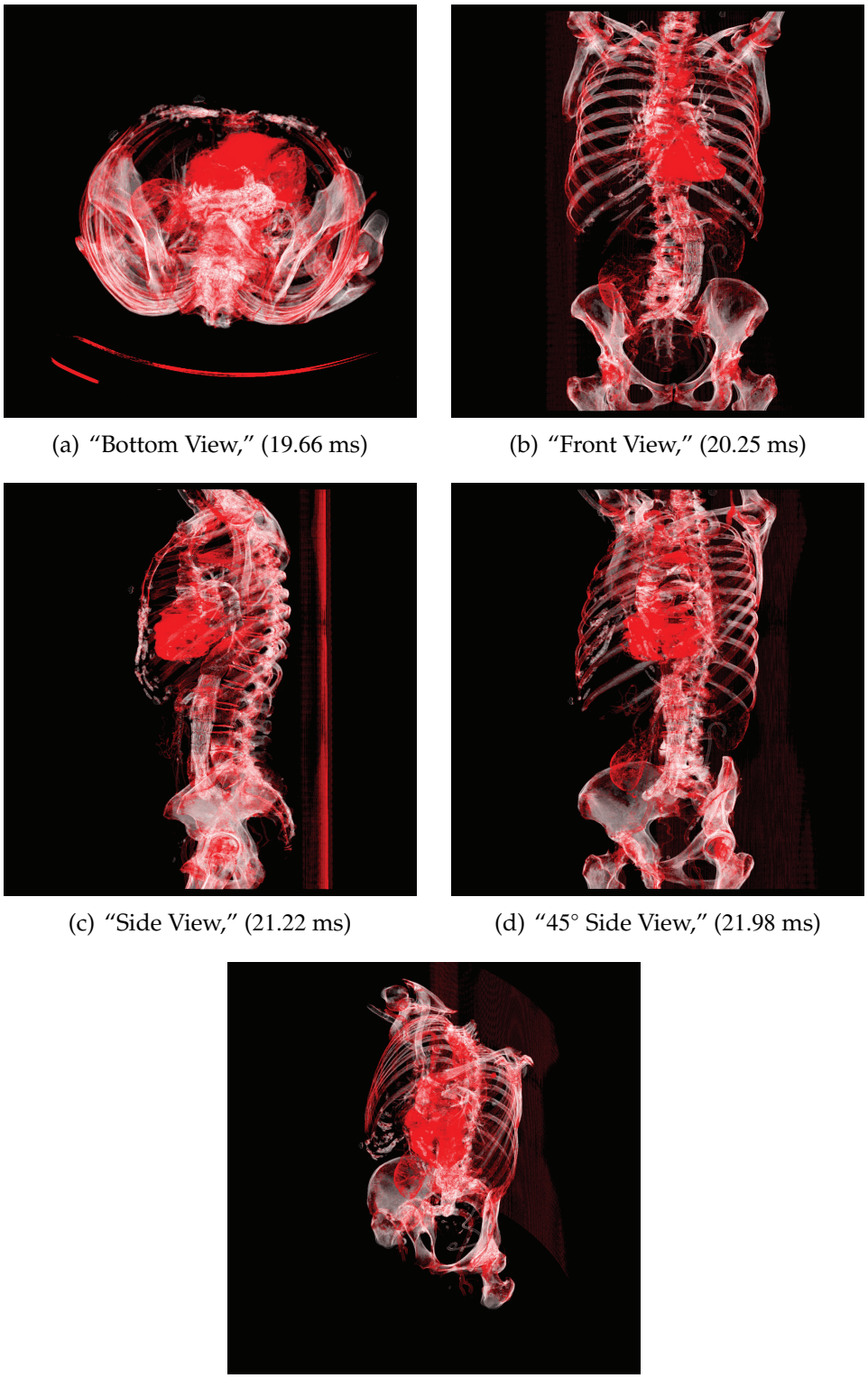
(b) "Front View," (20.51 ms)

(c) "Side View," (21.07 ms)

(d) "45° Side View," (21.39 ms)

(e) "45° Corner View," (20.96 ms)

Figure 5.6: Test views of the Pre-Operation Dataset. The times refer to the proposed bricked raycaster rendering a 1024^2 image using 32 nodes.



(a) "Bottom View," (19.66 ms)

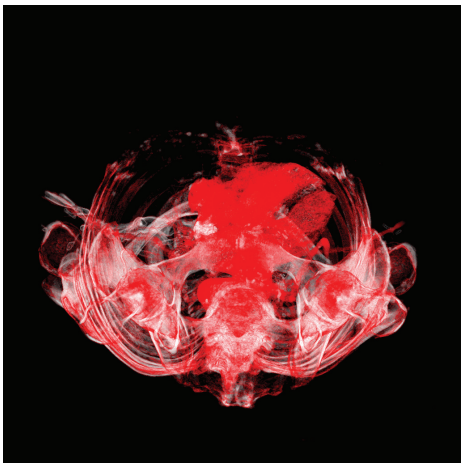
(b) "Front View," (20.25 ms)

(c) "Side View," (21.22 ms)

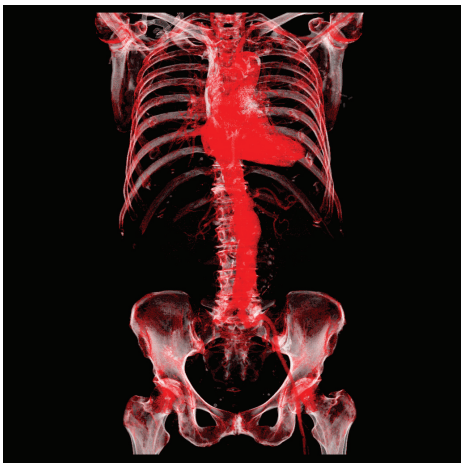
(d) "45° Side View," (21.98 ms)

(e) "45° Corner View," (20.4 ms)

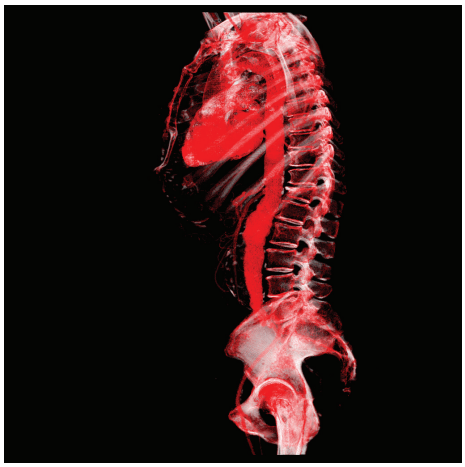
Figure 5.7: Test views of the Post-Operation Dataset. The times refer to the proposed bricked raycaster rendering a 1024^2 image using 32 nodes.



(a) "Bottom View," (19.46 ms)

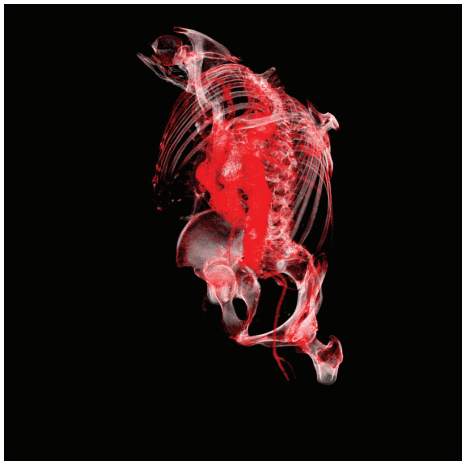


(b) "Front View," (20.05 ms)



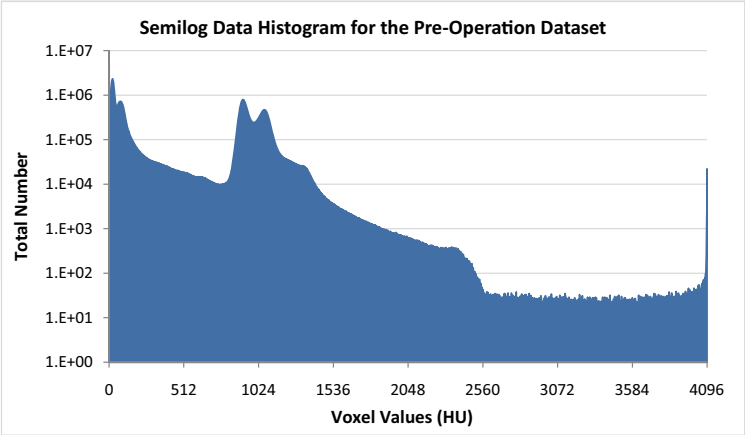
(c) "Side View," (20.6 ms)

(d) "45° Side View," (20.9 ms)

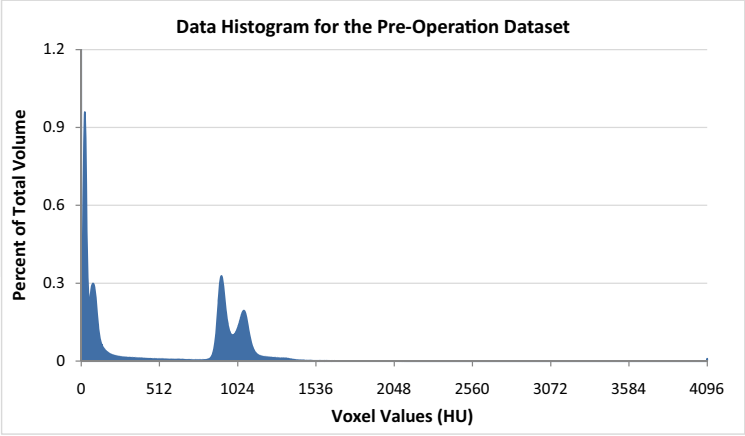


(e) "45° Corner View," (20.54 ms)

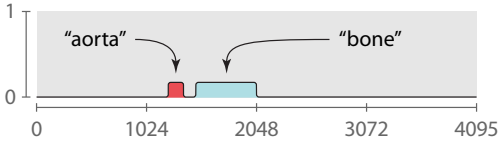
Figure 5.8: Test views of the CT14 Dataset. The times refer to the proposed bricked raycaster rendering a 1024^2 image using 32 nodes.



(a) Semi-Log plot of the data histogram



(b) Linear plot of the data histogram, 2.28% of voxels have the value zero



(c) Partial Transfer Function used for rendering

Figure 5.9: Data histograms for the Pre-Operation Dataset. The horizontal axis represents the density value of a voxel using the positive Hounsfield Units (HU). Note that the data histograms for all test datasets are similar and 2.62% of voxels in the Post-Operation Dataset and 5.21% of voxels in the CT14 Dataset are valued zero.

5.4 Test Transfer Functions

Transfer functions are directly responsible for what fraction of the dataset is visible to the user and affect the rendering times of our algorithm. In reality, the visualization system must also take into account the windowing function[1] Ω. This operation can be expressed by rendering a dataset with a modified transfer function $\Psi' = \Psi \circ \Omega$.

The two test opacity transfer functions included a "Partial Transfer Function" and a "Full Transfer Function." The "Partial Transfer Function," shown in Fig. 5.9(c), includes some tissue and bone. This transfer function has been used for all of the tests in Sections 3.7 and 4.3. The "Full Transfer Function" sets the opacity above zero for all voxels in a dataset. Although not useful for visualization, when all voxels have non-zero values, the computation times will be slowest thus providing a maximum bound.

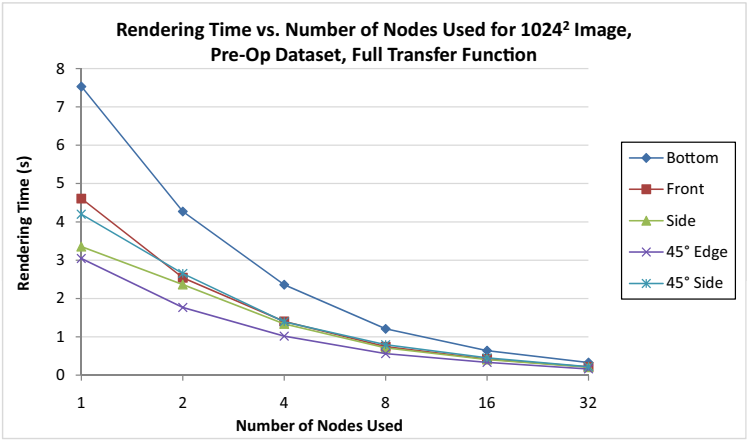
5.5 Dependence of Rendering Times on Transfer Functions

In this test, we vary the number of nodes used to render the datasets into an image with the resolution of 1024^2. Table 5.2 shows the percentages of each dataset that is visible to the user based on the transfer functions. Using the Full Transfer Function for visualization produces the rendering times shown in Fig. 5.10. On average, the rendering times are 4.27 times slower than using the Partial Transfer Function. The rendering rate for the "Bottom View" is 3.12 fps (320.6 ms per image) and the rate for the other views is 5.42 fps (184.3 ms per

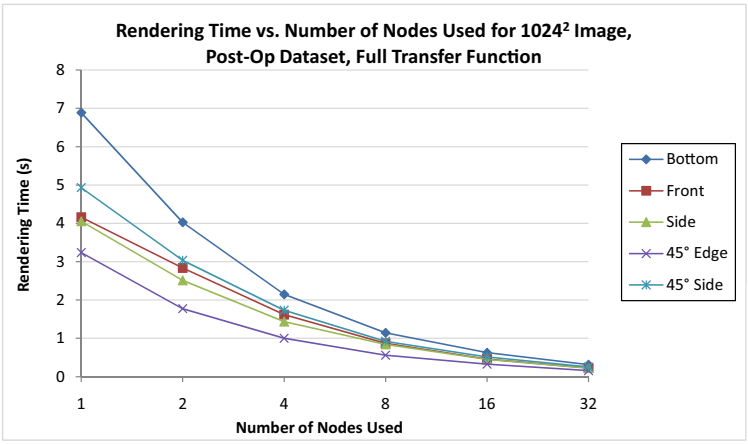
[1]The windowing function rescales the volume data in order to enhance certain features. The mapping is applied directly to the data prior to the application of the transfer function.

	Pre-Operation Dataset	Post-Operation Dataset	CT14 Dataset
Partial Transfer Function	1.43%	1.52%	1.33%
Full Transfer Function	100%	100%	100%

Table 5.2: Percentages of the datasets that are visible based on each of the transfer functions.



(a) Pre-Operation Dataset



(b) Post-Operation Dataset

Figure 5.10: Average rendering times for several number of nodes to generate an output image with the resolution of 1024^2 using the Full Transfer Function.

image). On the other hand, using the Partial Transfer Function can generate the images at the rate of 10.9 fps (91.5 ms per image) for the "Bottom View" and 20.2 fps (49.6 ms per image) for the other views.

Even though the amount of data that is visible increases from 1.5% to 100%,

the rendering time does not increase proportionally. This can be attributed to the early ray termination.

5.6 Dependence of Rendering Times on Integration Step Sizes

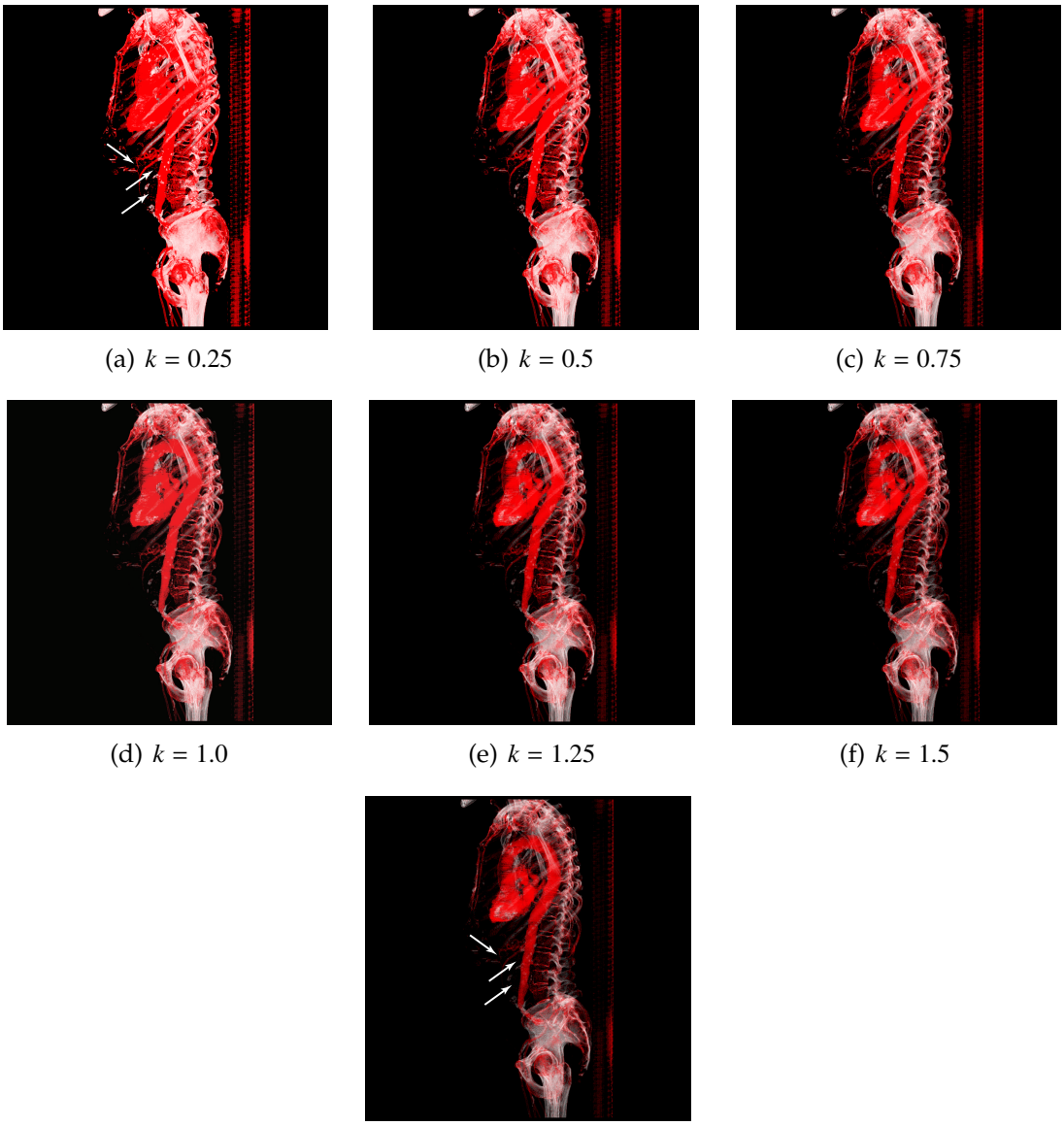
All of the previous tests considered the scalability of rendering times in terms of several parameters, like the number of nodes used for computation or output image resolutions. Each test used an integration step size that depends on the viewing direction and ranges from one for axis-aligned views to $\sqrt{3} \approx 1.73$ for the "45° Corner View." The step size sets the distance between sample points along any ray and, depending on the dataset, may violate the Nyquist-Shannon sampling theorem, [SAG+05].

This test considers the dependence of the rendering times on changes in the integration step size. We define the step size as:

$$\triangle s = k\left\|\mathbf{v} \cdot \mathbf{h}\right\|, \tag{5.1}$$

where $\mathbf{v}$ is the normalized viewing direction vector, $\mathbf{h}$ is the vector representing the spacing between voxels and k is a scaling factor. It was set to $k = 1.0$ for all of the previous tests. In order to satisfy the Nyquist-Shannon sampling theorem for the "45° Corner View," we must set k to be at most $k = 0.577$ to keep the integration step size as $\triangle s = k\sqrt{3} = 1$. To change the integration step size, we change the scaling factor k.

Fig. 5.11 shows the "Side View" of the Pre-Operation Dataset rendered with the different values of k. The images generated with $k = 0.25$ and $k = 1.0$ are visually similar besides the variation in brightness which originates from the



(a) $k = 0.25$ (b) $k = 0.5$ (c) $k = 0.75$

(d) $k = 1.0$ (e) $k = 1.25$ (f) $k = 1.5$

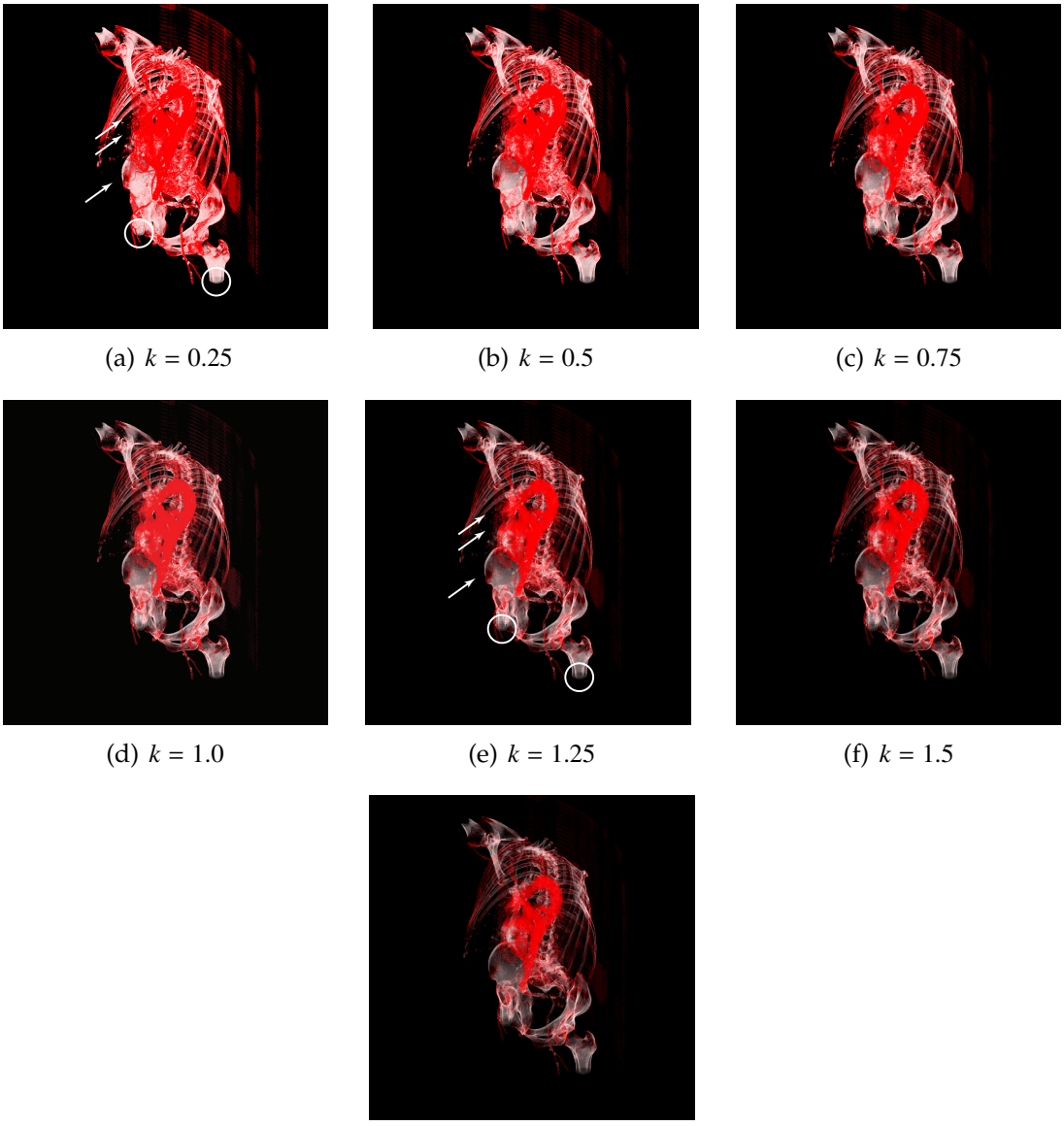
(g) $k = 2.0$

Figure 5.11: "Side View" of the Pre-Operation Dataset rendered with different integration step sizes that depend on the scaling k. White arrows indicate the lost features between images with different k values.

lack of updating the opacity transfer functions to compensate for the smaller step sizes. Note that when large step sizes are used, small features may be lost. For example, the small blood vessels visible in Fig. 5.11(a) where $k = 0.25$ (white arrow), are missing in Fig. 5.11(g) where $k = 2.0$.

Fig. 5.12 shows the "45° Corner View" of the Pre-Operation Dataset rendered using different values of k. Unlike the "Side View," the "45° Corner View" fails the Nyquist-Shannon sampling theorem when $k = 1.0$. The inside and the outside walls of the femur, marked by a white circle, become less defined as k increases. The blood vessels visible in Fig. 5.12(a) where $k = 0.25$ (white arrows) disappear in Fig. 5.12(e) where $k = 1.25$. As mentioned previously, to satisfy the Nyquist-Shannon sampling theorem at the axis-aligned views, k must be set no higher than one. We must set k to less than 0.707 for the "45° Side View" and $k \leq 0.577$ for the "45° Corner View." To ensure that the sampling theorem is satisfied at all times during rendering and no volume data is potentially missed by ray samples, the value of k must change depending on the viewing direction. The only time to use a larger step size is during fast interaction when the constraint on the output quality can be relaxed, similar to the multiple resolutions acceleration discussed in Section 3.1. To quantify the decrease in the computation time due to a larger step size, we tested our bricked raycaster with several values of k.

The bricked raycaster used 32 nodes to render all test views of the datasets at the resolution of 1024^2. We need to consider the absolute rendering times for the Post-Operation Dataset depending on the integration step size as a function of the scaling factor k. In our tests, changing the value for k from one to 0.5 to satisfy the Nyquist-Shannon sampling theorem increased the rendering times by 47%. Using an acceleration similar to the multiple resolutions but based on the integration step size, we can switch between producing a 1024^2 image of a lower quality at 20 fps (50 ms per image) and higher quality at 13.7 fps (73 ms per image). With processors currently available, such a difference in rendering times separates almost real-time performance from the interactive performance.



(a) $k = 0.25$ (b) $k = 0.5$ (c) $k = 0.75$

(d) $k = 1.0$ (e) $k = 1.25$ (f) $k = 1.5$

(g) $k = 2.0$

Figure 5.12: "45° Corner View" of the Pre-Operation Dataset with different integration step sizes that depend on the scaling k. White arrows and circles indicate the lost features between images with different k values.

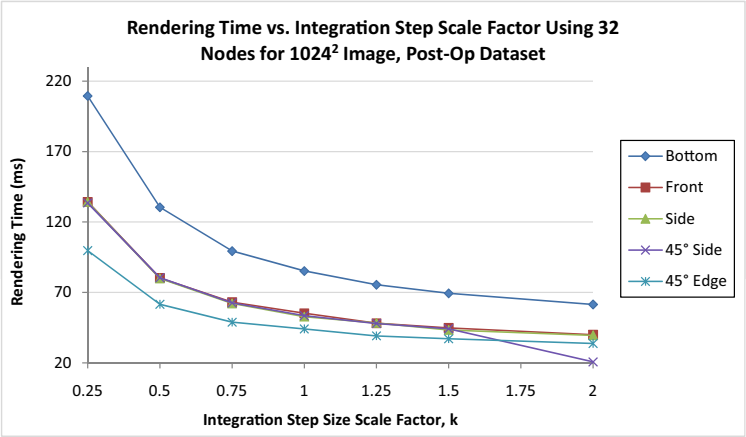


Figure 5.13: Rendering times for the Post-Operation Dataset for several integration step sizes as a function of scaling factor k. We used 32 nodes to render an image at the 1024^2 resolution.

5.7 Best Scalability of the Current Implementation

In this section, we use Amdahl's Law to approximate the parallel fraction of our implementation of the bricked raycasting algorithm running on our test cluster. Note that these specific constraints apply only to this section and prohibit us form arriving to general conclusions about the proposed algorithm. We can speculate the theoretical limit to the speedup achievable by our unoptimized raycaster and extrapolate it beyond 32 rendering nodes.

Amdahl's Law states that the speedup of the program execution time due to using multiple processors in parallel is limited by the sequential fraction of that program, [HP94]. To express this mathematically, let P be a fraction of the program runtime that accounts for the parallelizable portion. Setting the total execution time to one, the sequential portion of the runtime can be expressed as $1 - P$. Letting N be a number of processors used to execute the program in parallel, we can recompute the parallel fraction as $^P/_N$. The total execution time using parallelization becomes $(1 - P) + {}^P/_N$, which can be used to define the time

scalability in terms of the number of processors as:

$$S = \frac{1}{(1-P) + {}^{P}/_{N}}, \tag{5.2}$$

which is simply the ratio of the serial execution time and the parallel execution time.

Once the value of P is known, one can arrive at several conclusions specific to our unoptimized implementation running on our specific test cluster. The first one is the maximum theoretical speedup that an algorithm can achieve:

$$S_{max} = \lim_{N\to\infty} \frac{1}{(1-P) + {}^{P}/_{N}} = \frac{1}{1-P}. \tag{5.3}$$

To compute the value of P for our unoptimized raycaser, we use Least Squares to fit the function in Equation (5.2) to our experimental results obtained by rendering 1024^2 images with 32 nodes. The average value of P is 0.98087, which limits the maximum speedup to $S_{max} = 52.54$. The absolute rendering time for a 1024^2 image using a very large cluster of our machines is limited to 36.03 ms (27.75 fps per image) for the "Bottom View" and 19.66 ms (50.86 fps per image) for the other views.

Secondly, we can use the value of P to extrapolate the performance of the algorithm for a cluster made up of more than 32 nodes of the machines same as our cluster. Fig. 5.14 shows the extrapolated scalability of rendering times up to 1024 nodes derived from the Amdahl's Law with $P = 0.98097$.

5.8 Summary

In this chapter, we outlined the technical details of the rendering system's hardware. We described the datasets and defined five rendering views that were

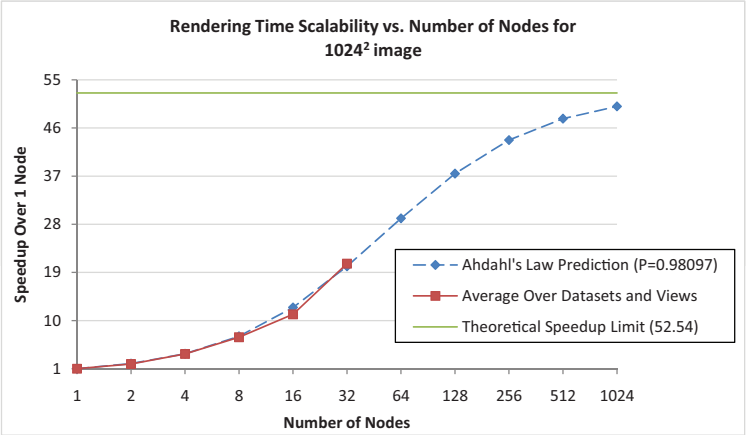


Figure 5.14: Extrapolated scalability of the rendering time for the unoptimized bricked raycaster up to 1024 nodes. It was computed using the scalability definition of Amdahl's Law in Equation (5.2) and $P = 0.98097$.

used to test the algorithms. We also evaluated the performance of the bricked raycaster in relation to transfer functions and the integration step size. We used Amdahl's law to approximate the parallel proportion of our unoptimized implementation of the proposed algorithm (98%), which limits the speedup to 52.5. This limits the maximum frame rate from 28 to 51 Hz for a 1024^2 image using a very large cluster made up of the same machines as our test cluster.

The integration step size is view dependent to satisfy the Nyquist-Shannon sampling theorem such that no data is potentially missed by a ray sample. Our tests have shown that decreasing the step size by two increases the rendering time by 47%. A large step size may be used when the constraint on the output quality is relaxed, such as during fast exploration of data. Switching to an appropriately small integration step size afterwards produces an acceleration method similar to the multiple resolutions technique.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Discussion

This thesis presents algorithmic investigations to achieve interactive volume visualization of 3D scalar medical data. In this work, we outline the basic raycasting algorithm, acceleration techniques and parallelization methods, which result in a bricked raycaster running on a cluster of multi-core machines. Although we have not achieved real-time rendering speeds, defined to be at least 30 fps, our unoptimized algorithm is able to render datasets above ten frames per second. We gauge the algorithm's performance by its scalability because future processor architectures will consist of tens of computational cores.

Following a comparison of several algorithms, we choose raycasting as the algorithm to build upon because it produces high quality output, extends to include a variety of features and parallelizes in a straightforward way. However to become a usable visualization tool, the brute-force raycaster must be accelerated. Several major improvements discussed in detail in Chapter 3 originate from using a brick hierarchy: global empty space skipping, increased data access coherency and local gradient cache. We also terminate rays early based on their accumulated opacity and skip transparent samples along them. We refer to the improved algorithm as the bricked raycaster. On our experimental cluster, these accelerations reduce rendering times by 25 – 37%, but not into the millisecond range required for the real-time visualization.

What seemed to be the most significant acceleration was selecting the data

subdivisions (brick sizes) to eliminate the memory and bus-bandwidth latencies and maximizing the utilization of the computing power of each core. If the brick sizes were chosen too large to fit into the available cache size, the rendering times were slower due to the poor cache coherency; the data requires extra time to be passed between RAM and the L2 cache. If the brick sizes were chosen too small, they subdivided the volume into too many pieces producing a similar issue. To achieve fast rendering times, a volume visualization algorithm must balance memory latency and processing power.

Our tests reveal that with careful selections of the brick sizes, one could obtain almost real-time volume visualization capabilities on a 32-node cluster. More importantly, the cache coherence of the data access due to the bricking scheme produced rendering times that are independent of the viewing direction. Clearly, as processor cache sizes and the relationship of computing power to memory accessibility change, brick sizes different from the ones reported may be necessary. However, very positive results indicated that real-time visualization can be achieved using standard hardware.

We considered and compared several parallelization approaches on both the multi-core and multi-node levels to take advantage of the specific architecture of our compute cluster. We subdivided the work within each level by considering distributing the volume data as well as the output image. Based on the scalability results, we proposed a system that utilizes output image subdivision. Using 32 nodes each with eight cores, we achieved rendering times that are 20.6 times smaller than using one node for a 1024^2 image. This 64% parallelization efficiency was fast enough for the average rendering rate of approximately 10 to 20 Hz for multiple views of our typical volume datasets. Using a resolution

of 512^2 for output images, the rate increases to roughly 27 to 40 Hz, sufficient for real-time image generation. We showed that this efficiency can be closely maintained with different dataset sizes, image viewing locations and output resolutions implying that the reported scalability is valid.

We have tested the response of the proposed algorithm to changing the distance between ray samples, the integration step size. Smaller distances are necessary to satisfy the Nyquist-Shannon sampling theorem while rendering datasets at any non-axial view. Doubling the sampling rate increases computation times by approximately 47%. The ability to select the integration step size is necessary for an acceleration where a bigger step size is used for rapid data exploration prior to rendering the higher quality result with a step size that satisfies the sampling theorem.

An important trend emerged with respect to the resolution of the output image. If it were increased from 256^2 to 2048^2, the rendering times increased 10 to 16 times instead of 64 times. Even though at such a high resolution we are severely oversampling the test datasets, this scaling hints at good performance for producing output at a high resolution.

The implications of these studies are vast. Currently the only way to achieve real-time visualization is to buy expensive workstations which are tailor-made for volume visualization. Most of these use a "marching cubes" type of algorithm which incorporates lots of pre-processing to establish interpolated surfaces which then can be viewed in real time. This interpolation procedure adds additional potential errors on the data originally obtained from the CT-scans, and therefore is not as appropriate for the medical profession.

On the other hand, our system displays the data directly without an intermittent construct decreasing the potential for error. The algorithm maps to a generic 32-node cluster which reduces the cost of a visualization system. The achieved rendering speeds without optimization promise real-time frame rates in the near future.

6.2 Future Work

There are several exciting opportunities for improvements to our work utilizing the computing potential of GPUs, the extension to rendering 4D datasets, and the algorithm use in an operating room.

Research into the utilization of custom hardware to visualize volumes has continued since the early 1990s. Algorithm designers can maximize rendering speeds by implementing computationally expensive functions with hardware. However, these systems could not be changed easily. The programmability of the current GPUs enables their use by software renderers and creates systems that are easily extended or updated. The GPUs are highly parallelized systems that surpass the CPU speed for floating point computation and memory access. In addition, these systems offer hardware that tri-linearly interpolates the data. As the result, GPUs provide an excellent platform as a visualization alternative to multi-core computers. Each rendering node in a computational cluster (or a "cloud") can utilize several GPUs for visualization.

Implementing our generic system to use the graphics hardware could see an additional improvement in performance because of faster memory access and floating point computations. However, the amount of on-board memory avail-

able to a GPU for storing data is much smaller than a workstation can provide. This limits the size of the dataset one can visualize with a GPU without out-of-core algorithms.

Even though rendering static 3D datasets enables their exploration, they lack the time dimension necessary in analyzing dynamic effects. For example, changes in the geometry of blood vessels occur due to the heart-beat or the impact of stent placement during intravascular surgeries. With the new paradigm of "cloud computing" and high enough internet bandwidth, it would be possible to use a cluster of machines to render time-dependent datasets in real time and to deliver these images directly into an operating room. The ability to produce visualizations in real time will enable matching fluoroscopic images to 3D volumetric images creating in a sense a virtual view, which potentially can enhance a surgeon's understanding.

BIBLIOGRAPHY

[Bli82] James Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *SIGGRAPH Computer Graphics*, 16(3):21–29, July 1982.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 Symposium on Volume Visualization*, pages 91–98, Tysons Corner, Virginia, USA, 1994. ACM.

[DBB06] Philip Dutré, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. A K Peters, Ltd, second edition, 2006.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS Workshop on Graphics Hardware*, pages 9–16, Los Angeles, California, USA, 2001. ACM.

[GBKGl04a] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Eduard Gröller. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. *IEEE Symposium on Volume Visualization and Graphics*, October 2004.

[GBKGl04b] Sören Grimm, Stefan Bruckner, Armin Kanitsar, and Eduard Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computers and Graphics*, ?(28):719–729, 2004.

[HMSC00] Jian Huang, Klaus Mueller, Naeem Shareef, and Roger Crawfis. Fastsplats: Optimized splatting on rectilinear grids. In *VIS '00: Proceedings of the Conference on Visualization '00*, pages 219–226, 2000.

[HP94] John Hennessy and David Patterson. *Computer Organization and Design*. Morgan Kaufmann Publishers, Inc., 1994.

[KH84] James Kajiya and Brian Von Herzen. Ray tracing volume densities. *Computer Graphics*, 18(3):165–174, July 1984.

[KH01] Alexander Keller and Wolfgang Heidrich. Interleaved sampling. In S. Gortler and K. Myszkowski, editors, *Rendering Techniques 2001*, pages 269–276. Springer, 2001.

[KK99] Kevin Kreeger and Arie Kaufman. Hybrid volume and polygon rendering with cube hardware. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 15–24, Los Angeles, California, USA, 1999. ACM.

[KM05] Arie Kaufman and Klaus Mueller. Overview of volume rendering. In Christopher Johnson and Charles Hansen, editors, *The Visualization Handbook*. Academic Press, 2005.

[KS97] Gúnter Knittel and Wolfgang Straßer. Vizard - visualization accelerator for realtime display. In *HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 139–146, Los Angeles, California, USA, 1997. ACM.

[KW03] Jens Krüger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 287–292. IEEE Computer Society, October 2003.

[LC87] William Lorensen and Harvey Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, July 1987.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Trans. on Graphics*, 9(3):245–261, July 1990.

[LK04] Sarah Lakare and Arie Kaufman. Light weight space leaping using ray coherence. *IEEE Visualization*, pages 19–26, October 2004.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA, 1994. ACM.

[LMK03] Wei Li, Klaus Mueller, and Arie Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 317–324. IEEE Computer Society, 2003.

[LS97] Xian Liu and G. F. Schrack. An algorithm for encoding and decoding the 3-d hilbert order. *IEEE Transactions on Image Processing*, 6(9):1333–1337, 1997.

[Max95] Nelson Max. Optical models for direct volume rendering. *IEEE Trans. on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[MC98] Klaus Mueller and Roger Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. In *VIS '98: Proceedings of the Conference on Visualization '98*, pages 239–245, October 1998.

[MFS06] Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek. Interactive volume rendering with ray tracing. *Eurographics*, 2006.

[MHB+00] Michael Meißner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis. A practical evaluation of popular volume rendering algorithms. In *Proc. of the 2000 Volume Visualization Symposium*, Salt Lake City, October 2000.

[Mis06] Oleg Mishchenko. Optimizing cache behavior of ray-driven volume rendering using space-filling curves. Master's thesis, Stony Brook University, May 2006.

[MKW+02] Michael Meißner, Urs Kanus, Gregor Wetekam, Johannes Hirche, Alexander Ehlert, Wolfgang Straßer, Michael Doggett, P. Forthmann, and R. Proksa. Vizard ii: A reconfigurable interactive volume rendering system. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 137–146, Saarbrucken, Germany, 2002. Eurographics Association.

[MKW+04] Gerd Marmitt, Andreas Kleer, Ingo Wald, Heiko Friedrich, and Philipp Slusallek. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 429–435, 2004.

[ML94] Stephen Marschner and Richard Lobb. An evaluation of reconstruction filters for volume rendering. In *VIS '94: Proceedings of*

the Conference on Visualization '94, pages 100–107. IEEE Computer Society Press, 1994.

[MMC99] Klaus Mueller, Torsten Möller, and Roger Crawfis. Splatting without the blur. In *VIS '99: Proceedings of the Conference on Visualization '99*, pages 363–370, October 1999.

[MSHC99] Klaus Mueller, Naeem Shareef, Jian Huang, and Roger Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.

[MY96] Klaus Mueller and Roni Yagel. Fast perspective volume rendering with splatting by utilizing a ray-driven approach. In *VIS '96: Proceedings of the Conference on Visualization '96*, pages 65–72, 1996.

[NEM08] NEMA. *Digital Imaging and Communications in Medicine (DICOM), Part 3. Information Object Definitions*. NEMA: National Electrical Manufacturers Association, 2008. ftp://medical.nema.org/medical/dicom/2008/.

[NT01] Masamitsu Nishihara and Norihiko Tamaki. Usefulness of volume-rendered three-dimensional computed tomographic angiography for surgical planning in treating unruptured paraclinoid internal cartoid artery aneurysms. *Kobe Journal of Medical Science*, (47):221–230, 2001.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, July 1984.

[PHK+99] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The volumepro real-time ray-casting system. In *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 251–260. ACM Press/Addison-Wesley Publishing Co., 1999.

[RGW+03] Stefan Röttger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Straßer. Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the Symposium on Data Visualisation 2003*, pages 231–238, Grenoble, France, 2003. Eurographics Association.

[SAG+05] Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen Marschner, Erik Reinhard, Kelvin Sung, William Thompson, and Peter Willemsen. *Fundamentals of Computer Graphics*. A K Peters, second edition, 2005.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, 2008.

[SM02] Jon Sweeney and Klaus Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 95–104, Barcelona, Spain, 2002. Eurographics Association.

[Ter09] TerraRecon. Terrarecon, inc., July 2009. www.terrarecon.com.

[TSH98] Ulf Tiede, Thomas Schiemann, and Karl Heinz Höhne. High quality rendering of attributed volume data. *In Proceedings of the IEEE Visualization 98*, pages 255–262, October 1998.

[TT84] H Tuy and L Tuy. Direct 2d display of 3d objects. *IEEE Computer Graphics and Applications*, 4(10):29–33, 1984.

[WBLS03] Yin Wu, Vishal Bhatia, Hugh Lauer, and Larry Seiler. Shear-image order ray casting volume rendering. In *I3D '03: Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pages 152–162, Monterey, California, USA, 2003. ACM.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. *SIGGRAPH Computer Graphics*, 24(4):367–376, August 1990.

[WG92] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transaction on Graphics*, 11(3):201–227, 1992.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus

Gross. Ewa volume splatting. In *VIS '01: Proceedings of the Conference on Visualization '01*, October 2001.

[ZRB+04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *GI '04: Proceedings of Graphics Interface 2004*, pages 247–254, May 2004.