# 1. Introduction

Volumes of data are created and analyzed in many scientific endeavors. Often their analysis is augmented by viewing the data in its true spatial distribution as opposed to a printed array of numbers or condensed data in the form of charts and graphs. Medical images, for example, are data that are almost-exclusively analyzed by direct viewing. Two-dimensional slices of data are readily obtained from CAT or MRI scanners. These are useful in their own right for direct viewing. A stack of two-dimensional data slices form a volume data set which has the advantage that it may be viewed from an arbitrary vantage point and shows spatial relationships in three dimensions. As another example, consider a finite-element analysis that computes temperature or stress through a three dimensional structure. In many of these applications the data at each point in the volume is a scalar or a vector of values. To display this data, some mapping must be applied to convert the volume of values to intensity or color. The issue of occlusion is also confronted; are the rear portions of the volume occluded by the front, and if so how? The answers to this question is often application-dependent. For some purposes, extracting and viewing an iso-valued surface is desirable. In such cases, the occlusion of data by the isosurface representation is often desirable. In other cases, three-dimensional spatial structure is important, so all structures should be represented transparently. Volume rendering applications should be flexible enough to accommodate these and other presentation styles. The viewer should be able to select the style most appropriate to the task. Renderers usually allow arbitrary mappings from scalar volume-data to color and opacity. Also, complex pre-processing methods may be applied to emphasize regions of interest by modifying opacity and color values. If the viewer's interactions with the system produce partially-transparent objects, the final image contains contributions from the entire volume. This is a worst-case condition from the perspective of computation. If volumes are very large and the entire volume contributes to an image, proportionally large amounts of computation are required to render that image.

The high cost of volume rendering is the motivation for this research. It focuses on two issues of concern to designers and implementers of volume rendering applications:

- finding the most efficient rendering-method that provides the best image possible, and
- parallelizing the computation on multicomputers to render images as quickly as possible.

The solutions proposed here are software-solutions suitable for general-purpose workstation and multicomputer architectures. Researchers designing volume-rendering applications for such systems should find this work useful. Those doing research into hardware architectures for volume rendering may also benefit from the cost and parallel-algorithm analysis of the problem.

Three volume-rendering methods are compared with respect to their accuracy and computational expense. Two error-measurement methods are used. The first is image-based and uses a heuristic metric for measuring the difference between rendered images and reference images. The second is analytical and uses a scale-space measure of feature-size to compute an error bound as a function of feature size and sampling density. The computational expense of each rendering method is modeled and measured on several workstations. The fastest rendering-method is shown to be splatting which performs reconstruction with a two-dimensional filter. The most-accurate reconstruction is shown to be obtained with a separable cubic filter.

Parallel volume-rendering algorithms may be classified and described by the partitioning of tasks and data within the system. A taxonomy of algorithms is presented and analyzed in terms of their communications requirements. A 2D mesh network-topology is shown to be sufficient for scalable performance with the *object-partition* class of algorithms. Furthermore, we show that the network-channel bandwidth-requirement actually decreases as the problem is scaled to a larger system and volume data size.

The following two sections (1.1 and 1.2) present analytical and algorithmic models for volume rendering. Following those, two more sections (1.3 and 1.4) introduce multicomputers and parallel volume-rendering algorithms. The last section (1.5) states the thesis of this work and summarizes its main contributions.

## 1.1. Volume Rendering Model

*Volume rendering* is the term that describes the viewing of volume data as a material of variable transparency and color. Its advantages are that much or all of the volume may be visible to the observer at one time and there is no need to introduce intermediate geometry that doesn't really exist in the data. The following three-step analytical model of the volume rendering process is based on previously published derivations that model the attenuation of light as it passes through a medium of varying transparency and color {Westover91] [Drebin$^{+}$88] [Blinn82] [Kajiya$^{+}$84]. Let the real-world volume be a scalar field $G$ which is sampled at the vertices of a three-dimensional regular grid - a situation often encountered in medical and simulation data. The samples $f$ are used to reconstruct a function $F$ which may be resampled to produce images from arbitrary view points. Reconstruction is performed by convolving the samples $f$ with a filter kernel $K$ in the spatial domain. (Note that the term *filter* will be used to denote both the spatial-domain kernel and its equivalent frequency-domain filter.) In general, the reconstructed function $F$ is not identical to $G$ for any practical filter $K$. This difference is the issue of concern in the first part of this work and will be explored further in the next chapter. The rendering operations are summarized below.

1 - Reconstruct the continuous 3D scalar function $F$ by convolving each sample point $f$ with a reconstruction filter kernel $K$.

$$F = \sum_{i,j,k} f_{i,j,k} * K \tag{1.1}$$

2 - Apply an opacity $O$ and shading $S$ function to the continuous scalar field. These user definable transfer functions yield a differential opacity $\Omega = O(F)$ and color emittance $E = S(F)$ at each point in the volume as a function of the scalar field properties at that point.

3 - Integrate an intensity and transparency function along sample viewing-rays through the volume. The integrals may be taken toward or away from the viewer. When taken towards the viewer, the accumulated transparency $T$ and intensity $I$ along the sample ray over $[0, p]$ ($p$ is closest to the viewer) is

$$T(p) = \exp(-\int_0^p \Omega(\nu)\, d\nu) \tag{1.2}$$

$$I(p) = T(p) \int_0^p E(\nu) / T(\nu)\, d\nu \tag{1.3}$$

The intensity function (Eq. 1.3) has a closed form solution only if $E$ and $\Omega$ are constant or multiples of each other over the interval. This is not generally the case, so numerical methods are used to approximate the whole integral by breaking the path into small segments each of which are themselves approximated with constant $\Omega$ and $E$ function values. For a segment with interval $[0, p]$, the approximate $T$ and $I$ values become

$$T(p) = \exp(-\Omega p) \tag{1.4}$$

$$I(p) = E / \Omega\, (1 - \exp(-\Omega p)) \tag{1.5}$$

The segments are then composited together to produce the integral of the entire path. Compositing combines the segments consistent with the model expressed by equations 1.2 and 1.3 [Drebin+88]. It is a sequential, associative operation that combines segments in front-to-back or back-to-front order. The back-to-front method is the simpler of the two and presented first. $\text{Segment}_i$ and $\text{segment}_{i-1}$ each have a ⟨color, alpha⟩ associated with them where color = $I$ and alpha = $1 - T$. If $\text{segment}_i$ is closest to the viewer, compositing yields a new combined segment with

$$\text{color} = \text{color}_i + \text{color}_{i-1}\, (1 - \text{alpha}_i) \tag{1.6}$$

This proceeds recursively until all segments are composited producing a color for the entire path. If $\text{segment}_i$ is closest to the viewer and compositing proceeds front-to-back,

the combined segment's color and opacity (or alpha value) must be maintained to allow recursion; the combined segment becomes the new front segment and the opacity of the front segment ($alpha_i$) must be known.

$$alpha = alpha_i + alpha_{i-1} (1 - alpha_i) \quad (1.7)$$

$$color = color_i + color_{i-1} (1 - alpha_i) \quad (1.8)$$

There are three parameters that affect the approximation accuracy for each segment: the interval or segment size, the function which approximates a constant $\Omega$ and $E$ over each interval, and the method of approximating the exponential terms. Wilhelms and Gelder [Wilhelms+91] analyze these issues and show some results obtained by varying these parameters. Several options are possible for evaluating the exponential terms of equations 1.4 and 1.5 quickly. If $\Omega$ is encoded as an integer of relatively few bits precision, a lookup table can compute the exponential term. We may simplify the exponential by approximating $e^{-\Omega p}$ with $\min(1,\Omega p)$. This has the added benefit of setting the range of the opacity function to [0, 1]. Similarly, the emittance function $S$ may be considered to embody the division by $\Omega$ simplifying equation 1.5 to

$$I(p) = E \ (1 - T(p)) \quad (1.9)$$

Both of these latter approaches are extensively used in actual implementations. Due to the nature of the study being conducted here, we will not complicate the rendering error measurements by allowing arbitrary $O$ and $S$ functions. These functions are application specific and frequently nonlinear. To isolate and focus on the reconstruction errors, we will consider $O$ to be the identity function and $S$ to always yield a constant value of one. These definitions for $O$ and $S$ equate to considering the data as glowing in proportion to its value. We are rendering the data as directly as possible. Any error in the rendering process can only be attributed to the reconstruction and resampling process. This is the part of the rendering process on which we focus.

## 1.2. Volume Rendering Algorithms

In addition to the above analytical model, an algorithmic approach is presented that focuses more on the computing process involved in rendering an image of a volume from an arbitrary view point. For simplicity, we'll consider a scalar field that is sampled on a regular grid in $\Re^3$. The small points connected by dark dashed lines in figure 1.1 correspond to such an array of samples. Their lattice is called the *object lattice*. When viewing the data, we embed another lattice in the volume which is aligned with the view direction. These image-space lattice points lie on pixel coordinates at multiple depths. The grey-line grids in figure 1.1 represent this *image lattice*. Each grid is shown as a plane of constant depth although this is not a necessary constraint. In fact, there may

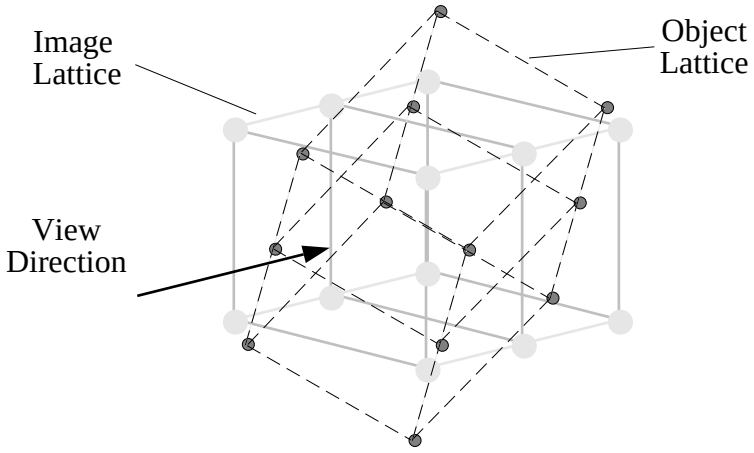


Fig. 1.1 - Embedded image and object lattices

be a different number of image lattice points along the view direction behind each pixel, and they may be different distances from each other.

The data field values on the object lattice are known. As the first step in rendering an image, we must calculate the field values on the image lattice points. This requires reconstructing the continuous 3D field from the object lattice data and resampling it at the image lattice points. It is in this step that volume rendering algorithms differ the most. Different algorithms use different reconstruction techniques and compute the image lattice values in different orders. These differences are explored in the first part of this work (Chapter's 3, 4, and 5). The accuracy of the reconstruction methods are compared and contrasted with their relative cost. The results allow a more informed choice among the methods and their trade-offs.

The second rendering step is to shade and classify the new sampled values on the image lattice. These functions are the emittance *E* and opacity *O* functions previously described. Shading and classification are functions of the local properties of the field; among these are its value, gradient magnitude and gradient vector. Classification assigns an opacity (alpha) value to each image lattice point. These opacity values are derived from a user-defined function that maps the resampled field values to an opacity. Alpha values range over [0, 1] and are a measure of the attenuation of any light passing through that point's local region. Shading may use the gradient vector at each point as a normal vector and apply diffuse and specular lighting models to derive a color. Since these functions are application specific and user-controlled, and we are only interested in comparing reconstruction techniques, we will use the identity function for opacity *O* and the unit constant function for emittance *E*. The important result of this second step is that each point of the image lattice will now have a four-tuple ⟨R, G, B, A⟩ of color and opacity associated with it. This tuple may also be written as ⟨color, alpha⟩.

The final rendering step is to composite the image-lattice tuples along the image z-axis to

arrive at a final color for each pixel in the image-plane. The image lattice tuples represent the light emission and attenuation in the local region about each point.

There are three common approaches to performing the above steps in actual implementations. These three rendering methods utilize the reconstruction filters we will compare. They are introduced here and will be described in greater detail in later sections.

**Ray Casting** - The volume is resampled along view rays [Levoy88] [Sabella88] [Upson+88]. Typically, reconstruction is done by trilinear interpolation of the eight neighboring object lattice values. The $\Omega$ and $E$ functions are applied to the new sample points along the rays. Successive samples along a ray are composited to produce the final ray color which corresponds to the color of one pixel in the final image. Affine and perspective projections are computed with approximately the same cost.

**Splatting** - This approach computes the effect of each object lattice point on the pixels near the point to which it projects. Object lattice slices are sorted in depth order and reconstructed by convolution with a 2D Gaussian filter kernel [Westover89]. The reconstructed 2D function is projected, resampled, and accumulated on an image lattice plane. Successive image lattice planes are composited to produce the final image. With splatting, affine projections are more efficient to compute than perspective projections.

**Volume Shearing** - An affine view transformation is decomposed into three sequential volume shear operations. Each shear is affected by a 1D transformation of the form $x' = Ax + B$ [Drebin+88] [Hanrahan90]. Since these transformations require only a 1D reconstruction filter, cubic splines are commonly used to facilitate the resampling. After three shears, the volume lies on the image lattice ready for compositing. Perspective projections are produced by performing two additional shear operations.

## 1.3. Multicomputer Architectures

Multicomputers are a source of the computation power and memory size that volume rendering requires. In this work, any system with multiple computing elements or nodes is considered a multicomputer. Each node is assumed to have some physically local memory. We distinguish between multicomputers by the communication model and hardware topology of the network between the nodes. These issues are orthogonal and form a useful classification for this work.

The major distinction with respect to an application's view of the network is the model for how non-local memory is accessed. In one model called *message passing*, each node considers its local memory to be the only directly addressable memory available. Remote memory accesses are explicitly controlled by the application program. If one node

requires data from another node's local memory, explicit operating system calls are made to send and receive messages containing that data. As an example, node A sends a message to node B requesting some data. Node B receives the request, interprets it, formats the requested data and sends it to node A. Node A receives the data and places it into its local memory. In the alternative model, communication is invisible to the application and occurs under the control of the operating system and hardware as an indirect result of the memory accesses made at each node. In this approach, nodes directly address all the memory in the system regardless of where it physically resides; this model is referred to as *shared memory*. At this point in the evolution of multicomputer systems, many of them use a message-passing model [Delta] [Paragon] [Pxpl5] [CM5]. Several experimental systems are evolving based on the shared-memory model [Dash] [JMach] [Mosaic]. Some commercial shared-memory systems have been built (BBN and Sequent), and are available at this time [Kendall]. A major difference between these two models is the latency with which remote memory is accessed. A system using a shared-memory model will have up to two orders of magnitude lower latency when accessing remote memory even when the communication channels have the same bandwidth. This arrises from differences in the low-level communication hardware and firmware, neither of which are modifiable by a user's program. This has important ramifications in terms of the reconstruction method and parallel algorithm suitable for such a system.

It is possible to implement a shared-memory model in software on top of a message-passing model (and vice-versa). A minimal requirement for a shared-memory system is the presence of memory management hardware to trap the remote accesses and re-map local memory. Once an access causes a trap, hardware and firmware mechanisms perform the actual data transfers. The class that a particular system is in depends on the operating system provided by the manufacturer. It is assumed that the manufacturer makes optimal use of the hardware capabilities in the system.

The network topology is the other distinguishing feature of a multicomputer. Common topologies are 2D and 3D meshes and tori [Delta] [Paragon] [Dash] [JMach] [Mosaic]. A mesh that wraps around in each dimension becomes a torus. These are both popular due to the property that each node has a constant valence, or number of network ports, for all system sizes. Examples of other topologies are fat-trees [CM5], and rings [Pxpl5] [Kendall]. Because of the commercial popularity and amount of published analysis done for meshes, the volume rendering algorithms are analyzed in the greatest detail for them.

Of greatest concern, from our standpoint, is the time spent on communication instead of productive computing. To the extent that a node expends its cycles on communications overhead or waiting for messages, the node is not rendering an image. This is one of the costs or penalties of a parallel algorithm. One goal of this work is to find the lowest cost partition for a mesh or torus topology. Rather than exhaustively test the cross-product of all rendering methods with all parallel algorithms, we treat them as orthogonal issues and

optimize each separately. The algorithm cost is related to the quantity of data communicated per-frame; the partition with the smallest communication requirement is likely to perform best when coupled with the fastest rendering method. In this work, an analysis of parallel algorithm costs for mesh and toroidal topologies is performed. The results show that if the image size is kept constant, meshes scale well for object partition algorithms. For an algorithm to *scale well,* its communication time on a given network topology must remain constant as the number of nodes and the volume size is increased proportionally. For object-partition algorithms on mesh topologies, when the image size is held constant as the volume and system sizes increase, the communication time actually decreases.

The writing of final image values into a frame buffer is not accounted for in the algorithm costs. It is assumed that frame buffer update is a constant-cost operation for all algorithms. Most multicomputers do not incorporate a frame buffer directly into their architecture, making it difficult to analyze the actual cost of this operation. The assumption that sufficient I/O bandwidth is available to allow frame buffer update is justified by the fact that the required network bandwidth for frame buffer update is usually a small fraction of that required by the rendering algorithm.

## 1.4. Parallel Algorithms

Since volume rendering is computationally expensive and data sets may be very large, it is only natural to turn to parallel methods. In designing a parallel algorithm, the tasks and data are distributed among the system resources in a fashion that minimizes the undesirable side-effects of parallelization. The four major considerations are outlined below.

1 - Duplicating computations on multiple nodes wastes computing cycles.

2 - Communication is expensive. In most message passing systems, latency and per-message overhead is high so algorithms that produce flurries of small messages should be avoided. It is usually more efficient to send fewer but larger messages.

3 - Volume data is often large. It is impractical or impossible to store a complete copy of the data at each node. The data must be distributed over the memory spaces in the system.

4 - Load imbalance can reduce the speedup we hope to obtain by parallelization. Since volume rendering is a view-dependent computation, load balancing should be done dynamically.

The space of parallel algorithm design choices is very large. To cope with the

complexity and systematically consider all the alternatives, the problem is broken into several orthogonal design alternatives. The selection of alternatives is helped by considering the parallelism inherent in the volume rendering process. Volume rendering algorithms offer primarily data parallelism in the following tasks:

1 - The reconstruction and resampling step may be performed in parallel for all image or object lattice points.

2 - The classification and shading may be performed in parallel for all image lattice points.

3 - The compositing task may be performed in parallel for all image pixels.

From this analysis of the problem, it is apparent that mapping lattice points to processing nodes is a good approach to the problem. There are, however, many choices as to which lattice points to map to which nodes. There are two ways to approach this mapping.

**Image Space Task Partition** - In this partition, a fixed, contiguous, sub-volume of image lattice points is mapped to each node. The object lattice data needed to reconstruct a node's assigned image lattice values must be communicated as the view changes. Classification, shading, and compositing can be performed for the entire sub-volume at that node without further communication. Compositing between nodes may be necessary if one node's sub-volume occludes another.

**Object Space Task Partition** - In this approach the mapping of image lattice points to nodes is a function of the view transformation. A fixed, contiguous, sub-volume of object lattice points is mapped to each node, and we compute the image lattice points that are embedded in them under the view transformation. Classification, shading, and compositing can then be performed for the entire sub-volume on that node without further communication. Compositing between nodes will be required since the occlusion relation is complex making it likely that one node's sub-volume partially occludes another's.

One assumption that has already been made is that the sub-volumes mapped to nodes are always contiguous. In many parallel algorithms load balance is achieved by interleaving data. Interleaving in three dimensions does not work well for volume rendering since compositing must be done in view order. Interleaved data would cause a large communications penalty. Two and one dimensional interleaving of the image lattice has been used with ray casting [Corrie+92] and found inefficient relative to contiguous sub-volumes which take advantage of the coherence in data access patterns. One dimensional interleaving has been used with splatting where it also caused increased communications cost [Westover91]. For this and other reasons to be discussed in chapter six, contiguous sub-volumes are usually desirable and other load balancing techniques

must be employed.

The data dependencies arising from a task partition must be examined since they give rise to communication between nodes and communication costs are a primary concern. For a given task partition, communication costs are a function of the sub-volume topology. The topology is related to the parallelism grain size. Sub-volume topologies are classified as *slabs*, *shafts*, and *blocks*. Slabs are distinguished by their spanning the total volume in two dimensions, while shafts span the whole volume in just one dimension. A block does not span the volume in any dimension. Opposite faces of slabs, shafts, and blocks are always parallel and aligned with the lattice in which they exist. This last condition is imposed to simplify array index computation.

There is structure to the communication involved in each step of the rendering process. During the resampling step, each image lattice point reconstructs its field value from multiple object lattice points that transform to nearby locations. The transformation is linear and therefore causes communication patterns that are regular; neighboring object points will be needed by neighboring image points. This regularity can be exploited to help minimize the communications costs of some algorithms.

The compositing step involves collapsing columns of image lattice points along the z- axis. This produces a regular communication pattern, if any at all, and the routing regularity again aids in the optimization of this step.

The classification and shading functions are application dependent. Depending on the selected approach, this step may or may not involve communication. The communication costs arising from complex classification functions are not considered in this work. Complex classification can be viewed as a batch process performed periodically but not necessarily within each frame.

## 1.5. Thesis and Contributions

### 1.5.1. Thesis

Volume rendering algorithms are distinguished by their reconstruction methods. The accuracy of the volume reconstruction directly determines the quality of the image.

- A separable cubic filter provides more accurate volume reconstruction than a pyramid filter or a Gaussian filter.
- Splatting is a more efficient volume reconstruction method than ray casting or volume shearing.

Parallel algorithms for volume rendering on multicomputers differ in how the tasks and data are partitioned over the architecture.

- An object partition has the lowest communication costs and, for a constant image-size, scales well on 2D mesh network topologies. The network-channel bandwidth-requirement actually decreases as the problem is scaled to larger systems and volume data sets.

### 1.5.2. Contributions

- Image comparisons and analytical techniques are used to compare the accuracy of three reconstruction filters used in volume rendering. The results demonstrate that a separable cubic filter is the most accurate of the three.

- Implementations on four workstations and analysis confirm that splatting with a 2D filter kernel is more efficient per reconstructed-point, than ray casting or volume shearing.

- A taxonomy of parallel algorithm task and data partitions is presented. Three algorithms are presented as optimal in their class, and their communication requirements are parameterized. Two of the three optimal algorithms are presented for the first time.

- The class of object partition algorithms is shown to scale better than image partitions on mesh and toroidal networks. Simulations and tests on the Touchstone Delta verify that the communication time decreases as the system and data size increase if the image size remains constant.

- Implementations on Pixel-Planes 5 and the Touchstone Delta demonstrate the performance of a new object partition algorithm.

- A new load balancing method for object partition algorithms is demonstrated.

- The third pass of the the three pass volume shear rendering method is shown to be unnecessary. This resulting in a thirty-percent savings of computation and memory.