University of California
Santa Cruz

Parallel Direct Volume Rendering of Intersecting Curvilinear and Unstructured Grids Using a Scan-line Algorithm and K-D Trees

A thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer and Information Sciences

by

Paul D. Tarantino

June 1996

The thesis of Paul D. Tarantino is approved:

Dr. Jane Wilhelms

Dr. Allen Van Gelder

Dr. Charlie McDowell

Dean of Graduate Studies and Research

Contents

List of Figures

List of Tables

Parallel Direct Volume Rendering of Intersecting Curvilinear and Unstructured Grids Using a Scan-line Algorithm and K-D Trees

Paul D. Tarantino

ABSTRACT

Scientific visualization of volume data is increasingly being used in applications such as Medical Imaging and Computational Fluid Dynamics (CFD). Due to the irregular nature of the geometry of CFD applications, volume data for CFD applications is often stored in curvilinear or unstructured grids. This increases the computational and memory resources needed to render the volume due to the limited amount of geometrical coherence that can be exploited.

A method of rendering multiple, possibly intersecting, curvilinear and unstructured grids is presented. This method, which has been implemented for a MIMD parallel architecture is based on a scan-line algorithm and provides ray-cast quality images efficiently by taking advantage of polygonal coherence between scan-lines. It is capable of rendering intersecting grids without the computational and spatial expense of subdividing intersecting cells, which most ray-casting and all cell projection methods must do. Further efficiency is obtained by the use of a k-d tree with prefix-order partitioning of triangles, to reduce the number of primitives that must be processed for one rendering. All rendering is done in software which eliminates the need for specialized graphics hardware.

The performance and rendering quality of this algorithm are compared with hardware-assisted cell projection algorithms, for both rectilinear and curvilinear grids, and with a ray-casting algorithm for rectilinear grids.

Keywords: Computer Graphics, Direct Volume Rendering, Curvilinear Grids, Unstructured Grids, Hierarchical Trees

Acknowledgments

I would like to thank all of my committee members - Dr. Jane Wilhelms, Dr. Allen Van Gelder, and Dr. Charlie McDowell for their dedication and time spent on this project. A special thanks goes to Dr. Jane Wilhelms for giving me the opportunity to pursue studies in scientific visualization and for supporting me throughout my attendance here at UCSC and to Dr. Allen Van Gelder for spending many hours working with me on implementation and optimization of the algorithms.

I would also like to thank my wife, Christine, and my mother, Ida, for their unending support and patience while I pursued my studies.

This project was made possible due to funds allocated by NAS/NASA-Ames Research Center, Moffett Field, California, Grant Number NAG-2-991, and by the National Science Foundation, Grants Number CCR-9503829 and CDA-9115268.

1. Introduction

Direct volume rendering is one of the few methods of displaying large, densely sampled sets of data. A useful aspect of direct volume rendering is that it provides a view of the entire data set in a single image, which can convey a large amount of information to the scientist or researcher that is viewing the image. Datasets from computational fluid dynamics and finite element analysis are often represented by curvilinear or unstructured grids, which are difficult to render. Multiple, intersecting grids make accurate rendering even more difficult for cell projection and ray-casting methods that can't handle two different cells sharing the same space. Large data sets also present problems for renderers that can't efficiently determine which parts of the volume contribute to the image. Accurate perspective viewing requires special care when projecting from world space to screen space.

The direct volume renderer presented in this thesis addresses four main topics:

1. A scan-line algorithm is presented that can handle multiple intersecting curvilinear grids, unstructured grids, and embedded surfaces without subdividing cells or polygons. Because it is implemented in software, it does not need specialized graphics hardware and is portable.
2. Parallelization of the algorithm has been implemented on a four-processor MIMD machine with an observed speedup factor of 3.25.
3. A new perspective transformation is presented that achieves greater numerical accuracy for depth calculations.
4. A hierarchical partitioning of the volume allows efficient identification of sub-volume regions that contribute to the image without exhaustively processing each volume primitive. It also allows approximation of sub-volumes which can be rendered extremely fast for preliminary viewing.

The next chapter discusses existing volume rendering techniques and related work. Chapter 3 presents the rendering application including the scan-line renderer, perspective transformation, parallelization, and hierarchy. Chapter 4 gives performance measurements

of the algorithm on various grids and compares it to other locally written rendering programs. Conclusions are presented in Chapter 5.

2. Background

Many papers and techniques for direct volume rendering have been written and implemented in the past ten years. Each of these techniques tries to address some or all of the desired properties of a good volume renderer which are:

- Picture Quality
- Speed (interactive rates are desirable)
- Parallelism (both SIMD and MIMD)
- Flexibility (different types and combinations of volume data)
- Space Efficiency (ability to render large data sets)
- Portability (hardware independent)

This section will introduce the reader to several of the most important direct volume rendering algorithms along with volume rendering terminology. Curvilinear and unstructured grids are described along with the problems associated with rendering them. Several approaches to solving these problems are discussed, including the use of hierarchical subdivision and resampling of the data into rectilinear space.

2.1 Volume Rendering

A general definition of volume rendering is – "any method used to render a volume of three-dimensional data." Sample volume data consists of information concerning locations at a set of points in space. This data may take the form of a single scalar value (such as density), a vector (such as velocity), or a combination of each. The algorithm presented in this paper deals with rendering a volume of scalar values. The color and opacity at a point in the volume can be specified as functions of a 3-D scalar field being visualized, and may also involve the gradient of the scalar field and the lighting direction. There are several styles of volume rendering which include cutting planes, iso-surfaces, and direct volume rendering.

Cutting planes are implemented by cutting a user-specified plane through the volume. The plane is drawn onto the display using colors that correspond to the scalar values at each point on the plane. Colors are obtained from scalar values through the use of a transfer function. The transfer function is a user-specified mapping of color and opacity to data values. NASA's Fast program [Bancroft *et al.*, 1990] has a cutting planes tool.

An iso-surface is the set of all points in the volume with a given scalar value [Foley *et al.*, 1990] and is a three-dimensional extension of two-dimensional contour lines. Iso-surfaces can be extracted from volume data as a polygonal approximation [Lorensen and Cline, 1987]. Cells, defined by the data sample points at the corners, are each examined to determine if an iso-surface intersects the cell. These intersections are combined to form a polygonal surface.

The goal of direct volume rendering is to produce an image of a varying density volume by projecting it onto an image plane. Direct volume rendering uses semi-transparency, therefore, every cell in the volume can contribute to the final image. This paper deals primarily with the topic of direct volume rendering. The following sections describe volume representations and techniques for direct volume rendering.

2.1.1 Curvilinear and Unstructured Grids

Regular rectilinear grids permit many kinds of optimizations since the spacing between samples is constant. This allows fast stepping from one cell to the next as well as easy calculation of spatial coordinates for any sample by its grid index (i, j, k) by multiplying each index with the constant delta spacing factor for that dimension $(i * dx, j * dy, k * dz)$. Orthographic projection of a rectilinear grid gives even more coherence, since the projection of all of the cells will be identical [Wilhelms and Van Gelder, 1991]. Medical imaging is one important application that makes use of regular rectilinear grids.

There are many other applications, however, which create volume data sets that are not rectilinear. These applications include computational fluid dynamics (CFD) and finite

element modeling (FEM). Due to the irregular nature of the geometry of CFD and FEM applications, volume data for these applications is often stored in curvilinear and unstructured grids.

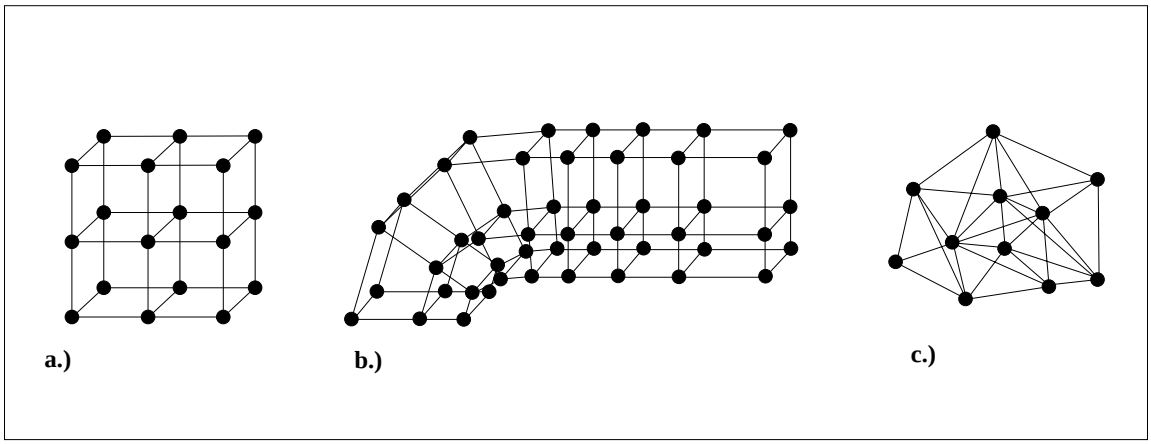

Figure 2.1: Sample grids with their cell representations. a) Regular Rectilinear b) Curvilinear c) Unstructured

Curvilinear grids are similar to regular rectilinear grids in that each point can be indexed by (i, j, k), however, curvilinear grids allow arbitrary positioning of the point locations. Curvilinear grids can be thought of as regular rectilinear grids that are warped in space to give a denser sampling of points around a particular region of space (see Figure 2.1). Since curvilinear grids can follow a curved surface, they are used in CFD applications and are usually fitted around the surface that is being studied by applying a formula to the space or by hand-fitting the grid to the desired space. Rendering algorithms do not usually have access to the method used to warp the grid, therefore, memory must be used to store the spatial coordinates of each point in the grid. One of the data sets that this paper focuses on is the NASA Space Shuttle [Buning *et al.*, 1989] which consists of nine curvilinear grids that intersect and occupy the same space, in some regions, with different sampling resolutions. Figure 2.2 shows four of the nine intersecting curvilinear grids that make up the space shuttle data set.

An unstructured grid is simply a set of sample points with arbitrary locations in space. Actually, unstructured grids are not defined over a grid at all, but consist of scattered data

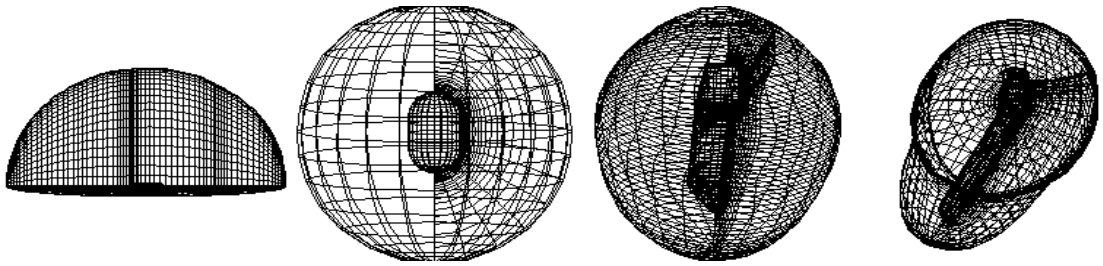

Figure 2.2: Four of the nine intersecting curvilinear grids that make up the space shuttle data set.

points. Some unstructured grids consist of a list of points with neighbor relationships that form tetrahedra cells. When neighbor relationships are not defined by the sample data, most rendering algorithms will triangulate between sample points to form a grid of tetrahedral cells (see Figure 2.1).

One of the problems with rendering curvilinear and unstructured grids is that a front-to-back traversal ordering of cells is not implicit in the volume and some sorting approach is necessary to find it [Van Gelder and Wilhelms, 1993, Max *et al.*, 1990, Williams, 1992b]. Methods such as ray-tracing (described below) require more computation to determine ray to cell face intersections, since a ray may exit a cell by any face, including the one that it entered [Wilhelms, 1993]. Multiple grids with intersecting cells present problems for cell projection algorithms, since cells that intersect must be subdivided to insure that the same region of space is not projected twice, thus producing an incorrect image. The method presented in this paper shows how to handle intersecting cells by decomposing and sorting the cell faces.

2.2 Approaches to Direct Volume Rendering of Curvilinear and Unstructured Grids

There are several techniques to direct volume rendering. The major methods include splatting, cell projection, ray-casting, and scan-line algorithms. Some recent papers [Ebert *et al.*, 1994, Yagel *et al.*, 1995] have suggested that the best rendering application should include a combination of several methods that would give scientists a wide range of rendering

capabilities including rapid navigation through data, emphasizing specific features of the data, and producing a realistic rendition of the volume.

Each of the methods described below has advantages over the other methods in one or more of the desired properties of a good volume renderer. Speed seems to be one of the most important properties of a good volume renderer. Wilhelms and Van Gelder [Wilhelms, 1993] give results that show cell projection methods of regular grids (resampled from curvilinear grids) to be many times faster than ray-tracing and they suggest that projection of curvilinear grids can provide a desirable option to ray-casting. Later in this paper, a curvilinear grid cell projection method by Wilhelms and Van Gelder [Van Gelder and Wilhelms, 1993] is compared, with respect to speed and picture quality, with the scan-line method described in this paper.

The following sections give a basic description of several direct volume rendering algorithms along with some of the good and bad points of each method. Basic descriptions of parallel implementations of these algorithms are also given.

2.2.1 Splatting

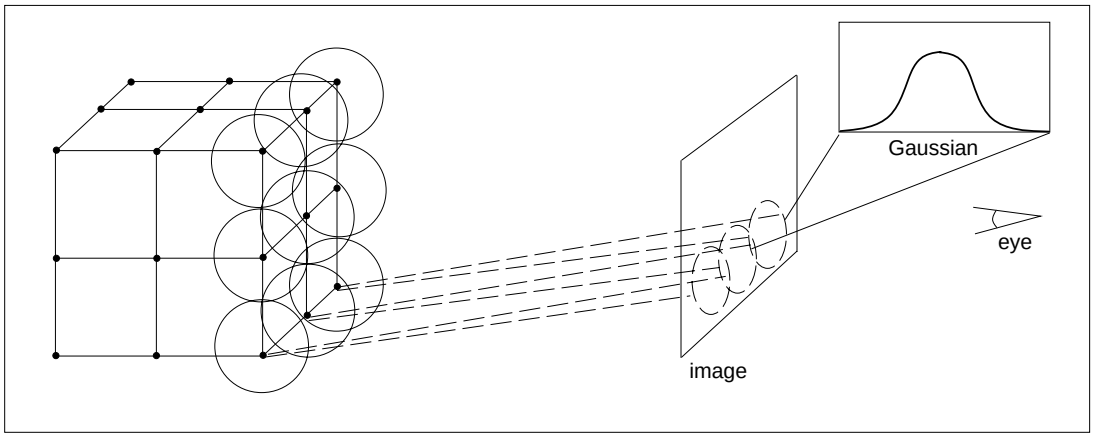

Figure 2.3: Splatting - Each voxel contributes to an elliptical region on the screen according to a Gaussian weight.

Splatting gets its name from the fact that data primitives are projected onto the screen similar to splatting a snowball against a wall. Westover [Westover, 1990] approximated the

splat of a voxel with a 2D spherical Gaussian where each data sample contributes to an overlapping elliptical region of the final image. Laur and Hanrahan [Laur and Hanrahan, 1991] also use RGBA polygons to approximate (splat) the projection of a cell on the screen. This technique was extended to curvilinear grids by adaptively resampling the grid in 3D space and producing a set of fast spherical and slower ellipsoid splats that approximate the curvilinear grid [Mao *et al.*, 1995]. Ihm and Lee [Ihm and Lee, 1995] accelerate the splatting of rectilinear grids by keeping an index of which volume elements (voxels) have scalar data values for a particular range of interest. Voxels that are not of interest can be easily ignored while those that are of interest can be splatted.

2.2.2 Cell Projection

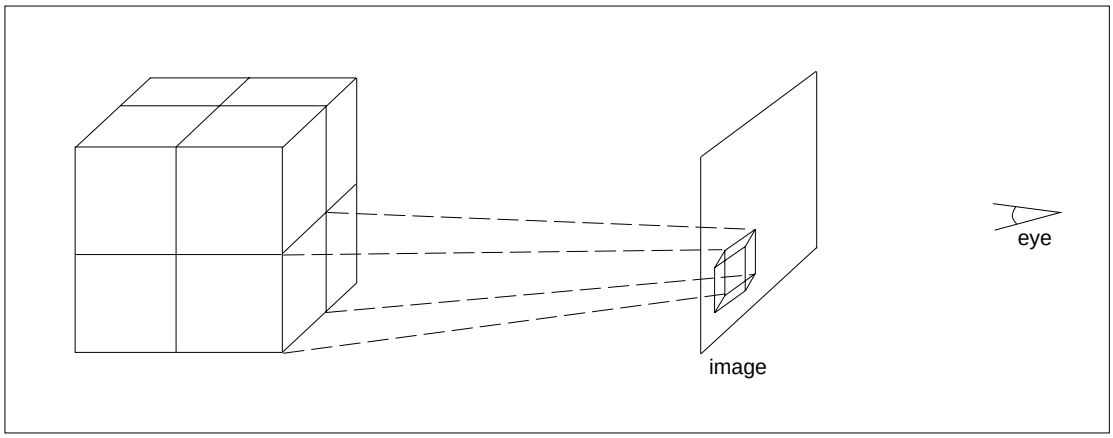

Figure 2.4: Cell Projection - Cells are projected as polygons onto the screen.

Cell projection is similar to splatting in that the volume is projected onto the screen; however, cell projection breaks the projection of the cell into a set of polygons. The set of polygons that represents a projected cell depends on the cell type and the geometric orientation of the cell with respect to the eye. The polygons are determined as being regions that have the same front and back faces in the projection of the cell. (see Figure 2.4). Each cell in the volume is projected in visibility order onto the screen to produce the final image.

Upson and Keeler [Upson and Keeler, 1988] describe an algorithm that projects each cell to the screen plane and then processes the projection using a scan conversion technique. Shirley and Tuchman [Shirley and Tuchman, 1990] decompose hexahedra cells from rectilinear and curvilinear grids into five tetrahedra cells and project them onto the screen. Unstructured grids can also be represented by tetrahedra, which makes their method very general with respect to the types of grids. They show that a tetrahedral projection can be represented by one to four triangles depending on the number of visible faces and the rotation of the cell.

Wilhelms and Van Gelder [Wilhelms and Van Gelder, 1991] show that a rectilinear hexahedral cell projection can be represented by one to seven triangular and/or quadrilateral polygons depending on the number of visible faces and the rotation of the cell. They describe a coherent projection approach [Wilhelms and Van Gelder, 1991] in which a regular rectilinear grid can be projected very efficiently due to the fact that every cell will project to the same set of polygons, so calculations for a single cell can be reused for the entire volume. All that is left to do is translate the polygons to the correct (x, y) positions on the screen, calculate the color and opacity of each vertex and pass the information to a polygon renderer, which could be implemented in hardware. They also point out that using Gouraud shading hardware, in general, can cause visual artifacts due to truncation and rounding of small contributions of color and opacity. Gouraud shading artifacts are also introduced by the linear interpolation of non-linear opacity gradients between pixels.

Williams [Williams, 1992a] uses the term *splatting* when referring to projecting tetrahedra cell faces with color and opacity approximations onto the screen. His technique, which is considered a form of cell projection, approximates the color contributions from each cell several ways including Gouraud-shaded polygons using a packed 32-bit color/opacity value at each vertex, a voxel model with a single 32-bit color/opacity value for each cell, and a uniform thickness slab approximation that pre-computes the color/opacity values of each vertex as well as the thickness of each cell. Each of these approximation methods trades off image quality for faster image generation times.

Wilhelms and Van Gelder [Van Gelder and Wilhelms, 1993] present three direct volume rendering methods of curvilinear grids via cell projection and hardware polygon rendering. Each method has a trade-off between speed and image quality. The method with the best image quality is called "Incoherent Projection", which is an extension of their rectilinear "Coherent Projection" approach, except that each hexahedra cell has to be analyzed individually. Comparisons with coherent and incoherent projection methods are presented later in this paper.

Recent work [Cignoni *et al.*, 1995] has been done on optimizing the tetrahedra cell projection of curvilinear and unstructured grids. These optimizations include -

- A fast visibility sorting method called *numeric distance sort* that was first described by Max, Hanrahan and Crawfis [Max *et al.*, 1990].
- Approximation of the projected tetrahedra polygons through the use of the tetrahedra's centroid.
- Data reduction of the grid via multi-resolution representation of the grid using adaptive incremental triangulation.

All of the optimization methods have a negative effect on image quality, however the authors suggest that these trade-offs are acceptable when compared to the benefits of interactive response times.

2.2.3 Scan-line

Some projection methods process the volume as cell faces [Max *et al.*, 1990, Lucas, 1992, Challinger, 1993] or line segments intersecting scan planes [Giertsen, 1992, Giertsen and Peterson, 1993], thus getting the name *scan-line* algorithms. They all take advantage of spatial coherence by transforming the volume into screen space and rendering the cells, or faces of cells, in front to back order for each scan-line. Coherence is achieved by processing the scan-lines and pixels in sequential order. Scan-line algorithms provide maximum flexibility, accommodate parallelization, permit the rendering of multiple intersecting grids of any type, and allow inclusion of intersecting polygonal data within the volume. They

also dissociate rendering from any particular platform, because the process occurs strictly in software. A detailed description of the scan-line algorithm used in this paper is given in later sections including figures, therefore, it is omitted here.

The scan-line algorithm described in this thesis is an extension from several related works [Max *et al.*, 1990, Challinger, 1993, Lucas, 1992]. Judy Challinger's paper [Challinger, 1993] presents a scan-line algorithm that is the most similar to the one in this paper except for the following differences.

- Her algorithm was only implemented for orthographic projection. The algorithm in this paper does both orthographic and perspective projection.
- She uses screen tiles to break up the work for parallelization, which gives results that are fairly scalable (up to 110 processors), but require more computation due to the reduced coherence of processing tiles individually. The algorithm in this paper uses a *next available scan-line* approach that maximizes the use of coherence and gets good scalability for 1 to 10 processors.
- Her implementation is only for regular and curvilinear grids. She stores each face of the hexahedral cells as five edges (four for the face and one to triangulate the quadrilateral face), and triangulates the face later in the algorithm, when calculating pixel intersections. The algorithm in this paper uses triangles as input from the data module, therefore, it handles rectilinear, curvilinear, and unstructured grids with no extra implementation issues.
- Her algorithm uses a flat shading model for surfaces. The algorithm in this paper uses a Phong lighting model at the vertices and interpolates the color values as if they were scalar data in the grid.
- She uses bounding boxes around each face to determine which tiles the face projects to, however, if the bounding box projects to a tile and the actual face does not project to that tile, then time is wasted checking that face with that tile. The algorithm in this paper determines pixels that intersect a triangle by using accurate Y-bucket and X-bucket calculations.

- Her paper does not mention user-defined clipping planes or how to handle external faces if some of them have been clipped by any clipping planes. The algorithm in this paper allows six orthogonal user-defined clipping planes that correspond to a region of interest in the volume. Determining which external faces enter the grid and which exit a grid for a given ray are handled and discussed in Section 3.1.5.

Another scan-line algorithm that is closely related to the one in this paper is that of Max, Hanrahan, and Crawfis [Max *et al.*, 1990]. Some of the differences between their algorithm and the algorithm presented in this paper are:

- Their scan-line algorithm renders polyhedral cells using edges in Y-buckets. The algorithm in this paper renders a set of triangles and uses triangle identifiers (ID's).
- Their algorithm splits rectilinear and curvilinear cells into five tetrahedra, if any cells are intersected by a contour surface. The algorithm in this paper does no subdivision and surfaces are represented as an independent set of triangles.
- Their algorithm achieves visibility ordering by depth sorting convex polyhedra. This means that they would have to subdivide intersecting cells such that no two cells share the same space. The algorithm in this paper is not cell-based, therefore no subdivision of intersecting cells is required. Triangles are depth sorted during X-bucket creation and maintained in sorted order when updating the X-buckets.

The algorithm described by Lucas [Lucas, 1992] is also similar to the one presented in this paper. Some of the differences between his algorithm and the algorithm presented in this paper are:

- Lucas uses screen tiles (similar to Challinger) to break up the work for parallelization. The algorithm in this paper uses a *next available scan-line* approach for parallelization.
- Lucas sorts cell faces by the z values of their centroids and states that "this results in few mis-sorted pixel face intersections". The algorithm in this paper maintains the cell faces in sorted order from one pixel to the next.
- Lucas uses two z-buffers to implement clipping of the volume by an arbitrary convex three-dimensional shape. The algorithm in this paper uses clipping planes which are

inserted into the same active lists as the cell faces.

- Lucas renders translucent objects in back-to-front order. The algorithm in this paper uses front-to-back ordering so that rendering stops if a pixel is opaque.
- In this paper, a new version of the perspective transformation is used to achieve greater numerical accuracy in the neighborhood of the transformed objects.

The algorithm in this paper makes use of a hierarchy over irregular data, based on k-d trees. This particular use of a hierarchy has not been done by any previously published scan-line based rendering algorithms, and is a current local research topic.

2.2.4 Ray Casting

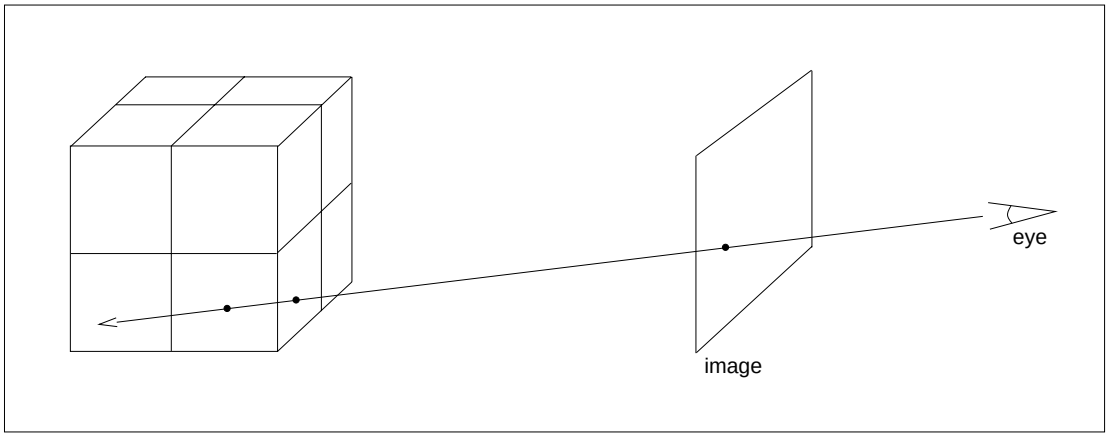

Figure 2.5: Ray-casting - Rays are sent from the eye through each pixel and into the volume.

In contrast to hardware-assisted cell projection, ray-casting produces high quality images at the expense of time. In ray-casting methods, rays are cast out from the viewing eye point through each pixel on the screen and into the volume (see Figure 2.5). If a ray intersects the volume, the contents of the volume along the ray are sampled, converted to color and opacity via the transfer function, composited, and the resulting color is used for that pixel. Samples taken along the ray can be either at equidistant intervals or at intersections with cell faces.

Early volume ray-casters were designed for regular rectilinear grids [Kajiya and Herzen, 1984, Sabella, 1988, Upson and Keeler, 1988, Drebin *et al.*, 1988, Levoy, 1988, Walsum *et al.*, 1991] and made use of the volume regularity. Ray-casting curvilinear volumes, however, is more difficult than rectilinear volumes due to the non-trivial calculations of cell locations and intersections along the ray's path. Wilhelms [Wilhelms *et al.*, 1990] describes a fast alternative to rendering curvilinear grids by resampling them into rectilinear grids. Dani [Dani *et al.*, 1994] developed a method for rendering cylindrical polar gridded volumes in particular.

Several algorithms for ray-casting curvilinear grids, without resampling into rectilinear space, follow the ray from face to face and calculate exit faces and intersection points for each cell that the ray hits [Garrity, 1990, Uselton, 1991, Koyamada, 1992, Ma and Painter, 1993, Ma, 1995]. Garrity [Garrity, 1990] decomposes hexahedra into 5 or 24 tetrahedra to eliminate complications due to possible concave cells. Ray-casting methods are CPU intensive, however, the algorithm is very simple and there are many parallel ray-casting methods, which are discussed later.

2.2.5 Hardware Assisted Rendering

Several volume rendering algorithms have been written to specifically take advantage of graphics capabilities that have been implemented in hardware, such as texture mapping, or polygon scan conversion. All projection methods that produce Gouraud-shaded polygons can take advantage of the speed of specialized graphics hardware [Shirley and Tuchman, 1990, Wilhelms and Van Gelder, 1991, Williams, 1992a], however, compositing large numbers of very small color and opacity values can create errors due to hardware rounding and truncation. Applications of hardware texture mapping have also been used to eliminate the error produced by cell projection methods that used linear approximation of non-linear opacity [Stein *et al.*, 1994]. Crawfis used hardware texture mapping in a splatting algorithm [Crawfis and Max, 1993] to show both scalar and vector values simultaneously.

The Silicon Graphics Reality Engine has the ability to process 3-dimensional texture

maps in hardware [Akeley, 1993]. This capability is the driver for several volume rendering methods that are based on 3-D texture maps [Cullip and Newman, 1993, Cabral *et al.*, 1994, Wilson *et al.*, 1994, Guan and Lipes, 1994].

Max [Max *et al.*, 1995] gives several uses for 2-dimensional hardware texture maps, including textured splats, polyhedron cell projection, spot noise, and texture advection.

2.2.6 Shear Factorization

A very fast direct volume rendering algorithm for rectilinear grids was presented in 1994 by Philippe Lacroute and Marc Levoy [Lacroute and Levoy, 1994] which is an extension of previous work [Hanrahan, 1990, Schroeder and Stoll, 1992]. It uses a technique called *shear-warp factorization*, and although it does not render curvilinear or unstructured grids, it deserves to be mentioned in this paper due to its impressive speed. The algorithm consists of three stages:

1. Factorization of the viewing transformation into a 3D shear parallel to the data slices.
2. Projection of the volume to form an intermediate but distorted 2D image.
3. A final 2D warp to form an undistorted final image.

Because the projection in the second stage is always orthogonal to the volume slices, the algorithm isn't spending any time computing tri-linear interpolations. The primary advantage of this method is speed – their implementation can render a 256^3 rectilinear data set in one second on an SGI Indigo workstation. The primary drawback to this method is that by using a 2D filter to resample the intermediate image into the final one, artifacts can be introduced into the image, especially when the viewing angle is 45 degrees to the square volume.

2.2.7 Parallel Algorithms

Direct volume rendering is a computationally intensive process. It is not uncommon for a dataset to contain a million or more samples. The time required to render an image from such a volume will vary greatly depending on which of the existing algorithms are

used along with the desired image resolution and quality. Achieving interactive or near-interactive rates has been the topic of much research. Parallel processing can address the need for computing power in volume rendering.

Parallel architectures are divided into two main categories SIMD and MIMD. A system in which the processors execute the same instruction stream in lockstep, but with different data values, is referred to as single-instruction multiple-data (SIMD). Alternatively, if each processor executes its own instruction stream, it is called multiple-instruction, multiple-data (MIMD). SIMD machines can easily be implemented with thousands of processors, while MIMD machines are usually implemented with less than 100 processors. While SIMD machines have several speed advantages over MIMD machines, the algorithms written for SIMD machines tend to be specific for the SIMD architectures and require that the data be in regular rectilinear grids. MIMD machines are more flexible in the variety of algorithms that they can support. The algorithm presented here is written for a MIMD architecture of 2-16 processors and runs well on single processor machines as well.

Parallel processing requires special care when dealing with irregular grids, this is why most parallel approaches are based on the ray-casting [Challinger, 1991, Ma and Painter, 1993, Wittenbrink and Harrington, 1994, Dani *et al.*, 1994, Ma, 1995] or scan-line algorithms [Lucas, 1992, Giertsen and Peterson, 1993, Challinger, 1993], since these algorithms are fairly simple and can be partitioned easily. Challinger discusses parallelization of projection and ray-casting [Challinger, 1991, Challinger, 1993], in terms of task generation (the decomposition of a large job into smaller tasks), synchronization, and memory management. She describes a hybrid approach using a scan-line algorithm to sort cells and X-buckets to sort edges.

Giertsen and Peterson [Giertsen and Peterson, 1993] parallelized their scan line algorithm to allow volume rendering over a network, parallelizing by dividing the image screen into rectangular regions. Each machine on the network works like a single processor and results are combined by a machine acting as a server. Ma parallelizes ray-casting of irregular grids by dividing the volume up along cell faces into irregularly sized chunks [Ma

and Painter, 1993]. Each processor is then responsible for a small piece of the volume and calculates the contributions that the piece makes to the final image by accumulating ray intersections for that piece [Ma, 1995]. The contributions, or ray segments, are composited to produce the final image. Similarly, Wittenbrink [Wittenbrink and Harrington, 1994] distributes the volume data over the processors, which locally create sub-frames that are composited into the final image. Lucas [Lucas, 1992] describes a general rendering algorithm that uses data partitioning of the volume by subdividing cells along divisional planes and by partitioning the screen into smaller rectangular regions.

2.3 Hierarchies

Hierarchies have been used to reduce unnecessary processing and to summarize regions of the data by encoding the data such that spatial coherence in the data is identified and exploited.

Meagher described a geometric modeling technique called "octree encoding" that uses a hierarchical tree structure in which the nodes represent disjoint cubes of exponentially decreasing size [Meagher, 1982]. Many variations based on the Octree have appeared since then [Yamaguchi *et al.*, 1984, Glassner, 1984, Mao *et al.*, 1987, Laur and Hanrahan, 1991, Wilhelms and Van Gelder, 1992].

Previous work has involved hierarchies over regular volumes [Wilhelms and Van Gelder, 1992, Laur and Hanrahan, 1991, Wilhelms and Van Gelder, 1994, Van Gelder *et al.*, 1995]. Wilhelms and Van Gelder present a hierarchical ray-casting approach [Van Gelder *et al.*, 1995] which achieves acceleration by taking large steps over regions where data need not be processed at a fine resolution. Their approach, which uses multi-dimensional trees [Wilhelms and Van Gelder, 1994], gives the user control over an error tolerance which directly affects the acceleration. Ma uses a k-d tree to subdivide the volume into spatial partitions that are passed to multiple parallel processors [Ma and Painter, 1993].

The research described here makes use of a hierarchy over irregular data, based on k-d trees, as introduced by Jon Bentley [Bentley, 1975]. The idea to associate primitives as

high in the hierarchy as necessary to avoid clipping was suggested by Greene [Greene *et al.*, 1993]. I.e., triangles are included in the lowest node that completely contains them.

3. Scan-line Renderer

The renderer described throughout this paper is the result of a joint research effort by Jane Wilhelms, Allen Van Gelder, Jonathan Gibbs, and me. Professor Wilhelms came up with the idea that a general scan-line algorithm could handle a wide range of volume formats and also might perform better than a ray-casting algorithm. Professor Van Gelder addressed many implementation issues including the development of the new perspective formula, code optimization, and the k-d tree. Jonathan Gibbs implemented the k-d tree partitioning of the volume as well as the rectilinear and unstructured volume data modules. My contributions include the coding and testing of the rendering module, design of the data structures, parallel implementation, implementation of embedded surfaces in the renderer, and an algorithm for determining vertices for a given polygon ID in curvilinear grids.

3.1 Algorithm and Data Structures

The algorithm described here is based on Watkins' scan-line algorithm [Watkins, 1970], with several modifications, which will be described later in this section. Although the algorithm could work with any type of polygon, triangles were chosen to be the only type of polygons that are rendered. Their simplicity makes it easier to identify active edges and also maintains data consistency when interpolating between edges at different rotations. It is also easy to break up the quadrilateral faces of a curvilinear grid into a set of triangles, however, it is shown in section 4.3 that different triangulations of quadrilateral cell faces can produce different pictures in some circumstances. Triangulation of grid cell faces increases the number of polygons by a factor of 2. Other methods [Williams, 1992b, Williams, 1992a] have been reported that decompose each hexahedral cell into five tetrahedra, resulting in an increase in the number of polygons by a factor of $3\frac{1}{3}$.

The algorithm begins with the conversion of the vertices from world space to screen space, preserving depth values. This is done by passing each vertex through the geometry and the projection matrices and storing the locations back in memory.

3.1.1 Y-buckets

The volume is decomposed into polygons by representing each face of each cell with two triangles. Each triangle is given a sequential identifier (ID). The next step is to process all of the polygons into the Y-buckets. A polygon is considered active for a scan-line if it contributes to the image for that scan-line. There is one Y-bucket for each scan-line and it contains the all of the triangles that become active (start contributing to the image at that scan-line) on that scan-line. The Y-buckets are implemented as one array that holds all of the ID's in scan-line order. Each Y-bucket can be accessed by a starting and ending subscript into the Y-bucket array. The algorithm can save time by eliminating any triangles that may not contribute to the final image because of the following reasons:

- It is entirely off of the visible screen space.
- It is entirely out of the region defined to be drawn. (i.e. a clipped region)
- It does not cross a scan-line. (The ceilings of the Y components of all three vertices are equal.)
- It does not cross a pixel in the X direction. (The ceilings of the X components of all three vertices are equal.)

Once the Y-buckets have been built, each scan-line is processed and drawn into the frame buffer. An active list containing the polygon ID's of those polygons that contribute to the scan-line (i.e. in the Y direction) is maintained. Before processing the next scan-line, the previous scan-line must be updated by adding in new triangles from the current Y-bucket and removing any triangles that are no longer active. Table 3.1 defines the data structure

Long	Long	Long	Long	Long	Long	Long
Poly ID	Grid Num	Bottom Vert	Left Vert	Right Vert	Max Y Value	Intermed Y Value
Poly ID	Grid Num	Bottom Vert	Left Vert	Right Vert	Max Y Value	Intermed Y Value
Poly ID	Grid Num	Bottom Vert	Left Vert	Right Vert	Max Y Value	Intermed Y Value

Table 3.1: Data Structure for Active Polygon List in Y direction. (Array)

used for the active list in the Y direction. The active list in Y direction is implemented as

an array. As each new triangle is added into the active list, its orientation to the scan-line must be determined by examining the X and Y screen space values of the vertices. Figure

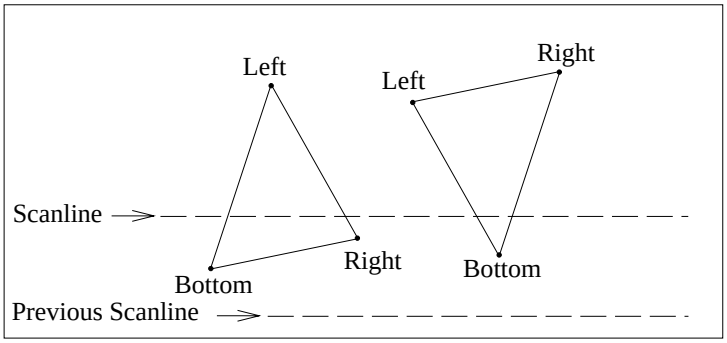

Figure 3.1: Figure of the two basic orientations of a triangle. 1) Two vertices below scan-line and 2) Two vertices above the scan-line.

3.1 shows the two basic orientations that a triangle can have with respect to the scan-line – two vertices below the scan-line or two vertices above the scan-line. There are actually twelve orientation cases since we do not know which of the three vertices are the bottom, left, and right. The bottom vertex is always below the scan-line. The left vertex is always above the scan-line. The right vertex is chosen such that by traversing the vertices of the triangle in counter-clockwise order, the vertices are visited in the order (right, left, bottom). In order to determine the bottom and right vertices (or left and right vertices for the second case), the slopes of the two edges that intersect the scan-line must be tested. In the case where two vertices are above the scan-line, an intermediate Y value is determined and saved that will signify a change in one of the two edges that are used. If the triangle falls into the case where two vertices are below the scan-line, then the intermediate Y value is set to the same value as the Y-bucket. When scan converting a triangle in screen space, you start with two active edges and work your way up sequential scan-lines. At some scan-line, it is possible that the third edge replaces one of the two active edges. If this is the case, the intermediate Y value signifies on which scan-line this change will occur.

After the active list is updated, the active triangles, which are not sorted in any particular order, must be processed into X-buckets.

3.1.2 X-buckets

There is an X-bucket for each pixel across the scan-line which contains the triangles that become active starting on that pixel. A polygon is considered active for a pixel if the pixel, which is represented by integers, lies inside or exactly on the edge of the polygon. The X-buckets, defined in Table 3.2, must be built and maintained in visibility-order for correct color and opacity calculations, therefore, its data structure has been implemented as a linked list.

Long	Float	Float	Float	Float	Float	Float	Float	Float	Int	Pointer
Polygon ID	Current Data Value	Current Depth Value	Left X Value	Right X Value	Left Z Value	Right Z Value	Left Data Value	Right Data Value	Max X Value	Next

Table 3.2: Data Structure for Active Polygon List in X direction. (Linked list)

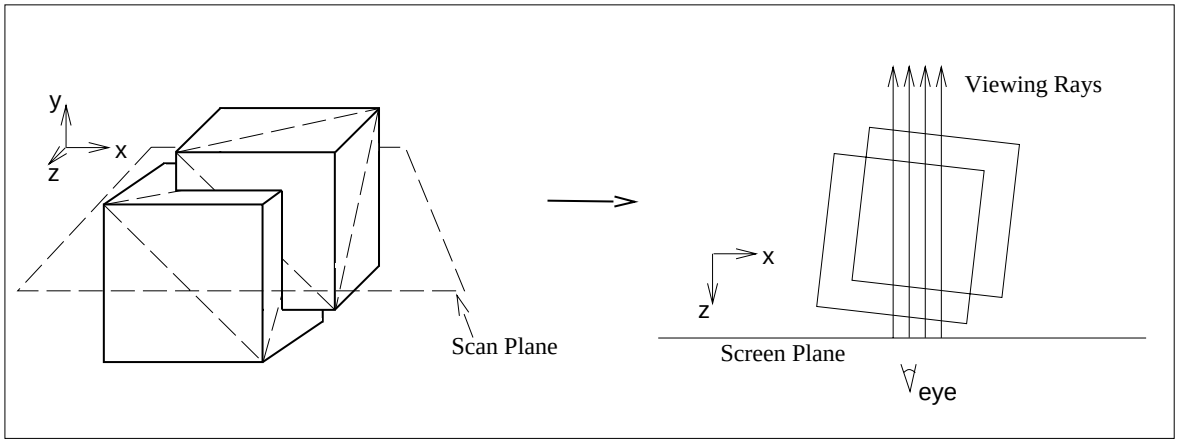

Figure 3.2: Figure of intersecting cell faces and their scan plane representation.

Figure 3.2 shows how intersecting cell faces are processed for a given scan-line. As the scan-line is processed, an active list is maintained and updated for each pixel. The active list contains the data value and the depth for each triangle that contributes to that pixel and is sorted by depth. It can then be traversed to accumulate the color and opacity for that pixel. When calculating the X-bucket for a triangle, the X values for the pixel where the triangle becomes active and where it becomes inactive on the current scan-line must be

determined by linearly interpolating the two active edges of the triangle. The code listed in Figure 3.3 determines which two of the three edges to use for a given scan-line.

```
if(scanline is below Y-intermediate)
   {
   use bottom->left and bottom->right
   }
else
   {
   if(left's Y value is greater than right's Y value)
      {
      use bottom->left and right->left
      }
   else
      {
      use left->right and bottom->right
      }
   }
```

Figure 3.3: Determining which edges to use for given scan-line.

3.1.3 Interpolation

During the course of updating the active lists, both in the Y and X direction, values for X, Z, and *data* are calculated through linear interpolation. The main requirement of the interpolation method is that the interpolated values for both end points must be exactly equal to the endpoints and never over shoot the end points by any amount. If the interpolated value of X did overshoot the endpoint by a small amount, and that amount resulted in X being greater than the next higher integer, the polygon would incorrectly contribute to an extra pixel in the final image. The equation below ensures that triangles that share a vertex and do not overlap in world space will not overlap on the screen at any point in this algorithm. The equation that was used for linear interpolation between vertices is

$$X = \frac{X_1}{(Y_1 - Y_0)}(Y - Y_0) + \frac{X_0}{(Y_1 - Y_0)}(Y_1 - Y).$$

This particular formula calculates X given the scan-line Y and the two points of a line (X_0, Y_0) and (X_1, Y_1) that intersects the scan-line. This formula will give exact values at

both end points, without over shooting and eliminates overlap between adjacent polygons. The order of calculation should never change throughout the algorithm to ensure that two polygons which share an edge always get the same value of X along that edge for any given scan-line.

3.1.4 Pseudo Code for Three Major Components

```
DoYScan(screenHeight, screenWidth)
    {
    ConvertVerticesToScreenSpace();
    BuildYBuckets();
    for(scanline goes from 1 to screenHeight)
        {
        ActiveYList = UpdateActiveYList(ActiveYList, scanline);
        X_Buckets = BuildXBuckets(ActiveYList, scanline);
        DoXScan(X_Buckets, screenWidth);
        }
    }
```

Figure 3.4: Outer loop of Y scan algorithm.

```
DoXScan(X_Buckets, screenWidth)
    {
    for(X goes from 1 to screenWidth)
        {
        ActiveXList = UpdateActiveXList(ActiveXList, X, X_Buckets);
        DrawPixel(ActiveXList);
        }
    }
```

Figure 3.5: Inner loop of Y scan algorithm.

Figure 3.4 is a simple pseudo code representation of the outer loop of the Y scan algorithm. Each scan-line calls the *DoXScan* procedure which could be thought of as the inner loop. The *DoXScan* procedure, represented as pseudo code in Figure 3.5, processes the scan-line from left to right. It calls the *DrawThisPixel* routine in Figure 3.6 which traverses the ActiveXList to determine the color and opacity for that pixel. Those polygons that are on a surface of a grid (i.e. the three vertices have one grid index that is either 0 or

```
DrawPixel(ActiveXList)
    {
    while(not at the end of ActiveXList)
        {
        if(ActiveXList.Polygon is on a grid surface or interior)
           {
           calculate and set inside/outside for that grid;
           }
        if(ActiveXList.Polygon is a clipping polygon)
           {
           toggle draw flag for clipping region;
           }
        if((draw flag is TRUE) and (inside a grid))
           {
           if(ActiveXList.Polygon is part of a surface)
              {
              add surface color to this pixel;
              return;
              }
           else
              {
              IntegrateColor(Depth to next, data of this and next poly);
              AddColorToThisPixel;
              }
           }
        ActiveXList = ActiveXList->next;
        }
    }
```

Figure 3.6: Draw a pixel routine.

max) will toggle a flag that keeps track of whether we are inside a grid or not. Similarly, those polygons that are part of the bounding box, or clipping region, will toggle a flag that controls whether we accumulate color or not.

3.1.5 Drawing Each Pixel

The active X-bucket list that is maintained for each pixel contains a list of polygons that includes the two polygons for the clipping region (one to enter the region and one to exit the region), any volume polygons that contribute to the pixel, and any polygons that are part of an embedded surface. This list is sorted in visibility order and must be traversed to calculate the final color for the pixel. Due to the fact that some polygons could intersect the clipping region polygons, it is likely that the first clipping region polygon is not the first polygon in the list, so the list must be traversed until the first clipping region polygon is

visited.

If the list traversal is inside a grid when the first clipping region polygon is visited (entering the clipping region), or when the second clipping region polygon is visited (exiting the clipping region), these partial regions must also be drawn. The depth of the partial region is simple to calculate, since the depth values are maintained for all polygons; however, the value of the scalar data does not exist for clipping region polygons and must be determined by either obtaining the data value from the polygon previous to the entering one, or from the polygon just after the exiting one. It is possible that there will not be any polygons in front of or behind the clipping region because of clipping from k-d tree. In that case, the data values from the closest polygon inside the clipping region is used and the color for the partial region is calculated using a homogeneous data value.

Since the grids being rendered may be concave and may not always occupy the same space, the traversal must maintain information regarding whether it is currently inside or outside of each grid. If the traversal is in inside the clipping region and also inside any of the grids, it accumulates color from one polygon to the next in the list. Maintaining the current grid information for a maximum of 32 grids is accomplished by setting and checking a 32-bit flag called *in_out*. During traversal, if a polygon is entering a grid (or if it is interior to the grid), then the *in_out* bit representing the grid ID is set to 1. If the polygon is exiting a grid, then the bit representing the grid ID is set to 0. If any of the bits in *in_out* are 1 (i.e. $in\_out > 0$), accumulation occurs, otherwise the polygon is skipped and the traversal continues to the next polygon.

Determining if a polygon is entering or exiting a grid involves keeping track of how the vertices project to screen space. The three vertices of each polygon are stored in the active list (Y-direction) in counter-clockwise order with the lowest vertex in screen space first (see Section 3.1.1). The three vertices are given to the renderer from the data module in an order such that a counter-clockwise cross product of the edges between the vertices produces a vector that always points away from the interior of the grid. This ordering is needed for triangles that are on an external face of the grid. The ordering does not matter

for triangles that are internal to a grid. If two vertices need to be swapped in order to maintain the counter-clockwise ordering in screen space, and if the triangle is also part of a grid boundary, then the polygon is an *exit polygon* for the grid to which it belongs.

3.1.6 Integration Method

During the list traversal, calculations are performed on each contributing region (from one polygon to the next) along the line of sight, to accumulate the color for the pixel. The scalar data values for the front and back polygons of each contributing region are averaged and converted into a color and opacity using a transfer function with lookup tables for red, green, blue, and opacity. The color and opacity values are used, along with the depth between the polygons, to calculate the contribution to the pixel, assuming that the region is homogeneous. Other methods average the color between the front and back polygons or they break the region into several smaller segments and interpolate the data values for those segments [Wilhelms and Van Gelder, 1991].

The method of integration used in this renderer is called *exponential homogeneous integration* and is briefly described below. A detailed derivation of the intensity and opacity formulas can be found in [Wilhelms and Van Gelder, 1991].

Each region is assumed to consist of semi-transparent material that emits light and occludes light from behind and within the region. Calculating the cumulative intensity and cumulative opacity involves solving the following equations

$$\begin{aligned} T(z) &= e^{-\int_0^z \Omega(u)du}, \\ I(z) &= T(z)\int_0^z \left(\frac{E(v)}{T(v)}\right) dv, \end{aligned}$$

where $T(z)$ is the fraction of light entering the region that is transmitted through distance z, $I(z)$ is the intensity of the color that is emitted within the cell and reaches z, and $E(v)$ is the differential intensity of the color. $\Omega(u)$ is the differential opacity which is the rate at which light is occluded as it travels through the material and is

$$\Omega = \log\left(\frac{1}{1 - Opacity_{material}}\right)$$

for a unit depth. If $\Omega(z)$ and $E(z)$ are constant throughout the region (which we assume they are), then

$$\begin{aligned} T(z) &= e^{-\Omega z}, \\ I(z) &= \frac{E}{\Omega}(1 - e^{-\Omega z}), \end{aligned}$$

and the cumulative intensity and cumulative opacity are

$$\begin{aligned} O_{cum} &= (1 - T(z)), \\ I_{cum} &= I(z). \end{aligned}$$

These values are then composited with color and opacity already accumulated for the pixel

$$\begin{aligned} C_{acc} &= C_{acc} + (1 - O_{acc})C_{new}, \\ O_{acc} &= O_{acc} + (1 - O_{acc})O_{new}, \end{aligned}$$

where C_{new} and O_{new} are the color and opacity values of the newly calculated region and C_{acc} and O_{acc} are the accumulated color and opacity values for the pixel before and after compositing. The regions are processed in front-to-back order.

3.1.7 Surfaces in the Volume

Embedded surfaces can greatly increase the effectiveness of direct volume renderings. Several graphics faculty and graduate students were unable to correctly identify the fighter jet rendering as a fighter until the surfaces were rendered along with the volume (see Figure 4.5).

The data module is responsible for handling the identification and administration of surface polygons, and surface vertices, including the calculation of vertex normals. In order to easily identify surface polygons, the grid number used for all surface polygons and vertices is 32. All surface polygons are given a polygon ID that starts sequentially from the maximum polygon ID in the volume that was loaded.

The renderer allocates space to hold the screen space locations of the surface vertices along with the volume vertices. Also, the color at each surface vertex is calculated and stored for each different view using a standard Phong illumination model [Watt and Watt, 1992] which is summarized as follows:

$$I_{tot} = I_a + k_d \left(\vec{L} \cdot \vec{N}\right) + k_s \left(\vec{N} \cdot \left(\frac{\left(\vec{L} + \vec{V}\right)}{2}\right)\right)^n$$

where,

I_a = the ambient light,

k_d = the diffuse coefficient,

k_s = the specular coefficient,

$\vec{L}$ = the normalized vector from the vertex to the light source,

$\vec{N}$ = the surface normal at the vertex which is calculated by averaging the normals of the polygons associated with the vertex,

$\vec{V}$ = the normalized vector from the vertex to the eye,

n = the shininess factor for sharper specular highlights.

The color (I_{tot}) for this implementation is a single scalar value representing a gray-scale color. This color is used and interpolated exactly the same as the data value that is associated with non-surface polygons.

3.2 Parallel Implementation

Watkins' scan-line algorithm [Watkins, 1970], although it appears sequential, can be implemented to take advantage of a shared-memory MIMD architecture. By breaking up the various sequential components of the scan-line algorithm into equal loads for each process, the algorithm can be almost entirely parallelizable. The only parts that are not easily parallelizable are the sections where the work is actually being broken up. This algorithm has three primary components that are implemented in parallel.

The first is the transformation of vertices from world space to screen space. The work for this part is easily partitioned for the number of processes that will be used.

The second component is the generation of the Y-buckets. This can also be broken up; however, it requires using a two-pass approach. During the first pass, each process is given an equal number of polygons to process. Each polygon either belongs in a Y-bucket, or is dropped because it does not cover a pixel in the drawing region. The Y-bucket value is kept for each polygon in a separate array of shorts called the *bucket_clip_info* array. In addition, if the polygon is discarded for any reason, an invalidation flag is set for that polygon in the *bucket_clip_info* array. This array is used by the second pass after the Y-bucket array has been sized properly. Also, a counter, which will be used later as a subscript, for that Y-bucket is incremented. After the first pass is completed, the subscripts that were generated by each process are combined to determine the total size of the Y-bucket array, and to determine the offset where each Y-bucket will start, as illustrated in Figure 3.7. The total counts determine the offsets for processor 0, then each processor calculates its own offset for each Y-bucket.

During the second pass each process handles the same polygons, minus those that were invalidated. The single Y-bucket array is filled by all of the processes, by using the *bucket_clip_info* array that was previously produced. For example, processor 2 has 3 triangles to put into Y-bucket 2, so it will place them in slots 32, 33, and 34, avoiding conflicts with other processors.

The third part involves the processing of each scan-line in order from bottom to top. There are several ways of implementing this part in parallel.

The first design, which is the one actually implemented, consists of a critical section of code where a processor updates the current active list for the scan-line, takes a copy of it, and then exits the critical section. After exiting the critical section, it builds the X-buckets and processes the scan-line. The next available processor waits to enter the critical section and get to the next available scan-line. This implementation does not scale linearly since it contains a critical section that will act as a bottle neck as more processors are added,

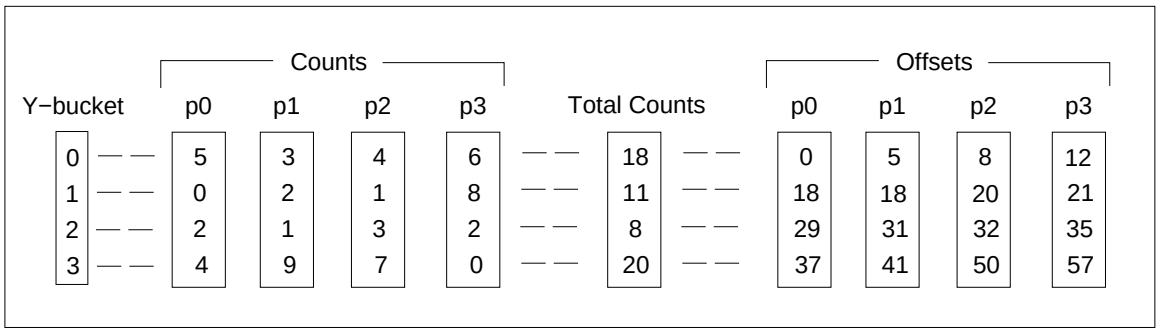

Figure 3.7: Counters and offsets used by 4 processors in Y-bucket creation. The counts signify how many polygons fall into each Y-bucket. The four processors use the offsets to place the polygons into the single Y-bucket array.

however, it was fairly easy to implement, and is the method that was used.

One way to eliminate waiting for the next scan-line to become available, in the above design, is to assign a region of sequential scan-lines to each processor. This will work if the volume is evenly distributed across the screen in the vertical direction. If it is not, then loading problems will arise resulting in one processor doing more work than the others.

Another alternative to the first approach is to have the processor update the current scan-line in small segments (100 triangles each) and after each segment is updated, signal to the next processor that the segment is ready. This way, all the processors can be updating their own scan-line and will only have to wait for the next segment. If all of the processors have approximately the same throughput, each processor will only have to wait for the first segment to be completed by the previous processor. Although this approach is not 100% scalable, it will reduce the time spent waiting for the previous scan-line to be completely updated. The communications between processes is more complicated in this approach and could cause delays if there is no efficient way to signal to other processes.

Uselton described scan-line processing as "embarrassingly parallel" for ray-casting [Uselton, 1993]. However, the expensive part of ray-casting in irregular structures with semi-transparent material is the calculation of exit faces from each cell entered by the ray. (Moreover, this technique does not directly apply to intersecting grids, where a ray can be in two cells simultaneously.) The technique presented here uses coherence to greatly reduce the

cost that corresponds to exit-face calculations, at the expense of introducing a short serial critical section.

3.3 Perspective Rendering

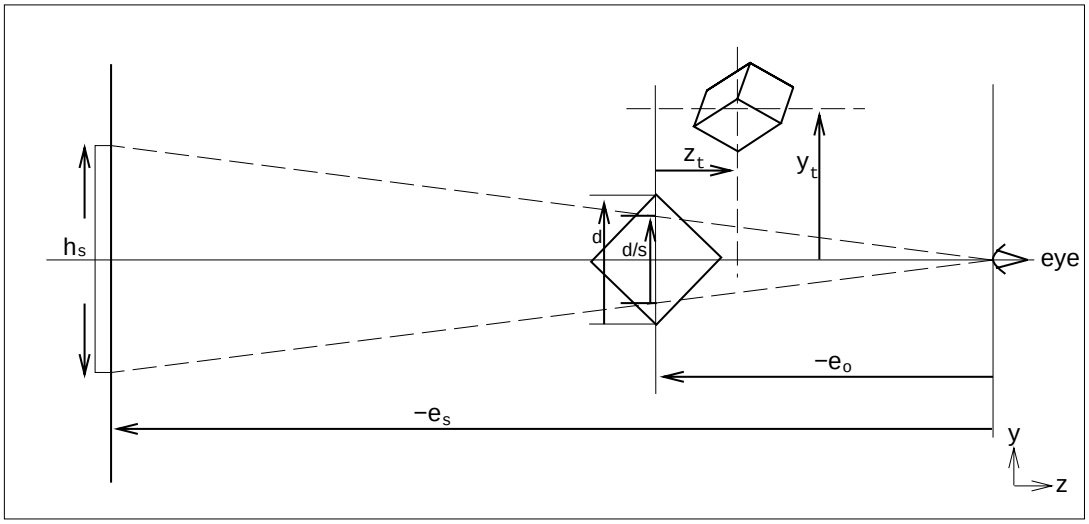

Figure 3.8: Figure of perspective view - eye is at (0,0,0) world space. Volume is projected onto screen at left.

During the initial testing of the program, we observed that the distance calculation between triangles was losing accuracy due to the fact that we were using a standard projection matrix which transforms the viewable world space into a unit viewing volume, with the eye at the origin [McLendon, 1991]. This meant that the Z values of the vertices were being transformed to values that were very close to each other and all close to 1.0. This introduced substantial floating point *relative* errors and resulted in some ordering inconsistencies. The problem was corrected by using a different projection transformation (described below). This transformation preserves accuracy of the Z values of the vertices near the center of the volume, by transforming that center to zero in screen Z. Symbols used in this section are summarized in Figure 3.3. The main innovation is the introduction of a translation in screen-Z, given by Eq. 3.2, which produces a numerically stable equation (Eq. 3.3) for screen-Z in terms of world-Z, near the world-Z center.

Symbol	Definition	Units
h_s	screen height	pixels
e_s	distance from the eye to the screen	pixels
e_o	distance from the eye to the volume origin	world units
d	diagonal of the volume's visible bounding box	world units
s	user's scale factor	pure number
z_t	translate in z direction	world units
y_t	translate in y direction	world units
x_t	translate in x direction	world units
z_v	vertex value of z	world units
y_v	vertex value of y	world units
x_v	vertex value of x	world units
z_s	transformed vertex value of z to screen	pixels
y_s	transformed vertex value of y to screen	pixels
x_s	transformed vertex value of x to screen	pixels
z_{so}	offset (defined below)	pixels

Table 3.3: Symbols used in perspective calculation.

The volume being viewed is inside a user definable visible bounding box (Vbb), of diagonal d (see Figure 3.8). The Vbb is first centered at (0,0,0) in world space, then it is rotated by the user rotations and translated back by e_o. Then the user translations are applied in screen space, with the restriction that

$$z_t + \tfrac{1}{2}d + NearClip \leq e_o$$

(our implementation uses $NearClip = .05e_o$), so that the denominator $(e_o - z_t - z_v)$ is positive. Using similar triangles, we get the following relationships

$$\begin{aligned} e_s &= \frac{s \cdot h_s \cdot e_o}{d}, \\ \frac{x_s}{e_s} &= \frac{x_v + x_t}{e_o - z_t - z_v}, \end{aligned}$$

$$\begin{aligned}\frac{y_s}{e_s} &= \frac{y_v + y_t}{e_o - z_t - z_v},\\ \frac{z_s + z_{so}}{e_s} &= \frac{z_v + z_t}{e_o - z_t - z_v} = 1 - \frac{e_o}{e_o - z_t - z_v}.\end{aligned}$$

Conversion to screen space is

$$\begin{aligned}x_s &= \frac{e_s(x_v + x_t)}{e_o - z_t - z_v}\\ y_s &= \frac{e_s(y_v + y_t)}{e_o - z_t - z_v}\\ z_s &= \frac{e_s(z_v + z_t)}{e_o - z_t - z_v} - z_{so}\end{aligned} \tag{3.1}$$

where the quantity z_{so} is an offset to the screen position that is chosen so that the center of the volume is mapped to 0 in z_s. (The center of the volume might map anywhere in screen X and Y, as it is not necessarily within the Vbb.) The required value is

$$z_{so} = \frac{e_s \cdot z_t}{e_o - z_t} \tag{3.2}$$

which gives the numerically stable expression

$$z_s = \frac{e_s \cdot z_v \cdot e_o}{(e_o - z_t)(e_o - z_t - z_v)} \tag{3.3}$$

To invert the z_v-to-z_s mapping, just solve for z_v. The following equations solve for the difference in world-Z between two points along one sight line $(z_{v2} - z_{v1})$, in terms of their screen-space values. The last line provides a numerically stable form:

$$\begin{aligned}e_o(z_s + z_{so}) - (z_t + z_v)(z_s + z_{so}) &= e_s(z_v + z_t)\\ z_v + z_t &= \frac{e_o(z_s + z_{so})}{e_s + z_s + z_{so}}\\ e_s + z_{so} &= e_s\left(1 + \frac{z_t}{e_o - z_t}\right) = \frac{e_s \cdot e_o}{e_o - z_t}\end{aligned}$$

$$\begin{aligned}z_{v2} - z_{v1} &= \frac{e_o[(z_{s2} + z_{so})(e_s + z_{s1} + z_{so}) - (z_{s1} + z_{so})(e_s + z_{s2} + z_{so})]}{(e_s + z_{s2} + z_{so})(e_s + z_{s1} + z_{so})}\\ &= \frac{e_o[e_s(z_{s2} - z_{s1})]}{(e_s + z_{so} + z_{s2})(e_s + z_{so} + z_{s1})}\\ &= \frac{e_o \cdot e_s(z_{s2} - z_{s1})}{\left(\frac{e_o \cdot e_s}{e_o - z_t} + z_{s2}\right)\left(\frac{e_o \cdot e_s}{e_o - z_t} + z_{s1}\right)}\end{aligned} \tag{3.4}$$

Finally, the world *distance* between two points on a sight line is inversely proportional to the cosine of the angle θ between the sight line and the screen-Z axis:

$$dist = \frac{z_{v2} - z_{v1}}{cos(\theta)} = (z_{v2} - z_{v1})\sqrt{\frac{z_s^2}{x_s^2 + y_s^2 + z_s^2}} \tag{3.5}$$

Accurate values of this distance are critical because they affect the compositing of color through possibly thousands of triangles that are very close to each other, as seen in the space-shuttle grid (see Figure 4.4).

3.4 Hierarchical Partitioning of Volume

A method of spatial partitioning in k dimensions, called k-d trees, was introduced by Jon Bentley [Bentley, 1975]. A binary tree is built that splits in one dimension at a time. The implementation used in this project recursively splits each of the dimensions (always in the same dimensional order). At each node, the region associated with the node is bisected with a plane in the splitting dimension for that tree level. Partitioning the polygons from the volume into k-d trees is described below.

3.4.1 Tree Creation

Each node in the tree consists of a list of polygon IDs. The node is built by checking all vertices of each polygon passed to the node against the splitting plane. If all vertices are located on the lesser side of the splitting plane, then the polygon is passed to the left subtree. If all vertices are located on the greater side of the splitting plane, then the polygon is passed to the right subtree. Otherwise, the polygon is retained in the current node (see Figure 3.9).

If the number of polygons passed into the node is below a user-defined threshold, no splitting occurs, and the node is a leaf, otherwise, the node is split along the next dimension in order and the left and right subtrees are processed recursively. Note this results in polygons being associated with the smallest node region that completely contains them, following Greene [Greene *et al.*, 1993], and avoids subdividing polygons.

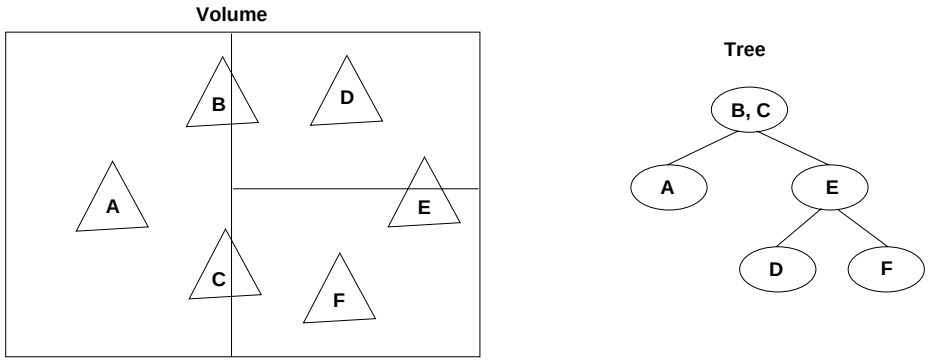

Figure 3.9: Distributing polygons in a k-d tree, as described in Section 3.4.

Polygon IDs are stored in one common array and are arranged in the order that they would appear during a prefix-order traversal of the k-d tree. A subtree is made up of a contiguous segment of polygon IDs with the root node's polygon IDs positioned at the beginning of the segment.

Each node holds the offset and length of its segment that contains the polygons associated with the node. Also, each node holds the offset and lengths of the segments that contain the polygon IDs for its left and right children. For approximation of subregions, each node must also calculate an approximation of the data in the subregion it represents, as well as an error term representing the deviation of the approximation from the data.

3.4.2 Rendering with no Approximation

The purpose of the tree is to quickly identify the polygon IDs that need to be rendered. A user-defined restrict box may limit the polygons to a small subset which can be easily identified by doing a tree traversal.

Since polygon IDs within the spatial region represented by one node are stored contiguously in the polygon array, the entire node, with its subtree can be quickly discarded if it is outside the restrict box. If the node is entirely inside the restrict box, then the subrange for that node is added to the list of subranges to be rendered, and the traversal is done with that particular subtree. If a node is partially inside the restrict box, then the segment of polygon IDs associated with the node itself is added to the set of subranges to be rendered,

and traversal continues into its children. After traversing the tree, the list of subranges of polygon IDs is passed to the renderer.

3.4.3 Rendering with Approximations

Rendering with approximations gives the user a very fast way to preview the volume and position it quickly. The tree is traversed in back-to-front visibility order and any regions that are inside the restrict box are drawn as a single rectangular box using approximated values at the eight corners of the region, which are stored in the node and are calculated during the creation of the tree.

When drawing an approximated region, cell projection [Wilhelms and Van Gelder, 1991] techniques are used, which take advantage of graphics hardware (if it exists). Figure 4.5 shows images rendered with approximation and without approximations. Speed is the primary reason for approximating each node in the tree. The speed of approximating is demonstrated by the fact that the non-approximated rendering of the fighter (see Figure 4.5) took approximately 40 seconds and the approximated version took only 1 second.

4. Results

This section analyzes run time, memory, and parallel speeds of the scan-line algorithm. The scan-line algorithm is also compared to other algorithms in speed and image quality.

4.1 Measurements Using a Large Multiple Curvilinear Dataset

This section describes results of testing the algorithm on a large dataset, the space shuttle [Martin Jr. and Slotnick, 1990] (see Table 4.1). The size of the dataset is discussed along with its impact on speed and memory requirements, as well as the effects of using different screen sizes and image scaling factors. Results are presented in graphical form with an emphasis on showing the gains in speed for one to four processors.

The dataset was rendered with a spatial rotation of ($-90°X$, $-10°Y$, $0°Z$), and scale factors that ranged from 1.0 to 5.16 (see Figure 4.1). The speedup data for four processors in particular is not exact since the machine, an SGI Onyx had four processors, which means that some time was spent by one or more processors on background processes. This introduced some variation in performance, but trends in speed can still be observed for four processors.

Grid 1	Grid 2	Grid 3	Grid 4	Grid 5	Grid 6	Grid 7	Grid 8	Grid 9
88x39x60	92x57x28	98x77x48	24x33x23	33x33x22	25x40x21	75x37x33	14x25x30	53x23x50
205920	146832	362208	18216	23958	21000	91575	10500	60950

Table 4.1: Grid dimensions and data points for NASA space shuttle.

4.1.1 Description of Dataset

The dataset used for the experiment was the NASA space shuttle [Martin Jr. and Slotnick, 1990], which is made up of nine intersecting curvilinear grids and 941,159 total data points. The data includes the position of every point in each grid along with values

Scale -	1.00	1.20	1.44	1.73	2.07	2.49	2.99	3.58	4.30	5.16
Active Polygons (Million)	0.865	0.995	1.153	1.346	1.495	1.660	1.836	2.022	2.192	2.360
Single processor	54.39	66.87	83.52	98.88	112.55	125.34	138.02	149.15	161.75	173.80
Two processors	30.66	37.46	46.34	56.10	63.30	71.32	77.67	84.33	90.17	97.57
Three processors	21.30	26.19	32.58	38.82	44.10	49.36	54.24	58.26	62.70	68.15
Four processors	16.85	20.16	25.06	30.10	34.14	37.89	41.62	44.89	49.15	53.03

Table 4.2: Elapsed time comparisons (in seconds) to render the NASA space shuttle data set on an SGI Onyx with four 150 MHz processors, 256MB memory.

for its density, energy and momentum vector. The space shuttle was chosen because it has multiple grids, and represents a relatively large dataset.

4.1.2 Timing Analysis

Transforming the point locations from world space to screen space was measured separately from the other sections of the algorithm. The average total CPU time to transform the shuttle points to screen space was 1.25 seconds and the total did not increase more than 0.02 seconds for four processors. This small increase is due to the amount of overhead needed for each processor. Figure 4.2 shows the speedups for transforming the points to screen space for multiple processors. The results show that this part of the algorithm is scalable with a greater than 3.5 speedup for four processors.

Speedup factors for creating the Y-buckets are shown on Figure 4.2. CPU times for creating Y-buckets varied from 16.8 seconds to 18.9 seconds. The total time needed to create Y-buckets increases slightly (less than 10% change from scale = 1.0 to scale = 5.16) when the number of active polygons increases. An active polygon is one that is not clipped from the drawing area due to size or position on the screen. This increase in time is due to the fact that the Y-bucket array grows in size and active polygon ID's must be stored in the array.

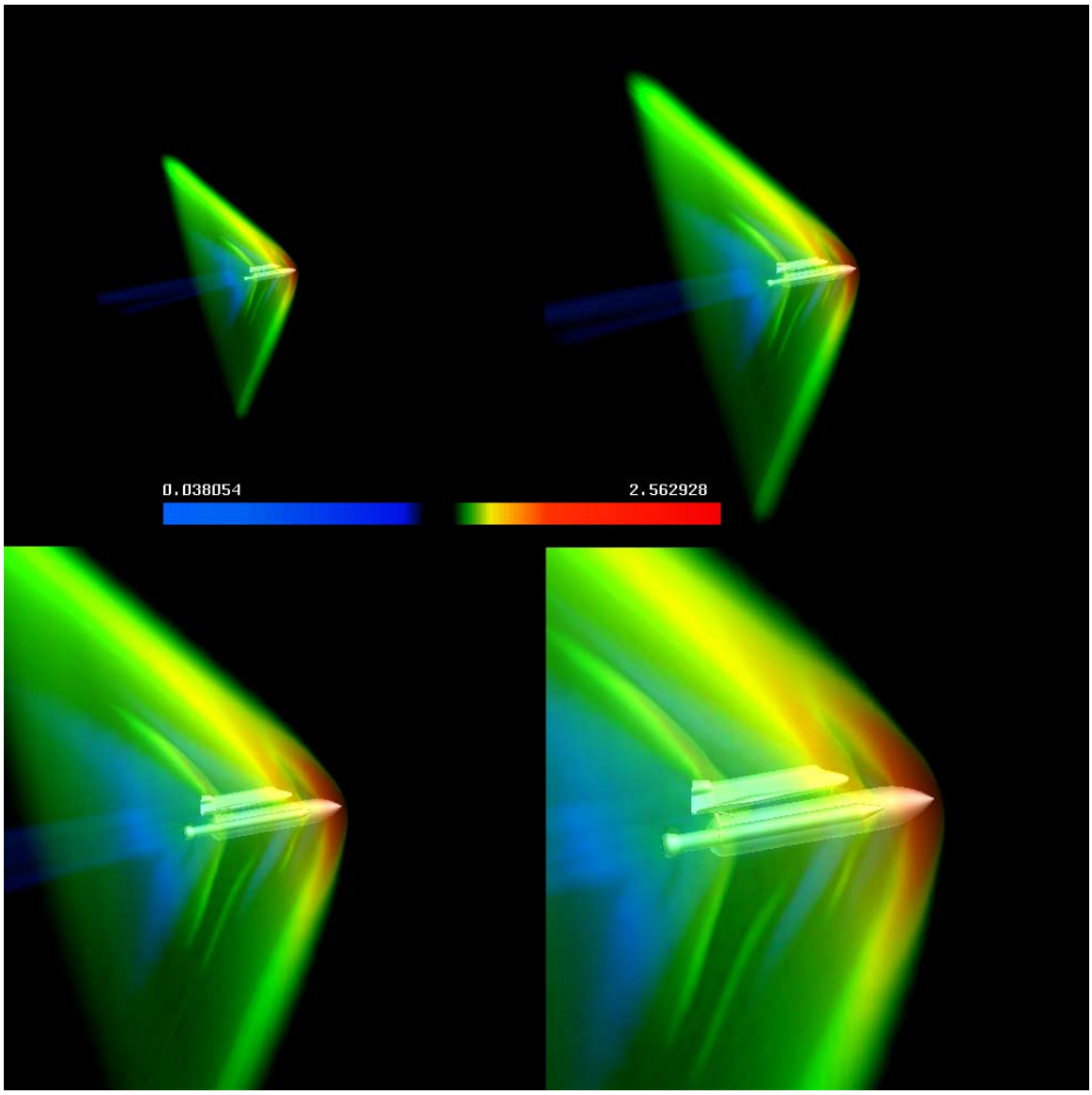

Figure 4.1: Shuttle images with transfer function. (scale = 1.0, 1.73, 2.99, 5.16)

Speedup factors for generating the scan-lines from the Y-buckets are shown on Figure 4.2. These speed ratios are not as high as those for transforming points to screen space or building Y-buckets because of the critical code mentioned in section 3.2, however, the increase in speed is still significant for four processors.

The total elapsed times to render each picture are presented in Table 4.2. Overall

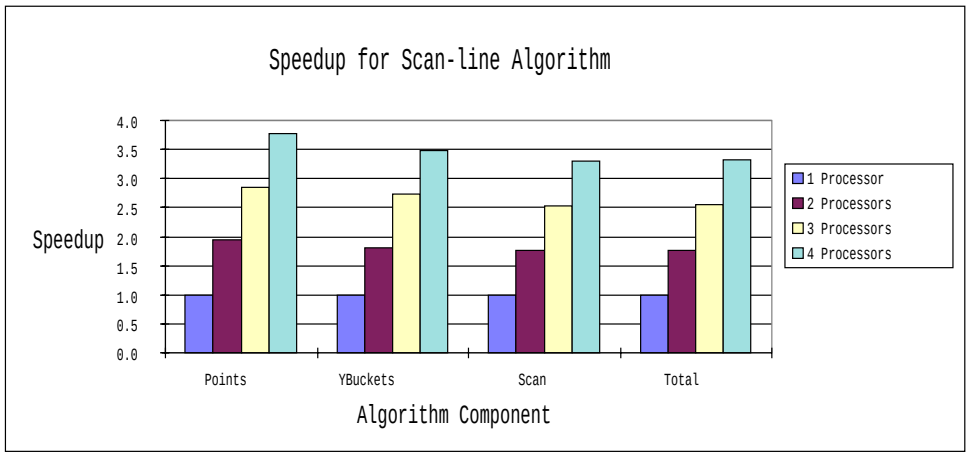

Figure 4.2: Speedup of algorithm for shuttle (scale = 3.58).

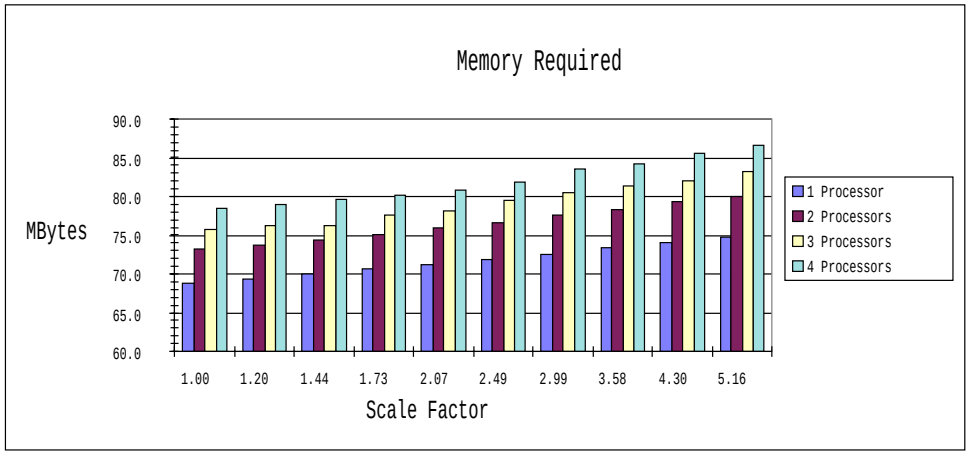

Figure 4.3: Amount of memory used by the program.

speedup factors are in Figure 4.2. It is clear that although 100% scalability is not achieved, there is a significant speedup (approx 3.25) for four processors.

4.1.3 Memory Analysis

Memory use is an important factor in measuring the efficiency of an algorithm. Figure 4.3 shows the memory (in megabytes) used by the program for a range of scale factors and multiple processors. This information was recorded by keeping statistical information associated with all calls to *malloc()* and *free()*. Most of the memory was used for storage

of the following major items:

- Initial space shuttle grids (point locations (x, y, z), momentum vector (u, v, w), density, energy) – 33.9 Mbytes.
- Transformed points – 11.3 Mbytes.
- Frame buffer (R,G,B,Alpha,Composite) – 5.0 Mbytes
- *bucket_clip_info* array – 11.0 Mbytes

The rest of the memory used by the program is to keep track of the polygons that it is rendering. It can be seen from Figure 4.3 that memory usage increases with the number of processors. Figure 4.3 includes memory that is used by dynamic arrays and is the maximum that is ever used during execution. This is due to the fact that each processor is rendering a scan line and needs the memory to hold the data structures for that scan line, such as the active polygon list and the X-buckets. Also, as the number of active polygons increase, the memory needed by active lists and buckets increases as well.

4.1.4 Speed Performance with Hierarchy

Use of the hierarchy improves performance by avoiding the computation time needed to test all polygons with the restrict box since whole invisible subregions can be discarded at one time. This is a significant time savings on the space shuttle only when zoomed in on the volume considerably, although with larger volumes, the gains may be more significant. Table 4.3 shows the speedup gained by using the hierarchy at different levels of zoom.

Scale Factor	1	4	16	64	256
Speedup With Hierarchy	0.95	0.98	1.15	1.37	1.96

Table 4.3: Speedup gained on space shuttle using a hierarchy to avoid invisible regions, on an SGI Onyx using one 150-MHz processor.

The hierarchy is also useful for approximating the information. Figure 4.5 compares images of the fighter jet rendered with approximation and without approximation. The left column shows the volume and embedded surface, the center column the volume only, and

the right column the approximation of the volume. The top row shows the whole data set, the bottom is zoomed in by a factor of two on an interesting area. The unapproximated images took between 38 and 44 seconds to render, while the two approximated images took about 1 second.

4.2 Comparison with Other Rendering Methods on Other Grids

4.2.1 Curvilinear Grid Rendering

A blunt fin dataset [Hung and Buning, 1985], which is smaller (40x32x32 = 40,960 points) than the space shuttle dataset and consists of a single curvilinear grid, was used to compare rendering times of the scan-line algorithm with times from a cell projection algorithm (*incoherent projection*) called *QP* [Van Gelder and Wilhelms, 1993]. Both algorithms were run on an SGI Onyx with four 150MHz processors, however, only one processor was used for this test.

Program	Scale = 1.0	Scale = 2.0
QP	17.01 sec	17.01 sec
Scan-line	22.07 sec	49.26 sec

Table 4.4: Scan-line and QP times for Blunt Fin data set.

Figure 4.6 shows images produced by the QP program (on top) and by the scan-line program (on bottom). Table 4.4 shows the CPU times for each program. It is clear that QP is the faster of the two programs, but at scales of 1.0 or less, the scan-line algorithm is competitive. The increase in time for the scan-line algorithm is offset by its superior image quality. Also, the scan-line algorithm handles intersecting grids where QP can not.

4.2.2 Rectilinear Grid Rendering

Rendering Method -	Software Scan Conversion	Incoherent Projection	Coherent Projection	Ray Casting
Space Shuttle scale 1	60.9			
Space Shuttle scale 5	202.1			
Blunt Fin scale 1	19.4	16.4		
Blunt Fin scale 5	51.4	16.5		
Fighter Jet scale 1	24.6			
Fighter Jet scale 5	92.6			
Hipip scale 1	74.5	109.4	12.2	63.0
Hipip scale 5	128.9	109.4	1.8	187.0
CTHead scale 1	180.1		56.7	48.0
CTHead scale 5	203.9		9.46	140.0

Table 4.5: Speed Comparison of Various Renderers (C.P.U. seconds) using a single processor 150 MHz SGI Onyx. Only renderers relevant to a particular data type are shown.

The scan-line renderer was compared against several locally developed renderers which, unfortunately, could not render multiple curvilinear grids (like the space shuttle) or unstructured grids (like the fighter). Comparisons of image quality and performance were made, however, using a (200x200x50) *CTHead* rectilinear volume and a (64x64x64) *Hipip* rectilinear volume with the following renderers:

1. *Ray Casting* for rectilinear grids [Van Gelder *et al.*, 1995].
2. *Coherent Projection* (hardware Gouraud shading) for rectilinear grids [Wilhelms and Van Gelder, 1991].
3. *Incoherent Projection* (hardware Gouraud shading) for curvilinear grids [Van Gelder and Wilhelms, 1993].

Table 4.5 compares the time taken to render each of the data sets described above by the renderers capable of handling them. Each volume is drawn in a 500x500 pixel window at a scale of 1 or 5 times.

Figure 4.7 compares image quality of the four methods on the Hipip data set, which is rectilinear and thus amenable to all methods.

We can provide some further general comments. First, software scan conversion generally produces pictures equivalent in image quality to ray casting much faster, because of the use

of coherence.

The ray-casting renderer did outperform the scan-line renderer on some tests due to the fact that the ray-caster was specifically written for rectilinear datasets, and that datasets that get opaque quickly are rendered faster by a ray-caster because a ray-caster processes through the volume front to back and can easily halt when the pixel becomes opaque. Thus on a data set such as the CTHead, the ray tracer can be competitive or even faster than other methods.

4.3 Other Factors

It is clear from Figures 4.8 and 4.9 that as the window size is increased or as the scale is increased, the number of active polygons is increased as well. As the scale is increased, there is an upper limit to the number of active polygons which is less than the total number of polygons in the data set because at high scale factors, more of the volume is projected off of the screen and the polygons that project off of the screen are discarded by the algorithm during Y-bucket creation. Increasing the window size will allow very small polygons, that may have been discarded because they did not cross a pixel, to become active since they are now projected onto a larger portion of the screen.

While testing the algorithm on the blunt fin, visual artifacts could be produced with a sharp transfer function. The cause of the artifacts was determined to be associated with the linear interpolation across the triangle face. When the direction of triangulation of the quadrilateral faces was reversed, (i.e. from lower-left$\rightarrow$upper-right to upper-left$\rightarrow$lower-right) the image was significantly affected for certain transfer functions, as shown in Figure 4.10. The triangle introduces a C1 discontinuity along the diagonal of the volume, which is accentuated by the high frequencies in the transfer function.

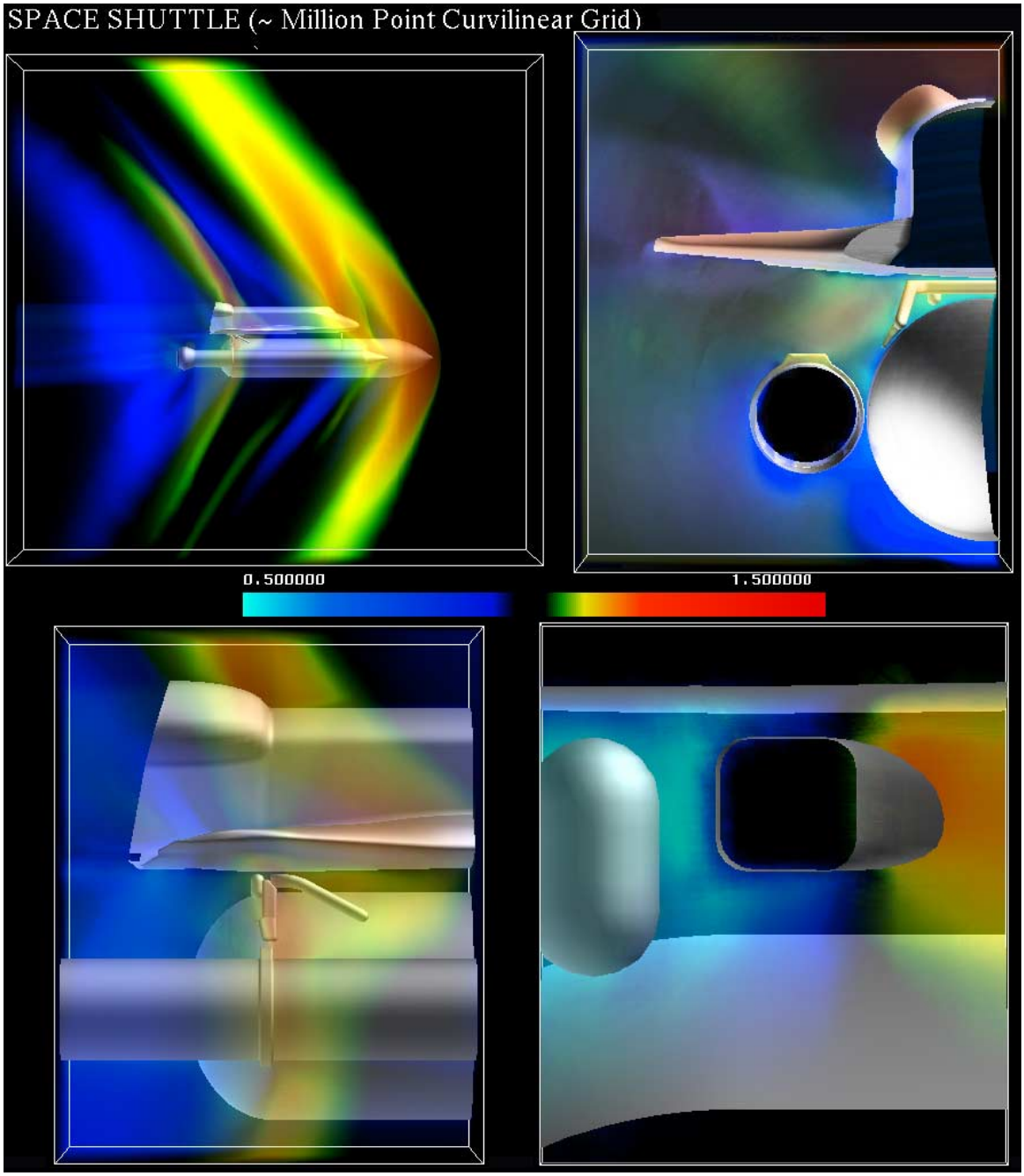

Figure 4.4: Four views of the space shuttle ranging from 1.8% to 0.000028% of the total volume.

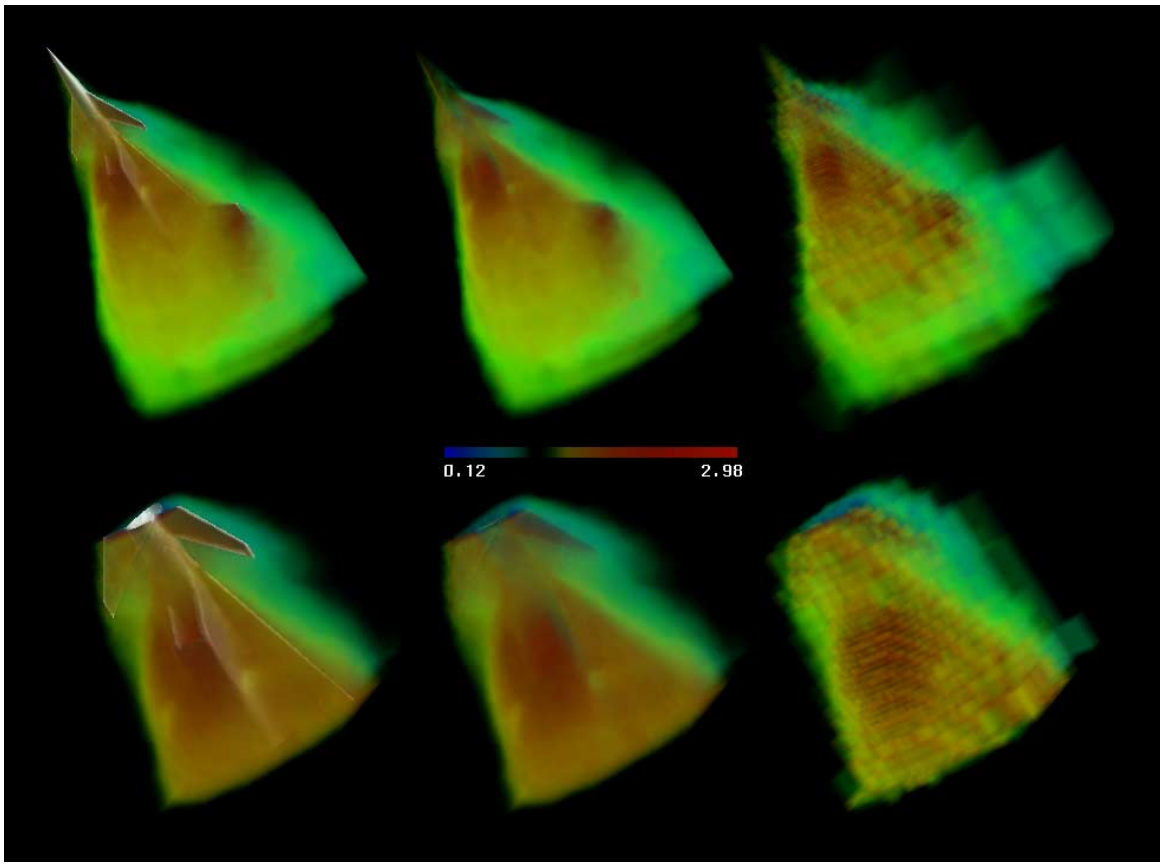

Figure 4.5: A tetrahedral grid of a fighter comparing scan-line rendering with nodal approximations.

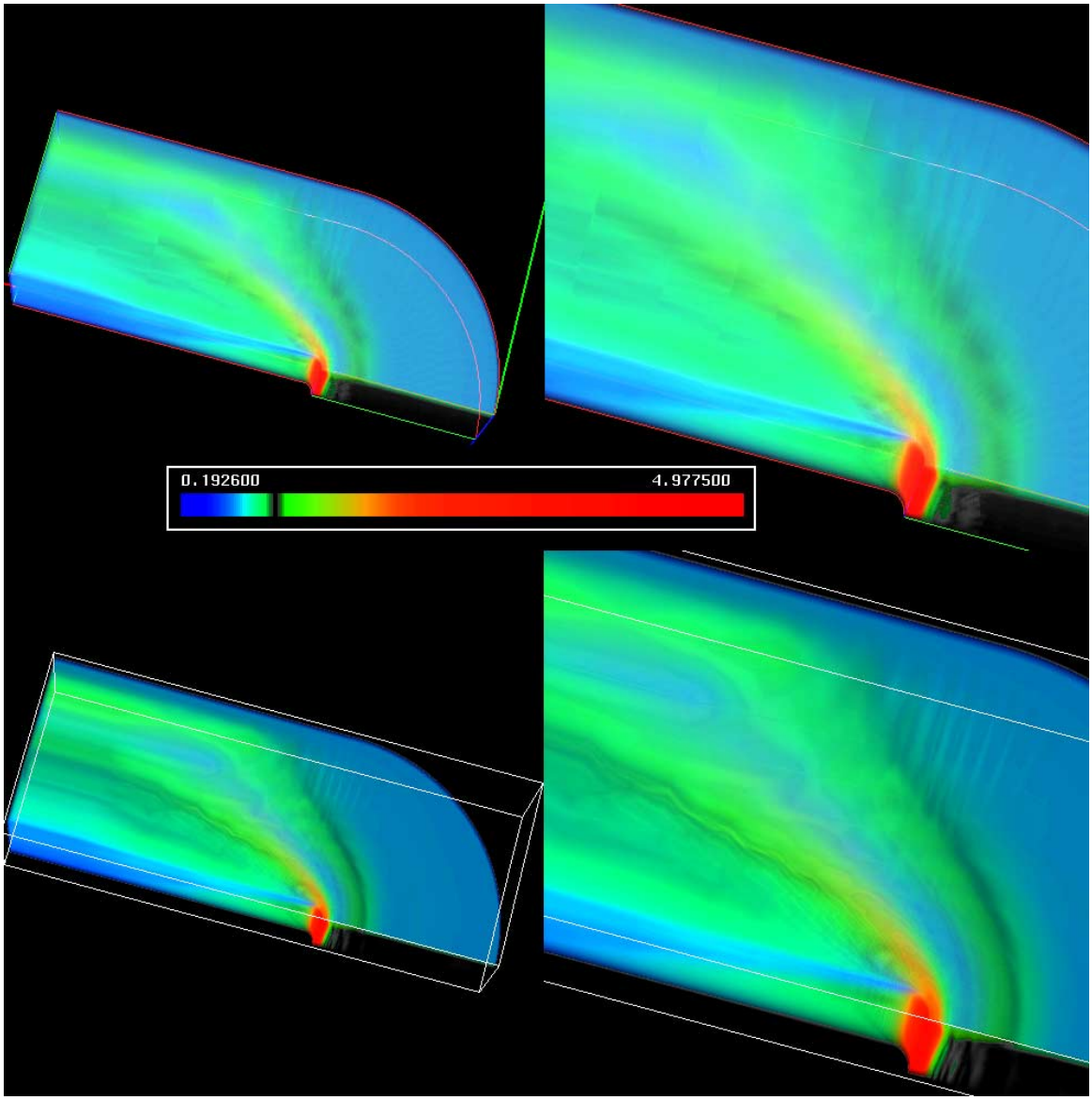

Figure 4.6: Comparison of blunt fin images with transfer function. (scale = 1.0, 2.0) QP (top) Scan-line (bottom)

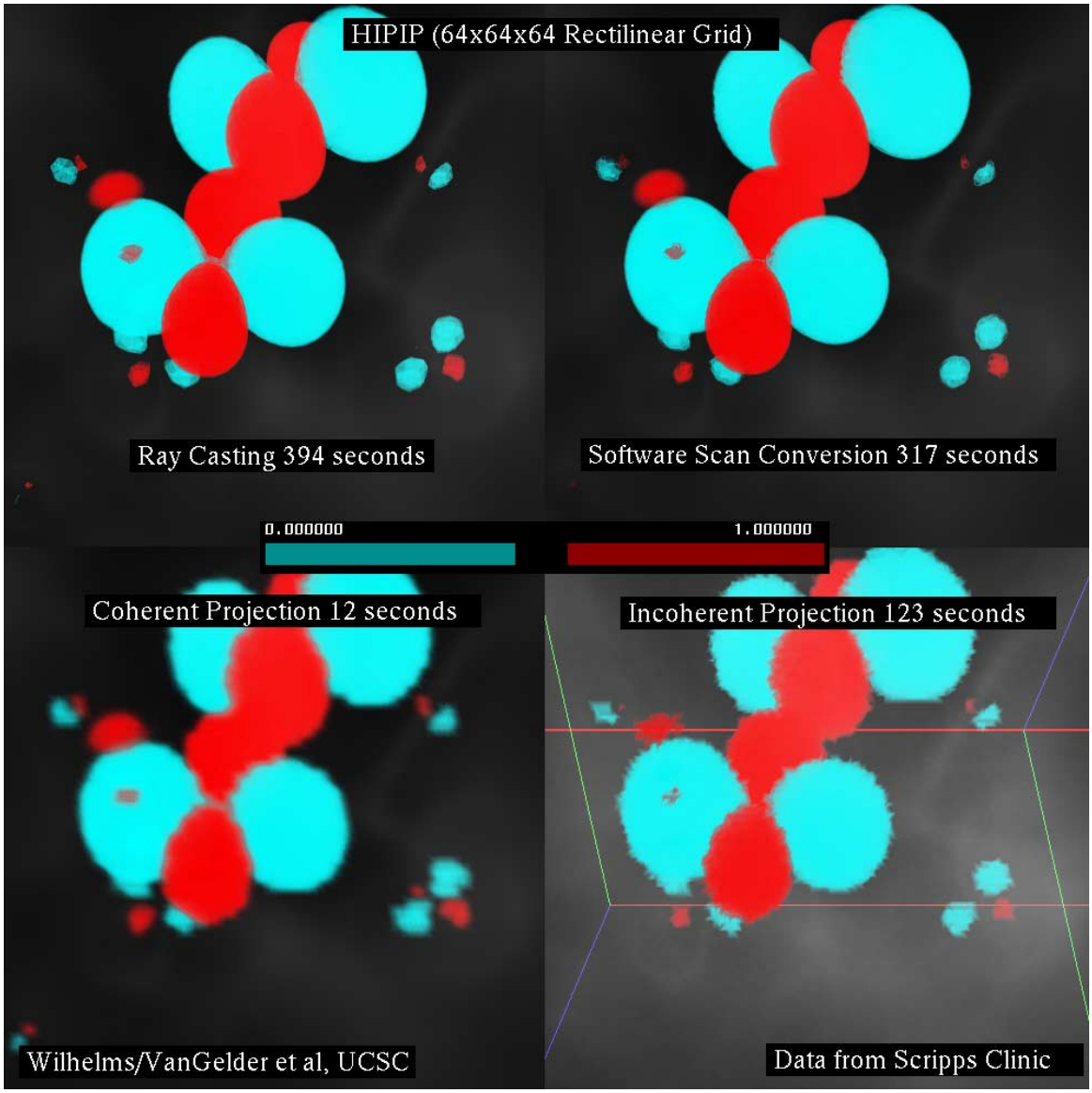

Figure 4.7: Rectilinear dataset (64x64x64) comparing ray-casting, scan-line, coherent projection, and incoherent projection algorithms.

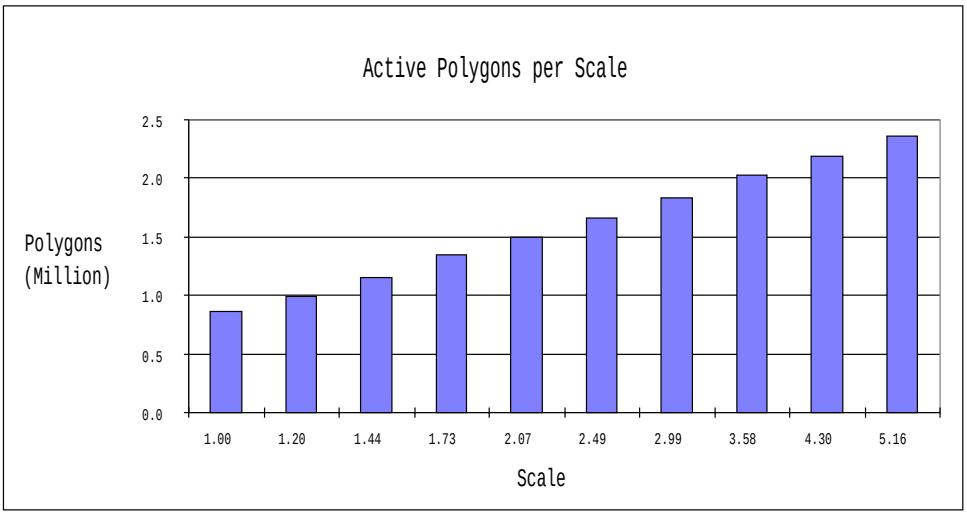

Figure 4.8: Number of active polygons for different scale factors (Shuttle data).

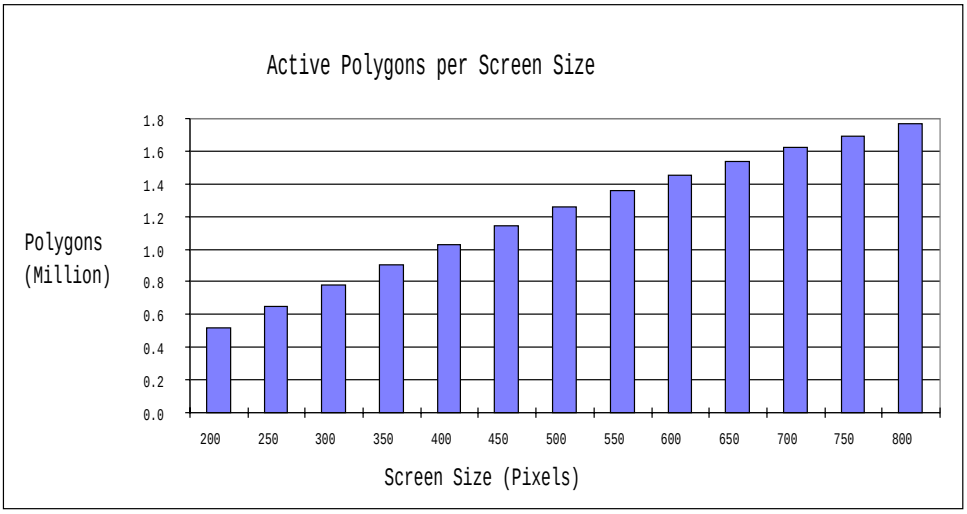

Figure 4.9: Number of active polygons for different screen sizes (Shuttle data).

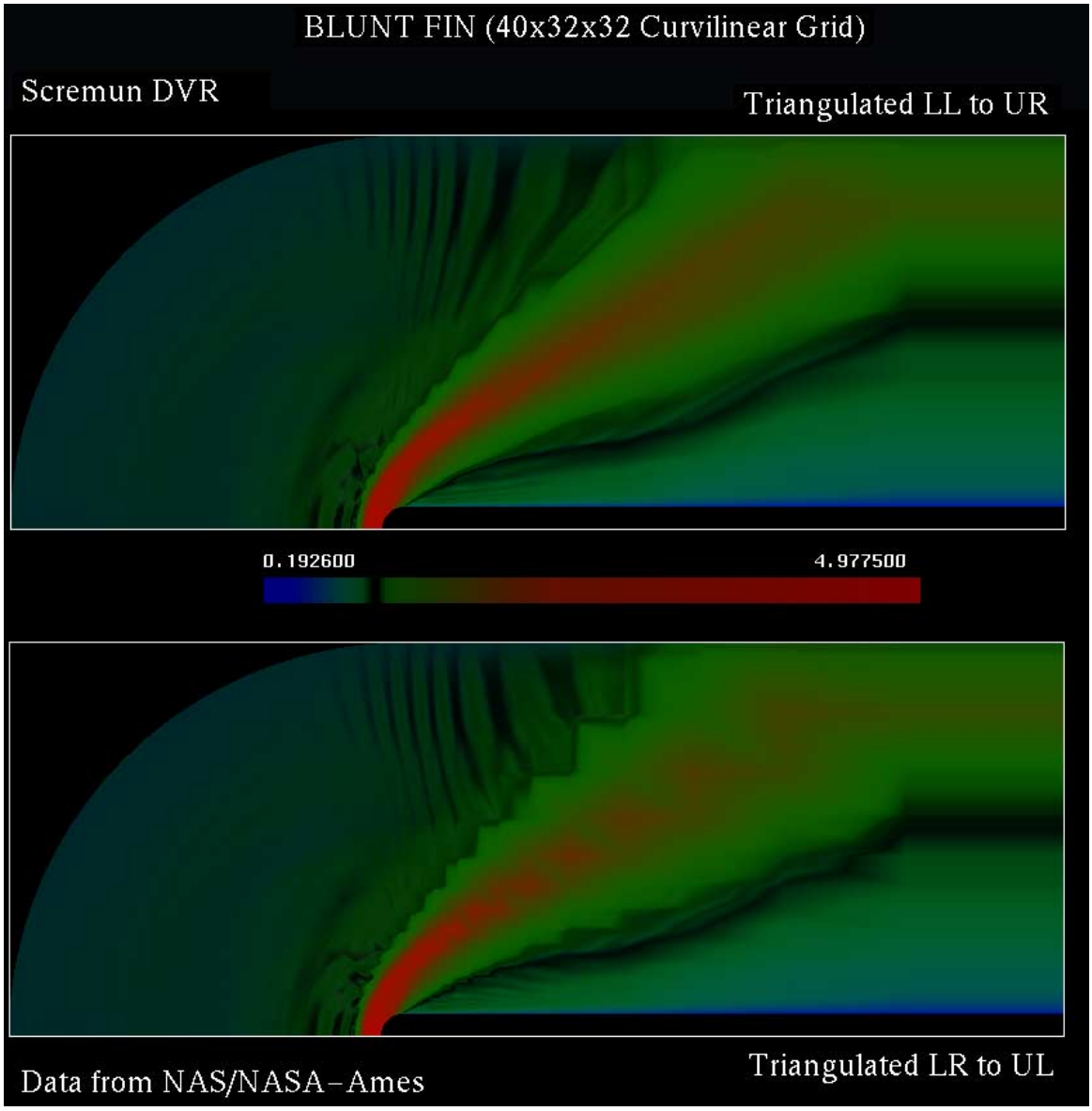

Figure 4.10: Two different triangulations of the blunt fin data set – the lower image shows resulting image discontinuities.

5. Conclusions

The renderer described in this paper allows direct volume rendering of large multiple intersecting curvilinear grids without the use of expensive graphics hardware. The general design allows rendering of any volume data that can be represented by triangles, including unstructured, curvilinear, and rectilinear grids. Factors such as screen size and scale can affect the time needed to render the volume, which provides a natural variable resolution feature. A hierarchical parsing of the grid into a tree can quickly identify, through a traversal, polygons that are inside the restrict region of the volume, which reduces the time needed to create Y-buckets. Approximation rendering can be useful for preliminary viewing of the volume. The renderer is parallelizable and measurements show that elapsed time can be greatly reduced without greatly increasing memory requirements. Accurate depth calculation can be achieved by using the projection method introduced in Section 3.3. In general, scan-line rendering produces a noticeably clearer image with less artifacts than the hardware Gouraud shading methods. While it is noticeably slower, this cost may often be worth it for the improvement in image quality.

References

[Akeley, 1993] Kurt Akeley. RealityEngine graphics. *Computer Graphics (ACM SIGGRAPH Proceedings)*, 27:109–116, August 1993.

[Bancroft *et al.*, 1990] G. V. Bancroft, F. J. Merritt, T. C. Plessel, P. G. Kelaita, R. K. McCabe, and Al Globus. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Visualization '90*, pages 14–27, San Francisco, CA., October 1990. IEEE.

[Bentley, 1975] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):214–229, 1975.

[Buning *et al.*, 1989] P.G. Buning, I.T. Chiu, Jr. F.W. Martin, R.L. Meakin, S. Obayashi, Y.M. Rizk, J.L. Steger, and M. Yarrow. Flowfield simulation of the space shuttle vehicle in ascent. *Fourth International Conference on Supercomputing*, 2:20–28, 1989. Space Shuttle data reference.

[Cabral *et al.*, 1994] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Symposium on Volume Visualization*, pages 91–98, Washington, D.C., October 1994.

[Challinger, 1991] Judy Challinger. Parallel volume rendering on a shared-memory multiprocessor. Technical Report UCSC-CRL-91-23, University of California, Santa Cruz, 1991.

[Challinger, 1993] Judy Challinger. *Scalable Parallel Direct Volume Rendering for Nonrectilinear Computational Grids*. PhD thesis, University of California, Santa Cruz, December 1993.

[Cignoni *et al.*, 1995] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno. On the optimization of projective volume rendering. In *Visualization in Scientific Computing 95'*, pages 58–71, Chia, Italy, May 1995. Sixth Eurographics Workshop Proceedings.

[Crawfis and Max, 1993] Roger Crawfis and Nelson Max. Texture splats for 3d scalar and vector field visualization. In *Visualization 93*, pages 261–265, San Jose, Ca, October 1993. IEEE.

[Cullip and Newman, 1993] T. J. Cullip and U. Newman. Accelerating volume reconstruction with 3d texture hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill, N. C., 1993.

[Dani *et al.*, 1994] Sandeep Dani, Joel E. Tohline, Jr. Warren N. Waggenspack, and David E. Thompson. Parallel rendering of curvilinear volume data. *Computers and Graphics*, 18(3):363–372, May-June 1994.

[Drebin *et al.*, 1988] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 22(4):65–74, July 1988.

[Ebert *et al.*, 1994] David S. Ebert, Roni Yagel, Jim Scott, and Yair Kurzion. Volume rendering methods for computational fluid dynamics. In *Visualization '94*, pages 232–239, Washington, DC, October 1994. IEEE.

[Foley *et al.*, 1990] James D. Foley, Andies Van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, Reading, Mass., 2 edition, 1990.

[Garrity, 1990] Michael P. Garrity. Raytracing irregular volume data. *Computer Graphics*, 24(5):35–40, December 1990.

[Giertsen and Peterson, 1993] Christopher Giertsen and Johnny Peterson. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics and Applications*, pages 16–23, November 1993.

[Giertsen, 1992] Christopher Giertsen. Volume visualization of sparse irregular meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.

[Glassner, 1984] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[Greene *et al.*, 1993] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. *Computer Graphics (ACM SIGGRAPH Proceedings)*, 27:231–238, August 1993.

[Guan and Lipes, 1994] S. Guan and R. G. Lipes. Innovative volume rendering using 3d texture mapping. In *SPIE: Medical Imaging 1994: Images Captures, Formatting and Display*. SPIE 2164, 1994.

[Hanrahan, 1990] Pat Hanrahan. Three-pass affine transforms for volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 24(5), November 1990.

[Hung and Buning, 1985] Ching-Mao Hung and Pieter G. Buning. Simulation of blunt-fin-induced shock-wave and turbulent boundary-layer interaction. *J. Fluid Mechanics*, 154:163–185, 1985.

[Ihm and Lee, 1995] Insung Ihm and Rae Kyoung Lee. On enhancing the speed of splatting with indexing. In *Visualization '95*, pages 69–76, San Jose, CA, November 1995. IEEE.

[Kajiya and Herzen, 1984] James T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. *Computer Graphics*, 18(4):165–174, July 1984.

[Koyamada, 1992] Koji Koyamada. Fast traversal of irregular volumes. In T. L. Kunii, editor, *Visual Computing - Integrating Computer Graphics and Computer Vision*, pages 295–312. Springer Verlag, 1992.

[Lacroute and Levoy, 1994] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics (ACM Siggraph Proceedings)*, pages 451–458, July 1994.

[Laur and Hanrahan, 1991] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 25(4):285–288, July 1991.

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

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

[Lucas, 1992] Bruce Lucas. A scientific visualization renderer. In *Visualization '92*, pages 227–233. IEEE, October 1992.

[Ma and Painter, 1993] Kwan-Liu Ma and James S. Painter. Parallel volume visualization on workstations. *Computers and Graphics*, 17(1):31–37, Jan.-Feb. 1993.

[Ma, 1995] Kwan-Liu Ma. Parallel volume ray-casting for unstructured grid data on distributed memory architectures. In *1995 Parallel Rendering Symposium*, pages 23–30. IEEE Computer Society Technical Committee on Computer Graphics in cooperation with ACM SIGGRAPH, The Association for Computing Machinery, Inc., 1995.

[Mao *et al.*, 1987] X. Mao, T. Kunii, I. Fuhishiro, and T. Nomoa. Hierarchical representations of 2D/3D gray-scale images and their 2D/3D two-way conversion. *IEEE Computer Graphics and Applications*, 7(12):37–44, December 1987.

[Mao *et al.*, 1995] Xiaoyang Mao, Lichan Hong, and A. Kaufman. Splatting of curvilinear volumes. In *Visualization '95*, pages 61–68, San Jose, CA, November 1995. IEEE.

[Martin Jr. and Slotnick, 1990] F.W. Martin Jr. and J.P. Slotnick. Flow computations for the space shuttle in ascent mode using thin-layer navier-stokes equations. *Applied Computational Aerodynamics, Progress in Astronautics and Aeronautics*, 125:863–886, 1990.

[Max *et al.*, 1990] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics (ACM Workshop on Volume Visualization)*, 24(5):27–33, December 1990.

[Max *et al.*, 1995] Nelson Max, Roger Crawfis, and Barry Becker. Applications of texture mapping to volume and flow visualization. In *Proceedings of Graphicon '95*, July 1995.

[McLendon, 1991] Patricia McLendon. *Graphics Library Programming Guide*. Silicon Graphics, Inc., Mountain View, CA, 1991.

[Meagher, 1982] Donald J. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.

[Sabella, 1988] Paolo Sabella. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics*, 22(4):51–58, July 1988.

[Schroeder and Stoll, 1992] Peter Schroeder and Gordon Stoll. Data parallel volume rendering as line drawing. In *1992 Workshop on Volume Visualization*, pages 25–32, Boston, Mass., October 1992. ACM.

[Shirley and Tuchman, 1990] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, December 1990.

[Stein *et al.*, 1994] Clifford M. Stein, Barry G. Becker, and Nelson L. Max. Sorting and hardware assisted rendering for volume visualization. In *Symposium on Volume Visualization*, pages 83–89, Washington, DC, March 1994. IEEE.

[Upson and Keeler, 1988] Craig Upson and Michael Keeler. The v-buffer: Visible volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 22(4):59–64, July 1988.

[Uselton, 1991] Sam Uselton. Volume rendering for computational fluid dynamics: Initial results. Technical Report RNR-91-026, NAS-NASA Ames Research Center, Moffett Field, CA, 1991.

[Uselton, 1993] Sam Uselton. Parallelizing volvis for multiprocessor sgi workstations. Technical Report RNR-93-013, NAS-NASA Ames Research Center, Moffett Field, CA, 1993.

[Van Gelder and Wilhelms, 1993] Allen Van Gelder and Jane Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. In *Visualization '93*, San Jose, CA, October 1993. IEEE. (extended abstract) Also, University of California technical report UCSC-CRL-93-02.

[Van Gelder *et al.*, 1995] Allen Van Gelder, Kwansik Kim, and Jane Wilhelms. Hierarchically accelerated ray casting for volume rendering with controlled error. Technical Report UCSC-CRL-95-31, University of California, Santa Cruz 95064, Santa Cruz, CA 95064, March 1995.

[Walsum *et al.*, 1991] Theo Van Walsum, Andrea J.S. Hin, Jack Versloot, and Frits H. Post. Efficient hybrid rendering of volume data and polygons. In Frits H. Post and Andrea J.S. Hin, editors, *Advances in Scientific Visualization*, pages 83–96, Delft, Netherlands, April 1991.

[Watkins, 1970] G.S. Watkins. *A Real Time Visible Surface Algorithm*. PhD thesis, University of Utah, Salt Lake City, June 1970.

[Watt and Watt, 1992] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. ACM Press, New York, New York, 1992.

[Westover, 1990] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 24(4):367–76, August 1990.

[Wilhelms and Van Gelder, 1991] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 25(4):275–284, 1991.

[Wilhelms and Van Gelder, 1992] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992. Extended abstract in ACM Computer Graphics 24(5) 57–62; also UCSC technical report UCSC-CRL-90-28.

[Wilhelms and Van Gelder, 1994] Jane Wilhelms and Allen Van Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *ACM Workshop on Volume Visualization 1994*, Washington, D.C., October 1994. See also technical report UCSC-CRL-94-02.

[Wilhelms *et al.*, 1990] Jane Wilhelms, Judy Challinger, Naim Alper, Shankar Ramamoorthy, and Arsi Vaziri. Direct volume rendering of curvilinear volumes. *Computer Graphics*, 24(5):41–47, December 1990. Special Issue on San Diego Workshop on Volume Visualization.

[Wilhelms, 1993] Jane Wilhelms. Pursuing interactive visualization of irregular grids. *The Visual Computer*, 9(8):450–458, 1993.

[Williams, 1992a] Peter Williams. Interactive splatting of nonrectilinear volumes. In *Visualization '92*, pages 37–44. IEEE, October 1992.

[Williams, 1992b] Peter Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[Wilson *et al.*, 1994] Orion Wilson, Allen Van Gelder, and Jane Wilhelms. Direct volume rendering via 3d textures. Technical Report UCSC-CRL-94-19, CIS Board, University of California, Santa Cruz, 1994.

[Wittenbrink and Harrington, 1994] Craig M. Wittenbrink and Michael Harrington. A scalable mimd volume rendering algorithm. In *Eighth International Parallel Processing Symposium*, pages 916–920, Cancun, Mexico, April 1994.

[Yagel *et al.*, 1995] Roni Yagel, David S. Ebert, James N. Scott, and Yair Kurzion. Grouping volume renderers for enhanced visualization in computational fluid dynamics. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):117–132, July 1995.

[Yamaguchi *et al.*, 1984] K. Yamaguchi, T. L. Kunii, and K. Fujimura. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, 4(1):53–59, January 1984.