Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids

Jane Wilhelms, Allen Van Gelder, Paul Tarantino, Jonathan Gibbs*
University of California, Santa Cruz

Abstract

A general volume rendering technique is described that efficiently produces images of excellent quality from data defined over irregular grids having a wide variety of formats. Rendering is done in software, eliminating the need for special graphics hardware, as well as any artifacts associated with graphics hardware. Images of volumes with about one million cells can be produced in one to several minutes on a workstation with a 150 MHz processor.

A significant advantage of this method for applications such as computational fluid dynamics is that it can process multiple intersecting grids. Such grids present problems for most current volume rendering techniques. Also, the wide range of cell sizes (by a factor of 10,000 or more), which is typical of such applications, does not present difficulties, as it does for many techniques.

A spatial hierarchical organization makes it possible to access data from a restricted region efficiently. The tree has greater depth in regions of greater detail, determined by the number of cells in the region. It also makes it possible to render useful "preview" images very quickly (about one second for one-million-cell grids) by displaying each region associated with a tree node as one cell. Previews show enough detail to navigate effectively in very large data sets.

The algorithmic techniques include 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, coarse-grain parallelism for a shared-memory MIMD architecture, a new perspective transformation that achieves greater numerical accuracy, and a scanline algorithm with depth sorting and a new clipping technique.

Keywords: Computer Graphics, Scientific Visualization, Scanline, Direct Volume Rendering, Curvilinear Grid, Irregular Grid, k-D Tree.

1 Introduction

Direct volume rendering is an attractive technique because it can convey a great deal of information in a single image by mapping the scalar data values in a sample volume to color and opacity. However, a great amount of computation is required to calculate the information presented, particularly when the samples are on irregular grids. The problem is further exacerbated when the data sets are very large.

This paper presents a direct volume rendering technique developed for a wide range of grids, including curvilinear grids with hexahedral cells, tetrahedral irregular grids, and rectilinear grids. Furthermore, this method can process multiple intersecting grids, which are often produced in complex computational fluid dynamics simulations [2]. Most previous methods will not work correctly on such grids. The principal contributions are as follows:

1. At the heart of the system is a *software scan conversion* method of direct volume rendering based on treating the faces of grid cells as independent polygons. This method generalizes polygon scanline methods in that it renders semi-transparent regions of space *between* polygons, as well as opaque polygonal surfaces. Also, it introduces a new clipping technique that makes it unnecessary to clip individual polygons. This algorithm can be used independently of any hierarchy. It does not require any graphics hardware and produces excellent quality images (See Section 3.)
2. The basic algorithm has been implemented to run in parallel on shared-memory MIMD machines. Parallelization is relatively easy to implement and fits naturally into the algorithm. We observed a speed-up of better than 3.25 on four processors. (See Section 4.)
3. To efficiently visualize very large data sets, the program builds a k-d tree over the polygons, organizing them by spatial location. The hierarchy is useful in two ways. First, if one zooms in or otherwise restricts viewing to a part of the volume, the hierarchy permits the program to avoid processing data that is clearly not visible. Second, an *approximate model* of the data present within any subtree can be stored in the root node of the subtree. An error term in each node indicates how closely the local model approximates the actual data. Approximate images can be quickly produced. (See Section 5.)
4. A new *perspective transformation* achieves greater numerical accuracy on geometrical elements spanning a great range of sizes. Standard techniques lose accuracy in screen-z, leading to significant errors, both in color integration and in element order (See Section 6).

We chose a projection-style algorithm because we were not designing our algorithm for massively parallel machines, we wanted to maximize coherence, and we wanted to avoid the difficulties of ray-casting intersecting cells.

We preferred the faced-based scanline approach because of its generality and speed (Section 3). We parallelized on a scanline basis because it limits duplication, is easy to implement, and provides reasonable speed-up for small-scale parallelization (see Section 4). We associated irregular grid data (and/or polygon mesh data) with a k-d tree. We believe this is the first use of a hierarchy with general irregular, possibly intersecting, grids.

2 Background and Related Work

Early approaches for direct volume rendering used *ray-casting*, *cell projection*, and *splatting* (voxel projection) [7]. Most research has addressed only rectilinear grids, and most previously reported acceleration and optimization techniques apply only to such grids. New methods including Fourier transforms, shear-warp transforms, or 3D texture maps suffer this limitation.

However, many applications create non-rectilinear volume data sets, such as computational fluid dynamics (CFD), finite element

*Computer Science Dept, Univ. of California, Santa Cruz, CA 95064, USA. E-mail: {wilhelms,avg,pault,jono}@cs.ucsc.edu

analysis (FEM), and atmospheric and oceanographic measurements. Such data is often found on *curvilinear grids* (where a computational regular grid is warped to fit around objects of interest), and *unstructured grids* (where data points are connected to form tetrahedral or other polyhedral cells). Sometimes non-tetrahedral cells are broken into tetrahedra to simplify processing; however, this can lead to artifacts and increases the number of primitives. Multiple overlapping and intersecting grids may be used to sample space around very complex shapes [2]. Our research concentrates on rendering such irregular data.

Many complexities are introduced when imaging data on irregular grids and multiple grids, as opposed to rectilinear grids. A visibility ordering (front-to-back) is not implicit. Many operations are much more expensive, such as intersecting rays with cells, projecting irregular cells, and interpolating across faces or through cells.

A number of algorithms have been developed for irregular grids. Ray-casting general irregular grids is complicated and slow (though it does parallelize beautifully) [3, 8, 16, 22]. Cell projection and splatting have been used for irregular grids in software [9, 13, 10, 18, 19, 29] and hardware [21, 24]. Though we process faces and they process volumetric regions, the methods of Max et al [19] and Giertsen [9, 10] are related to ours in that they use scan-conversion for efficiency.

Lucas [15] implemented a face projection method for irregular grids. Unlike ours (Section 3), he appears to fully sort all faces in depth and then scan convert them in visibility ordering. This is more closely related to the classic "Painter's Algorithm" for polygons, which is not widely used because of the costly z-sort. Lucas only sorts on the centroids of faces, which can produce an incorrect ordering. Challinger [4, 5] explored direct volume rendering of irregular volumes on massively parallel machines. She initially used ray-casting, but then developed a face projection method for speed. She partitioned the screen into "tiles" for each processor, but within each tile her algorithm is very close in philosophy to ours, though data structures differ.

Two approaches for ray-casting of irregular grids on massively parallel machines have been described [3, 16]. Challinger distributed data to processors after conversion to screen space, which must be redone after each transformation [3]. Ma distributed subvolumes to processors once, and composited contributions found by ray-casting the subvolumes [16]. In their projection methods, Lucas [15], Challinger [5], and Giertsen et al [10] all parallelize by breaking the screen space into rectangular tiles which are assigned to processors. This may require duplication, and as scanlines are broken into pieces, some coherence is lost.

Concerning hierarchies, Laur and Hanrahan used an octree on regular volumes, stored an average value at each node to approximate subtree data, and used splatting [14]. Two of the present authors implemented an octree on regular volumes with trilinear nodal models and a choice of error terms, explored more complex approximate nodal models, and used cell projection for rendering [28]. Cignoni et al [6] created a hierarchy over tetrahedral grids.

3 The Scanline Algorithm

The scanline technique has the merits of both generality and coherence. It is an extension of Watkin's scanline algorithm [7, 25]. Sample data values are supplied at the vertices of polyhedral cells whose faces are (possibly non-planar) polygons. These faces are processed independently in the algorithm. While a great deal of work has been done (mostly in the 1980's) on the problem of scanline rendering of polygonal data sets, we wish to emphasize that using scan conversion for direct volume rendering is a different problem. Most importantly, in polygon scan conversion it is the *surfaces* that have color properties (and these are usually opaque, permitting further simplifications). In direct volume rendering, it is the material *between* the cell surfaces that has color properties, and this material is often semi-transparent.

Figure 1 gives the high-level procedure. The remainder of this section describes the main steps further. See the technical report [26] for further details.

Procedure **scan:**

1. If a new geometrical transformation is specified, convert vertex locations from world space to screen space (Section 6).
2. For each (horizontal) scanline of the image, create its *y-bucket* list.
3. Initialize *y-actives* list as empty.
4. For each (horizontal) scanline of the image, in bottom-to-top order:
   (a) Update *y-actives* list from previous line's y-actives and this line's y-bucket.
   (b) For each pixel in scanline, create its *x-bucket* list.
   (c) Initialize *x-actives* list as empty.
   (d) For each pixel in scanline, in left-to-right order: update *x-actives* list from previous pixel's x-actives and this pixel's x-bucket; composite polygons in x-actives in front-to-back order.

Figure 1: The main processing steps are given here are described in Section 3.

3.1 Polygon Creation

The volume is decomposed into polygons, each given a unique integer identifier (polygon ID). Although many aspects of the design are compatible with a variety of polygons, the implementation permits only triangles. Because multiple grids are allowed, as well as surface polygon meshes, a grid ID is encoded in certain bits of the polygon ID, and specifies with which grid or surface this polygon is associated. The remaining bits of the polygon ID comprise either an index into an array of polygon structures (for irregular grids), or a direct encoding of the polygon's position within the grid (an option for curvilinear and rectilinear grids). Each polygon structure consists of an array of its vertex indices. For curvilinear and rectilinear grids, every cell has the same facial structure of 12 triangles, and the cell vertices can be inferred from the cell ID, so the vertex information can be encoded into the polygon ID, bypassing the use of polygon structures.

Any group of polygons to be rendered includes the faces of a *clipping box*, which delimits the subvolume to be rendered. The six faces (12 triangles) of the clipping box are treated much like other polygons during processing, but take on a special meaning for clipping during pixel processing (see Section 3.4).

3.2 Y-Bucket Processing

Each scanline has associated with it a *y-bucket* list consisting of the IDs for those polygons that first appear on that scanline (processed bottom-to-top). Polygons are excluded from y-buckets if they are outside the clipping box or do not cross any pixel center. After computing y-bucket lists, each scanline must be processed and drawn into the software frame buffer. A *y-actives* list containing the polygon IDs of those polygons contributing to a scanline is maintained. Before processing the current scanline, the *y-actives* list must be updated by removing polygons from the previous scanline's y-actives that are no longer active on this scanline, and adding new polygons from the current *y-bucket*. The y-actives list is implemented as an array of structures, each containing various polygon information.

For polygons on the boundary of a grid, the vertices are stored in counter-clockwise order when viewed from outside the grid, so that it can be easily recognized whether a boundary polygon is the first or last in its grid, in visibility order.

3.3 X-Bucket Processing

When processing a scanline, information for each active polygon is transferred to an *x-bucket* data structure associated with the leftmost pixel in which the polygon appears. A linked list of these structures is associated with each x-bucket in front-to-back visibility order. As the scanline is processed, an *x-actives* list is maintained and updated for each pixel. It contains the data value and the screen depth (screen-z) for each polygon contributing to that pixel. As with the y-actives list (Section 3.2), polygons that become inactive at the current pixel are deleted, and polygons in the current x-bucket are inserted, except that now the list is maintained in sorted order. The surviving polygons of the previous pixel's x-actives list must be updated with newly interpolated values of field data and screen-z. New screen-z values may require rearrangement of polygons among survivors, and new polygons must be merged in, maintaining screen-z order.

3.4 Pixel Processing

The *x-actives* list for a particular pixel is traversed front-to-back to accumulate the color and opacity. In our implementation, data values (interpolated to that pixel) for each pair of adjacent polygons are averaged, and the average is used as the parameter of a transfer function that provides a color and opacity value. Taking into account the line-of-sight distance between the two polygons (Section 6), the color-opacity contribution for that inter-polygon region is calculated. This contribution is composited into a software floating point frame buffer. Standard equations for this are found elsewhere, e.g. [27].

A set of *clipping* polygons encloses the region to be rendered, removing the need to explicitly clip any polygon. Therefore, there are two clipping polygons in any pixel's x-actives list. The first indicates pixel contributions should begin to be accumulated, and the second indicates contributions should stop being accumulated.

4 Parallelizing the Algorithm

Three primary components of this algorithm can, by and large, be easily parallelized. The first is the transformation of the vertices from world space to pixel space, and is trivially parallelizable, as each vertex can be transformed independently. The others require some discussion. Step numbers in this section refer to Figure 1.

The second parallelizable component is the task of grouping polygons by scanline into y-buckets (step 2). Its parallelization is more involved, but highly scalable. It proceeds in two passes. In the first pass, each processor is given an equal number of polygons to process. One temporary array stores, for each polygon, the lowest scanline where it appears, or an invalidation flag if it doesn't cross any pixel centers within the clipping box (as described in Section 3.2). A second, two-dimensional, array stores, for each scanline and processor, how many polygons the processor found that belong in that y-bucket.

In the second pass, the number of polygons belonging in each y-bucket is found by accessing the latter array, and space for each y-bucket is allocated by partitioning a common output array. The space for one y-bucket is further partitioned into space for each processor to fill within that y-bucket. (While this transitional step can also be parallelized by standard techniques, its work per processor is only proportional to the number of scanlines, not the number of polygons.) During the second pass, each process creates appropriate y-bucket structures for all polygons that it processed in the first pass, except those that were invalidated.

The third parallelizable component involves processing each scanline in order from bottom to top, which involves the bulk of the computation (steps 4a through 4d). There are several ways of implementing this part in parallel. Our implementation contains a "critical section" of code (step 4a). Only one processor can run the critical section at a time. During this section, a processor updates the current y-actives list for the scanline, takes a copy of it, and then exits the critical section. Then it builds the x-buckets and processes the scanline (steps 4b through 4d).

This implementation is not 100% scalable because the critical section can act as a bottleneck as more processors are added. However, it was fairly easy to implement and caused low contention with the four processors we had available (Section 7). Our measurements suggest that it can extend to about 16 processors before contention becomes a serious drawback.

5 Use with a Multi-Resolution Hierarchy

A method of spatial partitioning in k dimensions, called k-d trees, was introduced by Jon Bentley [1]. A binary tree is built that splits in one dimension at a time. Our current implementation splits in a round-robin fashion, but one could easily adopt a more sophisticated policy based on the locations of objects. At tree node v, the hyperplane that splits the region is orthogonal to the x_i-axis when splitting on dimension i. Our implementation bisects the region, but in general, any partitioning value X_v can be chosen.

5.1 Creation of the Hierarchy

Each node v in the tree has associated with it a list of polygon IDs. It is built as follows: for each polygon p passed into the tree node, which is splitting on dimension i, if all vertices have an x_i location less than X_v, place p in the set to be passed to the left subtree. If all vertices have an x_i location that is greater than the bisection value, place p in the set to be passed to the right subtree. Otherwise, retain p in the set to be stored at v (see Figure 2). Then, process the left and right subtrees. If the number of polygons passed into node v is below a user-defined threshold, no splitting occurs, and the node is a leaf. Note this results in polygons being associated with the smallest node region that completely contains them, following Greene [11], and avoids subdividing polygons.

We developed a new, flexible storage method that allows us to avoid the extra space of linked lists and still pass all objects associated with a subtree in one operation. Polygon IDs are stored in one common array Tid, and arranged in the order that they would appear during a prefix-order traversal of the k-d tree. (A post-fix order can also be easily made to work.) Thus, all polygons stored

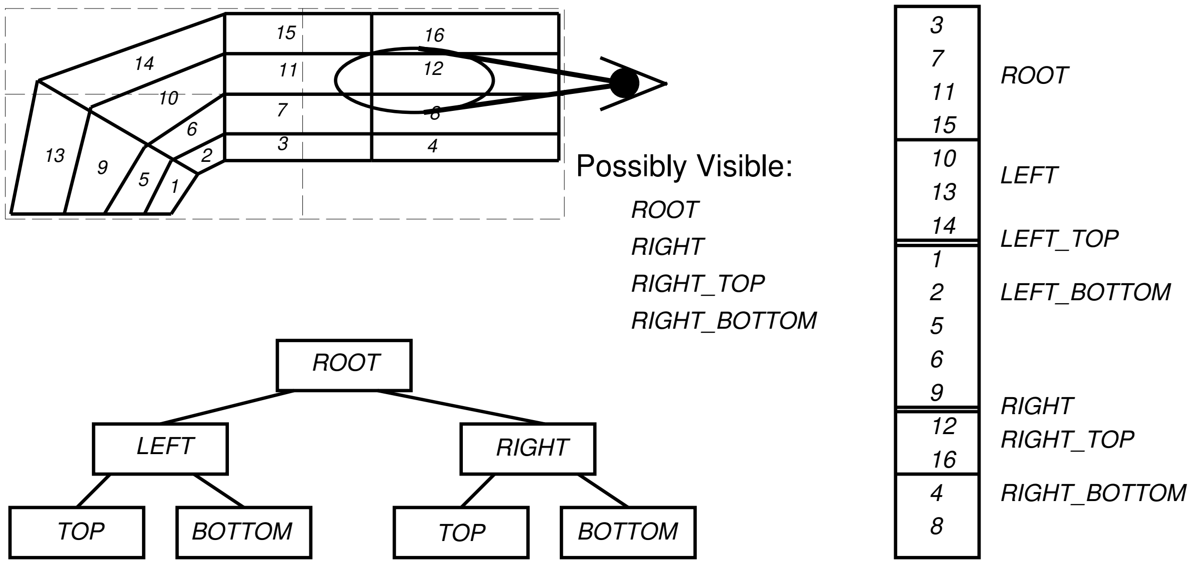

Figure 2: Distributing polygons in a k-d tree, as described in Section 5. Grid is shown as solid lines, with numbered cells, upper left. Dashed lines show partitioning by k-d tree, whose skeleton appears lower left. The *Tid* array is partitioned into segments, as shown at right. For the view indicated, the "Possibly Visible" list identifies the segments belonging to tree nodes whose regions intersect the subvolume to be imaged.

in a subtree form a contiguous section in Tid, and all polygons associated with that subtree's root node are at the beginning of that segment.

Assuming the array begins with a known subscript of 0, it is necessary to store globally the total number of polygons in the tree, and to store in each node the offset to the *beginning* of its set of polygons. The remaining values can be recovered during traversal, provided the end-point for each subtree is passed in as a parameter of the traversal. For the whole tree, this is just the total number of polygons. Thus if a tree node v is visited with end-point E passed in:

- The end-point for polygons associated with node v is the offset (begin-point) of v's left child;
- the end-point for the left child is the offset of v's right child;
- and the end-point for the right child is E.

In this convention, an end-point is the first index *above* the segment.

5.2 Fully Detailed Rendering

To prepare for rendering, the tree is traversed and a set of subranges of polygon IDs is developed, which represent all polygons within the user-defined restrict box. Recall that all polygon IDs within the spatial region represented by one node are stored contiguously in the polygon array, *Tid* (Section 5.1). If the entire node is outside the restrict box, the traversal returns from that branch immediately. If the node is entirely inside the restrict box, then the subrange for that node is added to the set of subranges to be rendered, and again the traversal returns without exploring the subtrees. The same applies for leaf nodes, and for nodes whose region contains fewer polygons than a user-specified cut-off. If a node is partially inside the restrict box and none of these exceptions applies, then the subrange of *Tid* associated with the node itself (but not its descendants) is added to the set of subranges to be rendered, and traversal continues into this node's children, which are treated recursively in the same manner.

When this traversal is completed, we have a *possibly visible* list (represented as a set of start and end positions within *Tid*) of all possibly relevant polygons (see Figure 2). This list is then sent to the renderer for processing (see Section 3).

5.3 Multi-Resolution Rendering

For *multi-resolution rendering*, we wish to display error-controlled approximations of the data for speed. For this, each k-d tree node must contain a model representing an approximation of the data in the subregion it represents, called a *nodal region*. The tree node also stores an error term representing the deviation of the model from the data. Our multi-resolution technique departs from that associated with wavelets [17] in that each level represents a complete model of its region, whereas wavelets represent just the detail information not represented at higher levels. For rendering, the user specifies an acceptable error. During the traversal, if the error associated with the node is less than the acceptable error, or the node is a leaf, traversal stops there and the nodal region is drawn.

At present, our nodal model is a rather simple one, but it constructs very fast "preview" images to help orient the user in a very large data set (see Figure 9). Eight data values are stored with each k-d tree node representing the values of the eight corners of its nodal region. These corner values induce a trilinear function throughout the nodal region. The data values are found, for each corner point, by locating the grid cell that contains the corner of the nodal region, then interpolating within that grid cell to compute the data value at that nodal corner point.

A future implementation is planned to support a mixture of rendering modes, with some nodal regions being approximated by the model and others being rendered in detail with their polygons. Tree traversal occurs in front-to-back visibility order. When rendering a nodal region by scan conversion, the algorithm of Section 3 will be used, with the local clipping box being the intersection of the boundary of the nodal region and the global

Symbol	Definition	Units
h_s	screen height	pixels
e_s	eye-to-screen distance	pixels
e_o	eye-to-Vbb-origin distance	world units
d	Vbb diagonal	world units
s	user's scale factor	pure number
x_t, y_t, z_t	user translation (after rot.)	world units
x_v, y_v, z_v	world coordinates (after rot.)	world units
x_s, y_s, z_s	screen coordinates	pixels

Figure 3: Perspective calculation notation. Visibility bounding box (Vbb) is centered at $(0, 0, -e_o)$ before user translation.

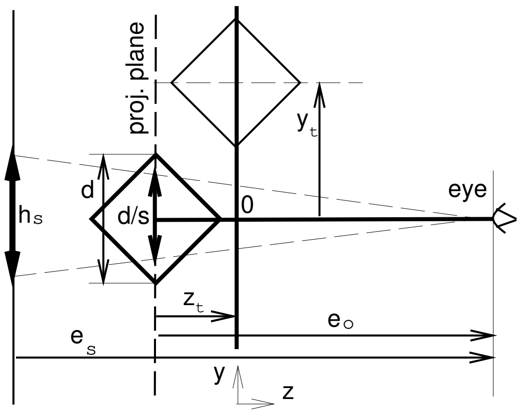

Figure 4: Figure of perspective view in world space. Eye is at (0,0,0). Vbb is projected onto screen at left.

restrict box. Nodal regions with smooth data will be rendered by an approximation technique. Separately rendered regions will be composited in the usual manner. There are numerous technical problems to be overcome to avoid discontinuities on the boundaries between nodal regions using different rendering methods.

If we restrict ourselves to only drawing nodal regions, then standard cell projection methods can be used [27], which may take advantage of the graphics hardware. To avoid color-interpolation inaccuracies (as produced by hardware Gouraud shading), we subdivide large nodal regions on the fly and interpolate data to their corners, until nodes that cover a reasonably small number of pixels are obtained.

6 Numerically Stable Perspective Transformation

Standard computer graphics software provides perspective transformations based on the eye being at the origin [7]. Standard preparation for a perspective transformation is to translate the eye to the origin. However, with our data, that resulted in translating many vertices *away* from the origin in screen-z, introducing substantial *relative* floating point errors. Some ordering inconsistencies arose in later computations. Double precision for all values involved would have imposed unacceptable space requirements. For numerical accuracy we needed a different geometrical basis, in which the eye is away from the origin. The transformation introduced here preserves accuracy of the screen-z values near the center of the *visibility bounding box* (Vbb), which is the portion of the volume being imaged. In fact, our transformation maps the center of the Vbb to 0 in screen-z.

Some texts give perspective transformations based on the projection plane passing through the origin [20]. While this puts the eye away from the origin, it did not suffice for our purposes because a user-specified translation might separate the center of the Vbb from the projection plane. (For a general 3D perspective transformation, the projection plane is the set of points whose *transformed* homogeneous coordinate is 1.)

This section presents a perspective transformation with a new geometrical basis, in which both the eye and the projection plane may be away from the origin (see Figs. 3 and 4). Specifically, the eye point is at $(0, 0, e_o - z_t)$, and the projection plane is $z = -z_t$. In addition, the Jacobian of this transformation is the identity at the origin. This transformation provided us with the numerical accuracy we needed in screen-z.

The Vbb is first centered at (0,0,0) in world space, then it is rotated by the user rotations. Now the eye is placed at $(0, 0, e_o)$. At this time the projection plane is $z = 0$. Then the user translates (exaggerated in the diagram) are applied, as follows. The user translate in z has the effect of moving the eye and the projection plane by $-z_t$. However, the user translates in x and y move the *Vbb* off the z-axis. The user translate in z is restricted to obey $z_t + \frac{1}{2}d + .05e_o \le e_o$. Finally, transformed points are uniformly scaled by a factor:

$$\frac{e_s}{e_o} = \frac{s\,h_s}{d}, \tag{1}$$

to convert world units into pixels. For additional details of the derivations, see [26].

$$\begin{aligned} x_s &= \frac{e_s(x_v + x_t)}{e_o - z_t - z_v} \\ y_s &= \frac{e_s(y_v + y_t)}{e_o - z_t - z_v} \\ z_s &= \frac{e_s\, z_v\, e_o}{(e_o - z_t)(e_o - z_t - z_v)} \end{aligned} \tag{2}$$

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

$$z_{v2} - z_{v1} = \frac{e_o\, e_s(z_{s2} - z_{s1})}{\left(\frac{e_o\, e_s}{e_o - z_t} + z_{s2}\right)\left(\frac{e_o\, e_s}{e_o - z_t} + z_{s1}\right)} \tag{3}$$

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:

$$dist = \frac{z_{v2} - z_{v1}}{cos(\theta)} = \frac{(z_{v2} - z_{v1})}{\sqrt{\frac{(z_{v1} - e_o)^2}{x_{v1}^2 + y_{v1}^2 + (z_{v1} - e_o)^2}}} \tag{4}$$

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 8).

7 Experimental Results

We first compare the performance of the method on a curvilinear multi-grid using single and multiple processors. Next, we compare

Scale	1.00	2.07	3.00	5.16
Active Polygons (Millions)	0.865	1.495	1.836	2.360
1 processor	54	113	138	174
2 processors	31	63	78	98
3 processors	21	44	54	68
4 processors	17	34	42	53
4 (ideal)	14	28	35	43

Figure 5: Elapsed time comparisons (in seconds) on an SGI Onyx using four 150-MHz processors, with 256MB memory. (NASA space shuttle data set)

the performance of the algorithm against other renderers we have implemented. Finally, we explore the ramifications of the hierarchy. For our results, we used the following data volumes: the *blunt fin* [12] (a single curvilinear grid of 40,960 data points); the *space shuttle* [2] (nine intersecting curvilinear grids consisting of 941,159 data points); the *Lockheed fighter*, courtesy of John Batina of NASA Langley Research Center (an unstructured tetrahedral grid with accompanying polygon surface file consisting of 13,832 data points and 70,125 tetrahedra); and, for comparison to rectilinear grid renderers, the *hipip* molecular data set courtesy of L. Noodleman and D. Case, Scripps Clinic (a rectilinear grid with 64x64x64 resolution, or 262,144 data points); and the rectilinear *CTHead* data set from UNC (at a resolution of 200x200x50 or 2,000,000 sample points).

7.1 Performance on Single and Multiple Processors

Performance with one to four processors on an SGI Onyx, with four 150-MHz processors, is shown in Figure 5. For this evaluation, we used the nine-grid space shuttle data set with a spatial rotation of ($-90°X$, $-10°Y$, $0°Z$), and scale factors that ranged from 1.0 to 5.16. A scale factor of 1.0 causes the long diagonal of the volume's bounding box to equal the width of the screen window. The window size is 500x500 pixels. We see that four-processor efficiency is about 80%.

We looked separately at the one-processor and four-processor times of different sections of the algorithm, including transformation of the points to screen space, creation of the Y-buckets, and generating scanlines. Times mentioned below are averaged over the four scale values reported in Figure 5.

Using a single processor, the average total CPU time to transform the shuttle vertex locations from world space to screen space was 1.25 seconds; four processors achieved a 3.8 speedup. The CPU time for creating Y-buckets averaged 17.8 seconds; four processors achieved a 3.5 speedup. The average CPU time for rendering the scanlines was 100 seconds. The speedup factor with four processors for this task was only 3.3, because of the critical section mentioned in Section 4.

From Figure 5, it is clear that, although 100% scalability is not achieved, there are significant speedups: 1.8, 2.5, and 3.3, respectively, for two, three, and four processors.

Memory use is another important factor in measuring the efficiency of an algorithm. Considering only the basic algorithm, without a hierarchy, the space shuttle grid itself takes 33.9 Mbytes, and the transformed vertex locations take 11.3 Mbytes. For a 500×500 window, the software frame buffer takes 5.0 Mbytes,

Rendering Method	Soft Scan	Incoh. Proj.	Coher. Proj.	Ray Cast
Shuttle scale 1	60.9			
Shuttle scale 5	202.1			
Blunt Fin scale 1	19.4	16.4		
Blunt Fin scale 5	51.4	16.5		
Fighter scale 1	24.6			
Fighter 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

Figure 6: Speed comparison of various renderers (CPU seconds) on an SGI Onyx with Reality Engine II graphics, using one 150-MHz processor. Only renderers relevant to a particular data type are shown. Projection methods utilize the Reality Engine graphics, while ray-casting and scan conversion do not.

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

Figure 7: Speedups on space shuttle using a hierarchy to avoid invisible regions. Times are CPU seconds on an SGI Onyx using one 150-MHz processor.

and the Y-bucket array takes 11.0 Mbytes. These sizes are view-independent, and total to 51 Mbytes.

The rest of the memory is used by the program to keep track of the polygons that it is rendering, so depends on the view. There is also some overhead for using multiple processors. For the scale 5.16, as reported in Figure 5, the total memory requirement ranged from 75 Mbytes for one processor to 87 Mbytes for four processors.

7.2 Comparison to Other Renderers

We have done speed comparisons of this new renderer against other direct volume renderers we have written. None of these other renderers have the generality to handle irregular multi-grids like the space-shuttle (which most renderers will not handle), and we have no other renderer that handles tetrahedral volumes. However, we can compare image quality and performance against the following: *Ray Casting* for rectilinear grids [23]; *Coherent Projection* (hardware Gouraud shading) for rectilinear grids [27]; and *Incoherent Projection* (hardware Gouraud shading) for single curvilinear grids [24].

Because of its extreme generality, our new software scan conversion algorithm cannot compete with methods designed to take advantage of the simplicity of regular grids. Figure 6 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.

We can provide some further general comments. First, software scan conversion generally produces pictures equivalent in image quality to ray casting, but much faster, because of the use of coherence. Also, our cell projection methods are not optimized for opaque data sets, while a ray caster processing front to back can easily halt when the pixel becomes opaque. Thus on a data set such as the head, the ray tracer can be competitive or even faster than other methods. In general, software scan conversion 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. Figure 8 shows three images of the space shuttle at different scales.

7.3 Performance with the Hierarchy

There are two aspects of the hierarchy. First, there is the temporal savings of discarding whole invisible subregions at one time, rather than examining their primitives individually. We found this a significant savings only when zoomed in on the volume considerably (see Figure 7), though with larger volumes, the gains may be more significant. For example, at a scale of 1, the hierarchy is slightly slower, whereas at a scale of 256, about twice as fast. As Figure 8 indicates, scales of 256 and larger are needed to view these volumes.

The second aspect is the use of a multi-resolution model to approximate the information. Figure 9 shows a comparison of the Lockheed fighter jet rendered using the scanline algorithm on every cell, and rendered using the hierarchy to draw an approximated version using hardware-assisted cell projection. The scanline image on the left is much better quality, but took over 3 minutes to render. The approximated version is suitable for exploring the volume and identifying regions of interest, and took less than three seconds to render.

8 Conclusions

The renderer described in this paper allows rendering of large multiple intersecting irregular and regular grids including polygonal meshes without the use of expensive graphics hardware. Factors such as screen size and scale can affect the time needed to render the volume. The renderer is parallelizable and measurements show that this can greatly reduce elapsed time without greatly increasing memory requirements. Use of a k-d tree makes the algorithm better able to handle very large data sets. Accurate depth calculation can be achieved by using the projection method described in Section 6.

Future work should investigate more sophisticated methods to use the multi-resolution model effectively, including seamless mixtures of polygons and approximated regions.

Acknowledgements

Funds for the support of this study have been 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.

References

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

[2] 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.

[3] Judy Challinger. Parallel volume rendering for curvilinear volumes. In *Proceedings of the Scalable High Performance Computing Conference*, pages 14–21. IEEE Computer Society Press, April 1992.

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

[5] Judy Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *IEEE Parallel Visualization Workshop*, October 1993.

[6] Paolo Cignoni, Leila De Floriani, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Multiresolution modeling and visualization of volume data based on simplicial complexes. In Arie Kaufman and Wolfgang Krueger, editors, *1994 Symposium on Volume Visualization*, Washington, D.C., October 1994. ACM.

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

[16] Kwan-Liu Ma. Parallel volume ray-casting for unstructured grid data on distributed memory architectures. In *1995 Parallel Rendering Symposium*, pages 23–30. ACM, 1995.

[17] S. G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, 1989.

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

[19] 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.

[20] David F. Rogers and J. Alan Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, New York, 2 edition, 1990.

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

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

[23] 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.

[24] 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.

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

[26] Jane Wilhelms, Paul Tarantino, and Allen Van Gelder. A scan-line algorithm for volume rendering of multiple curvilinear grids. Technical Report UCSC-CRL-95-57, Computer Sciences Board, University of California, Santa Cruz, November 1995.

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

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

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

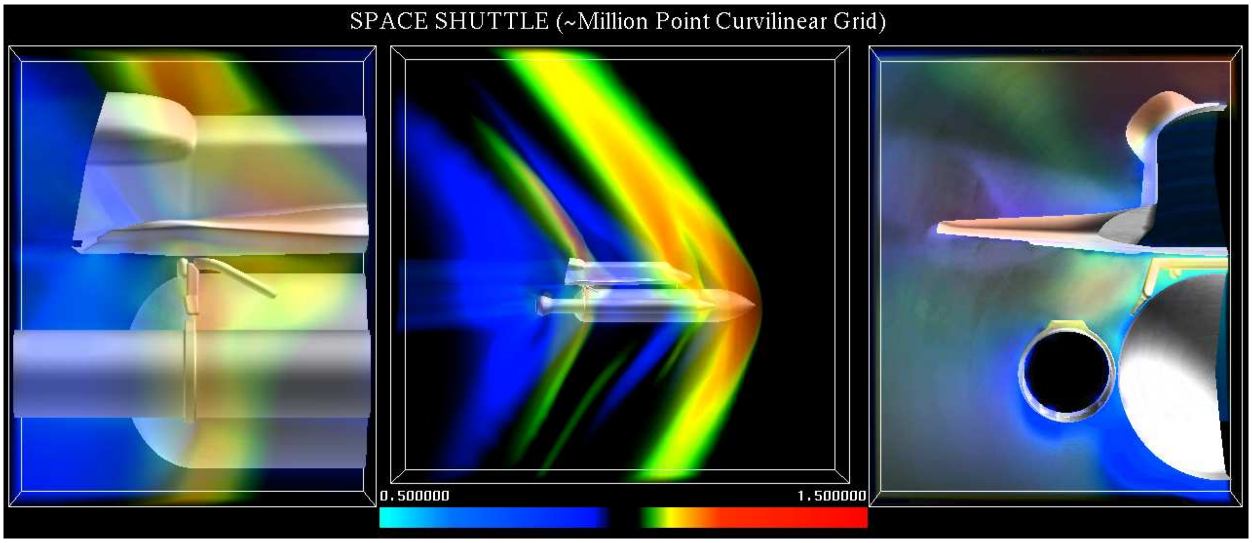

Figure 8: Three images of the space shuttle at different scales. Images took from 27 to 64 seconds on an SGI Onyx with four 150-MHz processors.

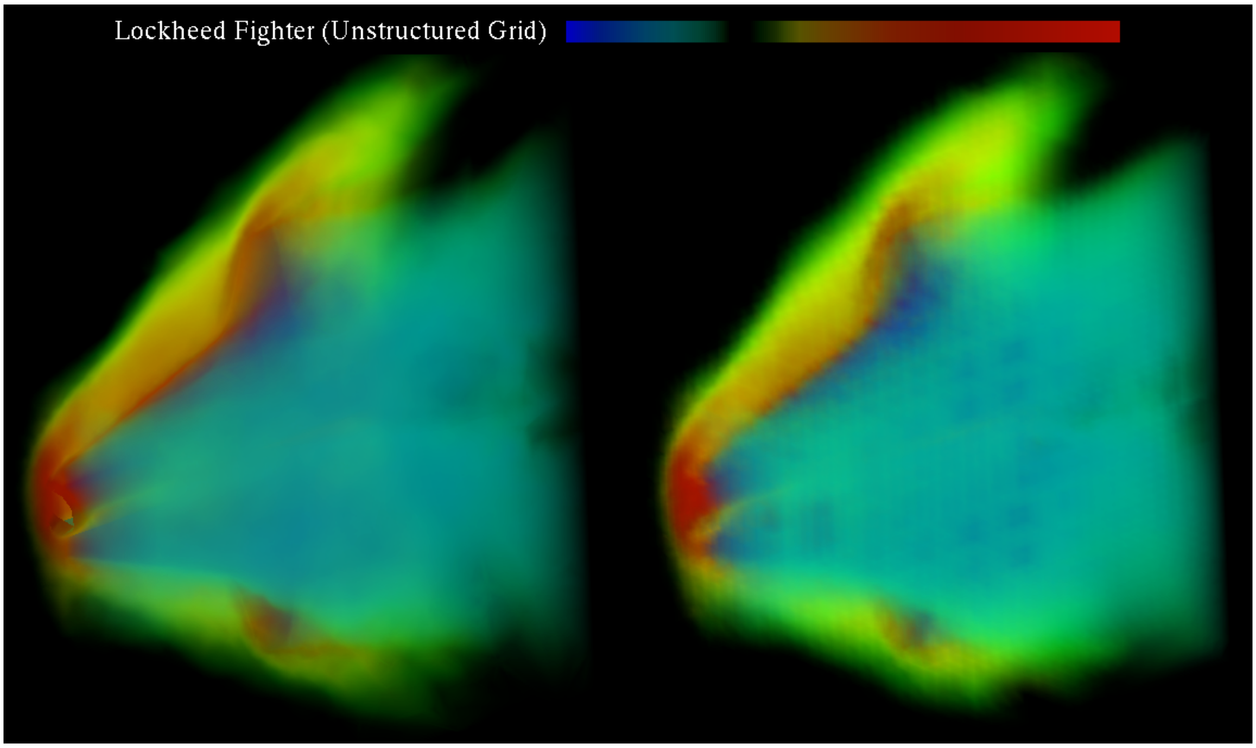

Figure 9: Software scan conversion of the Lockheed fighter (a tetrahedral data set) on the left (time about 3 minutes), and an approximate version rendered using hardware cell projection (Section 5.3) on the right (time about 3 seconds). Times are on a 150-Mhz SGI workstation.