Rapid Exploration of Curvilinear Grids Using Direct Volume Rendering

UCSC-CRL-93-02

Allen Van Gelder and Jane Wilhelms
Computer and Information Sciences
University of California, Santa Cruz 95064

March 16, 1993

Abstract

Fast techniques for direct volume rendering over curvilinear grids of hexahedral cells are developed. This type of 3D grid is common in computational fluid dynamics and finite element analysis. Four new projection methods are presented and compared with each other and with previous methods for tetrahedral grids and rectilinear grids. All four methods use polygon-rendering hardware for speed. A simplified algorithm for visibility ordering, which is based on a combination of breadth-first and depth-first searches, is described. A new *multi-pass blending* method is described that reduces visual artifacts that are introduced by linear interpolation in hardware where exponential interpolation is needed. Multi-pass blending is of equal interest to hardware-oriented projection methods used on rectilinear grids. Visualization tools that permit rapid data banding and cycling through transfer functions, as well as region restriction, are described.

1 Introduction

The visualization technique known as direct volume rendering is attractive because of its extreme flexibility, being able to map data values to color and opacity in any fashion. Direct volume rendering can be very useful for getting a general idea of volume contents, for scanning regions of interest, and for providing a context when combined with other methods (e.g., feature extraction). But it is hampered by computational cost. While some relatively fast methods have been introduced, they are subject to visual artifacts. Problems of speed and artifacts are exacerbated when volume-rendering non-rectilinear grids. However, when using direct volume rendering for a general perusal of volume information, improvements in speed may be worth even relatively significant artifacts.

Direct volume rendering is a visualization method for scalar sample data volumes where values within the volume are mapped to color and opacity and directly rendered by accumulating these color and opacity values to the screen pixels [DCH88, Lev88, UK88, Kru90, Wes90, MHC90, ST90, WVG91, Wil92b]. Any part of the volume may be visible in the final semi-transparent image. Direct volume rendering can be done by casting rays through pixels into the volume and traversing the rays [Lev88, UK88, Kru90], or by projecting sample regions or cells within the volume to the screen [UK88, Wes90, LH91, MHC90, ST90, WVG91]. Projection must be in front-to-back or back-to-front order for correct compositing if opacity values between zero and one occur.

If interpolation between sample points and integration in depth are not done accurately visual artifacts may occur [WVG91]. Further, because no geometric primitives such as polygonal isosurfaces are extracted, most or all of the work of direct volume rendering must be repeated if the viewpoint changes. For these reasons, direct volume rendering is an expensive technique to do well.

Significant speed-ups can be achieved by the use of coherence within the volume, by simplifying interpolation and integration, and by making use of graphics hardware (as in splatting and coherent projection)[UK88, MHC90, ST90, LH91, WVG91]. Speed-ups achieved on rectilinear grids, if not to realtime, are at least to the point where changing viewing parameters is comfortable to the user. We believe that the significant speed-up these methods provide are well worth the cost in image quality in many applications. The problem of achieving acceptably fast direct volume rendering is exacerbated if sample volumes are not on a regular rectilinear grid. Our goal in this investigation was to achieve some of these gains on curvilinear grids of hexahedral cells. Williams has studied related issues on tetrahedral grids [Wil92b, Wil92a].

In our application, computational fluid dynamics, curvilinear grids are common. A curvilinear grid can be thought of as a 3-dimensional rectilinear grid in computational space that is "warped" in physical space around regions of interest (e.g., aircraft wings). The grids present problems for direct volume rendering because cells vary greatly in size (e.g., neighbor distances in a commonly used grid vary by a factor of 10,000 [HB85]), cells may have irregular shapes, and they may be degenerate (e.g., multiple sample points in computational space may map to the same physical space location).

All methods described herein are designed for curvilinear grids that may not be convex as a volume, but whose cells are 6-sided convex polyhedra (hexahedra), possibly with some degeneracies in that some edges of the cell have zero length, and some faces have zero area.

Initial explorations convinced us that using ray-casting to directly render these volumes was unacceptably slow [RW92]. The main thrust of this paper is to explore more rapid, projection-based, methods. It is hard to compare these very different approaches exactly, because the ray-caster ran on a different machine and ray-casting is very sensitive to image size and volume orientation. However, we have found approximately, for similar images, that: (1) for rendering from a new viewpoint, our fastest method is a few hundred times faster than the ray-caster, and our slowest method is about 5 to 10 times faster; (2) for rendering from the same viewpoint, our fastest method is as above as viewpoint has no effect, and our slowest method is about

a hundred times faster. However, as window size increases, or the volume is zoomed, the cost of ray-casting may rise greatly while the cost of hardware-assisted projection will not be affected.

The faster methods may show significant artifacts in certain cases, however. Our new projection methods are described in Section 2.

To take into account opacity, a front-to-back ordering must be established, because cells (defined by eight corner sample points) in front may partially or totally obscure those behind. Because curvilinear grids can wrap around (see Figure 3), calculating this visibility ordering is nontrivial [MHC90, Wil92b]. Further, accumulating color and opacity values correctly in depth involves an exponential function [Kru90, MHC90, WVG91]. To approximate this quickly by a quadratic, we have developed a multi-pass blending method. These issues are discussed in Section 3.

Direct volume rendering is an intriguing and desirable method because of the amount of information that can be included in one image. But the fact that information may be layered and project to the same pixel makes rendering speed particularly important, because seeing the volume from different viewpoints clarifies it immensely. Generally, the mapping from data values to color and opacity is done using a simple transfer function. (For certain types of volumes, e.g., medical images, more complex methods such as material percentages may be desirable [DCH88].) We typically use an interactive transfer function editor to design the mapping from data to color and opacity, but as rendering became faster, we discovered that a significant amount of time was spent in trying to find desirable transfer functions [Ram90]. In Section 4 we discuss some new methods of rapidly designing and changing transfer functions for volume exploration.

The final area of research discussed in this paper is zeroing in on regions of interest and inverting the mapping from image to volume. We have designed a method whereby the user specifies a 3D region of interest, and the software renders only that region, indicating (if asked) the sample points and data values located in that region. In this way, the user can determine where interesting regions lie in the original grid. This topic is discussed in Section 5.

2 Rapid Direct Volume Rendering Approaches

We have implemented four projection methods for curvilinear grids. Previous work on projection methods for direct volume rendering is largely limited to a plethora of work on regular grids. Some work has been done on ray-casting irregular grids [Gar90, Use91, RW92]. A few researchers have explored projection methods on such grids. Max et al. describe a careful and general method that we felt was too slow for our needs [MHC90]. Williams takes the approach of breaking curvilinear grid cells into five tetrahedra each and projecting the tetrahedra [Wil92b, Wil92a]. While this is a quite rapid and reasonable approach, we hoped to achieve more speed and to avoid the explosion of primitives this involves, as well as some of the artifacts. At some point it would be worthwhile to do an in-depth comparison of this method with ours. Challinger implemented a kind of hybrid ray-caster/projection method where she sorted cells faces by scanline and pixel rather like a scan conversion algorithm for polygons, but then ray-cast faces present in a single pixel [WCA+90, Cha90]. This approach was also slower than we desired.

All four of our methods have some things in common:

- They convert cell projections to Gouraud-shaded polygons and use hardware for rendering.
- They use hardware compositing.
- They store cell information such as vertex locations, depth, and transfer function pointers in cell data structures. At a cost in space this provides better speed. Sorted lists of indices into these data structures are used by methods described in the following sections.

2.1 Depthless Cell Face Projection

Our first method is a very simple but admirably fast one: each data value is mapped to a color and opacity and the faces of each curvilinear grid cell is drawn as Gouraud-shaded polygons whose vertices have these mapped values. Data structures for this method record information for three adjoining faces of the cell, so each face is only drawn once. Usually this method is used with zero-opacity for maximum speed. The method takes about one second for 40,000 cells on a uniprocessor R3000-based Silicon Graphics VGX.

Two advantages to this method are that it is extremely fast and that it is trivial to implement. A feature that could be considered either advantageous or disadvantageous is that small cells contribute the same intensity as large cell, depth not being considered. On our grids, cell size is generally inversely proportional to interest, because volumes are finely gridded in areas of most interest. Some scientists may prefer this automatic weighting. Further, there is a problem with using hardware-compositing on these grids (or any grids with many tiny cells), because the typical intensity/opacity resolution is only eight bits per channel. Small cells may contribute well under 1/256 of the maximum possible intensity and, thus, never appear in the image at all. Using the depthless method, data is not ignored in this way, just improperly weighted.

A more serious problem is that noticeable visual artifacts appear from some angles because the distance between cell faces is not taken into account. These artifacts tend to delineate cell boundaries and probably would not be misinterpreted as data information.

2.2 Cell Face Projection with Depth

Our next method addresses the main problem with the former: that cell depth isn't taken into account. Here sample locations are first mapped to screen space. For each cell, vertices that lie on the convex hull are identified. If convex hull vertices are not coalesced, the depth through the cell at that point is zero and the vertex doesn't contribute color or opacity. If two vertices map to the same screen space location on the convex hull (e.g., in looking straight on at a regular cell), a depth is recorded for that vertex and the data value of the vertex is taken to be the average of the two vertices. This average is mapped to color and opacity. For interior vertices, the point at the same projected location on the opposite cell exterior is located and data values for that point found by interpolation. Depth and average color and opacity are again calculated and stored in the vertex. This information is calculated whenever the viewpoint is changed and is stored with the data structure for that cell. For rendering, each face is drawn once for each cell, and, thus, twice overall (except on the boundary). Scaling intensity can take care of intensity problems this may cause.

This method gives a more "realistic" (assuming some physical, colored medium being imaged) rendering, although there are some angles close to 90 degrees (for regular cells) for which it produces noticeable artifacts. It is much slower for new orientations than the depthless method, but takes opacity into consideration more accurately.

2.3 Silhouette Splatting

This method wasn't very successful and we mention it mainly to save others the work of trying it out. In this approach, we find the cell vertices that lie on the convex hull and connect them to form a polygon. Then we find the centroid of the cell and its depth and estimated value. By triangulation, the centroid is connected to the convex hull vertices, and these triangles sent to the screen. This is rather like the "splatting" method that is quite fast and successful for regular grids [Wes90, LH91]. Although, splatting is normally done as a region around a sample data point, not as a cell between sample data points.

Unfortunately, the images produced on our irregular grids were very blotchy from oblique angles. Further, this method was not much faster than the one described in the previous section and produced worse images, so we abandoned it. It is worth mentioning that, in splatting, the problem of blotchiness is partly removed

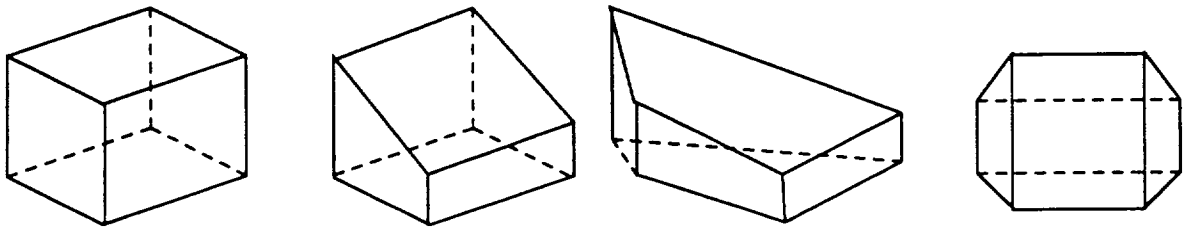

Figure 1: Typical projection of rectilinear cell and a few irregular hexahedron projections.

by making the splats overlap. This could be done on our grids at some additional expense by moving the convex hull away from the centroid for drawing. We didn't believe this would improve the image enough to warrant implementation, and it would involve more calculations.

2.4 Incoherent Projection

The method we have dubbed "incoherent projection" is the most careful of the projection methods presented, and also the most expensive. It builds upon the "coherent projection" technique for rectilinear grids [WVG91], and extends it to general convex hexahedra. The main idea is to render the 2-dimensional projection of each cell as an arrangement of polygons. For an orthogonal projection of a rectilinear cell there are 3 nondegenerate and 11 degenerate projection topologies. However, all cells in the volume fall into the same case from any given viewpoint. For irregular hexahedra the number of nondegenerate projection topologies is significantly higher (see examples in Figure 1), and the number of degenerate ones higher still. Moreover, different cells do not all fall into the same case, so a case analysis technique was not attractive to implement. Therefore, each cell is analyzed individually in screen space. A pleasant side-effect of this approach is that non-orthogonal perspective projections are no more difficult than orthogonal.

As shown in Figure 1, some vertices in the projection of a hexahedron do not correspond to vertices of the hexahedron, but are produced by intersection of edge projections. We call these *intersection vertices*. The first technical issue is the location of these intersection vertices. We used a sweep line algorithm which we now outline, assuming no degeneracies for the moment. The algorithm simultaneously finds the intersection vertices and the polygons that comprise the arrangement of the projection. (It also finds the convex hull as a by-product.) The hexahedron to be analyzed is given in screen space (x, y, z), so the objective is to find its projection on the x-y plane.

The algorithm maintains three data structures:

1. A priority queue of vertex *events*. The "minimum" of this event queue is the unprocessed vertex with minimum y-value.

2. An x-sorted list of *active edges*, where an edge is active if it goes from a processed vertex to an unprocessed vertex.

3. A *current boundary polygon* in the form of an edge list, which surrounds the processed vertices and edges.

Initially, the event queue contains the original hexahedron vertices sorted by y-value, and the active-edge list and current boundary are empty. The algorithm proceeds as follows.

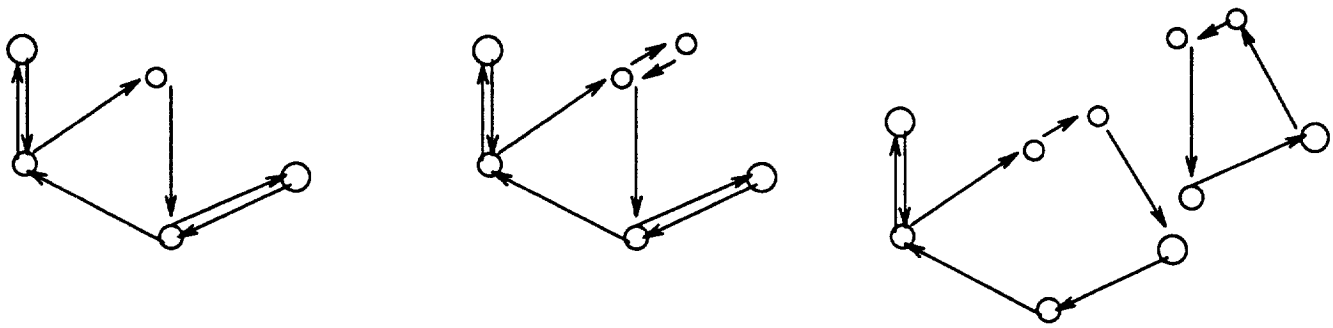

Figure 2: Current boundary updating and polygon creation steps. Left: a current boundary with one triangle previously removed; middle: one new edge pair to and from the new "event" node; right: the second edge pair formed a counterclockwise quadrilateral, which was spliced out.

```
while (eventQueue not empty)
    nextVertex = getMin(eventQueue);
    currentY = y-value of nextVertex;
    for (edge in ActiveEdges)
        currentX[edge] = x-value of edge when its y-value is currentY;
    if (ActiveEdges is still sorted on currentX[edge])
        remove edges that end at nextVertex from ActiveEdges,
                    call them remEdges;
        determine z-distance of nextVertex from face into which it projects;
        update currentBoundary with remEdges, creating some polygons of
                    the projection;   /* see text below */
        insert edges that begin at nextVertex into ActiveEdges in such a way
                    that all edges will be x-ordered if currentY is
                    increased slightly;
        deleteMin(eventQueue);
    else
        find (x,y) where two previously adjacent active edges intersected;
        build a new ''intersection vertex'', call it newVertex;
        chop off the two intersecting edges so they end at newVertex;
        the cut-off parts of the intersecting edges become new edges that
                    begin at newVertex;
        insert the new edges into ActiveEdges in correct x-order as described
                    above;
        insert(newVertex, eventQueue);   /* it will be the minimum */
```

This algorithm follows the standard pattern of sweep-line algorithms.

Updating the *current boundary* proceeds as suggested in Figure 2. Insert the first removed edge and its reversal into the current boundary edge list in such a way that the (nonconvex, nonsimple) polygon formed is planar (no edges cross); this forms a sort of needle. If there is a second removed edge, do likewise, but this completes a counterclockwise polygon, which is spliced out. (The last vertex of the cell has a third removed edge, which creates a second polygon to be spliced out.)

The above outline omits the details of handling "degeneracies". Projection degeneracies occur when any two projection vertices have the same x-value or y value. These can be removed by assuming a slight rotation of the screen space that does not change any nondegenerate topology. The details are tedious, but standard.

A more difficult and less standard degeneracy occurs when the original hexahedron is itself degenerate: if two points coincide in 3-space, no spatial transformation will separate them. Our solution was based on certain assumptions about what degeneracies could occur.

1. We assume no two adjacent edges have zero length;

2. We assume no two adjacent faces have zero area;

These assumptions leave a lot of flexibility: tetrahedra and pentahedra can be represented as degenerate hexahedra.

Suppose an edge of zero length is encountered. We want to perturb the hexahedron to give the edge some length while maintaining convexity. This requires finding a direction in which the two coinciding vertices can be "pulled apart". If the edge connects two faces with positive area, their planes intersect in a well-defined line that determines the required direction. The more difficult case is when the edge is adjacent to one face of zero area. Then we take advantage of the fact that, under the above assumptions, the diagonally opposite edge "in the same direction" cannot be degenerate. Form a triangle with this opposite edge as base and the coinciding vertices as apex, then slightly "pull apart" the coinciding vertices in the planes of the triangle and the face of positive area.

3 Opacity

Including opacity in direct volume rendering allows information to occlude that lying behind it. Sometimes this is desirable, but at other times it may not be necessary. If the purpose of the rendering is to get a general feel for information in the volume, zero-opacity rendering ensures that all information comes through. On the other hand, sometimes one would like an important feature or range of values to stand out, and opacity makes this possible. This is described in Section 3.1.

An additional issue becomes prominent in using opacity with hardware interpolation and blending. As mentioned in earlier papers, linear interpolation used in Gouraud-shading polygons produces an incorrect estimate of intensity and opacity across the projected polygon, even when the vertex values are calculated fairly accurately [MHC90, WVG91]. By using a method we call "multi-pass blending", we can approximate the correct intensity and opacity and still use the hardware polygon renderer. This is described in Section 3.2.

3.1 Visibility Ordering

A visibility ordering, which is an ordering on the cells such that no earlier cell occludes a later cell in screen space, is necessary to render cells with semi-transparency. Visibility ordering issues for tetrahedra were thoroughly explored by Williams [Wil92b], with attention to nonconvex volumes. This section outlines an implementation for curvilinear grids that is considerably simpler and is robust in practice. The main ideas are applicable to tetrahedral grids as well. Two issues concerning visibility ordering are: does one exist, and if so, how to find one. Although the theory is murky in the general case, in practice our method has never failed to find a visibility ordering. Williams reports similar practical experience.

When the volume is given as hexahedra there are significant advantages to keeping it in that form, rather than decomposing it into tetrahedra.

1. Adjacency and much other topology can be done by arithmetic on indices, without auxiliary data structures;

2. Decomposing into tetrahedra multiplies the number of cells by 5.

The other side of the coin is that tetrahedra are simpler to render.

The main idea that is well known for efficient visibility ordering is that of linear-time topological sorting [MHC90, Wil92b]. Recall that a topological sort of a directed acyclic graph is an ordering (or numbering) of its vertices such that there is no path from a smaller vertex to a larger one. This can be accomplished in linear time by a depth-first search and post-order numbering. For the visibility application the graph's vertices are cells and its directed edges given by the *immediately occludes* relation: cell A *immediately occludes* cell B if they share a face and A occludes B in screen space. For convex volumes, topological sort finds a visibility order if one exists and discovers a cycle otherwise [MHC90, Wil92b]. Nonconvex volumes occur often in practice, so it is important for an algorithm to work well on them, too. Here there is no definite theory known. Williams describes an heuristic for nonconvex volumes. We present an alternative that is considerably simpler, for connected, possibly nonconvex, volumes.

Our algorithm takes advantage of the fact that the underlying adjacency graph of the cells is undirected, where two cells are adjacent if they share a face. This undirected graph becomes directed by considering the orientation of the shared face in screen space, leading to the *immediately occludes* relation mentioned above. (The z component in screen space of the shared face normal determines which cell occludes the other.) We combine an undirected breadth-first search with directed depth-first searches.

For curvilinear grids, edges need not be represented explicitly, as they can be determined by arithmetic on cell indices. As it turns out, edge *directions* do not need to represented explicitly either!

The breadth-first search is implemented with a FIFO queue of cells. Initially, this FIFO queue contains one cell that has a vertex that is farthest from the viewpoint (minimum z in screen space), and all cells are unmarked.

```
nextNum = 0;
while (FIFOqueue not empty)
    nextCell = front(FIFOqueue);
    if (nextCell not marked)
        depthFirstsearch(nextCell,nextNum) yielding newNum;
        nextNum = newNum;
    dequeue(nextCell);
```

The depth-first search also needs to test edge directionality, and post "uphill" neighbors to the FIFO queue; otherwise it is quite standard.

```
depthFirstSearch(cell, nextNum)
    mark cell;
    for (neighbor adjacent to cell)
        if (neighbor not marked)
            if (neighbor immediately occludes cell)
                enqueue(neighbor, FIFOqueue);
            else
                depthFirstSearch(neighbor, nextNum) yielding newNum;
                nextNum = newNum;
        else if (neighbor is marked, but has no visNumber)
            Error - Cycle in vis order {has never happened in practice};
        else
            {neighbor has visNumber; do not visit}
    visNumber[cell] = nextNum;
    return  nextNum + 1;
```

Williams reports that about 60,000 tetrahedra per second can be ordered (SGI 4D/VGX). We found that

a comparable number of hexahedra per second were ordered by the above algorithm. Thus converting to tetrahedra would increase visibility-ordering cost by a factor of 5.

3.2 Multi-pass Blending

Linear interpolation is not always what is desired, but it is what the hardware offers. However, SGI workstations have a blend function setting that permits the "source" color to be multiplied by "one minus source alpha", and added to the background (cells already rendered). This permits some quadratic functions to be used for color interpolation, by multiplying two linear functions: color and alpha.

Assume a cell is filled with a semi-transparent light-emitting medium. When cell faces are planes, the depth δ of the cell varies linearly along any line, but the effective transmission of color varies as $(1 - e^{-\alpha\delta})$. This can be approximated between two vertices in the projection by a quadratic function of δ that is zero at vertices of 0 depth and gives the correct value of color at $\delta = \Delta$, the depth of the "thickest" vertex of the cell projection. The remaining parameter of the quadratic was chosen to minimize the squared error between the quadratic and the exponential function it is approximating, on the interval $[0, \Delta]$. Somewhat amazingly, this can be solved in closed form; some of the technical details are outlined in Appendix A. Many reasonable quadratics will give better interpolations than linear.

The above trick only provides nonlinear interpolation for the added contribution of the cell. The background needs to be reduced according to the cell's opacity, which again varies with depth according to an exponential decay. This time, the only trick is to use a linear function twice, with the blend function "one minus source alpha" applied to the background. The product of the two linear functions is a quadratic, and the alpha is rigged to give a reasonable quadratic approximation to the desired exponential decay.

Thus three passes are required in all. The first two apply opacity of the new cell to the background and the third adds the cell's own color. The extra time in computation and rendering are substantial (see Table 1.

4 Transfer Function Manipulation

For some time we have used an interactive transfer function editor to design the mapping between data values and color and opacity [Ram90]. While it is certainly more pleasant than designing mappings without it, we found it quite frustrating and time-consuming to try and guess which mappings bring out regions of interest.

Therefore, we developed a fast method of scanning the volume. The user interactively picks a data value and only data values within a user-controlled range around this center value are given a color. The user can designate the color and opacity of this banded region. The transfer function is a simple isoceles triangle centered at the designated center value. A slightly alternate approach is to use a pre-designed transfer function and define a transparent box around a designated data value. Only data within this box takes on color and opacity.

Rendering with such a single-band transfer function is extremely fast because only cells whose data values lie within the range of this triangle will have any visibility and need be rendered. Cells within this range can be rapidly found by using a supplemental data structure. This data structure is a two-dimensional array of size 256 X 256. Each location is a linked list of pointers into the data. (We assume 8-bit channels for color and opacity.) The minimum data value of a cell (scaled into the range 0–255) determines the row of the array and the maximum data determines the column of the array with which a cell is associated. (This could be more efficiently done but space was not a problem so we use this for simplicity.)

In drawing an image using the banded function, only those columns greater or equal to the minimum value of the band need be drawn; and within those columns, only rows less than or equal to the maximum

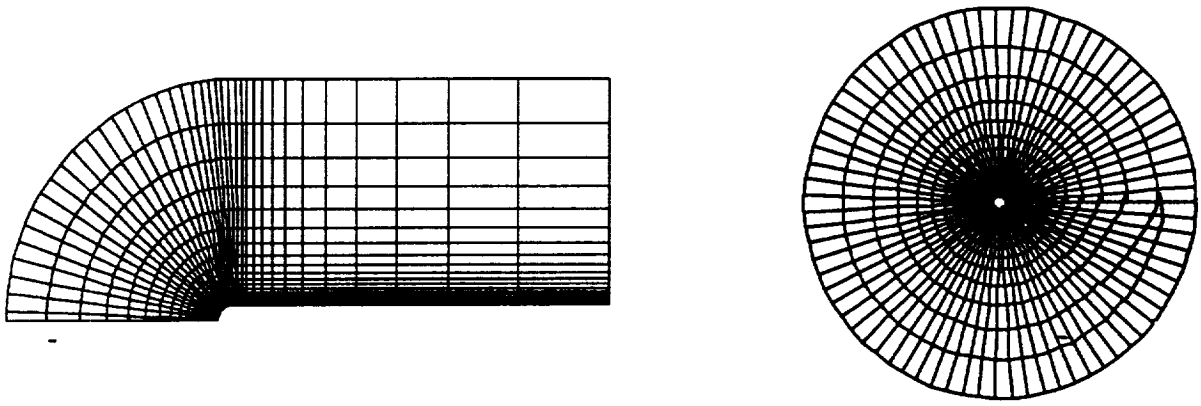

Figure 3: Single slices of the blunt fin (left) and post (right) curvilinear grids.

value of the band need be drawn. In many cases, this method takes a small fraction of the time it takes to render the whole volume.

A second feature of this method is that the band can easily and quickly be cycled, giving the effect of a moving fuzzy isosurface and helping to indicate the relation of neighboring data regions.

5 Restriction and Inverse Mapping

The final approach in our recent attempts to make direct volume rendering more useful was to implement a method of interactively restricting the parts of the volume rendered. At the suggestion of Arsi Vaziri of NAS/NASA-Ames, we invert the mapping for this restricted region to find the actual locations within the data that are being drawn.

Restriction is done by defining a simple bounding box whose location and size are defined interactively using sliders. Only cells whose origins lie within this box are drawn. Adding the option to draw cell origins as points helps clarify the relation of the volume rendering to the sample points.

If the restricted region is fairly small, it is practical to print the locations and values of the cells that lie within the box to the screen. This allows the user to determine the actual computational space location of features of interest.

6 Experimental Results

We explored these methods on two curvilinear grids. (The software works on a regular grid, but it is not optimized to take advantage of the greater simplicity of these grids.) The curvilinear grids tested were the "blunt fin" [HB85] and the "post" [RKK86], both from NASA-Ames Research Center. The curvilinear grid structure for these grids can be seen in Figure 3. The blunt fin is a 40x32x32 grid containing 40,960 samples, and the post is a 38x76x38 grid containing 109,744 samples.

Table 1 shows the rendering times for our four volume rendering methods using these two grids. Times are user and system CPU seconds on a Silicon Graphics uniprocessor VGX. For comparison, Williams reports times of around 15 seconds for volumes comparable in size to the blunt fin [Wil92a]. His images might be described as intermediate in quality. Coherent projection [WVG91] required about 4 to 7 seconds on a comparably sized *rectilinear* grid.

First we consider our four direct volume rendering methods. Cell face projection without depth is significantly faster than the others and is desirable for rapid scanning of the volume. We ignore the cost of

	Data Set	Visibility Sort	Making Data Structures	Single-Pass Rendering	Multi-pass Rendering
Faces Without Depth	Blunt Fin	0.64	–	1.14	–
	Post	1.65	–	3.02	–
Faces With Depth	Blunt Fin	0.64	6.63	2.94	11.72
	Post	1.65	21.76	10.99	39.76
Silhouette Splat	Blunt Fin	0.64	–	7.74	–
	Post	1.65	–	22.98	–
Incoherent Projection	Blunt Fin	0.64	37.56	3.52	12.05
	Post	1.65	104.48	13.70	37.38

Table 1: Rendering Times of Blunt Fin Data. (Time in CPU seconds.)

making the data structure for this method because it is done once as the data is read in and never changes despite orientation or mapping alternations. However, noticeable artifacts which delineate cell boundaries may occur from certain angles.

Cell face projection with depth uses the size and shape of cells more carefully, and requires rendering information that changes with orientation. We calculate this information (the "making data structures" cost in Table 1) and store it. Actually drawing the image from these structures takes much less time ("Rendering" columns in Table 1). This approach is desirable because the image can be scaled, rotated, transfer functions changed and intensity/opacity scaled, without recomputing the data structures.

Because we only pursued silhouette splatting briefly, this split between making data structures and rendering was never implemented for this method. We see from the costs that silhouette splatting is about 2/3 as expensive as cell face projection with depth. The quality of the images, however, is often somewhat worse.

As expected, our most careful and expensive method is more time-consuming. However, this method is much less likely to produce noticeable artifacts from any angle. Again we split the calculation into determining orientation-specific information and then rendering. The cost of re-rendering without orientation changes is not much worse than the method based on cell faces with depth, and does produce much better images.

The linear-time visibility sort used contributed only minimally to the cost of rendering. Multi-pass blending noticeably increased rendering time, by three or four times. However, the multi-pass method can produce smoother images, so it may be desirable when animation is not needed.

7 Conclusions

We discovered that projection methods do provide reasonable speed for volume rendering medium-sized curvilinear grids, far beyond what we could achieve with ray-tracing approaches. Rendering opacity with any degree of accuracy is the most costly function.

Acknowledgements

This research was supported in part by NSF PYI grant CCR-8958590, NSF New Technologies Program grant ASC-9102497, NSF Infrastructure grant CDA-9115268, and a NASA-Ames Research Center Cooperative Agreement Interchange No. NCC2-717.

References

[Cha90] Judith Challinger. Object-oriented rendering of volumetric and geometric primitives. Master's thesis, University of California, Santa Cruz, UCSC Computer and Information Sciences, Applied Sciences Building, Santa Cruz, CA 95064, 1990.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, July 1988.

[Gar90] Michael P. Garrity. Raytracing irregular volume data. *Computer Graphics*, 24(5):35–40, December 1990.

[HB85] 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.

[Kru90] Wolfgang Krueger. Volume rendering and data feature enhancement. *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, 24(5):21 - 26, 1990.

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

[LH91] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 25(4), July 1991.

[MHC90] 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.

[Ram90] Shankar Ramamoorthy. An interactive transfer function editor. Internal Technical Report, 1990.

[RKK86] S. E. Rogers, D. Kwak, and U. K. Kaul. A numerical study of three-dimensional incompressible flow around multiple posts, 1986. AIAA paper 86-0353, Reno, Nevada.

[RW92] Shankar Ramamoorthy and Jane Wilhelms. An analysis of approaches to ray-tracing curvilinear grids. Technical Report UCSC-CRL-92-07, UCSC, University of California, CIS Board, Santa Cruz, CA, January 1992.

[ST90] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, December 1990.

[UK88] Craig Upson and Michael Keeler. The v-buffer: Visible volume rendering. *Computer Graphics*, 22(4):59–64, July 1988.

[Use91] Sam Uselton. Volume rendering for computational flid dynamics: Initial results. Technical Report RNR-91-026, NAS-NASA Ames Research Center, Moffett Field, CA, 1991.

[WCA+90] Jane Wilhelms, Judy Challinger, Naim Alper, Shankar Ramamoorthy, and Arsi Vaziri. Direct volume rendering of curvilinear volumes. *Computer Graphics*, 24(5), December 1990.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–76, August 1990.

[Wil92a] Peter Williams. Interactive splatting of nonrectilinear volumes. In *Visualization '92*, pages 37–44. IEEE, October 1992.

[Wil92b] Peter Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[WVG91] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics (Proceedings ACM Siggraph)*, 25(4):275–284, 1991.

Appendix A Quadratic Fit to Exponential

This section sketches a minimum square-integral error fit of a quadratic function to a given exponential function, subject to boundary conditions. It is used to derive coefficients for the multi-pass blending method described in Section 3.2. The fit assumes that C, the color-per-unit-distance, and α, the opacity-per-unit-distance, are constant within the cell; that only the cell depth in z varies. The z-depth of the cell will be denoted simply by z; at the cell's thickest point $z = \Delta$.

Let $f(z)$ be the amount of light transmitted from the cell at a point of thickness z. The well-known formula is:

$$f(z) = \frac{C}{\alpha}\left(1 - e^{-\alpha z}\right)$$

Our goal is to approximate f on the range $[0, \Delta]$ by a quadratic function

$$g(z) = (c_0 + c_1 z)(a_0 + a_1 z)$$

Then using the *source-color* × *source-alpha* blend function in hardware, where source-color is computed as $(c_0 + c_1 z)$ and source-alpha is computed as $(a_0 + a_1 z)$, causes a close approximation of $g(z)$ to be rendered using Gouraud shading.

We impose end-point constraints $g(0) = f(0)$ and $g(\Delta) = f(\Delta)$. The first permits simplification, as it forces either c_0 or a_0 to be zero. We choose to make $a_0 = 0$. Then we can set $a_1 = 1/\Delta$ without loss of generality. Now

$$g(z) = (c_0 + c_1 z)\left(\frac{z}{\Delta}\right)$$

The second constraint will be incorporated with a LaGrange multiplier, λ. To minimize

$$\int_0^\Delta (f(z) - g(z))^2 dz - \lambda(c_0 + c_1\Delta)$$

we set the partial derivatives with respect to c_0 and c_1 to zero. This gives the constraints:

$$-2\int_0^\Delta (f(z) - g(z))\left(\frac{z}{\Delta}\right) dz - \lambda = 0$$

$$-2\int_0^\Delta (f(z) - g(z))\left(\frac{z^2}{\Delta}\right) dz - \lambda\Delta = 0$$

Performing the integrations leads to two simultaneous equations in three unknowns: c_0, c_1 and λ. A third equation is given by the right end-point constraint:

$$c_0 + c_1\Delta = f(\Delta)$$

Fortunately, the system is linear. To simplify the notation, several abbreviations will be used:

$$
\begin{aligned}
A &= \alpha\Delta \qquad \beta = \frac{C}{\alpha} \\
T &= (1 - e^{-\alpha\Delta}) \\
F_0 &= \frac{\beta}{\alpha}(A - T) && = \int_0^\Delta f(z)dz \\
F_1 &= \frac{\beta}{\alpha^2}\left(A + \tfrac{1}{2}A^2 - (1+A)T\right) && = \int_0^\Delta z f(z)dz \\
F_2 &= \frac{2\beta}{\alpha^3}\left(A + \tfrac{1}{2}A^2 + \tfrac{1}{6}A^3 - (1 + A + \tfrac{1}{2}A^2)T\right) && = \int_0^\Delta z^2 f(z)dz
\end{aligned}
$$

With the above notation, the solution is:

$$
\begin{aligned}
c_0 &= \frac{6}{\Delta^2}\left(5F_1 - 5\frac{F_2}{\Delta} - \tfrac{\beta}{4}\Delta^2 T\right) \\
c_1 &= \frac{10}{\Delta^3}\left(-3F_1 + 3\frac{F_2}{\Delta} + \tfrac{\beta}{4}\Delta^2 T\right) \\
\lambda &= \frac{1}{\beta\Delta}\left(3F_1 - 5\frac{F_2}{\Delta} + \tfrac{\beta}{4}\Delta^2 T\right)
\end{aligned}
$$

F_0, F_1, F_2 are well defined as $\alpha \to 0$, but care is needed to maintain numerical accuracy. The power series expansion $T = (A - \frac{1}{2}A^2 + \frac{1}{6}A^3 \ldots)$ exposes the low-order terms that cancel. For small values of A, we have

$$
\begin{aligned}
\left(F_1 - \frac{F_2}{\Delta}\right) &= \frac{\beta}{\alpha^2}\left(-2 + \tfrac{1}{6}A^2 + \left(1 + \frac{2}{A}\right)T\right) = \beta\Delta^2\left(\tfrac{1}{12}A - \tfrac{1}{40}A^2 + \tfrac{1}{180}A^3 - \tfrac{1}{1008}A^4 \ldots\right) \\
c_0 &= \left(A - \tfrac{1}{12}A^3 \ldots\right) \\
c_1 &= -\alpha\left(\tfrac{1}{2}A - \tfrac{1}{4}A^2 + \tfrac{25}{336}A^3 \ldots\right)
\end{aligned}
$$