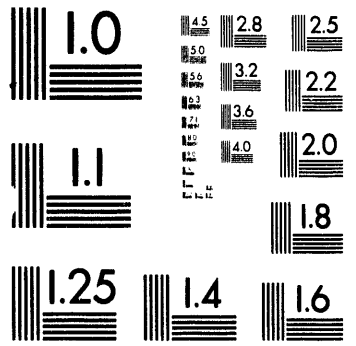

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0

1 of 1

LBL-33808
UC-405

ANALYTIC RENDERING OF CURVILINEAR VOLUME DATA

WES BETHEL

INFORMATION & COMPUTING SCIENCES DIVISION
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

MARCH 1993

This work was supported by the Director of the Scientific Computing Staff, Office of Energy Research, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

Analytic Rendering of Curvilinear Volume Data

Wes Bethel
Graphics Group
Computing Resources Department
Lawrence Berkeley Laboratory
Berkeley, California

ABSTRACT

A technique is presented for analytically rendering volume data from curvilinear grids. The method employs analytic techniques for manipulating and rendering curvilinear voxels. A curvilinear voxel is represented using a cubic triparametric solid formulation. The control points defining the cubic triparametric voxel are computed from the original curvilinear grid using the Catmull–Rom formulation, and subsequently rendered using a three–dimensional forward difference operator. The primary benefit from using such a representation is the fact the voxel shape and data values are C1–continuous across voxel boundaries. The issue of voxel opacity, both at the sub–voxel and super–voxel levels, is investigated. The use of both tricubic representation and rendering, along with our new approach to managing voxel opacity, results in images which are markedly different from those presented in previous work.

1. Introduction

A variety of scientific applications, such as computational fluid dynamics, are capable of producing volumetric data which lies on "warped" or curvilinear grids. A number of techniques have been recently developed to display data which comes from curvilinear grids. However, in each of these approaches, the curvilinear grid is approximated with a rectilinear grid.

[Wil90] describes two techniques for rendering curvilinear volume data. One method is to raytrace the curvilinear volume data "directly." However, the underlying structure used to approximate the curvilinear voxel is a set of twelve triangles. Each face of the curvilinear voxel is approximated by a triangle pair. The second technique is to resample the curvilinear volume into a rectilinear grid, then raytrace the rectilinear voxels.

[Shi90] presents a similar method for rendering curvilinear volume data by approximating each curvilinear voxel with a set of tetrahedra. The faces which bound the tetrahedra are subsequently rendered using hardware capable of displaying transparent triangles.

More recently, [Wil91] presents an algorithm for forward mapping from voxel space to image space, capitalizing on the notion of voxel footprints outlined in [Wes90], with particular attention to the interpolation of voxel opacity. Again, the underlying assumption is that the curvilinear grid has been approximated with a rectilinear grid.

With regard to volume rendering in general, a popular recent theme is to take advantage of local workstation rendering hardware. These techniques ([Wil91], [Shi90]) assume that a display list can be constructed which contains transparent polygons (usually triangles) which can then be rotated and Gouroud–shaded in hardware. The benefits of this architecture are the speed at which the volumes may be rendered, along with the fact that the same renderer is used for both geometric and volumetric primitives. A major drawback is that the display list can occupy a significant fraction of local memory, or more than is available on typical workstations for datasets which are modest in size. Additionally, results are inconsistent from hardware platform to hardware platform.

Another popular volume rendering technique involves the use of a voxel "footprint." The footprint is a template which is used in the compositing operation. For each voxel, the template is "filled in" (by the use of an appropriate interpolant) with pixel color and opacity values, and then composited into an accumulated image. This type of operation is of most benefit when the voxels are all the same size and shape, is is the case in rectilinear grids (with constant spacing along each of the component axes) using a parallel projection for viewing. If the size and shape constraints don't hold, then additional work is required to recompute the voxel footprints. In the worst case, the footprint must be re-evaluated for each and every voxel. The footprint architecture lends itself to efficient implementation through vectorization and parallelization.

In the methods we describe, it is assumed that the projected shape of each voxel is different. So, rather than construct a footprint for each voxel, each voxel is analytically rendered directly into an accumulating image. The underlying voxel renderer uses a three–dimensional forward difference operator, similar to that described in [Kau87]. Each tricubic voxel requires additional control points which are not present in the curvilinear grid.

To produce voxel shapes which "match" the curvilinear grid, a Catmull–Rom formulation is used to compute suitable internal voxel control points. The underlying data is interpolated using a tricubic formulation. The results are better than those resulting from a trilinear interpolation of the same data. We present examples which substantiate this claim.

2. Cubic Triparametric Solids

The cubic triparametric solid is a natural generalization of the cubic parametric curve and the cubic biparametric surface. The same properties which are attractive in parametric curves and surfaces are likewise attractive in solids. The cubic triparametric solid has the convex–hull property, which may used in clipping. The local control of shape property allows the specification of cubic triparametric voxels which are *C1* continuous in both the geometric and data domains.

The choice of a cubic function is based on the observation that the cubic is the "...lowest order parametric which can describe a nonplanar curve..." [Fol83]. Intuitively, this translates into "enough" degrees of freedom to allow the tricubic voxels to represent a large variety of deformed cubes. It is not the intent to approximate all such deformations, but enough to accommodate a majority of shapes resulting from typical curvilinear computational grids.

2.1 Deriving Triparametric Solids from Curvilinear Grids

In this section, we present the formulation used to compute the interior control points necessary to describe a cubic, triparametric Bezier solid.

Recall that a cubic parametric Bezier curve is described with four control points. Two of the points are at either end of the curve segment, while the other two do not lie on the curve, but define the direction and length of a vector tangent to the curve at each curve endpoint. The direction and magnitude of these vectors controls the shape of the curve. Those points which lie at the ends of the curve are here called *end points* while those control points which do not lie on the curve are called *interior points*. The set of both *end points* and *interior points* are referred to as *control points*.

In a cubic biparametric Bezier surface (two parametric dimensions), sixteen control points are required. Three parametric dimensions requires the use of sixty–four control points. The discussion below describes the computation of the interior points required, in addition to the eight vertices in the curvilinear grid (the end points), to describe a cubic triparametric Bezier solid (of sixty–four control points).

For the single parameter cubic Bezier curve, we are given points on the curve, which we use as curve endpoints for each curve segment, and we compute interior control points, two for each curve, which will result in a series of *C1* continuous curve segments. Each interior point adjacent to a given end point is a function of the direction and length of the chord joining the previous and next end points. From [Cat74] (see also [BBB87]), this is expressed analytically as:

$$D_i = \frac{1}{2}\beta[(P_{i+1} - P_i) + (P_i - P_{i-1})] \qquad (1)$$

or, equivalently,

$$D_i = \frac{1}{2}\beta(P_{i+1} - P_{i-1})$$

and is known as the Catmull–Rom formulation. From (1), a class of spline curves may be generated, where β controls the "tension" of the curve.

Hermite curves are generated from (1), but we are interested in Bezier curves. The appropriate transformation, from [Fol83], among others, is:

$$\begin{aligned} B_0 &= P_i & B_3 &= P_{i+1} \\ B_1 &= B_0 + \tfrac{1}{3}D_i & B_2 &= B_3 - \tfrac{1}{3}D_{i+1} \end{aligned} \qquad (2)$$

Figure 1 shows the relationship between the various components of (1) and (2). Boundary conditions with respect to the derivatives computed, for example, at P_{i+1} (and B_2), are not addressed here.

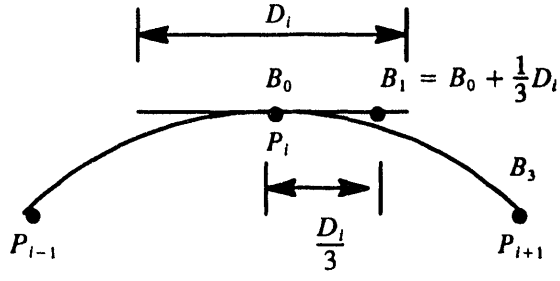


Figure 1

Extending this formula into two parametric dimensions is straightforward. If we consider the two parametric dimensions u and v, we can compute the interior points along a patch boundary directly from (1). Patches require the computation of additional interior points which do not lie on the patch boundary (Figure 2).

Since patches require the use of two parametric variables, we will compute the interior points which lie on the boundary using:

$$\frac{\partial f}{\partial u} = \frac{1}{6}\beta(P_{i+1,j} - P_{i-1,j})$$

along the u–axis and using:

$$\frac{\partial f}{\partial v} = \frac{1}{6}\beta\left(P_{i,j+1} - P_{i,j-1}\right)$$

along the v–axis.

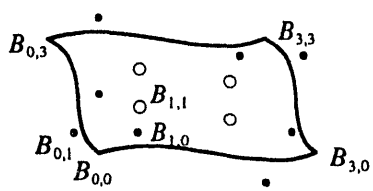
The interior patch points may be computed using a tensor product of (1):

$$\frac{\partial^2 f}{\partial u \partial v} = \frac{1}{36}\beta^2\left(P_{i+1,j+1} - P_{i-1,j+1} - P_{i+1,j-1} - P_{i-1,j-1}\right)$$

The formulation for a Bezier representation of a "non–boundary" control point (Figure 2) is:

$$B_{1,1} = P_{i,j} + \frac{\partial f}{\partial u} + \frac{\partial f}{\partial v} + \frac{\partial^2 f}{\partial u \partial v}$$

The tricubic non–boundary control points, the eight interior points which do not lie on any of the surface patches, are analogously computed using $\frac{\partial^3 f}{\partial u \partial v \partial w}$ as a linear combination of the eight values $P_{i+l,j+m,k+n}$ for $l, m, n = \pm 1$.



• Boundary Interior Points

o Non–Boundary Interior Points

Figure 2

2.2 Rendering Triparametric Solids

It is assumed that the reader is familiar with the rendering (i.e., evaluation) of both cubic parametric curves and cubic biparametric surfaces. It is also assumed that the reader is familiar with the notion of forward differencing. The discussion below continues from [Kau87].

[Kau87] gives the following formulation for the initial forward difference matrix used to evaluate a cubic biparametric surface:

$$\Delta f_{0,0} = E_\delta M G M^T E_\delta^T \qquad (3)$$

In this equation, the M matrix represents the Bezier basis matrix. G represents the control point array (sixteen points) used to specify the location and shape of cubic biparametric surface. The E matrix is a function of the number of steps to take in a given parametric dimensions when evaluating the parametric function, and is described in [Kau87], among others. The subscripts on the E matrices denote the step size for that particular parametric dimension.

[Kau87] did not provide the formulation to compute the corresponding $\Delta f_{0,0,0}$ which would then be used in the forward difference evaluation of the cubic triparametric Bezier solid. From [Mor85], we can conclude that:

$$\sum_{l=0}^{3}\sum_{k=0}^{3}\sum_{j=0}^{3}\sum_{i=0}^{3}\Delta f_{i,j,l} = B_{k,l}B_{i,j}G_{i,j,k}B_{i,j}^T \qquad (4)$$

It is important to observe that the G array consists of the sixty–four control points used to describe a tricubic solid, and that the Δf "matrix" in (4) is a three dimensional array (for each ordinate as well as the data). The B matrix in (4) is the matrix product of E and M in (3).

The reader is referred to [Kau87] for detailed information on how to proceed with the rendering, or evaluation, of the tricubic solid using this forward difference operator.

2.3 The Opacity Problem

As others have pointed out ([Lau91],[Wil91]), interpolation of the opacity in volumetric renderers merits scrutiny. Here we consider opacity at the sub–voxel level.

When rendering tricubic solids, "sheets" or "layers," each of which is a cubic biparametric surface, are successively composited, the total of which is the composited projection of the tricubic solid into image space. There arises a problem in dealing with the opacity at the layer level. If the original voxel has some opacity α_{voxel}, and each of the n layers of the voxel are composited using this α_{voxel}, the resulting opacity in the image is computed by:

$$O = 1 - e^{-\phi(\alpha_{voxel})n} \qquad (5)$$

which is a solution to the differential equation for performing front–to–back compositing [Por84]. This equation is:

$$O_n = (1 - O_{n-1}) * \alpha + O_{n-1} \qquad (6)$$

The $\phi(\alpha)$ function from (5) is computed so as to satisfy the boundary conditions of $n=0$ in (6) or $n=1$ in (5). This relationship is given by

$$\phi(\alpha) = -1 * \ln(1 - \alpha) \qquad (7)$$

We must recompute, in the spirit of [Lau91], the α_{layer} for the uniform–thickness layers as a function of the α_{voxel} along with the number of layers which are to be composited. After compositing each of the n layers, the resulting opacity in the image from the voxel will be α_{voxel} rather than the value obtained from (5) (which is the same as the result obtained after n iterations of (6)).

Given an initial α_{voxel} which will be composited using n layers, or sheets, the α_{layer} is computed:

$$\alpha_{layer} = 1 - \exp\left(\frac{-\phi(\alpha_{voxel})}{n}\right) \qquad (8)$$

These values may be computed prior to any rendering and stored in a table. The graph for this opacity composition function is shown in Figure 3. Note the similarity between this graph and that presented in [Lau91].

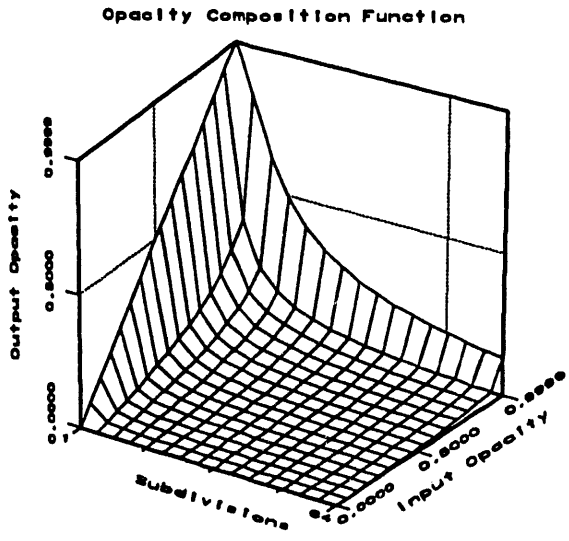


Figure 3

2.4 The Data to Opacity Transfer Function

The preceding discussion was focused on the issue of opacity at the sub–voxel level, and outlined the steps necessary to compute an appropriate opacity for a sub–voxel sheet or layer, given an initial voxel opacity and a number of constant–thickness sub–voxel sheets or layers. The discussion that follows focuses on how opacity is computed for each of the voxels, prior to rendering.

A convenient method for mapping data values to color and opacity is through a linear transfer function, the parameters of which are under user control. In this mapping, the user defines a lookup table of opacity and color values by some means. Then, a range of data values is linearly mapped into this table. The user specifies a minimum data value, which corresponds to the first entry in the table, and a maximum data value, which corresponds to the last entry in the table. We refer to this as a *linear transfer function*, and denote this transfer function as $L(d)$ where d represents the input data value.

$$\begin{aligned}
&\text{if}(data > data_{max})\\
&\quad index = table_size - 1;\\
&\textit{else}\ \text{if}(data < data_{min})\\
&\quad index = 0;\\
&\textit{else}\\
&\quad index = table_size * \frac{(data - data_{min})}{(data_{max} - data_{min})};
\end{aligned}$$

Linear Transfer Function

By not treating the entire set of volumetric data as a "unit–thickness" of some material, the linear transfer function can lead to unexpected or undesirable results. The behavior of (6) will tend to accumulate "too quickly" in the image, so that the overall volume appears too "opaque."

What we propose is a tool, under user control, which will take as input the results from $L(d)$, and apply a non–linear transformation which is a function of the "thickness" of the entire set of curvilinear voxels (such that "thickness" is viewpoint independent). It is important to realize that we are not trying to model "reality" using the alternate transfer function. Instead, we are providing a tool which overcomes some of the limits encountered when trying to model "reality."

We will use (8) from the previous section and $L(d)$ as the basis for a non–linear transfer function. In terms of providing the parametric values to (8), $L(d)$ will give an opacity, while some other function, say, $N(v)$, where v is the entire volumetric dataset, will give the "thickness" or "number of layers."

We could compute $N(v)$ as a function of the minimum and maximum extents along, say, the z–axis. This is not desirable since such a computation is sensitive to the orientation of the volume, and would have disastrous effects if a movie–loop is made of a rotating volume. We have had promising results using a function of the number of voxels to be rendered. In our examples, we use a value of $\gamma \log(m)$ where m is the minimum voxel count from each of the logical i,j,k dimensions, in computational (not physical) space, of the volume, and γ is under user control. Increasing γ has the effect of increasing the "overall" transparency of the volume. As γ goes to zero, the volume becomes more opaque. A good default value is 1.0, and is what was used in creating the example color images.

3. Implementation Issues

One major difficulty in implementing a forward difference operator is the choice of an appropriate step size, so that there are no voids in the image, and a minimal number of redundancies (a redundancy is the case where successive forward difference evaluations produce the same pixel–coordinate values, such as 32.17 on one it-

eration then 32.34 on the next — in this case, there is a redundancy at this pixel). There are a number of sources of information describing the computation of an appropriate step size ([Kau87],[Rap91]) to achieve this goal. We use a variation of the method presented in [Kau87] in computing parametric step sizes. We employ a first–order approximating technique that computes the extents in a two–dimensional projection for each of the parametric dimensions, resulting in different step sizes for each of the u,v and w axes. Such a technique favors faster execution time over what may be a more optimal step size.

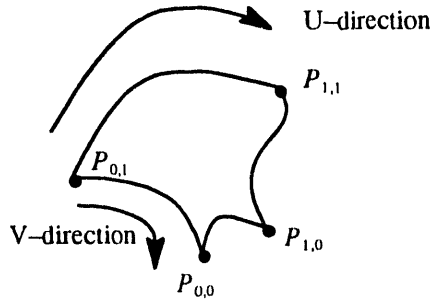


Figure 4

Another complication with using forward–differencing and non–adaptive step sizes (no matter how they are computed) arises when the sides of the voxels differ greatly in size. A two–dimensional example of this is shown in Figure 4. In this example, the same number of parametric steps is used along both the top and bottom edges. The result is that the bottom edge will be "over–composited," resulting in a voxel which appears to "glow" along the bottom.

A useful analogy is to think of a unit cube of jello. Some dye has been diffused throughout the jello (which corresponds to a homogeneous voxel). The result that we would like to have, if we consider Figure 4 above, is that there is the same concentration of dye throughout, no matter how thick or thin the jello along any particular line of sight. The jello will appear to be more transparent where thin, and more opaque where thick.

Using non–adaptive sampling will result in the dye being spread evenly through the jello, but in parametric space rather than physical space. The result is that the jello will appear more opaque where thin (since more parametric steps are being made per unit of physical space) than where the jello is thick.

There are at least three different ways around this problem. First is to use voxel subdivision rather than forward differencing. Another alternative is to employ adaptive sampling. These two approaches are discussed in more detail in a later section. Another alternative is to render the solids using a scan–converter, after "vertices" have been computed using the forward difference operator. In our implementation, we do not address this problem.

We provide in our implementation a "rendering throttle" whereby the user may adjust the step size once computed by some means. Increasing the step size decreases the amount of time required for rendering, but at the expense of "voids" in the resulting volume (i.e., there will not be 26–connectedness as described in [Kau87]). Going the other direction, the user may decrease the step size. This will increase the amount of rendering time required, but will also produce a lot of redundancies.

In our implementation, the curvilinear voxels are sorted front–to–back prior to processing. The cost of this sort is not included in the timings below for the various datasets, but is negligible (a second or two for each dataset using a Shellsort algorithm). The criterion used for the sort is the "minimum z" value in the voxel. There are cases where this type of sort will fail, but such pathological cases are not often encountered.

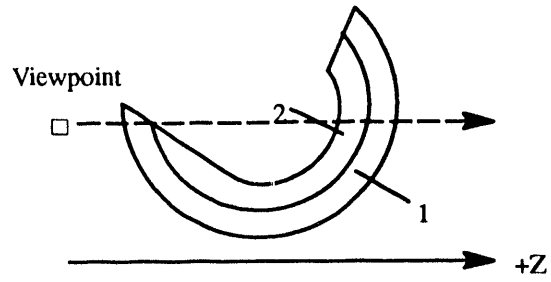


Figure 5

In Figure 5, the curvilinear voxel labelled "1" has a smaller Z value than voxel "2", so will be rendered first. The front–to–back compositing will fail (i.e., be incorrect) since voxel "2" is "mostly" "in front of" voxel "1". In this case, there is no good front–to–back ordering. The only plausible solution is to subdivide the voxels until there is a "good" front–to–back ordering. We discuss voxel subdivision in a later section.

After sorting, the voxels are then oriented (defining of the u and v parametric dimensions) so that "most" of the forward differencing occurs "parallel" to the image plane. The same pitfalls that apply to sorting voxels apply here, as well. The intent is to maximize the likelihood of detecting pixel collisions and to minimize forward difference calculations in depth.

```
1. construct voxel list.
2. sort voxels on minimum Z, orient voxels
for "best" scan converting.
3. for each voxel
        convert from eight point form to
          bezier solid form.
        render the voxel using either
            trillinear or tricubic forward
              differencing.
```

Rendering Algorithm Pseudocode

The memory consumption of this algorithm is linear with respect to the size of the input dataset. In addition to the constant–sized tables used (e.g., opacity composition), memory usage is as follows. For each input voxel (the area bounded by adjacent grid nodes), three integers and a float are required. The integers define the *(i,j,k)* indices into the grid for a particular voxel (one corner), while the float value holds the minimum *z* for that voxel. This scheme is required for representing the three dimensional grid of data as a one–dimensional list of voxels which may then be sorted. As each voxel is rendered one–at–a–time from the one–dimensional list, the memory required at the rendering stage is negligible (consists of four 64 element double precision arrays along with a few matrices).

4. Examples

The following examples illustrate, using three different datasets, a comparison between using trilinear and tricubic interpolation of both voxel shape and the underlying data as well as a comparison of the effects of using a linear or non–linear opacity transfer function. The cubic triparametric voxels have been computed using the Catmull–Rom formulation, so that the underlying data, as well as shape, is *C1* continuous across voxel boundaries.

The table below indicates the run times in CPU seconds on a Sun Microsystems 4/690MP (run on a single processor). All images were rendered into a 512 square pixel window.

The times for the trilinear images are in actuality much too high due to the fact that prior to rendering, internal voxel control points are computed using the Catmull–Rom method, even though these points are not used in the trilinear interpolation. This implementation choice was made for a valid cost comparison between the two types forward differencing schemes: trilinear and tricubic. The run times also point out there is a significant cost for computing the internal control points required by the tricubic representation.

Run Times in CPU Seconds

Dataset	Trilinear	Tricubic
Half–Torus	112	158
Blunt–Fin	1515	1890
Delta–Wing	22475	24624

The first example is a synthetic dataset. This dataset approximates a three–quarter torus in shape. This example shows the large–scale differences between using trilinear and tricubic interpolation of both shape and data, and shows the results of using the Catmull–Rom method of generating spline curves. The images which use the non–linear transfer function are quite dim. This is expected because this data set consists of only three voxels.

The second example sequence is computed using the "Blunt Fin" dataset [Hun84], provided courtesy of NASA Ames. The visualization uses the scalar density field from the dataset, and is composed of 40 by 32 by 32 curvilinear voxels, but cropped down to roughly 30 by 20 by 20.

The third sequence is computed using a CFD dataset depicting the vortical air flows over a Delta Wing at an extreme angle of attack [Eka90], again, courtesy of NASA Ames. The density scalar field is being visualized. The size of this dataset is 40 by 80 by 40 voxels.

For each of the example images, the following color and opacity scale is used. A hue ramp is used, with blue corresponding to low data values and red corresponding to high data values. Opacity is a linear ramp ranging from zero to one between low and high data values.

5. Conclusion

The images in the examples illustrate the the use of cubic triparametric interpolation of the data which produces images that appear to be more insightful since the tricubic renderings do a better job of representing the underlying data than when more crude approximatory techniques are used. There are small vortical structures present in these images of CFD datasets (aft of the turbulence boundary in the Blunt Fin data, in particular) when the cubic triparametric rendering is used which are not present, or are ill–defined, in the trilinear approximated renderings.

The tricubic renderings are more expensive than the trilinear renderings, as expected, but produce significantly better images than those presented in previous work in terms of the level of detail in the resulting images. It

should be noted that the run times required by our implementation are quite expensive, especially when compared to those achieved by others when using "splat" technology or hardware renderers. The tradeoff is one of image quality. As workstation performance continues to increase and prices drop, the significance of this drawback will decline.

The use of a non–linear opacity transfer function results in images which allow the regions of high opacity to show through the regions of low opacity much better than when a linear opacity (or no) transfer function is used. From a qualitative standpoint, these images are better than those presented in previous work in which no global opacity modification is performed.

The source code for this renderer is available in the form of an AVS module by anonymous ftp from the International AVS Center (avs.ncsc.org). It has been compiled and tested on the Sun Sparcstation platform.

6. Future Work

A significant amount of effort is required in terms of computing the internal control points which specify a cubic triparametric solid, along with the work required to set up the initial forward difference matrices. It would be worthwhile to implement a hierarchical version of this renderer. In the hierarchical version, "adjacent" cubic triparametric voxels would be represented with a single cubic triparametric voxel (the opposite of voxel subdivision). Such an implementation would be quite similar to the work presented in [Lau91], with the added complication that it is more effort to produce and traverse a space–subdividing tree for a curvilinear grid than to produce and traverse a similar tree for a rectilinear grid.

[Rap91] presents work which describes a method for adaptive forward differencing (a dynamic change in the parametric step size based upon some criteria) along with a hybrid algorithm which combines curve subdivision with forward differencing. Exploring the use of these techniques will be beneficial, particularly when the area of the projected voxel is large. However, as [Rap91] points out, doing either hybrid subdivision with forward differencing or adaptive forward differencing produces diminishing returns in efficiency when the size of the curves are small. When viewing large curvilinear volumetric datasets, this is typically the case.

It may be impossible to sort curvilinear voxels front–to–back so that there is a single rendering order (for the front–to–back compositing process). If this is the case, it must first be detected, then remedied via voxel subdivision. Note that the sorting problem exists regardless of the type of renderer being used; splat versus analytic versus polygonal/polyhedral approximation.

The notion of non–linear opacity transfer functions, based upon variables in addition to the volumetric scalar field, merit further investigation. We have presented a first approximation to a solution of this problem.

7. Acknowledgement

This work was supported by the U. S. Department of Energy, Energy Research Division under Contract No. DE–AC03–76SF00098.

8. References

[BBB87] Richard H. Bartels, John C. Beatty and Brian A. Barsky, *An Introduction to Splines for Use In Computer Graphics and Geometric Modeling*, Morgan–Kaufman, 1987.

[Cat74] E. E. Catmull and R. Rom, *A Class of Local Interpolating Splines*, Computer–Aided Geometric Design, R. E. Barnhill and R. F. Riesenfeld, eds., Academic Press, 1974.

[Eka90] J. A. Ekaterinaris and L. B. Schiff, *Vortical Flows over Delta Wings and Numerical Prediction of Vortex Breakdown*, AIAA Paper 90–0102, AIAA Aerospace Sciences Conference, Reno NV, January 1990.

[Fol83] J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addision–Wesley 1983.

[Gar90] Michael P. Garrity, *Raytracing Irregular Volume Data*, Computer Graphics (ACM Siggraph Proceedings of the San Diego Workshop on Volume Visualization) 24(5):35–40.

[Hun84] C. M. Hung and P. G. Buning, *Simulation of Blunt–Fin Induced Shock Wave and Turbulent Boundary Layer Separation*, AIAA Paper 84–0457, AIAA Aerospace Sciences Conference, Reno NV, January 1984.

[Kau87] Arie Kaufman, *Efficient Algorithms for 3D Scan–Conversion of Parametric Curves, Surfaces and Volumes*, Computer Graphics (ACM Siggraph 87 Proceedings) 21(4):171–179.

[Lau91] David Laur and Pat Hanrahan, *Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering*, Computer Graphics (ACM Siggraph 91 Proceedings) 25(4):285–288.

[Lev90] Marc LeVoy, *A Taxonomy of Volume Visualization Algorithms*, Volume Visualization Algorithms and Architectures, Siggraph 1990 Course Notes.

[Mor85] Michael E. Mortenson, *Geometric Modeling*, John Wiley & Sons, 1985.

[Por84] Thomas Porter and Tom Duff, *Compositing Digital Images,* Computer Graphics (ACM Siggraph 84 Proceedings) 18(3):253–260.

[Rap91] Ari Rappoport, *Rendering Curves and Surfaces with Hybrid Subdivision,* ACM Transactions on Graphics, (October 1991) 10(4):323–341.

[Shi90] Peter Shirley and Allan Tuchman, *A Polygonal Approximation to Direct Scalar Volume Rendering,* Computer Graphics (ACM Siggraph Proceedings of the San Diego Workshop on Volume Visualization) 24(5):63–70.

[Wes90] Lee Westover, *Footprint Evaluation for Volume Rendering,* Computer Graphics (ACM Siggraph 90 Proceedings) 24(4):367–376.

[Wil90] Jane Wilhelms, Judy Challinger, Naim Alper, Shankar Ramamoorthy, Arsi Vaziri, *Direct Rendering of Curvilinear Volumes,* Computer Graphics (ACM Siggraph Proceedings of the San Diego Workshop on Volume Visualization) 24(5):41–47.

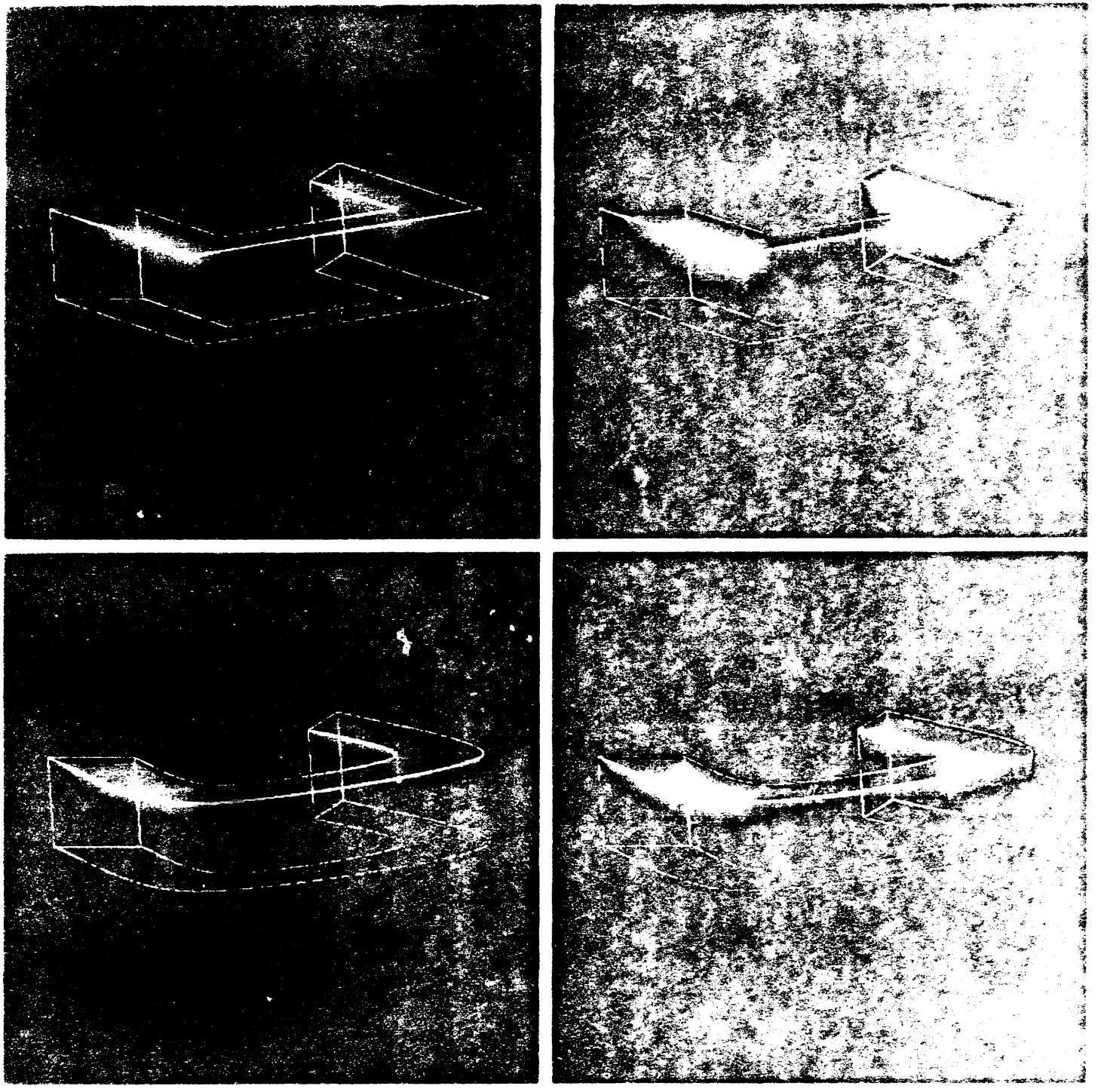
[Wil91] Jane Wilhelms, Allen Van Gelder, *A Coherent Projection Approach for Direct Volume Rendering,* Computer Graphics (ACM Siggraph 91 Proceedings) 25(4):275–284.

9. NOTE TO REVIEWERS

Some explanations regarding the artifacts in the images of CFD datasets are in order.

In the trilinear images, there are artifacts along voxel boundaries, particularly where a "plane" of voxel edges are viewed edge–on. Work is in progress to eliminate these by attentuating the opacity along voxel boundaries.

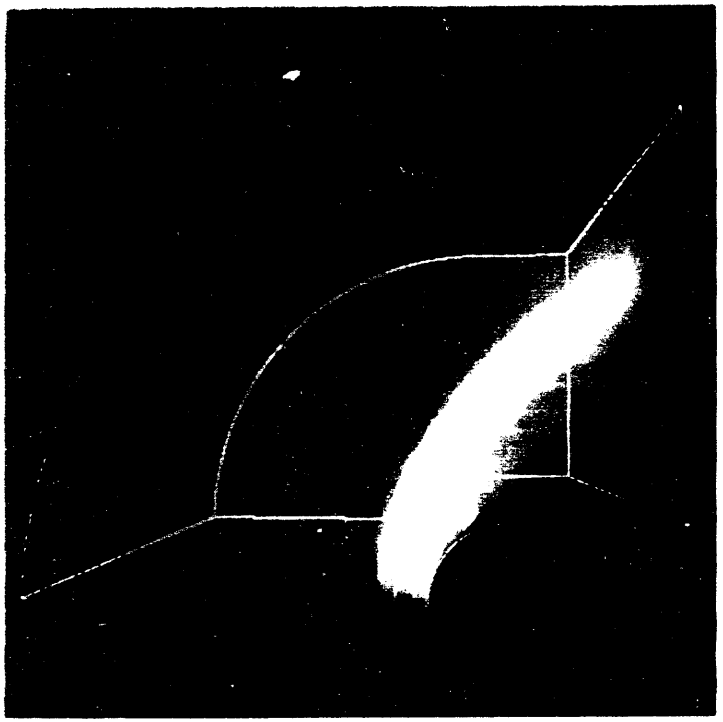
Similar artifacts are present in the tricubic images. There is an added complication of selecting a "good" Beta value for use in the Catmull–Rom computation of internal control points. The goal is to achieve a more or less uniform step size through physical space, as is the case in the trilinear renderings. The bottom line is that the tricubic images look worse than the trilinear images. It is my opinion that once the artifacts are corrected, the tricubic images will look better than the trilinear ones, even if only marginally so. This begs the question of whether or not this is a worthwhile technique for use with volume rendering.

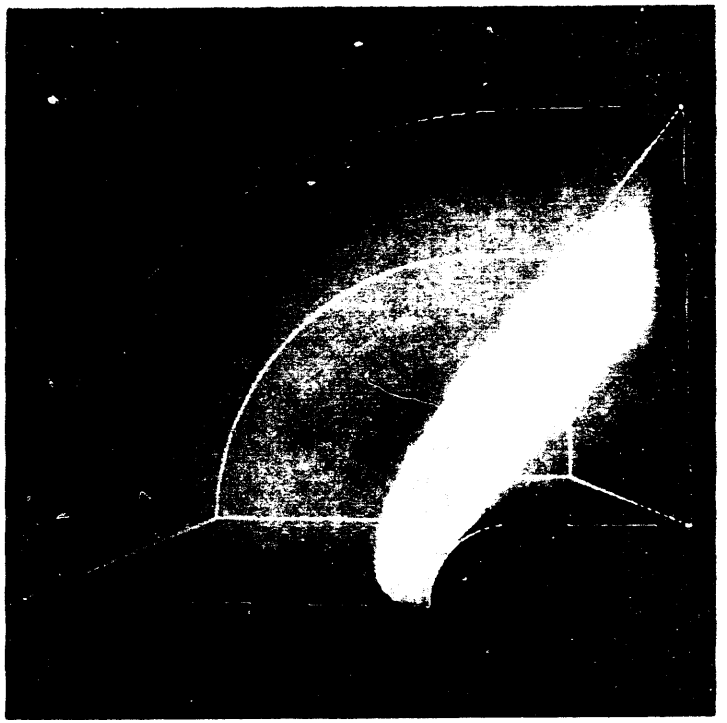

...g a correction
no a correction

log α correction

No α correction

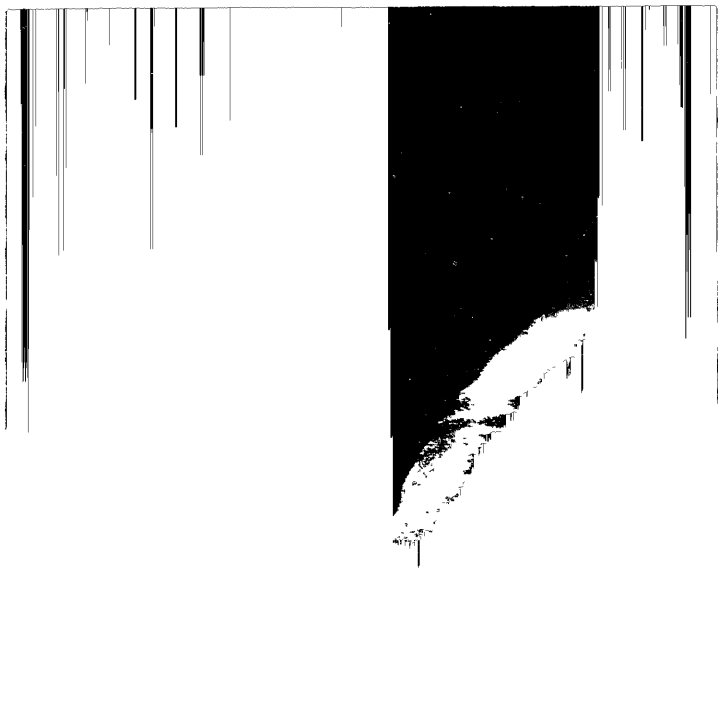
trilinear





Example 2

tricubic



DATE
FILMED
2/10/94
END