Nonpolygonal Isosurface Rendering for Large Volume Datasets

James W. Durkin
Program of Computer Graphics
Cornell University
Ithaca, NY 14853

John F. Hughes
Computer Science Department
Brown University
Providence, RI 02912

Abstract

Surface-based rendering techniques, particularly those that extract a polygonal approximation of an isosurface, are widely used in volume visualization. As dataset size increases though, the computational demands of these methods can overwhelm typically available computing resources. Recent work on accelerating such techniques has focused on preprocessing the volume data or postprocessing the extracted polygonization. Our new algorithm concentrates instead on streamlining the surface extraction process itself so as to accelerate the rendering of large volumes. The technique shortens the conventional isosurface visualization pipeline by eliminating the intermediate polygonization. We compute the contribution of the isosurface within a volume cell to the resulting image directly from a simplified numerical description of the cell/surface intersection. Our approach also reduces the work in the remaining stages of the visualization process. By quantizing the volume data, we exploit precomputed and cached data at key processing steps to improve rendering efficiency. The resulting implementation provides comparatively fast renderings with reasonable image quality.

1 Introduction

1.1 Background

Increasingly complex environments present an ongoing challenge to computer graphics. A dominant source of increased complexity in volume visualization is growth in data size. Early volume datasets typically ranged from 64^3 to 128^3 voxels, while many of today's volumes are reaching the 512^3 to 1024^3 voxel range. The introduction of higher resolution data acquisition devices and more complex simulations suggests this growth trend will continue.

The essential difficulty such growth poses is that the computational complexity of most volume rendering algorithms is $O(n^3)$ for a dataset of size $n \times n \times n$.[1] Thus doubling volume dimension, say from 256^3 to 512^3, yields an eightfold increase in computational cost. Even today's fastest workstations are, at best, barely keeping pace with the demands of rendering large volumes in reasonable amounts of time.

1.2 Prior work

Volume data has, by itself, no visible manifestation. Implicit in its visualization is the creation of an intermediate representation, some visible object or phenomenon, that can be rendered. Levoy [3] classifies volume rendering algorithms by the intermediate representation they employ. Among the classes are *surface-based* techniques, those using polygons or surface patches as the representation. Such techniques have proved popular due to their ease of use, range of applicability, and comparatively fast execution.

Surface-based techniques are characterized by the application of a surface detector to the data, followed by a fitting of geometric primitives to the detected surface, and the rendering of the resulting geometric representation. The techniques differ primarily in their choice of primitives and the scale at which they are defined. The primitives are typically fitted to an approximation of an isosurface of the continuous scalar field within cells of the volume.[2]

The best known of these techniques is the *Marching Cubes* algorithm [4]. Processing the volume cell by cell, the algorithm classifies each cell based on the value of its voxels relative to that of the isosurface being reconstructed. The classification yields a binary encoding that provides an index into a table describing the polygonal approximation of the isosurface within the cell. Polygon vertex positions are computed by interpolating voxel values, as specified by the indexed table entry. The generated polygons are transferred to a hardware or software polygon renderer for display. Gouraud shading is often used to achieve a smoother image. To do this, the algorithm approximates the volume gradient at voxel positions and interpolates these gradient vectors to produce normals at polygon vertices.

Wyvill et al. [12] present a very similar technique. They too classify cell voxels relative to isosurface value and calculate polygon vertex positions by voxel value interpolation. Their technique differs from Marching Cubes in that it uses an approximate value at the center of a cell face to select among alternate polygon configurations.

An alternative to isosurface polygonization is the point-based *Dividing Cubes* algorithm [1]. It subdivides volume cells into sub-cells with lattice spacing equal to the image grid spacing. Data values for sub-cell vertices are interpolated from the divided cell's vertex voxels. Sub-cells intersecting the surface are identified as those having values both above and below the isosurface value. For these sub-cells, a normal vector is interpolated from volume gradients as in Marching Cubes. This normal is used to shade the intersection point, considered to lie at the sub-cell center, which is then projected onto the image plane where the computed intensity is assigned to the appropriate pixel.

Recent work has focused on improving the performance of such techniques. Wilhelms and Van Gelder [11] use spa-

[1] The notable exception is *frequency-domain volume rendering* [5, 9], with a complexity of $O(n^2 \log n)$.

[2] We adopt the terminology of Wilhelms [10], referring to individual volume data points as *voxels*, and to a region of space bounded by a set of voxels (typically eight for regular volumes) as a *cell*.

tial data structures as a preprocess to reduce the work devoted to regions within the volume of little or no interest. Schroeder et al. [8] reduce the number of triangles required for the polygonal representation of objects through a post-process, making the extracted representation renderable on typical graphics hardware.

1.3 Motivation

As dataset size grows, the processing demands of conventional techniques can severely tax even the fastest workstations. Consider an example. The industrial CT dataset of the turbine blade in Figure 6 (also illustrated by Schroeder et al. [8]) contains 300 slices, each of size 512×512. The isosurface created from this data by Marching Cubes contains approximately 1.7 million triangles. Several stages in the algorithm are particularly expensive when processing such complex surfaces. The number of floating point operations required to calculate position, normal, and color information for the 5.1 million triangle vertices is enormous, even when reusing data at shared vertices. The amount of data transferred to the display system is also enormous: at 50 bytes per vertex, it is roughly 250 megabytes of information. Finally, rendering 1.7 million polygons is beyond the capabilities of all but the most advanced workstations.

The necessity of such expensive processing is an open question. In a typical image, say 512×512 pixels, these 1.7 million polygons are each rendered at sub-pixel size. One can well suggest that, given the small contribution of each polygon to the final image, the tremendous work involved in processing polygons for such a surface is probably excessive. An alternative to preprocessing the volume data or post-processing the polygonal surface is to concentrate instead on the surface extraction process itself. Our technique streamlines the isosurface visualization pipeline by eliminating the intermediate polygonization stage and reducing the work required at the remaining stages.

2 Foundations

Our algorithm is based, in part, on three observations about the surface-based rendering of *large* volumes:

Cell projections are small

In a 'complete' image of a large volume, a cell projects to about the size of a pixel. If we make the correspondence exact (i.e., volume inter-voxel spacing equals image inter-pixel spacing), an orthographic projection of an n^3 volume has a maximum image size of $\sqrt{3}\,n \times \sqrt{3}\,n$ pixels. For n between 512 and 1024, the image occupies from 60–240% of a typical workstation screen, suggesting that using such a correspondence produces sufficiently large images.

Isosurfaces are locally almost planar

The intersection of an isosurface with a cell is almost always well approximated by a plane. The function whose isosurface we are reconstructing was sampled in some way to generate the volume data. Unless the original function was band-limited before sampling, the data will contain aliasing. We therefore assume we are reconstructing the isosurface of the (unique) band-limited function f whose samples constitute our volume. Such band-limited functions are always C^∞. Sard's Theorem [6] guarantees that for almost every (in the measure-theoretic sense) isosurface value v, the isosurface $f^{-1}(v)$ contains no zeroes of the gradient of f. Thus almost every isosurface is locally smooth (by the implicit function theorem). Smooth surfaces can be approximated by their tangent planes, to an accuracy that depends on the surface curvature. The inaccuracy of our method is thus quantified by the local curvature of the isosurface. Near singular points, this can become arbitrarily large, but surface-based methods typically suffer from this: cells containing singular points are generally ambiguous.

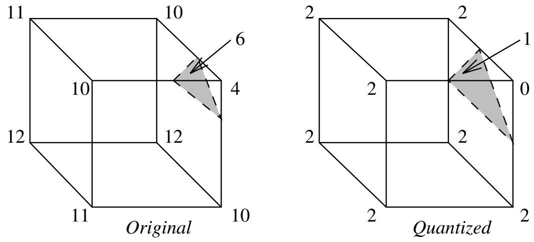

Figure 1: Geometric change resulting from data quantization. *Left*: original values and placement of planar approximation of the level 6 isosurface. *Right*: data quantized to a range of 3 centered on the original isolevel, and resulting shift in isosurface position.

Data can be quantized

To reduce rendering expense, we want to exploit precomputed and cached data whenever possible (e.g., using a precomputed approximation of the isosurface within a cell). To keep such data of reasonable size, we need to index it not by full-range volume data, but by a quantized representation of a cell's voxel values. Quantized data can produce a reasonably accurate isosurface approximation. Figure 1 shows that quantizing the data introduces some error; increasing the allowable range for the quantized representation reduces the error, but cannot eliminate it. Although we have not formally analyzed the error due to quantization, our empirical results suggest that the visual artifacts that result are acceptable. In any event, the quantization error is in the sub-cell placement of the isosurface, and hence (after projection) in the sub-pixel placement of the surface image. If the original data indicates the presence of surface within a cell, so too will the quantized data (if processed properly): isosurface topology is not altered, so no spurious surfaces or erroneous holes are introduced.

3 Algorithm overview

To render an image, we start with a volume dataset, an isolevel ℓ, a viewing direction, and lighting information.

As we precompute an approximation of the isosurface within a cell and index it by voxel values, our first step is to limit the data range within the volume. We choose a range r and quantize voxel values to an interval of length r that contains ℓ. To illustrate, let us assume that ℓ lies halfway along this interval (in practice this is not a requirement). The quantization may be as simple as clamping values to the range $\ell - r/2 \ldots \ell + r/2$, or may involve a more complicated scaling of values from a larger range, encompassing both r and ℓ, into the range $\ell - r/2 \ldots \ell + r/2$.

Next is the computation of the planar isosurface approximation data, the *isoplane table* (see Section 4.1). This table depends only on r and ℓ, and can therefore be precomputed. For each possible eight-tuple of values at the cell vertices, the table contains a description of the planar approximation of the isosurface within that cell, including the area of the plane within the cell and the plane normal.

The second step of the algorithm proper is initialization of the image, corresponding to a region on the projection plane. The inter-pixel spacing on this plane is made equal to the inter-voxel spacing in the volume. Thus the projection of a cell overlaps at most nine image pixels. The color and α channels of the image are initialized to zero.

The third step in the algorithm uses the isoplane table to compute a rendering of the isosurface. The volume is traversed from front to back and each cell is examined. The quantized voxel values at a cell's vertices are used as an index into the isoplane table, which contains the area and normal vector for the isosurface approximation within the cell.

If the area is zero (the isosurface does not intersect the cell), or if the dot product of the normal and the projection direction is negative (the surface is back-facing), the cell is ignored. Otherwise, the area is multiplied by this dot product to find the projected area of the cell's isosurface on the image plane. We compute the light reflected from the surface fragment (if not previously computed) and record it in a cached data structure called the *intensity table* (see Section 4.2), indexed by the cell's eight voxel values.

We accumulate into the image the light reflected from the surface fragment towards the image plane. Lacking precise geometric information describing the position of the surface within the cell, we assume that the projection of the surface fragment is evenly distributed across that of the entire cell onto the image plane. We can therefore clip the cell's projection against the pixels in the image plane to compute the fraction of the reflected light that should be composited into each of the nine pixels the cell projection may overlap. We avoid repeated clipping by precomputing a table describing the projection/pixel overlap for a representative set of the possible projection positions (see Section 4.3).

Using a modification of standard compositing (see Section 4.4) we accumulate values in the image until the α value for a pixel is 1.0, after which no more light is composited into the pixel.

4 Algorithm details

This algorithm exploits precomputed and cached data wherever possible, an approach that might well be termed *look-up tables everywhere*. We discuss the most important of these tables, and other implementation details, below.

4.1 Isoplane table

The precomputed planar approximation of the isosurface within volume cells depends on the quantized data range r and the isosurface level ℓ — more precisely, on the location of ℓ within an interval of length r. We therefore create a table whose entries are indexed by eight-tuples of values $[v_0,\ldots,v_7]$ (v_i in the range $0 \ldots r-1$) and associated with a level ℓ' between 0 and $r-1$.

Suppose the vertices of the unit cube are labeled by binary numbers so that, for example, vertex 6 has coordinates $[x,y,z] = [1,1,0]$ (as $6_{10} = 110_2$). Table entry $[v_0, v_1, \ldots]$ corresponds to a cube whose vertex 0 has value v_0, vertex 1 has value v_1, and so on. Given the values $v_0,\ldots,v_7$ at these corners (whose positions we denote $p_0,\ldots,p_7$), we approximate the isosurface by a plane determined by these values.

We do this by one of three methods, all variants of the same technique. Certain vertices of the cube are marked, and those vertices alone are used to find a least-squares best-fit plane. That is to say, we seek the function

$$f(x,y,z) = Ax + By + Cz + D$$

such that

$$\sum_{i \in M} (f(p_i) - v_i)^2, \text{ where } M = \{\textit{marked vertices}\}$$

is minimized. This is a straightforward least-squares problem in the unknowns A, B, C, and D.

The three methods differ in the choice of which cell vertices to mark. The first method marks all vertices, yielding a least-squares solution using all available data. We call this method *all-voxels*. In the second, we mark only cube edge endpoints with values on opposite sides of the isosurface level. The method, called *edge-crossings*, is analogous to Marching Cubes' identification of polygon vertex locations. In the third method, if any vertex has a value at either extreme of the quantized data range, and all three neighbor vertices (those connected to it by an edge) share the same value, then that vertex is unmarked; all others vertices are marked. This approach reduces the error in the approximation of the plane equation by eliminating data values that violate the assumption of linearity due to the limited range of the table. We term this technique *sans-clamped*. For any cell intersecting the isosurface, there are *at least* four marked voxels for any of the three methods, so the least-squares problem always has a unique solution.

Having found the function f above, we consider the plane $f(x,y,z) = \ell'$ to be our 'best linear approximation' to the isosurface. We clip this plane to the bounds of the cell, compute the area remaining, and record this in the table along with A, B, and C, which constitute the plane normal.

The table as described has r^8 entries. We store the area and normal data as floating-point numbers. At four bytes per value, we have sixteen bytes per table entry. Using ten megabytes as a rough limit for such precomputed data, the maximum allowable r is 5.

Fortunately, we can exploit the symmetry of the cube to give us a table compression scheme. A cube centered at the origin has many geometric symmetries: rotations about the x-, y- and z-axes, reflections in the xy-, yz-, and xz-planes, and combinations of these. For any eight-tuple of values labeling the cube's vertices, we consider the labelings derived from it by applying such symmetries to the cube as equivalent. That is, the area of the planar isosurface approximation for each is the same, and the surface normals are simple transformations of one another. We wish to map each such equivalence class to a single entry in our compressed isoplane table. Doing so requires that we produce just one

Data range	*Uncompressed* size	*Compressed* size	Compression factor
2	256	65	3.94
4	65,536	5,995	10.93
6	1,679,916	100,446	16.72
8	16,777,216	793,650	21.14
10	100,000,000	4,076,215	24.53

Table 1: Isoplane table information for r in the range 2–10. Shown are the number of entries for the uncompressed and compressed forms, and the compression factor for each r value.

entry for the equivalence class in generating the table, and that we can identify that entry and appropriately transform the stored data when accessing the table.

To identify a canonical element for the equivalence class, we permute the eight-tuple of cell vertex values $[v_0, \ldots, v_7]$ (using only permutations allowable under rotation and reflection) so that the value at p_0 is the smallest of the eight and the values at p_1, p_2, and p_4 satisfy the relation $v_1 \leq v_2 \leq v_4$. In generating the table, we limit ourselves to one entry per equivalence class by iterating over values that obey the stated restrictions. Using this scheme, the size of the isoplane table for a range r is

$$\sum_{a=0}^{r-1} \sum_{b=a}^{r-1} \sum_{c=b}^{r-1} \sum_{d=c}^{r-1} (r-a)^4,$$

an eighth-degree polynomial in r. In the limit, as $r \to \infty$, the size approaches $1/48\, r^8$. Table 1 gives size information for representative values of r.

In accessing the compressed table, the voxel values at cell vertices are permuted according to the above scheme, and the permuted tuple is used for table look-up. The permutation is a linear transformation of the cell, so we apply the inverse adjoint of this transformation in extracting the normal vector. The processing required to access data from the compressed table is approximately double that for the uncompressed table. Only with such compression though, is this precomputation technique feasible for larger values of r. In practice, we observe an actual increase in rendering time with compressed isoplane tables of only 1–10%. That this is lower than the raw increase in access time suggests, makes sense; isoplane table access is not the only step in rendering, nor do all cells intersect the isosurface (access time for ‘empty’ cells is the same for both table forms).

The incentive to use as large an r as possible is strong, as the quantized data range is a major factor in determining image quality. Figure 2 shows the result of using tables of varying r value. As expected, image quality improves with larger tables.[3] We typically keep precomputed tables for ranges of 2, 4, 6, and 8, with ℓ' at the range midpoint. The first two tables are usually kept uncompressed, and the latter two in compressed form.

4.2 Intensity table

In examining each cell, we look up its isosurface approximation in the isoplane table and apply the user-defined

[3]The test sphere, with its large, constant-curvature surface, highlights the effects of the quantization and linear approximation; lower r values frequently produce satisfactory images on real-world datasets.

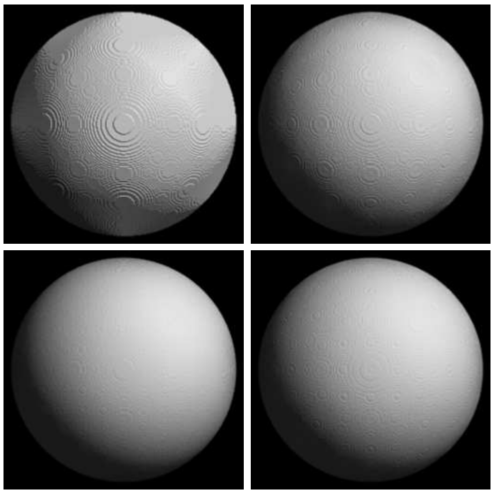

Figure 2: Test volume containing a ‘spherical’ isosurface, rendered using isoplane tables with different values of r. Clockwise from upper left, values for r are 2, 4, 6, and 8.

lighting definition to the data found there, computing the color of light reflected from the cell’s isosurface fragment. To avoid repetitive calculations, we cache this information in the intensity table, indexed by the cell’s voxel values. With an uncompressed isoplane table, we could add the information to that table at run-time, but for a compressed table, the need to permute the surface normal makes this impossible.

In principle, this table is of size r^8. To limit the size in practice, we allocate space for the intensity table in parts. We first allocate a table of pointers of size r^4, indexed by the cell’s first four voxel values. This part of the table is quite small, requiring only 4,096 pointers for $r = 8$. We allocate ‘data pages’ only when we must compute the light intensity for a given eight-tuple of values on that page. The data pages are of size r^4, and are indexed by the cell’s last four voxel values. Individual entries on the data pages are computed only as needed, and are reused by later cells with the same eight voxel values.

To minimize space requirements, we use only four bytes per entry, one byte each for the red, green, and blue color values plus a byte for flags used in table management. This follows the heuristic that eight bit color values are adequate in image compositing, but higher accuracy is required for the α-channel. We compute the α value elsewhere in the rendering process (using the projected area of the cell’s isosurface) and do not cache it in the intensity table.

This approach provides what we feel is the best tradeoff among access time, memory usage, and time spent computing intensity values. In practice, cell voxel values are not evenly distributed (they tend to cluster), so for typical r values (4 and above) we rarely allocate all data pages or compute a large percentage of the entries on allocated pages.

4.3 Cell projection table

As we restrict ourselves to parallel projections, the projections of any two cells onto the image plane are congruent. By precomputing a limited set of cell projections, we can approximate the projection of any volume cell and avoid the expense of repeated projection operations. We exploit this in computing the contribution to the image of a cell's projected isosurface fragment.

At the start of rendering, we project the vertices of a single cell onto the image plane and compute their convex hull, thus providing the polygonal projection of the whole cell. We calculate the position of the lower left corner of the projected polygon's bounding rectangle and designate it as the *projection marker* (see Figure 3). The polygon is translated so that the projection marker lies at the origin of the image plane: the polygon now lies on a 3×3 pixel grid whose lower left corner has coordinates $[0,0]$. We clip the polygon to each of the nine pixels and record the fraction of its area lying within each. If we offset the position of the polygon so that its marker lies at each of a discrete set of sub-pixel positions in the region $[0,1)\times[0,1)$, performing the clip and record operation each time, we then have a reasonable approximation of the cell's projected area over the 3×3 grid for any projection (and hence any cell) position. We use a 20×20 array of sub-pixel positions, which can be computed quickly and provides reasonable accuracy.

We can compute the position of the projection marker for each cell intersecting the isosurface, and use the fractional part of that position (modulo the discretization rate) as an index into the cell projection table. The corresponding table entry gives the fraction of the light contributed by the cell to be composited into each of the nine pixels its projection may overlap. Exploiting the congruency further, we need perform the complete projection marker calculation only for the first cell visited; the position of any other projection marker can be computed via a simple offset from the first cell's marker, based on the x, y, and z offset of the cell's position relative to that of the initial cell.

Distributing the light contributed by a surface fragment evenly across the cell's projection is an approximation. Some approximation is necessary to avoid the prohibitive expense of storing in the isoplane table a precise geometric description of the isosurface intersection. Using an even distribution is roughly equivalent to averaging over all positions within the cell of a fragment with that area and normal.

Clipping the cell's projection to pixel boundaries is equivalent to convolving the projection with a box filter and point sampling at pixel centers. It would be straightforward to extend the method to use arbitrary filters, and as the cell projection table is precomputed the cost would be negligible. Our experience, however, indicates that box filtering, in conjunction with the averaging operation just discussed, yields sufficiently good results.

4.4 Compositing

As described above, for each cell containing a front-facing piece of the isosurface we compute the amount of light reflected toward the image plane, and then distribute that light to the nine pixels onto which the cell projects. In essence we are compositing a series of 3×3 pixel sub-images into an accumulating larger image. Porter and Duff's image compositing algebra [7] assumes that the contents of two pixels being composited are randomly distributed. In our situation this assumption is almost always false, so we extend the algebra by a new operator to compensate.

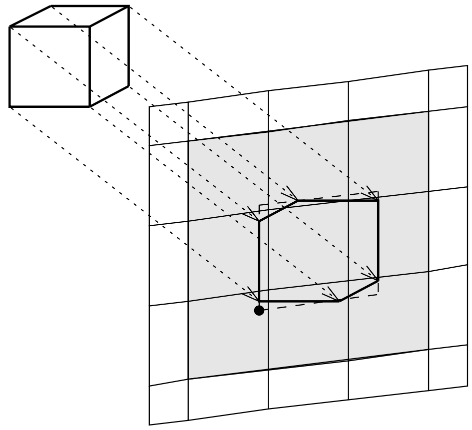

Figure 3: Projection of a cell onto the image plane. The fraction of the projected area lying within each of the nine pixels is stored in the cell projection table, which is indexed by the sub-pixel location of the *projection marker*.

Consider the case where adjacent cells both project onto the same image pixel and contain adjoining bits of the isosurface (see Figure 4). As the first cell's surface fragment is composited onto the target pixel, the pixel becomes partly covered; as the second cell's surface fragment is composited, the previously uncovered part of the pixel becomes completely covered. Let us call the first cell's projection the *foreground pixel A* and the second cell's projection the *background pixel B*, and assume that the α value for each is 0.5. Using the Porter-Duff **over** operator, where

$$F_A = 1.0 \text{ and } F_B = 1.0 - \alpha_A ,$$

the α value of the composited pixel (A **over** B) is

$$(F_A\times\alpha_A)+(F_B\times\alpha_B) = (1.0\times 0.5)+(0.5\times 0.5) = 0.75 .$$

The composited pixel has less coverage than expected and appears too dark.

In this case, the contents were not at all independently distributed. To address this situation we replace **over** with a new compositing operator, **add**. For **add** the values of F_A and F_B are

$$F_A = 1.0 \text{ and } F_B = \min(\,(1.0-\alpha_A)/\alpha_B,\, 1.0\,) .$$

The α value of the composited pixel (A **add** B) is now

$$(F_A\times\alpha_A)+(F_B\times\alpha_B) = (1.0\times 0.5)+(1.0\times 0.5) = 1.0 .$$

This gives the composited pixel the expected (full) coverage and the correct intensity.

The assumption that the contributions from multiple cells to a single pixel is *not* independent fails in some cases. If non-neighboring cells contribute to the same pixel, the Porter-Duff independence assumption is valid. We therefore expect slightly-too-bright edges when multiple silhouette edges project onto the same pixel. We have not observed this in practice, nor do we expect to: a pixel generically is the projection of the interior of a surface, and being on the projection of a silhouette is unusual. Being on

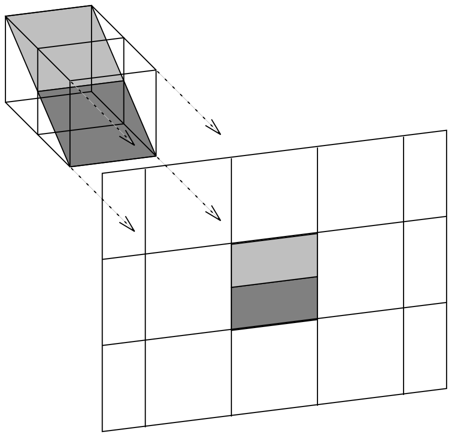

Figure 4: The isosurface projected from adjacent cells is not independently distributed within the image pixel.

the projection of *two* silhouettes is very unlikely, and should happen only at isolated pixels. The error is dual to that made by a Z-buffer renderer, in which a nearby polygon that partially covers a pixel can totally obscure a distant polygon that completely covers the pixel, yielding a too-dim image.

5 Algorithm extensions

The algorithm as described makes every effort to reduce the work at key steps in the rendering process. The characteristics of the volume data and its rendered image that make the acceleration possible also limit the types of data we can process and the kinds of image we can render. Limitations include quantizing the volume as a whole prior to rendering, restricting processing to regular-uniform data[4], and rendering only orthographic views.

These limitations are not, however, fundamental: each can be relaxed or eliminated, at some additional cost. We have extended the algorithm in various ways. The more important extensions implemented so far are described below. We give the user control in enabling these extensions, so that the visualization needs can dictate the tradeoff between flexibility and speed.

5.1 Cell-based quantization

Our extensive use of precomputed and cached data requires restricting the volume's data range, so that we can use the cell voxel values as a look-up table index. Normally this involves processing the volume once per isosurface level, prior to the start of rendering. This quantization requires visiting each voxel only once, and is comparatively fast.

Quantizing prior to rendering gives the best speed-up, but limits our accuracy in approximating the isosurface within a cell. The range of values across cells intersecting the isosurface is not always the same, and can be much less than the data range over which we quantize the volume as a whole. We can therefore achieve a more accurate approximation if we quantize voxels on a cell-by-cell basis rather than once per volume. Unfortunately, cell-by-cell quantization may give a voxel different quantized values depending on the cell being evaluated. To do this as a preprocess would require storing multiple values per voxel, which is prohibitively expensive for large volumes.

[4] *Regular* volumes are those with voxels arranged on a regular lattice, with constant spacing along each axis. *Uniform* refers to equal spacing along all axes.

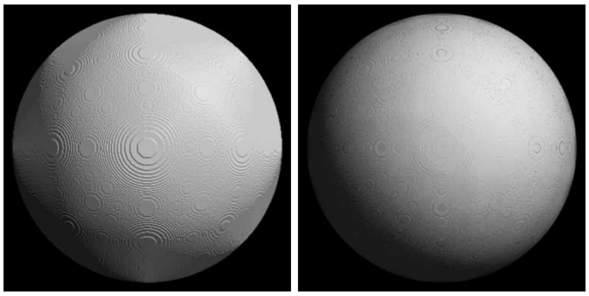

Figure 5: Test volume illustrating the potential difference between quantization strategies. *Left*: volume quantized as a whole prior to rendering. *Right*: volume processed during rendering using cell-based quantization.

We do, however, allow cell-based quantization. When specified, we skip the quantization preprocess and quantize during the rendering. For each cell that contains isosurface, we copy its voxel values to local storage and quantize using the data range over just that cell. We then use these quantized values as our table look-up index.

For certain data, this processing can produce a marked improvement in image quality, particularly surface smoothness. The dataset for Figure 5 is a 256^3 volume, with full-range eight bit data. The volume contains a 'spherical' isosurface, of radius 120, at a level of 127.5. The data in the vicinity of the surface ranges from 0 to 255, via a linear ramp, over a distance of eight voxels. Prequantizing the volume from the full range of 256 into a range of 8 (corresponding to the isoplane table range used for rendering) leaves us roughly one bit of accuracy per voxel value, for cells intersecting the surface. Using cell-based quantization allows us to achieve almost the full three bits allowed by the isoplane table. The resulting improvement in image quality is clear. While in practice, only a limited number of volumes actually exhibit such characteristics, the feature is always available for use as desired.

5.2 Non-uniform volumes

The isoplane table data is computed for cubical cells. If the voxel spacing in the volume is not uniform, so that the cells are parallelepipeds, we typically resample the volume to make it uniform. By modifying our algorithm slightly we can avoid this resampling. The transformation from a cube to a general parallelepiped is just a non-uniform scaling, so we can still use the precomputed isoplane table data. The area and normal information extracted from the table must, however, be transformed prior to its use.

Let us denote the inter-voxel spacing along the three axes of the non-uniform volume by S_x, S_y, and S_z. We index into the isoplane table using the quantized cell values, as described previously. Before using the isosurface fragment's area and normal though, we must first compute its 'scale-transformed' normal and area, N' and A' respectively.

We transform the normal vector N by the scale transformation, saving the result as the vector U to be used in computing N' and A'. U is calculated as

$$U = [(N_x/S_x), (N_y/S_y), (N_z/S_z)].$$

To compute N', we normalize U,

$$N' = U/\|U\|.$$

The transformed area A' is

$$\begin{aligned} A' &= A\left((N'_x S_y S_z)^2 + (N'_y S_x S_z)^2 + (N'_z S_x S_y)^2\right)^{1/2} \\ &= A\,(S_x S_y S_z)\ \|U\|. \end{aligned}$$

The factor $(S_x S_y S_z)$ can be computed once for the entire volume. The net expense of this non-uniform scaling is therefore three divides, one vector normalization, and two multiplications per cell.

We must also adjust the size of our pixel grid in the cell projection table, since the projection of a cell in the non-uniform volume may not always lie within a 3×3 pixel area (for a $1\times1\times2$ inter-voxel spacing, the projection may cover any of the pixels in a 4×4 region).

6 Results

We illustrate the algorithm's use with datasets from the medical and industrial communities. The data was processed and rendered on Hewlett-Packard Series 700 workstations, typically with machines having sufficient real memory to contain both the volume data and the precomputed/cached tables. No graphics hardware acceleration was used. All images were rendered with a $r = 8$, edge-crossings, compressed isoplane table.

Figure 6 shows a CT scan of a turbine blade. The original dataset comprised 300 slices, each of size 512×512, with a $1\times1\times2$ cell aspect ratio. To obtain cubical cells, we resampled the volume to size $512\times512\times600$, using trilinear interpolation. The image shows fine detail and smooth shading. The holes on the leading edge of the blade, the slots along its tail, and the serial number on the base are all clearly visible. The concave surface of the blade is smoothly shaded. The slightly rough texture on the surface of the base results from features in the original data (either noise or actual surface information) that are visible when viewing slices in isolation. The few minor artifacts visible along some flat surfaces are caused by the limited range of normals available due to data quantization. The image took 10 minutes, 9.6 seconds to render.

Figure 7 shows a human pelvis CT study. The original data contained 56 slices, each of size 256×256. To obtain cubical cells we again interpolated intermediate slices, this time using a cubic B-spline. The resulting $256\times256\times111$ volume was rendered in 18.1 seconds. Notice the smooth shading along bone surfaces and the fine detail visible on the spinal column.

Figure 8 is an image of an angiography dataset showing vasculature in the pelvic region. The dataset was 80 slices, with 256×256 samples each. We resampled the volume to $384\times384\times240$ for rendering, both to obtain cubical cells and provide a larger volume for testing. The original data

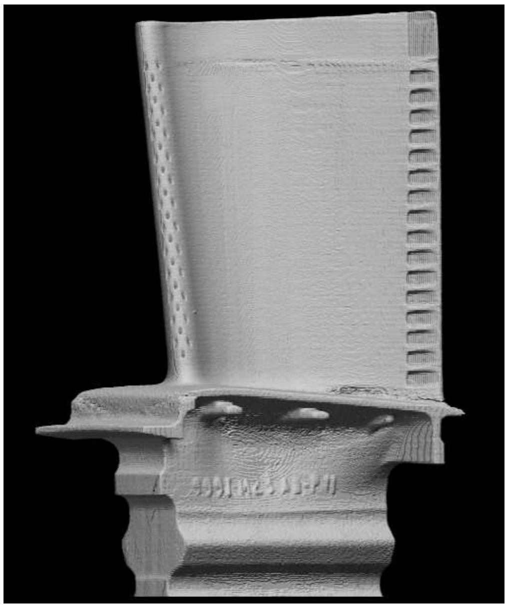

Figure 6: Industrial CT scan of a turbine blade [volume size: $512\times512\times600$ — isoplane table range: 8].

is fairly noisy, but the algorithm nonetheless extracts sufficient fine detail to interpret the key elements of the blood vessel structure. Although rendered with a $r = 8$ isoplane table, for data of this type smaller ranges (e.g., $r = 4$) produce nearly identical results. Rendering time for the image was 86.5 seconds.

In considering the rendering performance, note that the algorithm currently make no use of spatial data structures, such as octrees or bounding volumes, to accelerate the rendering. As such, every cell in the volume is examined in generating the surface representation. Clearly spatial data structures can reduce the effort expended on regions of the volume not intersecting the isosurface, yielding a corresponding improvement in performance. We chose to focus on reducing the work at cells containing isosurface, knowing that spatial acceleration techniques could be integrated later. Based on existing work using such approaches (e.g., Wilhelms and Van Gelder [11], Laur and Hanrahan [2]) and our own observations on the small percentage of cells that intersect a typical isosurface, we expect a substantial speed-up from using such a technique, quite likely a factor of ten or more.

7 Future work

After integrating a spatial acceleration mechanism, next on our list of future work items is an error analysis of the algorithm. The approximations made at various stages of the rendering process to minimize cost introduce some 'errors' into the resulting image. The main sources of error are data quantization, the approximation of the surface within a cell by a plane, and the projection of that surface approximation onto the image plane. It is fairly straightforward to quantify the error at each stage. The more difficult problem is relating the separate error metrics in a way relevant to actual image quality.

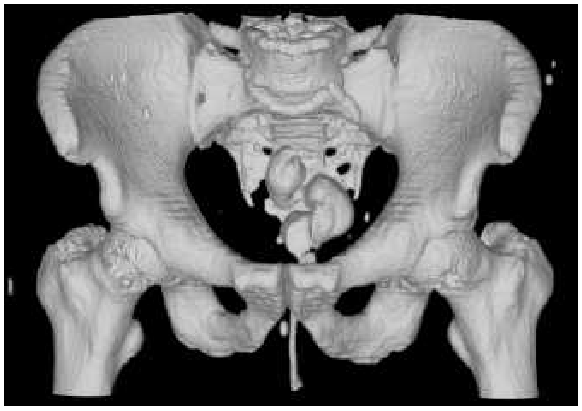

Figure 7: CT study of a human pelvis [volume size: $256 \times 256 \times 111$ — isoplane table range: 8].

Figure 8: Angiography dataset showing pelvic vasculature [volume size: $384 \times 384 \times 240$ — isoplane table range: 8].

We could use such error measures to improve accuracy, even if we cannot develop a single quality metric. For instance, we could store an error value based on the closeness of the planar surface approximation, for each isoplane table entry. We could use this value, alone or in combination with some quantization error measure, to decide whether the surface approximation for a given cell is sufficiently accurate. If not, we could subdivide the cell, interpolating interior values as necessary, and recursively apply our surface approximation method to the resulting sub-cells.

We are also interested in examining higher-order isosurface approximations that could be stored in a minimum of space. One motivation for this stems from what we term *ambiguous cells*. For a cell intersecting the isosurface, it is possible for the least-squares solution in Section 4.1 to have $A = B = C = 0$. Such ambiguous cases have reflectance functions that are not well approximated by the reflection function of a plane. An extended algorithm might enhance the isoplane table in these cases by storing a second-order approximation of the reflectance function.

Finally, we hope to develop a parallel or distributed implementation of the algorithm. The minimal processing in the main rendering loop, mostly table index generation and look-up table access, makes the algorithm comparatively simple to implement in a multiprocessor environment. Its object-order nature also provides for convenient data partitioning without replication across processing nodes. The only communications-intensive step would be compositing sub-images to produce a complete image of the isosurface.

8 Acknowledgements

This work was supported by the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-8920219). The first author also received support as an ONR-NDSEG fellow. We gratefully acknowledge the equipment grants from Hewlett-Packard Company and Sun Microsystems, on whose workstations this research was conducted. We also wish to thank those who provided us with volume data: Terry Yoo of the University of North Carolina at Chapel Hill, William Schroeder of General Electric Company Corporate Research and Development, and Dr. E. Kent Yucel of Massachusetts General Hospital – Harvard Medical School. Finally, special thanks to Richard Lobb for his encouragement during the writing and his assistance with volume resampling.

References

[1] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, May/June 1988.

[2] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (SIGGRAPH '91 Conference Proceedings)*, 25(4):285–288, July 1991.

[3] Marc Levoy. A taxonomy of volume visualization algorithms. *Introduction to Volume Visualization – SIGGRAPH '91 Course Notes*, July 1991.

[4] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (SIGGRAPH '87 Conference Proceedings)*, 21(4):163–169, July 1987.

[5] Tom Malzbender. Fourier volume rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[6] John W. Milnor. *Topology from the Differentiable Viewpoint*. The University Press of Virginia, Charlottesville, VA, 1965.

[7] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics (SIGGRAPH '84 Conference Proceedings)*, 18(3):253–259, July 1984.

[8] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Conference Proceedings)*, 26(2):65–70, July 1992.

[9] Takashi Totsuka and Marc Levoy. Frequency domain volume rendering. In *Proceedings: SIGGRAPH '93 Conference*, pages 271–278. ACM SIGGRAPH, August 1993.

[10] Jane Wilhelms. Decisions in volume rendering. *State of the Art in Volume Visualization – SIGGRAPH '91 Course Notes*, July 1991.

[11] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[12] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structures for soft objects. *The Visual Computer*, 2(4):227–234, April 1986.