Voxels as a Computational Representation of Geometry

Arie E. Kaufman

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

ari@cs.sunysb.edu

Abstract

This paper is a survey of volume visualization and volume rendering techniques. It focuses specifically on the use of voxel representation and volumetric techniques for geometric applications.

1. Introduction

Volume data are 3D entities that may have information inside them, might not consist of surfaces and edges, or might be too voluminous to be represented geometrically [30]. They are obtained by sampling, simulation, or modeling techniques. For example, a sequence of 2D slices obtained from Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) is 3D reconstructed into a volume model and visualized for diagnostic purposes or for planning of treatment or surgery. The same technology is often used with industrial CT for non-destructive inspection of composite materials or mechanical parts. Similarly, confocal microscopes produce data which is visualized to study the morphology of biological structures. In many computational fields, such as in computational fluid dynamics, the results of simulation typically running on a supercomputer are often visualized as volume data for analysis and verification. Recently, many traditional geometric computer graphics applications, such as CAD and simulation, have been exploiting the advantages of volume techniques for modeling, computation, and visualization. This paper focuses on and discusses this trend, called *volume graphics*.

Over the years many techniques have been developed to visualize 3D data. Since methods for displaying geometric primitives were already well-established, most of the early methods involve approximating a surface contained within the data using geometric primitives. When volumetric data are visualized using a surface rendering technique, a dimension of information is essentially lost. In response to this, volume rendering techniques were developed that attempt to capture the entire 3D data in a 2D image. Volume rendering convey more information than surface rendering images, but at the cost of increased algorithm complexity, and consequently increased rendering times. To improve interactivity in volume rendering, many optimization methods have been developed.

This paper begins with an introduction to volume rendering terminology. Section 3 covers briefly surface rendering techniques for volume data. Section 4 covers in details volume rendering techniques, including image-order, object-order, and hybrid techniques. Optimization methods for volume rendering are discussed in Section 5. Section 6 introduces global illumination of volumetric data, including volumetric ray tracing and volumetric radiosity. Volume graphics and volume based modeling are introduced in Section 7, while several volume modeling techniques, such as voxelization, texture mapping, amorphous phenomena, and block operations and constructive solid geometry are discussed in Sections 8-11. Surface graphics is contrasted with volume graphics in Section 12, and volume graphics features and weaknesses are discussed in Sections 13 and 14.

2. Volume Rendering Terminology

Volume rendering is the process of creating a 2D image directly from 3D volumetric data. The data is typically a set S of samples (x, y, z, v), representing the value v of some property of the data, at a 3D location (x, y, z). If the value is simply a 0 or a 1, with a value of 0 indicating background and a value of 1 indicating the object, then the data is referred to as binary data. The data may instead be multivalued, with the value representing some measurable property of the data, including, for example, color, density, heat or pressure. The value v may even be a vector, representing, for example, velocity at each location.

In general, the samples may be taken at purely random locations in space, but in this paper we will consider only sets which contain samples taken at regularly spaced intervals along three orthogonal axes. The spacing between samples along each axis is a constant, but there may be three different spacing constants for the three axes. Since the set of samples is defined on a regular grid, a 3D array (called also *volume buffer, cubic frame buffer, 3D raster*) is typically used to store the values, with the element location indicating position of the sample on the grid. For this reason, the set S will be referred to as the array of values $S(x, y, z)$, which is defined only at grid locations.

The array S only defines the value of some measured property of the data at discrete locations in space. A function $f(x, y, z)$ may be defined over R^3 in order to describe the value at any continuous location. The function $f(x, y, z) = S(x, y, z)$ if (x, y, z) is a grid location, otherwise $f(x, y, z)$ approximates the sample value at a location (x, y, z) by applying some interpolation function to S. There are many possible interpolation functions. The simplest interpolation function is known as zero-order interpolation, which is actually just a nearest-neighbor function. The value at any location in R^3 is simply the value of the closest sample to that location. With this interpolation method there is a region of constant value around each sample in S. Since the samples in S are regularly spaced, each region is of uniform size and shape. The region of constant value that surrounds each sample is known as a *voxel*, with each voxel having six faces, twelve edges, and eight corners.

Higher-order interpolation functions can also be used to define $f(x, y, z)$ between sample points. One common interpolation function is a piecewise function known as trilinear interpolation. With this interpolation function, the value is assumed to vary linearly along directions parallel to one of the major axes. Let the point P lie at location (x_p, y_p, z_p) within the regular hexahedron, known as a *cell*, defined by samples A through H. For simplicity, the distance between samples in all three directions is 1, with sample A at $(0, 0, 0)$ with a value of V_A, and sample H at $(1, 1, 1)$ with a value of V_H. The value V_P, according to trilinear interpolation, is then:

$$
\begin{aligned}
V_P = {} & V_A(1 - x_p)(1 - y_p)(1 - z_p) + V_E(1 - x_p)(1 - y_p)(z_p) + \\
& V_B(x_p)(1 - y_p)(1 - z_p) + V_F(x_p)(1 - y_p)(z_p) + \\
& V_C(1 - x_p)(y_p)(1 - z_p) + V_G(1 - x_p)(y_p)(z_p) + \\
& V_D(x_p)(y_p)(1 - z_p) + V_H(x_p)(y_p)(z_p)
\end{aligned}
\tag{1}
$$

In general, A will be at some location (x_A, y_A, z_A), and H will be at (x_H, y_H, z_H). In this case, x_p in Equation 1 would be replaced by $\frac{(x_p - x_A)}{(x_H - x_A)}$, with similar substitutions made for y_p and z_p.

3. Surface Rendering Techniques

Several surface rendering techniques have been developed which approximate a surface contained within volumetric data using geometric primitives, which can be rendered using conventional graphics hardware. A surface can be defined by applying a binary segmentation function $B(v)$ to the volumetric data. $B(v)$ evaluates to 1 if the value v is considered part of the object, and evaluates to 0 if the value v is part of the background. The surface is then the region where $B(v)$ changes from 0 to 1. If a zero-order interpolation function is being used, then the surface is simply the set of faces which are shared by voxels with differing values of $B(v)$. If a higher-order interpolation function is being used, then the surface passes between sample points according to the interpolation function.

For zero-order interpolation functions, the natural choice for a geometric primitive is the 3D rectangular cuboid, since the surface is a set of faces, and each face is a rectangle. An early algorithm for displaying human organs from computed tomograms [21] uses the square as the geometric primitive. This algorithm was developed before graphics hardware was generally available, so to simplify the projection calculation and decrease rendering times, the assumption is made that the sample spacing in all three directions is the same. A software Z-buffer algorithm is then used to project the shaded squares onto the image plane to create the final image.

With continuous interpolation functions, a surface, known as an *iso-valued surface* or an *iso-surface*, may be defined by a single value. Several methods for rendering iso-surfaces are briefly described here. The Marching Cubes algorithm [40] was developed to approximate an iso-valued surface with a triangle mesh. The algorithm breaks down the ways in which a surface can pass through a cell into 256 cases, reduces by symmetry to 15 topologies. For each of these 15 cases, a generic set of triangles representing the surface is stored in a look-up table. Each cell through which a surface passes maps to one of the 15 cases, with the actual triangle vertex locations being determined using linear interpolation on the cell vertices. A normal value is estimated for each triangle vertex, and standard graphics hardware can be utilized to project the triangles, resulting in a smooth shaded image of the iso-valued surface.

When rendering a sufficiently large data set with the Marching Cubes algorithm, many of the triangles map to a single pixel when projected onto the image plane. This fact led to the development of surface rendering algorithms that use 3D points as the geometric primitive. One such algorithm is Dividing Cubes [6], which subdivides each cell through which a surface passes into subcells. The number of divisions is selected such that the subcells project onto a single pixel on the image plane. Another algorithm which uses 3D points as the geometric primitive is the Trimmed Voxel Lists method [49]. Instead of subdividing, this method uses only one 3D point per visible surface cell, projecting that point on up to three pixels of the image plane to insure coverage in the image.

4. Volume Rendering Techniques

Representing a surface contained within a volumetric data set using geometric primitives can be useful in many applications, however there are several main drawbacks to this approach. First, geometric primitives can only approximate the surface contained within the original data. Adequate approximations may require an excessive amount of geometric primitives. Therefore, a trade-off must be made between accuracy and space requirements. Second, since only a surface representation is used, much of the information contained within the data is lost during the rendering process. For example, in

CT scanned data useful information is contained not only on the surfaces, but within the data as well. Also, amorphous phenomena, such as clouds, fog, and fire cannot be adequately represented using surfaces, and therefore must have a volumetric representation, and must be displayed using volume rendering techniques.

In the next subsections we will explore various volume rendering techniques. Although several of the methods described in these subsections render surfaces contained within volumetric data, these methods operate on the actual data samples, without the intermediate geometric primitive representations used by the algorithms in Section 3.

Volume rendering can be achieved using an *object-order* technique or an *image-order* technique. Object-order volume rendering techniques use a *forward mapping* scheme where the volume data is mapped onto the image plane. In image-order algorithms, a *backward mapping* scheme is used where rays are cast from each pixel in the image plane through the volume data to determine the final pixel value.

4.1. Object-Order Techniques

Object-order techniques involve mapping the data samples to the image plane. One way to accomplish a projection of a surface contained within the volume is to loop through the data samples, projecting each sample which is part of the object onto the image plane. An algorithm of this type was developed by Frieder, Gordon, and Reynolds [16]. For this algorithm, the data samples are binary voxels, with a value of 0 indicating background and a value of 1 indicating the object. Also, the data samples are on a grid with uniform spacing in all three directions.

If an image is produced by projecting all voxels with a value of 1 to the image plane in an arbitrary order, we are not guaranteed a correct image. If two voxels project to the same pixel on the image plane, the one that was projected later will prevail, even if it is farther from the image plane than the earlier projected voxel. This problem can be solved by traversing the data samples in a back-to-front order. For this algorithm, the strict definition of back-to-front can be relaxed to require that if two voxels project to the same pixel on the image plane, and first processed voxel must be farther away from the image plane than the second one. This can be accomplished by traversing the data plane-by-plane, and row-by-row inside each plane. If, for example, the image plane is placed such that the origin of the data is farthest from the image plane, then the data samples can be traversed in a back-to-front manner by considering the elements in the order of increasing x, y, and z. For arbitrary orientations of the data in relation to the image plane, we may traverse some axes in an increasing order, while others may be considered in a decreasing order. The traversal can be accomplished with three nested loops, indexing on x, y, and z. Although the relative orientations of the data and the image plane specify whether each axis should be traversed in an increasing or decreasing manner, the ordering of the axes in the traversal (and therefore the nesting of the loops) is arbitrary.

An alternative to back-to-front projection is a front-to-back method in which the voxels are traversed in the order of increasing distance from the image plane. Although a back-to-front method is easier to implement, a front-to-back method has the advantage that once a voxel is projected onto a pixel, we need not process any more voxels which project to the same pixel, since they would be hidden by the first voxel. Another advantage of front-to-back projection methods is that if the axis which is most parallel to the viewing direction is chosen to be the outermost loop of the data traversal, meaningful partial image results can be displayed to the user. Partial image results can be displayed to the user during a back-to-front method also, but the value of a pixel may change many times during image generation. With a front-to-back method, once a pixel value is set, its value remains unchanged. The partial image when only half the voxels have been traversed is often a close approximation of the final image. This allows the user to terminate the image generation if, for example, an incorrect view direction was selected.

Clipping planes orthogonal to the three major axes, and clipping planes parallel to the view plane are easy to implement using either a back-to-front or a front-to-back algorithm. For orthogonal clipping planes, we need only limit our traversal of the data to a smaller rectangular region within the full data set. To implement clipping planes parallel to the image plane, we would ignore data samples whose distance to the image plane is less than the distance between the cut plane and the image plane. This ability to explore the whole data set is a major difference between volume rendering techniques and the surface rendering techniques described in Section 3. In surface rendering techniques, we may need to modify our geometric primitive representation of the object in order to implement cut planes, which could be a time-consuming process. In this back-to-front method, cut planes can be achieved by simply modifying the bounds of our data traversal, and utilizing a condition when placing depth values in the image plane pixels.

For each voxel, we could store its distance to the image plane in the pixel to which it maps. If we traverse the data in the method described above, at the end of this traversal we will have a 2D array of depth values, called a Z-buffer, where the value at each pixel in the Z-buffer is the distance to the closest voxel. A 2D discrete shading technique can then be applied to the image, resulting in a shaded image suitable for display. The 2D discrete shading techniques described here take as input a 2D array of depth values, and produce as output a 2D image of intensity values. The simplest 2D discrete shading method is known as depth shading, or *depth-only shading* [22, 57], where the intensity value stored in each pixel of the output image is inversely proportional to the depth of the corresponding input pixel. This produces images where features far from the image plane appear dark, while close feature are bright. Since surface orientation is not considered in this shading method, most details such as surface discontinuities and object boundaries are lost.

A more accurately shaded image can be obtained by passing the 2D depth image to a gradient-shader [18] which can take into account the object surface orientation and the distance from the light at each pixel to produce a shaded image. This method evaluates the gradient at each (x, y) pixel location in the input image by

$$\nabla z = \left(\frac{\delta z}{\delta x}, \frac{\delta z}{\delta y}, 1 \right) \tag{2}$$

where $z = D(x, y)$ is the depth associated with pixel (x, y). The estimated gradient vector at each pixel is then used as a normal vector for shading purposes.

The value $\frac{\delta z}{\delta x}$ can be approximated using a backward difference $D(x, y) - D(x-1, y)$, a forward difference $D(x+1, y) - D(x, y)$, or a central difference $\frac{1}{2}(D(x+1, y) - D(x-1, y))$. Similar equations exist for approximating $\frac{\delta z}{\delta y}$. In general, the central difference is a better approximation of the derivative, but along object edges where, for example, pixels (x, y) and $(x+1, y)$ belong to two different objects, a backward difference would provide a better approximation. A context sensitive normal estimation method [67] was developed to provide more accurate normal estimations by detecting image discontinuities. In this method, two pixels are considered to be in the same "context" if their depth values, and the first derivative of the depth at these locations do not greatly differ. Similar depth values indicate C^0 continuity, and similar first derivatives indicate C^1 continuity. The gradient vector at some pixel p is then estimated by considering only those pixels which lie within a user-defined neighborhood, and belong to the same context as p. This ensures that sharp object edges, and slope changes are not smoothed out in the final image.

The previous rendering methods consider only binary data samples where a value of 1 indicates the object and a value of 0 indicates the background. Many forms of data acquisition (e.g., CT) produce data samples with 8, 12, or even more bits of data per sample. If we consider these data samples to represent the values at some sample points, and let the value vary according to some convolution applied to the data samples which will reconstruct the original 3D signal, we then have a scalar field which approximates the original 3D signal.

One way to reconstruct the original signal is, as described previously, to define a function $f(x, y, z)$ which determines the value at any location in space based on an interpolation function applied to the nearest data samples. This is the technique typically employed by backward-mapping (image-order) algorithms. In forward mapping algorithms, the original signal is reconstructed by spreading the value at a data sample into space. Westover describes a splatting algorithm [62] for object-ordered volume rendering in which the value of the data samples represents a density. Each data sample $s = (x_s, y_s, z_s, \rho(s))$, $s \in S$, has a function defining its contribution to every point (x, y, z) in the space:

$$contribution_s(x, y, z) = h_v(x - x_s, y - y_s, z - z_s)\rho(s) \qquad (3)$$

where $h_v()$ is the volume reconstruction kernel and $\rho(s)$ is the density of sample s which is located at (x_s, y_s, z_s). The contribution of a sample s to an image plane pixel (x, y) can then be computed by integration:

$$contribution_s(x, y) = \rho(s) \int_{-\infty}^{\infty} h_v(x - x_s, y - y_s, w)dw \qquad (4)$$

where the w coordinate axis is parallel to the view ray. Since this integral is independent of the sample's density, and depends only on its (x, y) projected location, a footprint function can be defined as follows:

$$footprint(x, y) = \int_{-\infty}^{\infty} h_v(x, y, w)dw \qquad (5)$$

where (x, y) is the displacement of an image sample from the center of the sample's image plane projection. The weight at each pixel can then be expressed as:

$$weight(x, y)_s = footprint(x - x_s, y - y_s) \qquad (6)$$

where (x, y) is the pixel's location, and (x_s, y_s) is the image plane location of the sample s.

A footprint table can be generated by evaluating the integral in Equation 5 on a grid with a resolution much higher than the image plane resolution. All table values lying outside of the footprint table extent have zero weight and therefore need not be considered when generating an image. If we have a footprint table for a data sample s, we can center this table on the projected image plane location of s, and sample it in order to determine the weight of the contribution of s to each pixel on the image plane. Multiplying this weight by $\rho(s)$ then gives us the contribution of s to each pixel.

Computing a footprint table can be difficult due to the integration required. Discrete integration methods can be used to approximate the continuous integral, but generating a footprint table is still a costly operation. Luckily, for orthographic projections, the footprint of each sample is the same except for an image plane offset. Therefore we need only calculate one footprint table per view. Since this still would require too much computation time, only one generic footprint table is built for the kernel. For each view, a view-transformed footprint table is created from the generic footprint table. The generic footprint table can be precomputed, therefore it does not matter how long the computation takes.

Generating a view-transformed footprint table from the generic footprint table can be accomplished in three steps. First, the image plane extent of the projection of the reconstruction kernel is determined. Next a mapping is computed between this extent and the extent that surrounds the generic footprint table. Finally, the value for each entry in the view-transformed footprint table is determined by mapping the location of the entry to the generic footprint table, and sampling. The extent of the reconstruction kernel is either a sphere, or is bounded by a sphere, so the extent of the generic footprint table is always a circle. If the grid spacing of the data samples is uniform along all three axes, then the reconstruction kernel is a sphere and the image plane extent of the reconstruction kernel will be a circle. The mapping from this extent to the extent of the generic footprint table is simply a scaling operation. If the grid spacing differs along the three axes, then the reconstruction kernel is an ellipsoid and the image plane extent of the reconstruction kernel will be an ellipse. In this case, a mapping from this ellipse to the circular extent of the generic footprint table must be computed.

There are three modifiable parameters in this algorithm which can greatly affect image quality. First, the size of the footprint table can be varied. Small footprint tables will produce blocky images, while large footprint tables may smooth out details and require more space. Second, different sampling methods can be used when generating the view-transformed footprint table from the generic footprint table. Using a nearest-neighbor approach is fast, but may produce aliasing artifacts. On the other hand, using bilinear interpolation will produce smoother images at the expense of longer rendering times. The third parameter which can be modified is the reconstruction kernel itself. The choice of, for example, a cone function, Gaussian function, sync function or bilinear function affects the final image.

Drebin, Carpenter, and Hanrahan [11] developed a technique for rendering volumes that contain mixtures of materials, such as CT data containing bone, muscle and flesh. In this method, various assumptions about the volume data are made. First, it is assumed that the scalar field was sampled above the Nyquist frequency, or a low-pass filter was used to remove high frequencies before sampling. The volume contains either several scalar fields, or one scalar field representing the composition of several materials. If the latter is the case, it is assumed that material can be differentiated either by the scalar value at each point, or by additional information about the composition of each volume element.

The first step in this rendering algorithm is to create new scalar fields from the input data, known as material percentage volumes. Each material percentage volume is a scalar field representing only one material. Color and opacity are then associated with each material, with composite color and opacity obtained by linearly combining the color and opacity for each material percentage volume. A matte volume, that is, a scalar field on the volume with values ranging between 0 and 1, is used to slice the volume, or perform other spatial set operations. Actual rendering of the final composite scalar field is obtained by transforming the volume so that one axis is perpendicular to the image plane. The data is then projected plane by plane in a back-to-front manner and composited to form the final image.

4.2. Image-Order Techniques

Image-order volume rendering techniques are fundamentally different from object-order rendering techniques. Instead of determining how a data sample affects the pixels on the image plane, in an image-order technique we determine for each pixel on the image plane which data samples contribute to it.

One of the first image-order volume rendering techniques, which may be called *binary ray casting* [55], was developed to generate images of surfaces contained within binary volumetric data without the need to explicitly perform boundary detection and hidden-surface removal. In order to provide all possible views, the data is kept in a fixed position and the image plane is allowed to move. The image plane is defined initially with the center located at C_0. A new image plane with its center at C can then be defined by C_0 and three angular parameters α, β, and γ which specify three rotations R_α, R_β, and R_γ about the X, Y, and Z axes, respectively. Applying these three rotations to the initial image with center C_0 produces the new image plane location with center C. If the view vector for the initial image is $V_0 = (0, 0, 1)$, the view vector V for the current image plane can be computed by applying R_α, R_β, and R_γ to V.

For each pixel on the image plane, we send a ray from that pixel and determine if it intersects the surface contained within the data. For parallel projections, all rays are parallel to the view direction, where as for perspective projections, rays are cast from the eye point according to the view direction and the field of view. If an intersection does occur, shading is performed at the intersection, and the resulting color is placed in the pixel. In order to determine the first intersection along the ray a stepping technique is used where the value is determined at regular intervals along the ray until the object is intersected. Data samples with a value of 0 are considered to be the background while those with a value of 1 are considered to be part of the object. A zero-order interpolation technique is used, so the value at a location along the ray is 0 if that location is outside of the bounding box of the data, otherwise it is the value of the closest data sample. For a step size d, the i^{th} point sample p_i would be taken at a distance $i \times d$ along the ray. For a given ray, either all point samples along the ray have a value of 0 (the ray missed the object entirely), or there is some sample p_i taken at a distance $i \times d$ along the ray, such that all samples p_j, $j < i$, have a value of 0, and sample p_i has a value of 1. Point sample p_i is then considered to be the first intersection along the ray.

In this algorithm, the step size d must be chosen carefully. If d is too large, small features in the data may not be detected. On the other hand, if d is small, the intersection point is more accurately estimated at the cost of higher computation time.

There are two optimizations which can be made to this algorithms. First, we can reduce the number of steps which must be made along each ray if we traverse only the part of the ray contained within the bounding box of the data. If P_0 is the location of the pixel on the image plane, then any location P along the view ray V through that pixel can be expressed as $P = P_0 + kV$. We can then find the two values k_1 and k_2 such that if $k_1 < k < k_2$, P lies within the bounding box of the data. Once we know these two values, we can skip over the steps which would be made along the ray between the pixel and where the ray enters the bounding box, and we can terminate our search when the ray exits the bounding box.

The second optimization involves the representation of the data in memory. This algorithm was originally developed on a machine with only 32K of RAM, so data compression was a critical issue. Instead of simply storing the data as a binary array of 1's and 0's, a scan-line representation can be used.

Since the data is binary, each scan-line can be thought of as a sequence of alternating-valued segments. For each scan-line in the data, a list of end points can be stored which represent the segments belonging to the object. This representation is compact, yet doesn't add too much time to the intersection calculation.

The previous algorithm deals with the display of surfaces within binary data. A more general algorithm can be used to generate surface and composite projections of multivalued data. Instead of traversing a continuous ray and determining the closest data sample for each step with a zero-order interpolation function, we could traverse a discrete representation of the ray generated using a 3D Bresenham-like algorithm, or a 3D line scan-conversion (voxelization) algorithm [26, 30] (see Section 8.2).

As in the previous algorithms, we need to determine for each pixel in the image plane which data samples contribute to it. This could be done by casting a ray from each pixel in the direction of the viewing ray. This ray would be discretized, and the contribution from each voxel along the path is considered when producing the final pixel value. This technique is referred to as *discrete ray casting*.

In order to generate a 3D discrete ray using a voxelization algorithm, the 3D discrete topology of 3D paths has to be understood. There are three types of connected paths: 6-connected, 18-connected, and 26-connected, which are based upon three adjacency relationships between consecutive voxels along the path. Assuming a voxel is represented as a box centered at the grid point, two voxels are said to be 6-connected if they share a face, they are 18-connected if they share a face or an edge, and they are 26-connected if they share a face, an edge or a vertex. A 6-connected path is a sequence of voxels, v_1, v_2, ... v_N, where for each pair of voxels v_i, v_{i+1} $(1 \le i < N)$, v_i and v_{i+1} are 6-connected. Similar definitions exist for 18- and 26-connected paths. An example of these three types of connected paths is given in Figure 1.

In *discrete ray casting*, a ray is discretized into a 6-, 18-, or 26-connected path, and only the voxels along this path are considered when determining the final pixel value. If a surface projection is required, we would traverse the path until we encounter the first voxel which is part of the object. We

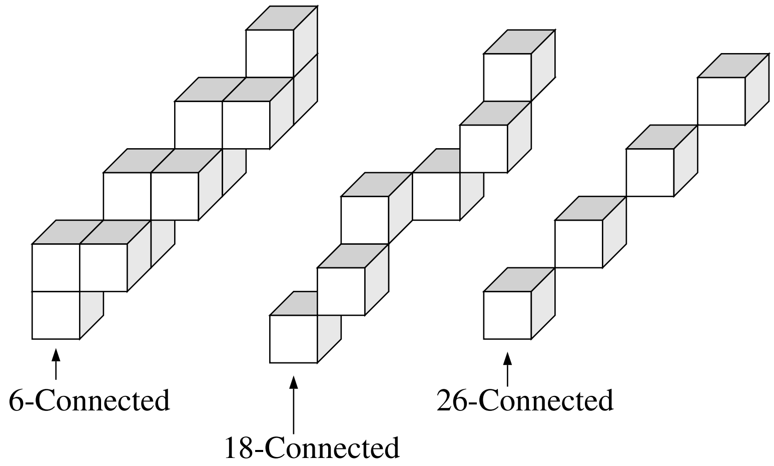

Figure 1: 6-, 18-, and 26-connected paths

would then shade at this voxel and store the resulting color value in the pixel. 6-connected paths contain almost twice as many voxels as 26-connected paths, so an image created using 26-connected paths would require less computation, but a 26-connected path may miss an intersection that would be detected using a 6-connected path.

To produce a shaded image we could store at each pixel in the image the distance to the closest intersection, then pass this image to a 2D discrete shader, such as those described previously. However, better results can be obtained by performing a 3D discrete shading operation at the intersection point. One 3D discrete shading method, known as *normal-based contextual shading* [4], can be used to estimate the normal when zero-order interpolation is used. The normal for a face of a voxel that is on the surface of the object is determined by examining the orientation of that face, and the orientation of the four faces on the surface that are edge connected to that face. Since a face of a voxel can have only six possible orientations, the error in the approximated normal can be significant. More accurate results can be obtained using a technique known as *gray-level* shading [3, 6, 23, 52, 53]. If the intersection occurs at location (x, y, z) in the data, then the gray-level gradient at that location can be approximated with a central difference:

$$
\begin{aligned}
G_x &= \frac{f(x+1, y, z) - f(x-1, y, z)}{2S_x}, \\
G_y &= \frac{f(x, y+1, z) - f(x, y-1, z)}{2S_y}, \\
G_z &= \frac{f(x, y, z+1) - f(x, y, z-1)}{2S_z},
\end{aligned}
\tag{7}
$$

where (G_x, G_y, G_z) is the gradient vector, and S_x, S_y, and S_z are the distances between neighboring samples in the x, y, and z directions, respectively. The gradient vector is used as a normal vector for shading calculation, and the intensity value obtained from shading is stored in the image. A normal estimation can be performed at p_i, and this information, along with the light direction, and the distance $i \times d$ can be used to shade p_i.

Actually, stopping at the first opaque voxel and shading there is only one of many operations which can be performed on the voxels along a discrete path or continuous ray. Instead, we could traverse the whole ray, storing in the image plane pixel the maximum value encountered along the ray. Figure 2 (a) is a first opaque, or surface, projection of a bullfrog sympathetic ganglion cell, which was reconstructed from confocal microscope data, while Figure 2 (b) is a maximum projection of the same cell. Figure 2 was generated using the PARC algorithm, which is described in Section 5 As opposed to a surface projection, a maximum projection is capable of revealing internal features of the data. For example, the high intensity regions of Figure 2 (b) are boutons, which lie below the surface of the cell and are therefore not visible in Figure 2 (a). Another option is to store the sum (simulating X-rays) or average of all values along the ray. More complex techniques exist (described below), which may involve defining an opacity and color for each scalar value, then accumulating intensity along the ray according to some compositing function.

The previous two rendering techniques, binary ray casting and discrete ray casting, use zero-order interpolation in order to define the scalar value at any location in R^3. One advantage to using zero-order interpolation is simplicity and speed, since many of the calculations required can be done using integer arithmetic. One disadvantage though is the aliasing effects in the image. Higher-order interpolation functions can be used to create a more accurate image, but generally at the cost of algorithm complexity

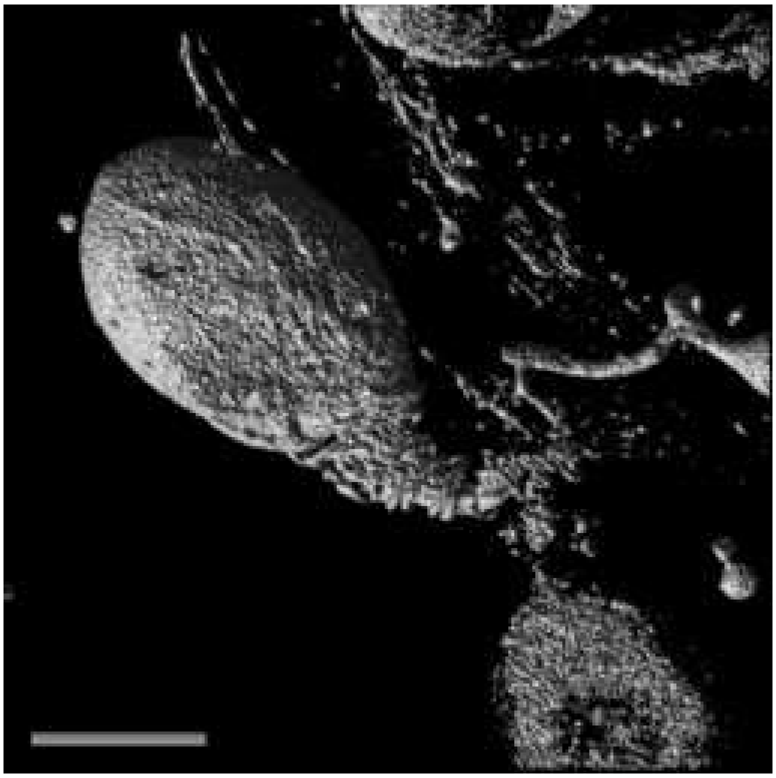

Figure 2(a): A surface projection of a nerve cell.

and computation time. The next three algorithms described in this section all use higher-order interpolation functions.

When creating a composite projection of a data set, there are two important parameters, the color at a location, and the opacity at that location. An image-order volume rendering algorithm developed by Levoy [38] states that given an array of data samples S, we can use preprocessing techniques to generate two new arrays S_c and S_α, which define the color and opacity at each grid location. We then define the interpolation functions $f(x, y, z)$, $f_c(x, y, z)$, and $f_\alpha(x, y, z)$, which define the sample value, color, and opacity at any location in R^3.

Generating the array S_c of color values involves performing a shading operation, such as gray-level shading, at each data sample in the original array S. For this purpose, the Phong illumination model could be used. The normal at each data sample is the unit gradient vector at that location. The gradient vector at any location can be computed by partially differentiating the interpolation function with respect to x, y, and z to get each component of the gradient. If the interpolation function is not first derivative continuous, aliasing artifacts will occur in the image due to the discontinuous normal vector. A smoother set of gradient vectors can be obtained using a central differencing method similar to the one described previously in this section.

Calculating the array S_α is essentially a surface classification operation. There are various different ways to classify surfaces within a scalar field, and each way requires a new mapping from $S(x, y, z)$ to $S_\alpha(x, y, z)$. If we wish to view an iso-surface at some constant value v with an opacity α_v, we could

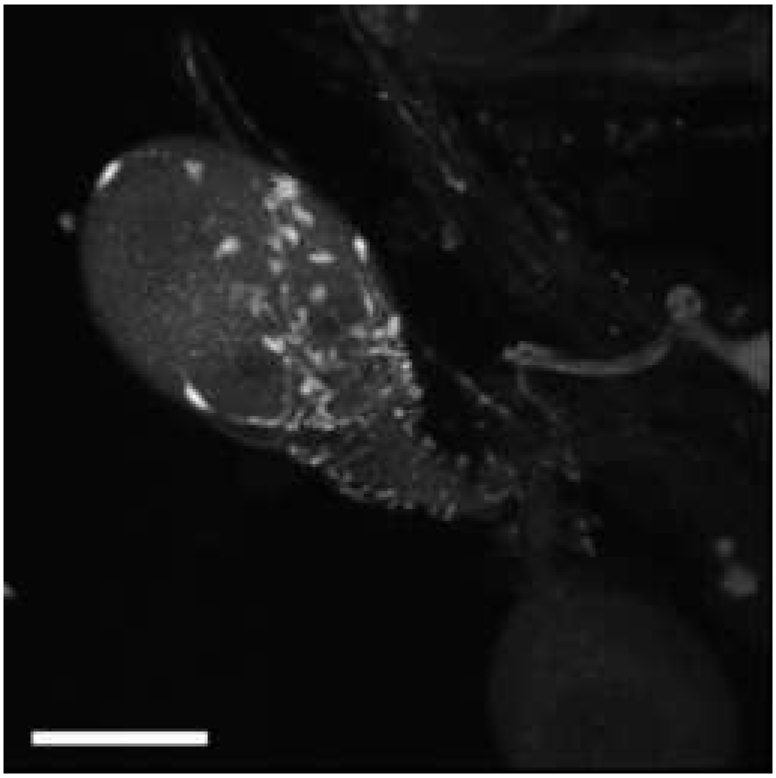

Figure 2(b): A maximum projection of a nerve cell.

simply assign $S_\alpha(x, y, z)$ to α_v if $S(x, y, z)$ is v, otherwise $S_\alpha(x, y, z) = 0$. This would produce aliasing artifacts, which can be reduced by setting $S_\alpha(x, y, z)$ close to α_v if $S(x, y, z)$ is close to v. The best results are obtained when the thickness of the transition region is constant throughout the volume. This can be approximated by having the opacity fall off at a rate inversely proportional to the magnitude of the local gradient vector.

Multiple iso-surfaces can be displayed in a single image by separately applying the classification mappings, then combining the opacities. For example, if N iso-value surfaces are to be displayed with values v_n and opacities α_{v_n}, $1 \le n \le N$, we would define N opacity mappings, and compute the total opacity by:

$$\alpha_{total}(x, y, z) = 1 - \prod_{n=1}^{N} (1 - \alpha_n(x, y, z)). \tag{8}$$

Once we have the $S_c(x, y, z)$ and $S_\alpha(x, y, z)$ arrays, we then cast rays from the pixels, through these two arrays, sampling at evenly spaced locations. To determine the value at a location, the trilinear interpolation functions $f_c()$ and $f_\alpha()$ are used. Once we have these point samples along the ray, we add in a fully opaque background, then composite the values in a back-to-front order to produce a single color that is placed in the pixel.

Two rendering techniques for displaying volumetric data, known as the V-Buffer method, were developed by Upson and Keeler [56]. One of the methods for visualizing the scalar field is an image-order ray-casting technique. In this method, rays are cast from each pixel on the image plane into the volume. For each cell in the volume along the path of this ray, the scalar value is determined at the point where the ray first intersects the cell. The ray is then stepped along until it traverses the entire cell, with calculations for scalar values, shading, opacity, texture mapping, and depth cuing performed at each stepping point. This process is repeated for each cell along the ray, accumulating color and opacity, until the ray exits the volume, or the accumulated opacity reaches unity. At this point, the accumulated color and opacity for that pixel are stored, and the next ray is cast.

The goal of this method is not to produce a realistic image, but instead to provide a representation of the volumetric data which can be interpreted by a scientist, or an engineer. For this purpose, the user is given the ability to modify certain parameters in the shading equations which will lead to an informative, rather than physically accurate shaded image. A simplified shading equation is used where the perceived intensity as a function of wavelength, $I(\lambda)$ is define as:

$$I(\lambda) = K_a(\lambda) I_a + K_d(\lambda) \sum_j [(N \cdot L_j) I_j] \tag{9}$$

In this equation, K_a is the ambient coefficient, I_a is the ambient intensity, K_d is the diffuse coefficient, N is the normal approximated by the local gradient, L_j is the vector to the j^{th} light source, and I_j is the intensity of the j^{th} light source. In order to highlight certain features in the final image, the diffuse coefficient can be defined as a function of not only wavelength, but also scalar value and solid texture:

$$K_d(\lambda, S, M) = K(\lambda)\ T_d(\lambda, S(x, y, z)\ M(\lambda, x, y, z)), \tag{10}$$

where K is the actual diffuse coefficient, T_d is the color transfer function, S is the sample array, and M is the solid texture map. The color transfer function is defined for r, g, and b, and maps scalar value to intensity. When we accumulate along the ray in this method, we are approximating the following intensity integral:

$$I(\lambda) = \int_w \left[att(d) O(s) \left[K_a(\lambda) I_a + K_d(\lambda, S, M) \sum \left[(N \cdot L_j) I_j \right] \right] + (1 - att(d) bg(\lambda)) \right] dw \tag{11}$$

where $att(d)$ represents atmospheric attenuation as a function of distance, O is the opacity transfer function, bg is the background color, and w is a vector in the direction of the view ray. The opacity transfer function is similar to the color transfer function in that it defines opacity as a function of scalar value. Different color and opacity transfer functions can be defined to highlight different features in the volume.

The second method for visualizing the scalar field is a cell-by-cell processing technique, where within each cell an image-order ray-casting technique is used, thus making this a hybrid technique. In this method, each cell in the volume is processed in a front-to-back order. Processing begins on the plane closest to the viewpoint, and progresses in a plane-by-plane manner. Within each plane, processing begins with the cell closest to the viewpoint, then continues in order of increasing distance from the viewpoint. Each cell is processed by first determining for each scan line in the image plane, which pixels are affected by the cell. Then for each pixel an integration volume is determined. Within the bounds of the integration volume, an intensity calculation similar to Equation 11 is performed according to:

$$I(\lambda) = \int_x \int_y \int_z \left[att(d) O(s) \left[K_a(\lambda) I_a + K_d(\lambda, S, M) \sum \left[(N \cdot L_j) I_j \right] \right] + (1 - att(d) bg(\lambda)) \right] dxdydz \tag{12}$$

This process continues in a front-to-back order, until all cells have been processed, with intensity accumulated into pixel values. Once a pixel's opacity reaches unity, a flag is set and this pixel is not processed further. Due to the front-to-back nature of this algorithm, incremental display of the image is possible.

In order to simulate light coming from translucent objects, volumetric data with data samples representing density values can be considered as a field of density emitters (as proposed by Sabella [47]). A density emitter is a tiny particle that both emits and scatters light. The amount of density emitters in any small region within the volume is proportional to the scalar value in that region. These density emitters are used to correctly model the occlusion of deeper parts of the volume by closer parts, but both shadowing and color variation due to differences in scattering at different wavelengths are ignored. These effects are ignored because it is believed that they would complicate the image, detracting from the perception of density variation. Similar to the V-Buffer method, rays are cast from the eye point, through each pixel on the image plane, and into the volume. The intensity I of light for a given pixel is calculated according to:

$$I = \int_{t_1}^{t_2} e^{-\tau \int_{t_1}^{t} \rho^{\gamma}(\lambda) d\lambda} \rho^{\gamma}(t) dt \tag{13}$$

In this equation, the ray is traversed from t_1 to t_2, accumulating at each location t the density $\rho^{\gamma}(t)$ at that location attenuated by the probability $e^{-\tau \int_{t_1}^{t} \rho^{\gamma}(\lambda) d\lambda}$ that this light will be scattered before reaching the eye. The parameter τ is modifiable, and controls the attenuation, with higher values of τ specifying a medium which darkens more rapidly. The parameter γ is also modifiable, and can be used to control the spread of density values. Low γ values produce a diffuse cloud appearance, while higher γ values highlight dense portions of the data.

For each ray, three values in addition to I are computed. The value M is the maximum value encountered along the ray with D being the distance at which that maximum occurred. The value C is the center of gravity of density emitters along the ray. By mapping these four values to different color parameters, interesting effects can be achieved. For example, M may be mapped to hue, D to saturation, and I to value, to highlight certain characteristics of the volume data.

Krueger [35] showed that the various existing volume rendering models can be described as special cases of an underlying transport theory model of the transfer of particles in inhomogeneous media. The basic idea is that a beam of ‘‘virtual’’ particles is sent through the volume, with the user selecting the particle properties and the laws of interaction between the particles and the data. The image plane then contains the ‘‘scattered’’ virtual particles, and information about the data is obtained from the scattering pattern. If, for example, the virtual particles are chosen to have the properties of photons, and the laws of interaction are governed by optical laws, then this model essentially becomes a generalized ray tracer. Other virtual particles and interaction laws can be used, for example, to identify periodicities and similar hidden symmetries of the data.

Using Krueger's transport theory model, the intensity of light I at a pixel can be described as a path integral along the view ray:

$$I = \int_{S_{near}}^{S_{far}} Q(S) e^{-\int_{S_{near}}^{S} \sigma_a(S') + \sigma_{sc}(S') dS'} dS \tag{14}$$

The emission at each point S along the ray is scaled by the optical depth to the eye to produce the final intensity value for a pixel. The optical depth is a function of the total extinction coefficient, which is composed of the absorption coefficient σ_a, and the scattering coefficient σ_{sc}. The generalized source $Q(S)$ is defined as:

$$Q(S) = q(S) + \sigma_{sc}(S) \int \rho_{sc}(\vec{\omega}' \rightarrow \vec{\omega}) I(S, \vec{\omega}') d\vec{\omega}' \tag{15}$$

This generalized source consists of the emission at a given point $q(S)$, and the incoming intensity along all direction scaled by the scattering phase ρ_{sc}. Typically, a low albedo approximation is used to simplify the calculations, reducing the integral in Equation 15 to a sum over all light sources.

The algorithms described so far in this section are concerned primarily with conveying useful information in the final image, rather than generating photo-realistic images. A 3D raster ray tracing (RRT) method, developed by Yagel, Cohen and Kaufman [66, 70]), produces realistic images of volumetric data using a global illumination model. The RRT algorithm is a discrete ray-tracing algorithm similar to the discrete ray-casting algorithm described earlier in this section. Discrete primary rays are cast from the image plane, through the data to determine pixel values. Secondary rays are recursively spawned when a ray encounters a voxel belonging to an object in the data. To save time, the view-independent parts of the illumination equation can be precomputed and added to the voxel's color, thereby avoiding the calculation of this quantity during the ray tracing. Also, two bits per light source per voxel can be precomputed, indicating whether the light is definitely visible, possibly visible, or definitely not visible from that voxel. Shadow rays need only be cast during the ray tracing if the bits indicate that the light is possibly visible through a translucent object. Actually, all view-independent attributes (including normal, texture, and antialiasing) can be precomputed and stored with each voxel.

There are several advantages to using RRT instead of conventional ray tracing. One such advantage is that sampled or computed data, possibly intermixed with voxelized geometric data, can be ray traced directly without having to approximate the sampled data using geometric primitives. Another advantage is that there is only one primitive to deal with − the voxel, which greatly simplifies ray-object intersection calculations. Unlike conventional ray tracing that computes expensive continuous ray-object intersections, RRT traverses discrete rays through discrete data and therefore it is basically insensitive to scene complexity and object complexity. RRT is also very effective for ray tracing voxelized geometric models, such as constructive solid geometry (CSG) models. This is an example for the emerging field of volume graphics [32] in which geometric scenes are modeled using voxelized objects and efficiently rendered using a volume rendering algorithm such as RRT. Volume graphics has the potential to supersede surface graphics for handling and visualizing volumes, as well as for modeling and rendering synthetic scenes composed of surfaces.

5. Volume Rendering Optimizations

Volume rendering can produce informative images that can be useful in data analysis, but a major drawback of many of the techniques described in this paper is the time required to generate a high-quality image. In this section, several volume rendering optimizations are described that decrease rendering times, and therefore increase interactivity and productivity. Other optimizations have been discussed briefly earlier in the paper, along with the original algorithm.

Object-order volume rendering typically loops through the data, calculating the contribution of each volume sample to pixels on the image plane. This is a costly operation for moderate to large sized data sets (e.g., 128M bytes for a 512^3 sample data set, with one byte per sample), leading to rendering times that are not interactive. Viewing the intermediate results in the image plane may be useful, but these partial image results are not always representative of the final image. For the purpose of interaction, it is useful to be able to generate a lower quality image in a shorter amount of time. For data set with binary sample values, we could pack our bits into bytes such that each byte represents a 2×2×2 portion of the data [55]. We would process the data bit by bit to generate the full resolution image, but we could generate a lower resolution image by processing the data byte by byte. If more than four bits of the byte are set, the byte is considered to represent an element of the object, otherwise it represents the background. This will produce an image with one-half the linear resolution in approximately one-eighth the time.

A more general method for decreasing data resolution is to build a pyramid data structure, which for an original data set of N^3 data samples, consists of a sequence of $\log N$ volumes. The first volume is the original data set, while the second volume is created by averaging each 2×2×2 group of samples of the original data set to create a volume of one-eight the resolution. The third volume is create from the second volume in a similar fashion, with this processes continuing until all $\log N$ volumes have been created. An efficient implementation of the splatting algorithm, called hierarchical splatting [36], uses such a pyramid data structure. According to the desired image quality, this algorithm scans the appropriate level of the pyramid in a back-to-front order. Each element is splatted onto the image plane using the appropriate sized splat. The splats themselves are approximated by polygons which can efficiently be rendered by graphics hardware.

Image-order volume rendering involves casting rays from the image plane into the data, and sampling along the ray in order to determine pixel values. In discrete ray casting, the ray would be discretized, and the contribution from each voxel along the path is considered when producing the final pixel value. It would be quite computationally expensive to discretize every ray cast from the image plane. Fortunately, this is unnecessary for parallel projections. Since all the rays are parallel, one ray can be discretized and used as a ‘‘template’’ for all other rays. This technique, developed by Yagel and Kaufman [69], is called *template-based volume viewing*. A 26-connected template v_1, v_2, ... v_N is created such that removing any v_i $(1 < i < N)$ would destroy the connectivity of the path. If this template were used to cast a ray from each pixel in the image plane, some voxels in the data may contribute to the image twice while others may not be considered at all. To solve this problem, the rays are cast instead from a base plane, that is, the plane of the volume buffer most parallel to the image plane. This ensures that each data sample can contribute at most once to the final image, and all data samples could potentially contribute. But when we extend these rays back to the image plane, we notice that we have not cast the rays exactly from the pixels. Once all the rays have been cast, we need a simple final step of resampling which uses bilinear interpolation to determine the pixel values from the ray values we have calculated.

An extension can be made to this template-based ray casting to allow higher-order interpolation [68]. The template for higher-order interpolation consists of connected cells, as opposed to the connected voxel template used for zero-order interpolation. Since the value varies within a cell, it is desirable to take multiple samples along the continuous ray inside of each cell. Since these samples are taken at regular intervals, and the same template is used for every ray, there is only a finite number of 3D locations (relative to a cell) at which sampling occurs. This fact allows us to precompute part of the interpolation function and store it in a table, allowing for faster rendering times.

Another extension to template-based ray casting allows for screen space supersampling to improve image quality [65]. This is accomplished by allowing rays to originate at sub-pixel locations. A finite

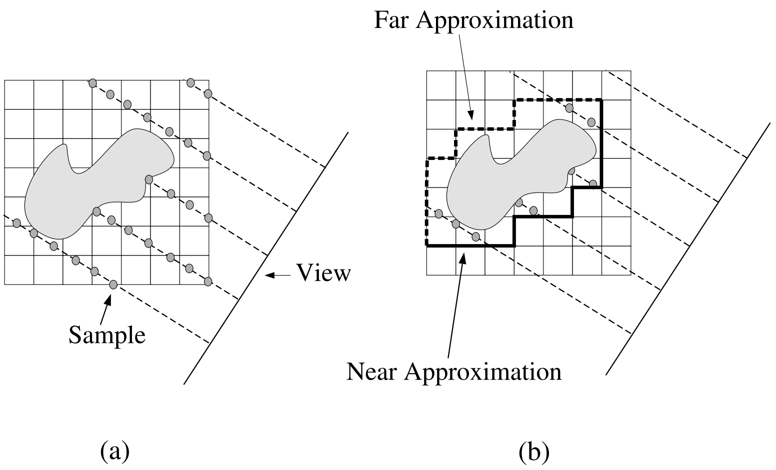

Figure 3: (a) Brute-force ray casting. (b) Polygon assisted ray casting (PARC).

Figure 4 (a): VolVis polyhedral representation of a complex scene.

number of sub-pixel locations from which a ray can originate is selected, and a template is created for each. When a ray is cast, its sub-pixel location determines which template is used. For example, to accomplish a 2×2 uniform supersampling, four rays would be cast per pixel, and therefore four sub-pixel locations are possible. Stochastic supersampling can also be supported by limiting the possible ray origins to a finite number of sub-pixel locations, and precomputing a template for each.

One obvious optimization for both discrete and continuous ray casting which has already been discussed is to limit the sampling to the segment of the ray which intersects the data, since samples outside of the data evaluate to 0 and do not contribute to the pixel value. If the data itself contains many zero-valued data samples, or a segmentation function is applied to the data that evaluates to 0 for many samples, the efficiency of ray casting can be greatly enhanced by further limiting the segment of the ray in which samples are taken. One algorithm of this sort is known as *polygon assisted ray casting*, or *PARC* [1]. This algorithm approximates objects contained within a volume using a crude polyhedral representation. The polyhedral representation is created so that it completely contains the objects. Using conventional graphics hardware, the polygons are projected twice to create two Z-buffers. The first Z-buffer is the standard closest-distance Z-buffer, while the second is a farthest-distance Z-buffer. Since the object is completely contained within the representation, the two Z-buffer values for a given image plane pixel can be used as the starting and ending points of a ray segment on which samples are taken. A comparative illustration of brute-force ray casting and PARC for rendering a surface contained within a volume are shown in Figure 3, with the closest distance Z-buffer values shown as a solid line, and the farthest distance Z-buffer values shown as a dashed line.

The PARC algorithm is part of the VolVis volume visualization system [1], which provides a multi-algorithm progressive refinement approach for interactivity. By using available graphics hardware, the user is given the ability to manipulate in real-time the polyhedral representation of the data. When the user is satisfied with the placement of the data, light sources, and view, the Z-buffer information is

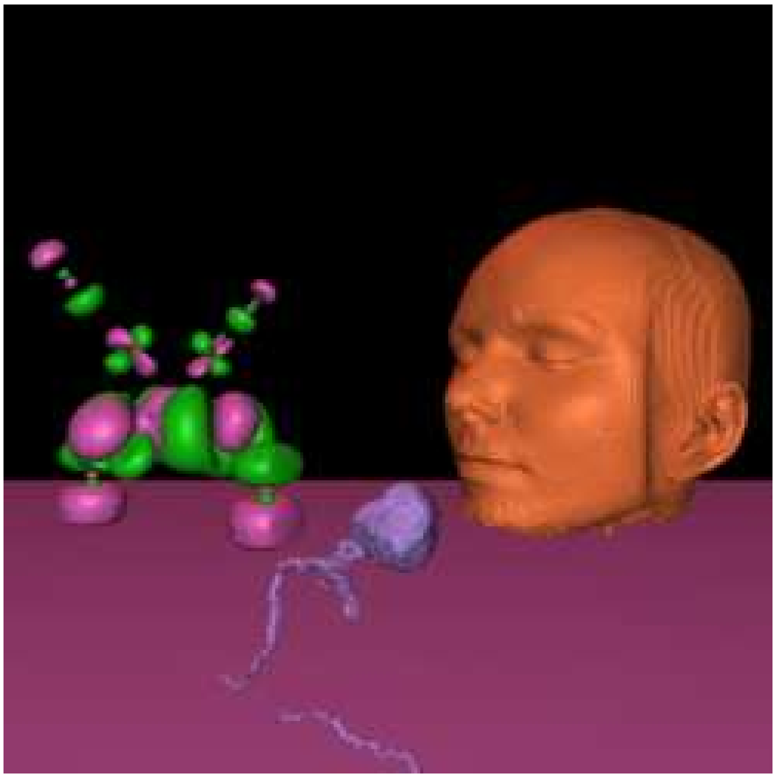

Figure 4 (b): VolVis PARC rendering of the scene.

passed to the PARC algorithm, which produces a ray-cast image. In a final step, this image is further refined by continuing to follow the PARC rays which intersected the data according to a volumetric ray tracing algorithm [50] in order to generate shadows, reflections, and transparency. The ray tracing algorithm uses various optimization techniques, including uniform space subdivision and bounding boxes, to increase the efficiency of the secondary rays. Surface rendering, as well as transparency with color and opacity transfer functions, are incorporated within a global illumination model.

Figure 4 illustrate a typical sequence of images in this progressive refinement. The scene shown consists of three volumetric data sets, and one geometric object. The leftmost object is a quantum mechanics simulation of a high potential iron protein, the middle object is a bullfrog sympathetic ganglion cell acquired with a laser-scanning confocal microscope, and the right object is a human head obtained from MRI. Below these three objects, a geometric polygon was placed. Figure 4 (a) is the polyhedral representation of the scene used for interactive navigation and to create the Z-buffers for the PARC algorithm. Figure 4 (b) is the PARC rendering, and Figure 4 (c) is the volumetric ray traced version of the scene.

Another optimization method, which is also implemented as part of *VolVis*, is a data compression technique for volume rendering [5]. Figure 5 shows the basic compression and rendering process. This volume compression technique is a generalization of the JPEG still image compression algorithm [58] to the 3D case, with one important exception: the transform is a discrete Fourier transform rather than a

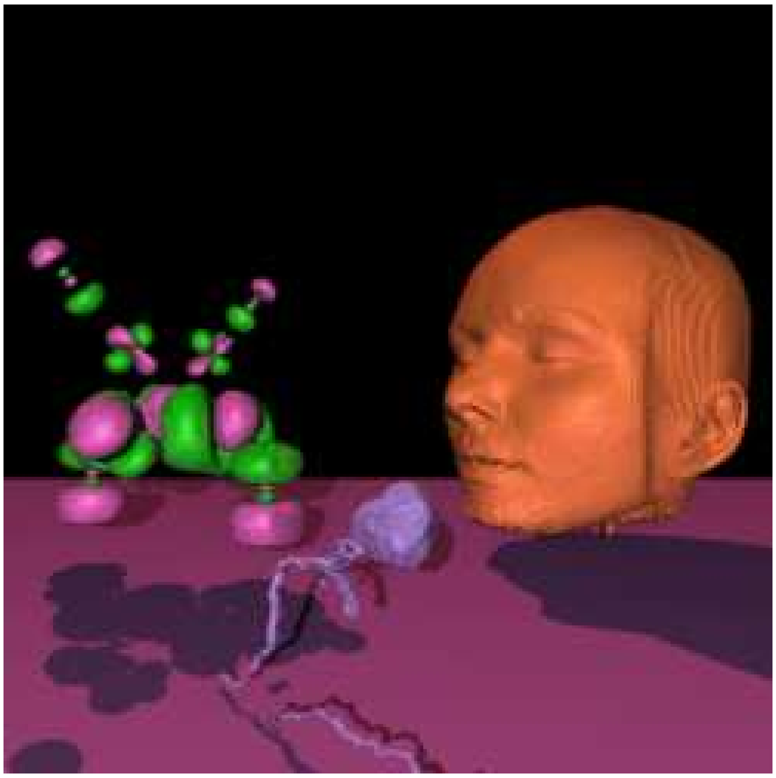

Figure 4 (c): VolVis ray traced rendering of a complex scene.

discrete cosine transform. The original 3D data is subdivided into $M \times M \times M$ subcubes, where each subcube is Fourier transformed to the frequency domain through a 3D discrete Fourier transform. Each of the 3D Fourier coefficients in each subcube is then quantized, and the resulting 3D quantized frequency coefficients within a cube are organized as a linear sequence through a 3D zig-zag order. The resulting sequence of Fourier transform coefficients is fed into an entropy encoder that in turn consists of run-length coding and Huffman coding.

To render in the compressed domain, a new class of Fourier domain volume rendering algorithms is used [13, 39, 41, 42, 54]. All of these algorithms are based on the Fourier projection slice theorem, which states that a projection of the 3D data volume from a certain view direction can be obtained by extracting a 2D slice perpendicular to that view direction out of the 3D Fourier spectrum and then inverse Fourier transforming it. In the compression-domain approach the Fourier projection slice theorem is applied to each subcube in the Fourier domain, which results in a set of 2D planes in the spatial domain called subimages that are composited using a spatial compositing function to get the final projection of the original 3D dataset.

Using the compression-domain rendering approach, it is possible to achieve very high compression ratios with no visible image degradation after rendering. Furthermore, image quality is improved over previous frequency domain rendering techniques by compositing subimages, while still maintaining a similar speed advantage compared to spatial volume rendering due to a lookup table-driven implementation. Figure 6 shows a CT scan of a lobster that was rendered out of the compressed domain.

6. Volumetric Global Illumination

Speed and accuracy of the final image are both important, yet often conflicting aspects of the rendering process. For this reason, a comprehensive volume rendering system, such as *VolVis*, includes a rang of rendering algorithms from the fast, rough approximation of the final image, to the comparatively slow, accurate rendering within a global illumination model. Also, each rendering algorithm itself should

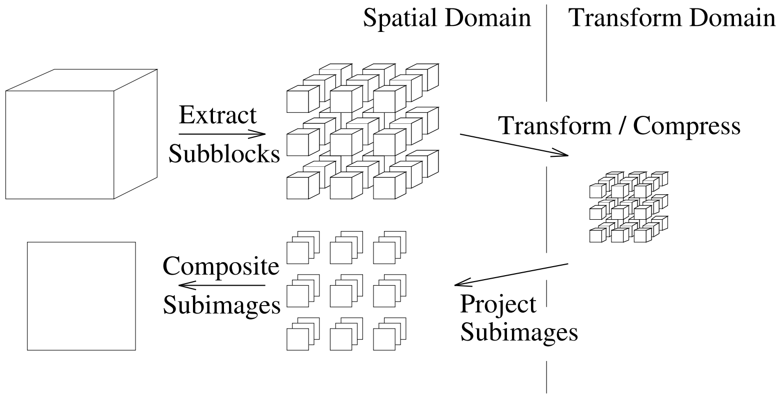

Figure 5: The compression and rendering process.

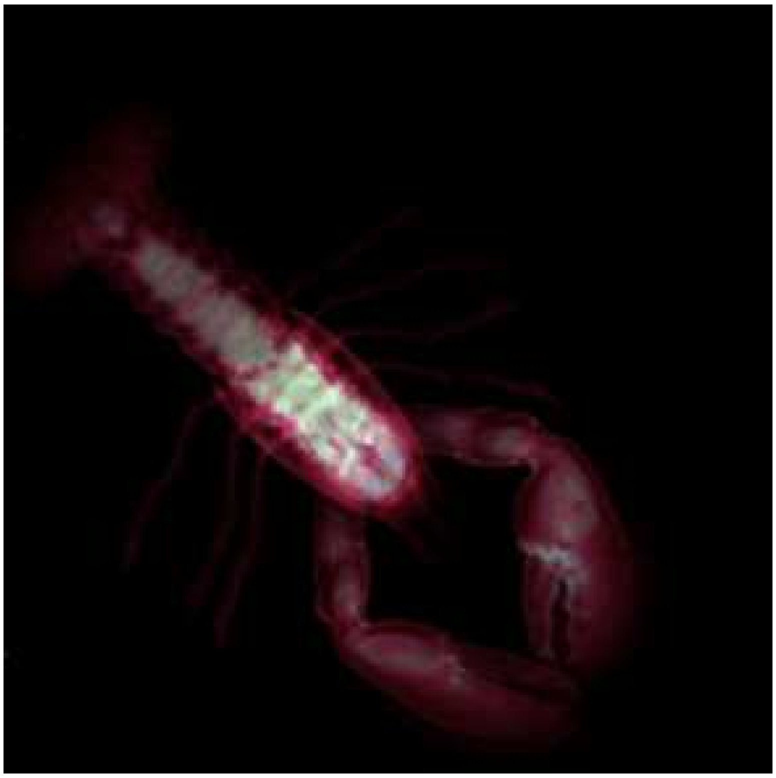

Figure 6: Compression domain volume rendering of a CT scan of a lobster.

support several levels of accuracy, giving the user an even greater amount of control over the speed and accuracy of the final image.

Standard volume rendering techniques typically employ only a local illumination model for shading, and therefore produce images without global effects. Including a global illumination model within a visualization system has several advantages. First, global effects can often be desirable in scientific applications. For example, by placing mirrors in the scene, a single image can show several views of an object in a natural, intuitive manner leading to a better understanding of the 3D nature of the scene. Also, complex surfaces are often easier to render when represented volumetrically than when represented by high-order functions or geometric primitives. Volumetric ray tracing is described in Section 6.1 and volumetric radiosity is discussed in Section 6.2.

6.1. Volumetric Ray Tracing

The volumetric ray tracer [50] is intended to produce accurate, informative images. In classical ray tracing, the rendering algorithm is designed to generate images that are accurate according to the laws of optics. A volumetric ray tracer must handle classical geometric objects as well as volumetric data, and strict adherence to the laws of optics is not always desirable. For example, a scientist may wish to view the maximum value along the segment of a ray passing through a volume, instead of the optically-correct composited value. Figure 7 illustrates the importance of including global effects in a maximum-value projection of a hippocampal pyramidal neuron dataset which was obtained using a laser-scanning confocal microscope. Since maximum-value projections do not give depth information, a floor is placed below the cell, and a light source above the cell. This results in a shadow of the cell on the floor, adding back the depth information lost by the maximum value projection.

In order to incorporate both geometric and volumetric objects into one scene, the standard ray tracing intensity equation must be expanded. Since the illumination equation in classical ray tracing is evaluated only at surface locations, volumetric data cannot be incorporated into the scene. This problem can be solved by extending the standard illumination equation to include volumetric effects. The intensity of light, $I_\lambda(x, \vec{\omega})$, for a given wavelength λ, arriving at a position x, from the direction $\vec{\omega}$, can be computed by:

$$I_\lambda(x, \vec{\omega}) = I_{v\lambda}(x, x') + \tau_\lambda(x, x') I_{s\lambda}(x', \vec{\omega}). \tag{16}$$

where x' is the first surface intersection point encountered along the ray $\vec{\omega}$ originating at x. $I_{s\lambda}(x', \vec{\omega})$ is the intensity of light at this surface location, and can be computed with a standard ray tracing illumination equation [63]. $I_{v\lambda}(x, x')$ is the volumetric contribution to the intensity along the ray from x to x', and $\tau_\lambda(x, x')$ is the attenuation of $I_{s\lambda}(x', \vec{\omega})$ by any intervening volumes. These values are determined using volume rendering techniques, based on a transport theory model of light propagation [35]. The basic idea is similar to classical ray tracing, in that rays are cast from the eye into the scene,

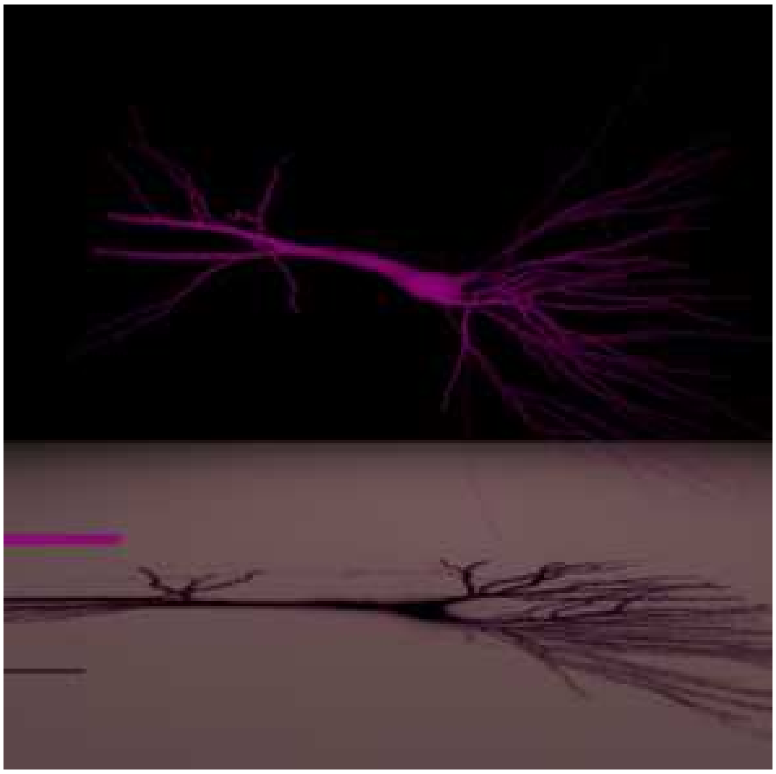

Figure 7: A maximum-value projection of a cell casting a shadow on the floor using the *VolVis* volumetric ray tracer.

and surface shading is performed on the closest surface intersection point. The difference is that shading must be performed for all volumetric data that are encountered along the ray while traveling to the closest surface intersection point.

For photo-realistic rendering, the user typically wants to include the maximum amount of shading effects that can be calculated within a given time limit. For visualization however, the user may find it necessary to view volumetric data with no shading effects, such as when using a maximum-value projection. For example, in Figure 7 no shading effects were included for the maximum-value projection of the cell, while all parts of the illumination equation were considered when shading the geometric polygon. In another example, the user may place a mirror behind a volumetric object in a scene in order to capture two views in one image, but may not want the volumetric object to cast a shadow on the mirror. This can be accomplished easily by ‘‘turning off’’ the shadowing calculations for the mirror, as shown in Figure 8. The head was obtained using magnetic resonance imaging, with the brain segmented from the same dataset. The mirror is a voxelized polygon, which was created using the non-binary voxelization technique described in Section 8.3

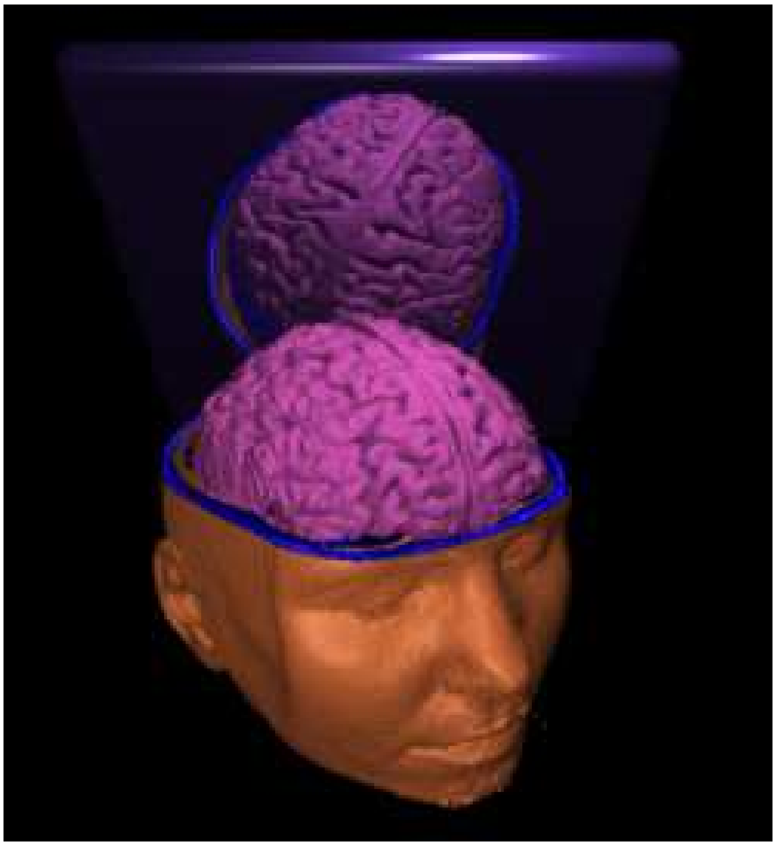

Figure 8: A ray traced image of a human head. Shadowing effects were not included in this image in order to produce a clear reflection in the mirror (see also animation segment).

6.2. Volumetric Radiosity

The ray tracing algorithm described in the previous section can be used to capture specular interactions between objects in a scene. In reality, most scenes are dominated by diffuse interactions, which are not accounted for in the standard ray tracing illumination model, but accounted for by a radiosity algorithm for volumetric data [51]. In volumetric radiosity, the basic ‘‘patch’’ element of classical radiosity is replaced by a ‘‘voxel’’. As opposed to previous methods that use participating media to augment geometric scenes, [24, 46] this method moves the radiosity equations into volumetric space, and renders scenes consisting solely of volumetric data. Each voxel can emit, absorb, scatter, reflect, and transmit light. Both isotropic and diffuse emission of light are allowed, where ‘‘isotropic’’ implies directional independence, and ‘‘diffuse’’ implies Lambertian reflection (i.e., dependent on normal or gradient). Light is scattered isotropically, and is reflected diffusely by a voxel. Light entering a voxel that is not absorbed, scattered, or reflected by the voxel is transmitted unchanged.

In order to cope with the high number of voxel interactions required, a hierarchical technique similar to [19] is used. The basic hierarchical concept is that the radiosity contribution from some voxel v_i to another voxel v_j is similar to the radiosity contribution from v_i to v_k if the distance between v_j and v_k is small, and the distance between v_i and v_j is large. For each volume, a hierarchical radiosity structure is built by combining each subvolume of eight voxels at one level to form one voxel at the next higher level. An iterative algorithm [7] is then used to shoot voxel radiosities, where several factors govern the highest level in the hierarchy at which two voxels can interact. These factors include the distance between the two voxels, the radiosity of the shooting voxel, and the reflectance and scattering coefficients of the voxel receiving the radiosity. This hierarchical technique can reduce the number of interactions required to converge on a solution by more than four orders of magnitude.

After the view-independent radiosities have been calculated, a view-dependent image is generated using a ray casting technique, where the final pixel value is determined by compositing radiosity values along the ray. Figure 9 shows a scene containing a volumetric sphere, polygon, and light source. The light source is isotropically emitting light, and both the sphere and the polygon are diffusely reflecting light. The light source is above the sphere, and therefore the top half of the sphere is directly illuminated. The bottom half of the sphere is indirectly illuminated by light diffusely reflected from the red polygon.

7. Volume Graphics

The 3D raster representation seems to be more natural for empirical imagery, due to its ability to represent interiors and digital samples. Nontheless, the advantages of this representation are also attracting traditional surface-based applications that deal with the modeling and rendering of synthetic scenes represented by geometric models. The geometric model is *voxelized (3D scan-converted)* into a set of voxels that ‘‘best’’ approximate the model which is then stored in the volume buffer together with pre-computed view-independent attributes. The voxelized model can be either binary (see [8, 26-28]) or volume sampled [59] which generates alias-free density voxelization of the model. Some surface-based examples are the rendering of fractals [44], hyper textures [45], fur [25], gases [15], and other complex models [48] including CAD models and terrain models for flight simulators [9, 32, 64]. Furthermore, in many applications involving sampled data, such as medial imaging, the data need to be visualized along with synthetic objects that may not be available in digital form, such as scalpels, prosthetic devices, injection needles, radiation beams, and isodose surfaces. The geometric objects can be voxelized and intermixed with the sampled organ in the voxel buffer [29].

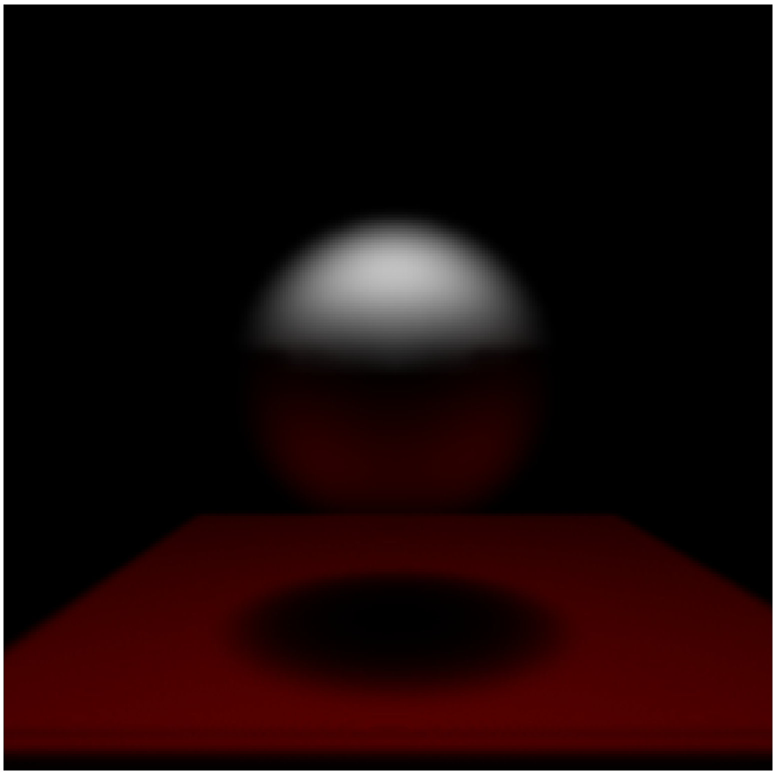

Figure 9: A volumetric radiosity projection of a voxelized sphere and polygon.

Volume graphics [32], which is an emerging subfield of computer graphics, is concerned with the synthesis, manipulation, and rendering of volumetric geometric objects, stored in a volume buffer of voxels. Unlike volume visualization which focuses primarily on sampled and computed datasets, volume graphics is concerned primarily with modeled geometric scenes and particularly with those that are represented in a regular volume buffer. As an approach, volume graphics has the potential to greatly advance the field of 3D graphics by offering a comprehensive alternative to traditional surface graphics.

In the next sections we describe the volumetric approach to several common modeling techniques. We describe the generation of object primitives (voxelization), 3D antialiasing, texture and photo mapping, solid-texturing, modeling of amorphous phenomena, modeling by block operations, and constructive solid modeling. For each of the techniques, we identify the advantages of the volume graphics approach.

8. Voxelization

An indispensable stage in volume graphics is the synthesis of voxel-represented objects. This stage, which is called *voxelization*, is concerned with converting geometric objects from their continuous geometric representation into a set of voxels that ‘‘best’’ approximates the continuous object. As this process mimics the scan-conversion process that pixelizes (rasterizes) 2D geometric objects, it is also referred to as *3D scan-conversion*. In 2D rasterization the pixels are directly drawn onto the screen to be visualized and filtering is applied to reduce the aliasing artifacts. However, the voxelization process does not render the voxels but merely generates a database of the discrete digitization of the continuous object.

Intuitively, one would assume that a proper voxelization simply "selects" all voxels which are met (if only partially) by the object body. Although this approach could be satisfactory in some cases, the objects it generates are commonly too coarse and include more voxels than are necessary. For example, when a 2D curve is rasterized into a connected sequence of pixels, the discrete curve does not "cover" the entire continuous curve, but it is connected and concisely and successfully "separates" both "sides" of the curve.

One practical meaning of separation is apparent when a voxelized scene is rendered by casting discrete rays from the image plane to the scene. The penetration of the background voxels (which simulate the discrete ray traversal) through the voxelized surface causes the appearance of a hole in the final image of the rendered surface. Another type of error might occur when a 3D flooding algorithm is employed either to fill an object or to measure its volume, surface area, or other properties. In this case the nonseparability of the surface causes a leakage of the flood through the discrete surface.

Unfortunately, the extension of the 2D definition of separation to the third dimension and to surfaces is not straightforward since voxelized surfaces cannot be defined as an ordered sequence of voxels and a voxel on the surface does not have a specific number of adjacent surface voxels. Furthermore, there are important topological issues, such as the separation of both sides of a surface, which cannot be well-defined by employing 2D terminology. The theory that deals with these topological issues is called *3D discrete topology*. We sketch below some basic notions and informal definitions used in this field.

8.1. Fundamentals of 3D Discrete Topology

The 3D discrete space is a set of integral grid points in 3D Euclidean space defined by their Cartesian coordinates (x, y, z). A voxel is the unit cubic volume centered at the integral grid point. The voxel value is mapped into $\{0, 1\}$: the voxels assigned "1" are called the "black" voxels representing opaque objects, and those assigned "0" are the "white" voxels representing the transparent background. In Section 8.3 we describe non-binary approaches where the voxel value is mapped onto the interval [0,1] representing either partial coverage, variable densities, or graded opacities. Due to its larger dynamic range of values, this approach supports 3D antialiasing and thus may support higher quality rendering.

Two voxels are *26-adjacent* if they share either a vertex, an edge, or a face (see Figure 1). Every voxel has 26 such adjacent voxels: eight share a vertex (corner) with the center voxel, twelve share an edge, and six share a face. Accordingly, face-sharing voxels are defined as *6-adjacent*, and edge-sharing and face-sharing voxels are defined as *18-adjacent*.

The prefix N is used to define the adjacency relation, where $N = 6, 18$, or 26. We say that a sequence of voxels having the same value (e.g., "black") is an *N-path* if all consecutive pairs are N-adjacent. A set of voxels A is *N-connected* if there is an N-path between every pair of voxels in A. An *N-connected component* is a maximal N-connected set.

Given a 2D discrete 8-connected black curve, there are sequences of 8-connected white pixels (8-component) that passes from one side of the black component to its other side without intersecting it.

This phenomenon is a discrete disagreement with the continuous case where there is no way of penetrating a closed curve without intersecting it. To avoid such a scenario, it has been the convention to define "opposite" types of connectivity for the white and black sets. "Opposite" types in 2D space are 4 and 8, while in 3D space 6 is "opposite" to 26 or to 18.

Assume that a voxel space, denoted by Σ, includes one subset of "black" voxels S. If $\Sigma - S$ is not N-connected, that is, $\Sigma - S$ consists of at least two white N-connected components, then S is said to be *N-separating* in Σ. Loosely speaking, in 2D, an 8-connected black path that divides the white pixels into two groups is 4-separating and a 4-connected black path that divides the white pixels into two groups is 8-separating. There are no analogous results in 3D space.

Let A be an N-separating surface. A voxel $p \in A$ is said to be an *N-simple voxel* if $A - p$ is still N-separating. An N-separating surface is called *N-minimal* if it does not contain any N-simple voxel. A *cover* of a continuous surface is a set of voxels such that every point of the continuous surface lies in a voxel of the cover. A cover is said to be a *minimal cover* if none of its subsets is also a cover. The cover property is essential in applications that employ space subdivision for fast ray tracing [17]. The subspaces (voxels) which contain objects have to be identified along the traced ray. Note that a cover is not necessarily separating, while on the other hand, as mentioned above, it may include simple voxels. In fact, even a minimal cover is not necessarily N-minimal for any N.

8.2. Binary Voxelization

In the past, digitization of solids was performed by spatial enumeration algorithms which employ point or cell classification methods in either an exhaustive fashion or by recursive subdivision [37]. However, subdivision techniques for model decomposition into rectangular subspaces are computationally expensive and thus inappropriate for medium or high resolution grids. Instead, objects should be directly voxelized, preferably generating an N-separating, N-minimal, and covering set, where N is application dependent. The voxelization algorithms should follow the same paradigm as the 2D scan-conversion algorithms; they should be incremental, accurate, use simple arithmetic (preferably integer only), and have a complexity that is not more than linear with the number of voxels generated.

The literature of 3D scan-conversion is relatively small. Danielsson [10] and Mokrzycki [43] developed similar 3D curve algorithms where the curve is defined by the intersection of two implicit surfaces. Voxelization algorithms have been developed for 3D lines, 3D circles, and a variety of surfaces and solids, including polygons, polyhedra, and quadric objects [26]. Efficient algorithms have been developed for voxelizing polygons using an integer-based decision mechanism embedded within a scan-line filling algorithm [27], for parametric curves, surfaces, and volumes using an integer-based forward differencing technique [28], and for quadric objects such as cylinders, spheres, and cones using "weaving" algorithms by which a discrete circle/line sweeps along a discrete circle/line [8]. Figure 10 consists of a variety of objects (polygons, boxes, cylinders) voxelized using these methods. These pioneering attempts should now be followed by enhanced voxelization algorithms that, in addition to being efficient and accurate, will also adhere to the topological requirements of separation, coverage, and minimality.

Figure 10: A volumetric model of terrain enhanced with photo mapping of satellite images. The buildings are synthetic voxel models raised on top of the terrain. The voxelized terrain has been mapped with satellite or aerial photos during the voxelization stage.

8.3. 3D Antialiasing

The previous sub-section discussed binary voxelization, which generate topologically and geometrically consistent models, but exhibit object space aliasing. These algorithms have used a straightforward method of sampling in space, called *point sampling*. In point sampling, the continuous object is evaluated at the voxel center, and the value of 0 or 1 is assigned to the voxel. Because of this binary classification of the voxels, the resolution of the 3D raster ultimately determines the precision of the discrete model. Imprecise modeling results in jagged surfaces, known as *object space aliasing* (see Figure 10). In this section, a 3D object-space antialiasing technique is presented. It performs antialiasing once, on a 3D view-independent representation, as part of the modeling stage. Unlike antialiasing of 2D scan-converted graphics, where the main focus is on generating aesthetically pleasing displays, the emphasis in antialiased 3D voxelization is on producing alias-free 3D models taht are stored in the view-independent volume buffer for various volume graphics manipulations, including but not limited to the generation of aesthetically pleasing displays.

To reduce object space aliasing, a *volume sampling* technique have been developed [59], which estimates the density contribution of the geometric objects to the voxels. The density of a voxel is attenuated by a filter weight function which is proportional to the distance between the center of the voxel and the geometric primitive. To improve performance, precomputed lookup tables of densities for a predefined set of geometric primitives can be used to select the density value of each voxel. For each voxel visited by the binary voxelization algorithm, the distance to the predefined primitive is used as an index into a lookup table of densities.

Since the voxelized geometric objects are represented as volume rasters of density values, we can essentially treat them as sampled or simulated volume datasets, such as 3D medical imaging datasets, and employ one of many volume rendering techniques for image generation. One primary advantage of this approach is that volume rendering or volumetric global illumination carries the smoothness of the volume-sampled objects from object space over into its 2D projection in image space [61]. Hence, the silhouette of the objects, reflections, and shadows are smooth. Furthermore, by not performing any geometric ray-object intersections or geometric surface normal calculations, the bulk of the rendering time is saved. In addition, CSG operations between two volume-sampled geometric models are accomplished at the voxel level after voxelization, thereby reducing the original problem of evaluating a CSG tree of such operations down to a fuzzy Boolean operation between pairs of non-binary voxels [60] (see Section 11). Volume-sampled models are also suitable for intermixing with sampled datasets, since they can be treated uniformly as one common data representation. Furthermore, volume-sampled models lend themselves to alias-free multi-resolution hierarchy construction [60]. The upper right window in Figure 11 shows a ray traced image of a nut and bolt modeled by CSG operations on volume-sampled geometric objects (see also animation segment). The upper left window is a navigation window allowing interactive speeds of a progressively refined marching cubes approximation of the nut and bolt [2].

9. Texture Mapping

One type of object complexity involves objects that are enhanced with texture mapping, photo mapping, environment mapping, or solid texturing. Texture mapping is commonly implemented during the last stage of the rendering pipeline, and its complexity is proportional to the object's complexity. In volume graphics, texture mapping is performed during the voxelization stage, and the texture color is stored in each voxel in the volume buffer.

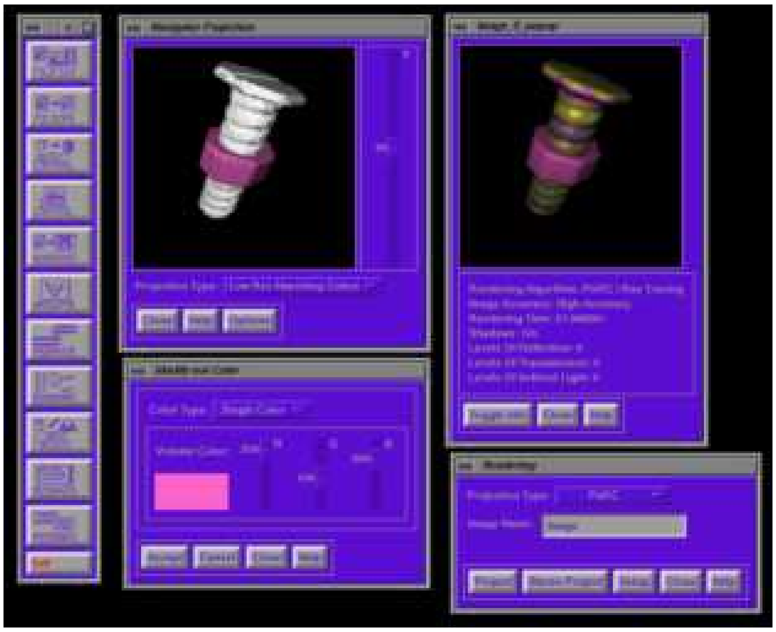

Figure 11: An example *VolVis* session. The nut and bolt are volume-sampled geometric models.

In photo mapping six orthogonal photographs of the real object are projected back onto the voxelized object. Once this mapping is applied, it is stored with the voxels themselves during the voxelization stage, and therefore does not degrade the rendering performance. Texture and photo mapping are also viewpoint independent attributes implying that once the texture is stored as part of the voxel value, texture mapping need not be repeated. This important feature is exploited by voxel-based flight simulators (see Figure 10) and in CAD systems (see Figure 11).

A central feature of volumetric representation is that, unlike surface representation, it is capable of representing inner structures of objects, which can be revealed and explored with appropriate manipulation and rendering techniques. This capability is essential for the exploration of sampled or computed objects. Synthetic objects are also likely to be solid rather than hollow. One method for modeling various solid types is solid texturing, in which a function or a 3D map models the color of the objects in 3D (see Figure 12). During the voxelization phase each voxel belonging to the objects is assigned a value by the texturing function or the 3D map. This value is then stored as part of the voxel information. Again, since this value is view independent, it does not have to be recomputed for every change in the rendering parameters.

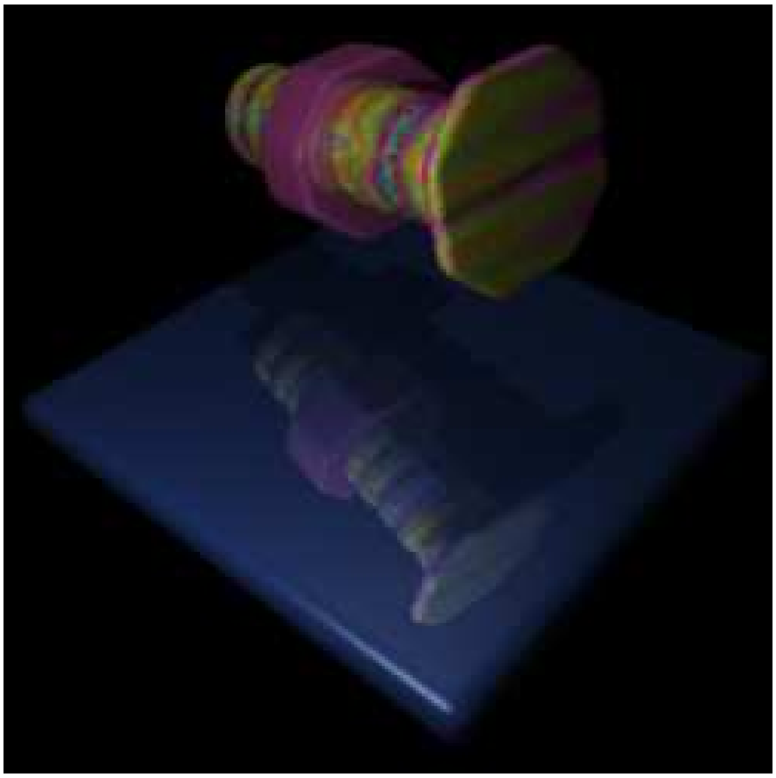

Figure 12: Volume-sampled bolt and nut generated by a sequence of CSG operations on hexagonal, cylindrical, and helix primitives, reflected on a volume-sampled mirror (see animation segment).

10. Amorphous Phenomena

While translucent objects can be represented by surface methods, these methods cannot efficiently support the modeling and rendering of amorphous phenomena (e.g., clouds, fire, smoke) that are volumetric in nature and lack any tangible surfaces. A common modeling and rendering approach is based on a function that, for any input point in 3D, calculates some object features such as density, reflectivity, or color. These functions can then be rendered by ray casting, which casts a ray from each pixel into the function domain. Along the passage of the ray, at constant intervals the function is evaluated to yield a sample. All samples along each ray are combined to form the pixel color. Some examples for the use of this or similar techniques are the rendering of fractals [20], hypertextures [45], fur [25], and gases [15].

The process of function evaluation at each sample point in 3D has to be repeated for each image generated. In contrast, the volumetric approach allows the pre-computation of these functions at each grid point of the volume buffer. The resulting volumetric dataset can then be rendered from multiple viewpoints without recomputing the modeling function. As in other volume graphics techniques, accuracy is traded for speed, due to the resolution limit. Instead of accurately computing the function at each sample point, some type of interpolation from the precomputed grid values is employed.

11. Block Operations and Constructive Solid Geometry

An intrinsic characteristic of the volume buffer is that adjacent objects in the scene are also represented by neighboring memory cells. Therefore, rasters lend themselves to various meaningful grouping-based operations, such as *bitblt* in 2D, or *voxblt* in 3D [31]. These include transfer of volume buffer rectangular blocks (cuboids) while supporting voxel-by-voxel operations between source and destination blocks. Block operations add a variety of modeling capabilities which aid in the task of image synthesis and form the basis for the efficient implementation of a 3D "room manager", which is the extension of window management to the third dimension.

Since the volume buffer lends itself to Boolean operations that can be performed on a voxel-by-voxel basis during the voxelization stage, it is very advantageous to use CSG as the modeling paradigm. Subtraction, union, and intersection operations between two voxelized objects are accomplished at the voxel level, thereby reducing the original problem of evaluating a CSG tree during rendering time down to a 1D Boolean operation between pairs of voxels during a preprocessing stage.

For two point-sampled binary objects the Boolean operations are trivially defined. However, the Boolean operations of CSG or *voxblt* applied to volume-sampled models are analogous to those of fuzzy set theory (cf. [12]). The volume-sampled model is a density function $d(x)$ over R^3, where d is 1 inside the object, 0 outside the object, and $0 < d < 1$ within the soft region of the filtered surface. Some of the common operations, intersection, complement, difference, and union, between two objects A and B are defined as follows:

$$d_{A \cap B}(x) \equiv \min\left(d_A(x), d_B(x)\right) \tag{17}$$

$$d_{\bar{A}}(x) \equiv 1 - d_A(x) \tag{18}$$

$$d_{A-B}(x) \equiv \min\left(d_A(x), 1 - d_B(x)\right) \tag{19}$$

$$d_{A \cup B}(x) \equiv \max\left(d_A(x), d_B(x)\right) \tag{20}$$

The only law of set theory that is no longer true is the excluded-middle law (i.e., $A \bigcap \bar{A} \neq \phi$ and $A \bigcup \bar{A} \neq Universe$). The use of the min and max functions causes discontinuity at the region where the soft regions of the two objects meet, since the density value at each location in the region is determined solely by one of the two overlapping objects.

Complex geometric models can be generated by performing the CSG operations in Equations 17-20 between volume-sampled primitives. Volume-sampled models can also function as matte volumes [11] for various matting operations, such as performing cut-aways and merging multiple volumes into a single volume using union operation. However, in order to preserve continuity on the cut-away boundaries between the material and the empty space, one should use an alternative set of Boolean operators based on algebraic sum and algebraic product [12, 45] :

$$d_{A \bigcap B}(x) \equiv d_A(x)\ d_B(x) \tag{21}$$

$$d_{\bar{A}}(x) \equiv 1 - d_A(x) \tag{22}$$

$$d_{A-B}(x) \equiv d_A(x) - d_A(x)\ d_B(x) \tag{23}$$

$$d_{A \bigcup B}(x) \equiv d_A(x) + d_B(x) - d_A(x)\ d_B(x) \tag{24}$$

Unlike the min and max operators, algebraic sum and product operators result in $A \bigcup A \neq A$, which is undesireable. A consequence, for example, is that during modeling via sweeping, the resulting model is sensitive to the sampling rate of the swept path [60].

Once a CSG model has been constructed in voxel representation, it is rendered in the same way any other volume buffer is. This makes, for example, volumetric ray tracing of constructive solid models straightforward [50] (see Figures 10, 12 and 13 and animation segment).

12. Surface Graphics vs. Volume Graphics

Contemporary 3D graphics has been employing an object-based approach at the expense of maintaining and manipulating a display list of geometric objects and regenerating the frame-buffer after every change in the scene or viewing parameters. This approach, termed *surface graphics*, is supported by powerful geometry engines, which have flourished in the past decade, making surface graphics the state-of-the-art in 3D graphics.

Surface graphics strikingly resembles vector graphics that prevailed in the sixties and seventies, based on vector drawing devices. Like vector graphics, surface graphics represents the scene as a set of geometric primitives kept in a display list. In surface graphics, these primitives are transformed, mapped to screen coordinates, and converted by scan-conversion algorithms into a discrete set of pixels. Any change to the scene, viewing parameters, or shading parameters requires the image generation system to repeat this process. Like vector graphics that did not support painting the interior of 2D objects, surface graphics generates merely the surfaces of 3D objects and does not support the rendering of their interior.

Instead of a list of geometric objects maintained by surface graphics, volume graphics employs a 3D volume buffer as a medium for the representation and manipulation of 3D scenes. A 3D scene is discretized earlier in the image generation sequence, and the resulting 3D discrete form is used as a

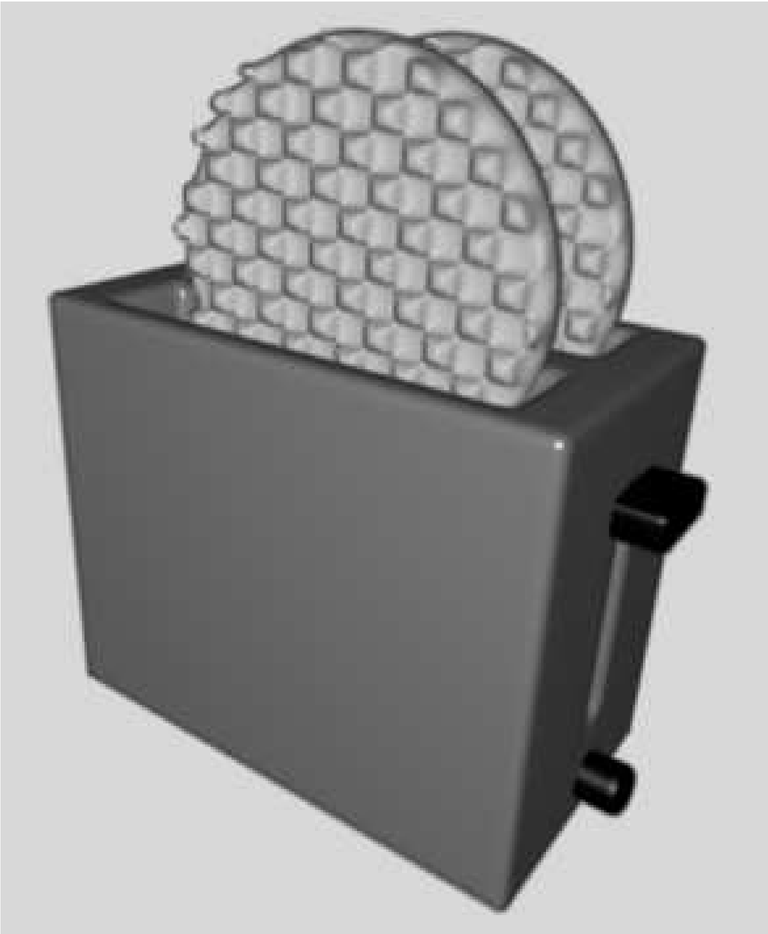

Figure 13: CSG modeling of a toaster and waffles. The waffles were formed by CSG operations of subtracting a tesselation of cubes from two disks.

database of the scene for manipulation and rendering purposes, which in effect decouples discretization from rendering (viewing and shading). Furthermore, all objects are converted into one uniform meta-object – the voxel. Each voxel is atomic and represents the information about at most one object that resides in that voxel.

Volume graphics offers the same benefits as surface graphics, with several advantages that are due to the decoupling, uniformity, and atomicity features. The rendering phase is viewpoint independent and insensitive to scene complexity and object complexity. It supports Boolean and block operations and constructive solid modeling. When 3D sampled and simulated data are used, such as that generated by medical scanners (e.g., CT, MRI) or scientific simulations (e.g., CFD), volume graphic is suitable for their representation too. It is capable of representing amorphous phenomena and both the interior and exterior of 3D objects. These features of volume graphics as compared with surface graphics are discussed in detail in Section 13. Several weaknesses of volume graphics are related to the discrete nature of the representation, for instance, transformations and shading are performed in discrete space. In addition, this approach requires substantial amounts of storage space and specialized processing. These weaknesses are discussed in detail in Section 14.

Table 1: Comparison between vector graphics and raster graphics and between surface graphics and volume graphics.

2D	Vector Graphics	Raster Graphics
Scene/object complexity	−	+
Block operations	−	+
Sampled data	−	+
Interior	−	+
Memory and processing	+	−
Aliasing	+	−
Transformations	+	−
Objects	+	−
3D	Surface Graphics	Volume Graphics

Table 1 contrasts vector graphics with raster graphics. A primary appeal of raster graphics is that it decouples image generation from screen refresh, thus making the refresh task insensitive to the scene and object complexities. In addition, the raster representation lends itself to block operations, such as *bitblt* and CSG. Raster graphics is also suitable for displaying 2D sampled digital images, and thus provides the ideal environment for mixing digital images with synthetic graphic. Unlike vector graphics, raster graphics provides the capability to present shaded and textured surfaces, as well as line drawings. These advantages, coupled with advances in hardware and the development of antialiasing methods, have led raster graphics to supersede vector graphics as the primary technology for computer graphics. The main weaknesses of raster graphics are the large memory and processing power it requires, and the discrete nature of the image. These difficulties delayed the full acceptance of raster graphics until the late seventies when the technology was able to provide cheaper and faster memory and hardware to support the demands of the raster approach. In addition, the discrete nature of rasters makes them less suitable for geometric operations such as transformations and accurate measurements, and once discretized the notion of objects is lost.

The same appeal that drove the evolution of the computer graphics world from vector graphics to raster graphics, once the memory and processing power became available, is driving a variety of applications from a surface-based approach to a voxel-based approach. Naturally, this trend first appeared in applications involving sampled or computed 3D data, such as 3D medical imaging and scientific visualization, in which the datasets are in volumetric form. The diverse empirical applications of volume visualization still provide a major driving force for advances in volume graphics.

The comparison in Table 1 between vector graphics and raster graphics strikingly resembles a comparison between surface graphics and volume graphics. Actually Table 1 itself is used also to contrast surface graphics and volume graphics. Section 3 discusses the features of volume graphics: its insensitivity to scene and object complexities, its viewpoint independence, its ability to represent sampled and simulated datasets, its ability to represent inner information and amorphous phenomena such as clouds and smoke. and its ability to support various block operations. Section 14 discusses the weaknesses of volume graphics relative to surface graphics, which are due to the memory and processing power it requires, its discrete form, and the lack of geometric information.

13. Volume Graphics Features

Insensitivity to Scene Complexity

One of the most appealing attributes of volume graphics is its insensitivity to the complexity of the scene, since all objects have been pre-converted into a finite size volume buffer. Although the performance of the pre-processing voxelization phase is influenced by the scene complexity [8, 26-28], rendering performance depends mainly on the constant resolution of the volume buffer and not on the number of objects in the scene. Insensitivity to the scene complexity makes the volumetric approach especially attractive for scenes consisting of a large number of objects.

Insensitivity to Object Complexity

In volume graphics, rendering is decoupled from voxelization and all objects are first converted into one meta object, the voxel, which makes the rendering process insensitive to the complexity of the objects. Thus, volume graphics is particularly attractive for objects that are hard to render using conventional graphics systems. Examples of such objects include curved surfaces of high order and fractals which require the expensive computation of an iterative function for each volume unit [44]. Constructive solid models are also hard to render by conventional methods especially those that mix data and geometric primitives, but are straightforward to render in volumetric representation (see below).

Viewpoint Independence

Anti-aliasing and texture mapping are commonly implemented during the last stage of the rendering pipeline, and their complexity is proportional to object complexity. Solid texturing, which employs a 3D texture image, has also a high complexity proportional to object complexity. In volume graphics, however, anti-aliasing, texture mapping, and solid texturing are performed only once, during the voxelization stage, where the color is calculated and stored in each voxel. The texture can also be stored as a separate volumetric entity which is rendered together with the volumetric object, as in the *VolVis* software system for volume visualization [1].

The textured objects in Figures 10, 11, and 12 have been assigned texture during the voxelization stage by mapping each voxel back to the corresponding value on a texture map. Once this mapping is applied, it is stored with the voxels themselves during the voxelization stage, which does not degrade the rendering performance. In addition, texture mapping and photo mapping are also viewpoint independent attributes, implying that once the texture is stored as part of the voxel value, texture mapping need not be repeated.

In anticipation of repeated access to the volume buffer (such as in animation), all viewpoint independent attributes can be precomputed during the voxelization stage, stored with the voxel, and be readily accessible for speeding up the rendering. The voxelization algorithm can generate for each object voxel its color, its texture color, its normal vector (for visible voxels), antialising information [59], and information concerning the visibility of the light sources from that voxel. Actually, the viewpoint independent parts of the illumination equation, can also be precomputed and stored as part of the voxel value.

Once a volume buffer with precomputed view-independent attributes is available, a rendering algorithm such as a discrete ray tracing or a volumetric ray tracing algorithm can be engaged. Either ray tracing approach is especially attractive for complex surface scenes and constructive solid models, as well as 3D sampled and computed datasets (see below). Figures 4(c), 5, 7, 8, 10 and 12 show examples of objects that were ray traced in discrete voxel space. In spite of the complexity of these scenes, volumetric ray tracing time was approximately the same as for much simpler scenes and significantly faster than traditional space-subdivision ray tracing methods. Moreover, in spite of the discrete nature of the volume buffer representation, images indistinguishable from the ones produced by conventional surface-based ray tracing can be generated by employing auxiliary object information and screen supersampling techniques.

Sampled and Simulated Datasets

Sampled datasets, such as in 3D medical imaging (see Figures 1, 2, 8 and 12), volume microscopy (see Figures 2, 4 and 7), and geology, and simulated datasets, such as in computational fluid dynamics, chemistry (see Figure 4), and materials simulation are often reconstructed from the acquired sampled or simulated points into a regular grid of voxels and stored in a volume buffer. Such datasets provide for the majority of applications using the volumetric approach. Unlike surface graphics, volume graphics naturally and directly supports the representation, manipulation, and rendering of such datasets, as well as provides the volume buffer medium for intermixing sampled or simulated datasets with geometric objects [29], as can be seen in Figures 5, 7, 8 and 12. For compatibility between the sampled/computed data and the voxelized geometric object, the object can be volume sampled [59] with the same density frequency as the acquired or simulated datasets. In volume sampling the continuous object is filtered during the voxelization stage generating alias-free 3D density primitives. Volume graphics also naturally supports the rendering of translucent volumetric datasets (see Figure 14).

Inner Information

A central feature of volumetric representation is that, unlike surface representation, it is capable of representing inner structures of objects, which can be revealed and explored with the appropriate manipulation and rendering techniques. Natural objects as well as synthetic objects are likely to be solid rather than hollow. The inner structure is easily explored using volume graphics and cannot be supported by surface graphics (see Figures 2(b), 6, 8 and 14). Moreover, while translucent objects can be represented by surface methods, these methods cannot efficiently support the translucent rendering of volumetric objects (see Figure 14), or the modeling and rendering of amorphous phenomena (e.g., clouds, fire, smoke) that are volumetric in nature and do not contain any tangible surfaces [15, 25, 45].

Block Operations

An intrinsic characteristic of rasters is that adjacent objects in the scene are also represented by neighboring voxels. Therefore, rasters lend themselves to various meaningful block-based operations which can be performed during the voxelization stage. For example, the 3D counterpart of the *bitblt* operations, termed *voxblt* (voxel block-transfer), can support transfer of cuboidal voxel blocks with a variety of voxel-by-voxel operations between source and destination blocks [31]. This property is very useful for *voxblt* and CSG. Once a CSG model has been constructed in voxel representation, it is rendered like any other volume buffer. This makes rendering of constructive solid models straightforward.

The spatial presortedness of the volume buffer voxels lends itself to other types of grouping or aggregation of neighboring voxels. For example, the terrain images shown in Figure 10 was generated

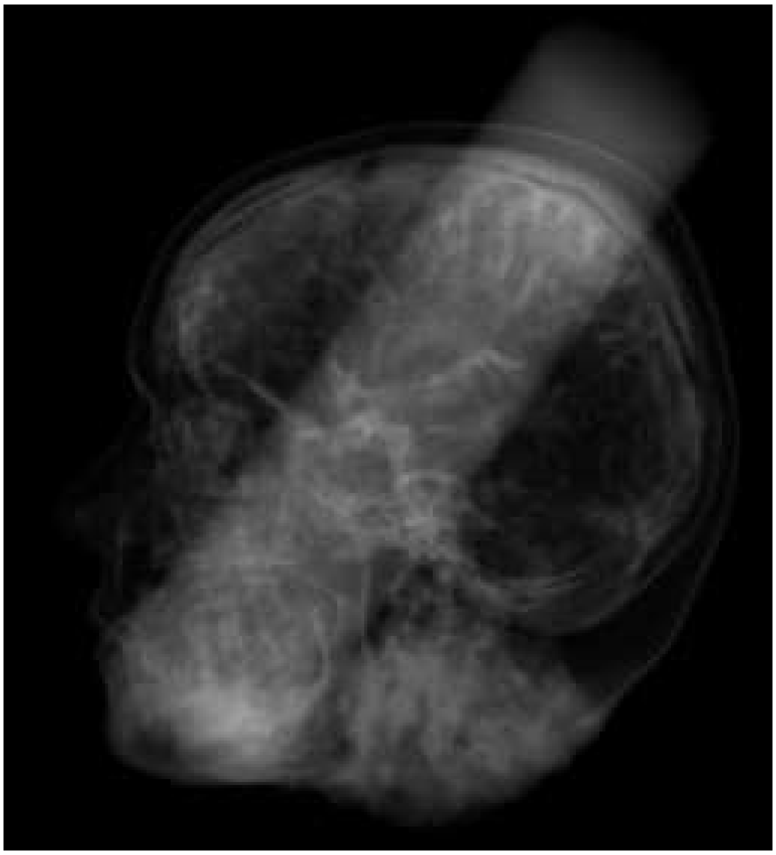

Figure 14: Intermixing of a volume-sampled cylinder with an MRI head using a union operation.

by the voxel-based Hughes Aircraft Co. flight simulator [64]. It simulates a flight over voxel-represented terrain enhanced with satellite or aerial photo mapping with additional synthetic raised objects, such as buildings, trees, vehicles, aircraft, clouds and the like. Since the information below the terrain surface is invisible, terrain voxels can be actually represented as tall cuboids extending from sea level to the terrain height. The raised and moving objects, however, have to be represented in a conventional voxel-based form.

Similarly, voxels can be aggregated into super-voxels in a pyramid-like hierarchy. For example, in a voxel-based flight simulator, the best resolution can be used for takeoff and landing. As the aircraft ascends, fewer and fewer details need to be processed and visualized, and a lower resolution suffices. Furthermore, even in the same view, parts of the terrain close to the observer are rendered at high resolution which deceases towards the horizon. A hierarchical volume buffer can be prepared in advance or on-the-fly by subsampling or averaging the appropriate size neighborhoods of voxels.

14. Weaknesses of Volume Graphics

Memory Size

A typical volume buffer occupies a large amount of memory. For example, for a resolution of 512^3, two bytes per voxel, the volume buffer consists of 256M bytes. However, since computer memories are significantly decreasing in price and increasing in their compactness and speed, such large memories are becoming common place. This argument echoes a similar discussion when raster graphics emerged as a technology in the mid-seventies. With the rapid progress in memory price and compactness, it is safe to predict that, as in the case of raster graphics, the memory will soon cease to be a stumbling block for volume graphics.

Processing Power

Yet, the extremely large throughput that has to be handled requires a special architecture and processing attention (see [30] Chapter 6). *Volume engines*, analogous to the currently available geometry engines, are emerging. Because of the presortedness of the volume buffer and the fact that only a simple single type of object has to be handled, volume engines are conceptually simpler to implement than current geometry engines. We predict that, consequently, volume engines will materialize in the near future, with capabilities to synthesize, load, store, manipulate, and render volumetric scenes in real time (e.g., 30 frames/sec), configured possibly as accelerators or co-systems to existing geometry engines.

Discrete Form

Unlike surface graphics, in volume graphics the 3D scene is represented in discrete form. This is the cause of many of the problems of voxel-based graphics, which are similar to those of 2D rasters [14]. The finite resolution of the raster poses a limit on the accuracy of some operations, such as volume and area measurements, that are based on voxel counting.

Since the discrete data is sampled during rendering, a low resolution volume yields high aliasing artifacts. This becomes especially apparent when zooming in on the 3D raster. When naive rendering algorithms are used, holes may appear between voxels. Nevertheless, this can be alleviated in ways similar to those adopted by 2D raster graphics, such as employing either reconstruction techniques, a higher-resolution volume buffer, or volume sampling.

Manipulation and transformation of the discrete volume are difficult to achieve without degrading the image quality or losing some information. Rotation of rasters by angles other than 90 degrees is especially problematic since a sequence of consecutive rotations will distort the image. Again, these can be alleviated in ways similar to the 2D raster techniques.

Lack of Geometric Information

Once an object has been voxelized, the voxels comprising the discrete object do not retain any geometric information regarding the geometric definition of the object. Thus, it is advantageous, when exact measurements are required (e.g., distance, area), to employ conventional modeling where the geometric definition of the object is available. A voxel-based object is only a discrete approximation of the original continuous object where the volume buffer resolution determines the precision of such measurements. On the other hand, several measurement types are more easily computed in voxel space (e.g., mass property, adjacency detection, and volume computation).

The lack of geometric information in the voxel may inflict other difficulties, such as surface normal computation. In voxel-based models, a discrete shading method is commonly employed to estimate the normal from a context of voxels. A variety of image-based and object-based methods for normal estimation from volumetric data has been devised (see [67], [30, Chapter 4]). Most methods based on fitting some type of a surface primitive to a small neighborhood of voxels.

A partial integration between surface and volume graphics is conceivable as part of an object-based approach in which an auxiliary object table, consisting of the geometric definition and global attributes of each object, is maintained in addition to the volume buffer. Each voxel consists of an index to the object table. This allows exact calculation of normal, exact measurements, and intersection verification for discrete ray tracing [70]. The auxiliary geometric information might be useful also for re-voxelizing the scene in case of a change in the scene itself.

15. Conclusions

This paper has presented many of the important concepts and computational methods of volume visualization. Surface rendering algorithms for volume data were briefly described in which an intermediate representation of the data is used to generate an image of a surface contained within the data. Object order, image order, and hybrid volume rendering techniques were presented for generating images of surfaces within the data, as well as volume rendered images that attempt to represent all three dimensions of information in the 2D image. Several optimization techniques that aim to decrease the rendering time for volume visualization as well as realistic global illumination rendering were also described in this paper.

Although volumetric representations and visualization techniques seem more natural for sampled or computed data sets, their advantages are also attracting traditional geometric-based applications. This trend implies an expanding role for volume visualization, and it has thus the potential to revolutionize the field of computer graphics, by providing a better alternative to surface graphics, called volume graphics. We have introduced recent trends in volume visualization that brought about the emergence of volume graphics. As summarized in Table 1, volume graphics has advantages over surface graphics by being viewpoint independent, insensitive to scene and object complexity, and lends itself to the realization of block operations, CSG modeling, and hierarchical representation. It is suitable for the representation of sampled and simulated datasets and their intermixing with geometric objects, and it supports the visualization of internal structures. The problems associated with the volume buffer representation, such as memory size, processing time, aliasing, and lack of geometric representation, echo problems encountered when raster graphics emerged as an alternative technology to vector graphics and can be alleviated in similar ways.

The progress so far in volume graphics, in computer hardware, and memory systems, coupled with the desire to reveal the inner structures of volumetric objects, suggests that volume graphics may develop into a major trend in computer graphics. Just as raster graphics in the seventies superseded vector graphics for visualizing surfaces, volume graphics has the potential to supersede surface graphics for handling and visualizing volumes as well as for modeling and rendering synthetic scenes composed of surfaces.

Acknowledgments

Special thanks are due to Lisa Sobierajski, Roni Yagel, Dany Cohen, Sid Wang, Taosong He, and Hanspeter Pfister who made substantial contributions to this paper, and co-authored with me realted papers [2, 32-34]. This work has been supported by the National Science Foundation under grant CCR-9205047, and a grant from the Deaprtment of Energy under PICS grant. The MRI head data in Figures 4, 8 and 14 is courtesy of Siemens Medical Systems, Inc., Iselin, NJ. Figure 10 is courtesy of Hughes Aircraft Company, Long Beach, CA. This image has been voxelized using voxelization algorithms, a voxel-based modeler, and a photo-mapper developed at Stony Brook Visualization Lab. The protein simulated data in Figure 4 is courtesy of Scripps Clinic, La Jolla, CA. The confocal microscope data in Figures 2, 4 and 7 are courtesy of Howard Hughes Medical Institute at Stony Brook, NY. The CT lobster data in Figure 6 is courtesy of AVS, Waltham, MA. Thanks are due to Lisa Sobierajski, Sid Wang, and Rick Avila who helped with the *VolVis* software and the production of many of the color pictures of this paper. *VolVis* can be obtained by sending email to: volvis@cs.sunysb.edu.

16. References

1. Avila, R., Sobierajski, L. and Kaufman, A., ‘‘Towards a Comprehensive Volume Visualization System’’, *Visualization ’92 Proceedings*, October 1992, 13-20.

2. Avila, R., He, T., Hong, L., Kaufman, A., Pfister, H., Silva, C., Sobierajski, L. and Wang, S., ‘‘VolVis: A Diversified Volume Visualization System’’, *Visualization ’94 Proceedings*, Washington, DC, October 1994.

3. Barillot, C., Gibaud, B., Luo, L. M. and Scarabin, I. M., ‘‘3D Representation of Anatomic Structures From CT Examinations’’, *Proceedings SPIE*, **602**, (1985), 307-314.

4. Chen, L. S., Herman, G. T., Reynolds, R. A. and Udupa, J. K., ‘‘Surface Shading in the Cuberille Environment’’, *IEEE Computer Graphics & Applications*, **5**, 12 (December 1985), 33-43.

5. Chiueh, T., He, T., Kaufman, A. and Pfister, H., ‘‘Compression Domain Volume Rendering’’, Technical Report 94.01.04, Computer Science, SUNY at Stony Brook, April 1994.

6. Cline, H. E., Lorensen, W. E., Ludke, S., Crawford, C. R. and Teeter, B. C., ‘‘Two Algorithms for the Three-Dimensional Reconstruction of Tomograms’’, *Medical Physics*, **15**, 3 (May/June 1988), 320-327.

7. Cohen, M. F., Chen, S. E., Wallace, J. R. and Greenberg, D. P., ‘‘A Progressive Refinement Approach to Fast Radiosity Image Generation’’, *Computer Graphics (Proc SIGGRAPH)*, 1988, 75-84.

8. Cohen, D. and Kaufman, A., ‘‘Scan Conversion Algorithms for Linear and Quadratic Objects’’, in *Volume Visualization*, A. Kaufman, (ed.), IEEE Computer Society Press, Los Alamitos, CA, 1990, 280-301.

9. Cohen, D. and Shaked, A., ‘‘Photo-Realistic Imaging of Digital Terrain’’, *Proceedings Eurographics ’93 Conference*, Barcelona, Spain, September 1993, 363-374.

10. Danielsson, P. E., ‘‘Incremental Curve Generation’’, *IEEE Transactions on Computers*, **C-19**, (1970), 783-793.

11. Drebin, R. A., Carpenter, L. and Hanrahan, P., ‘‘Volume Rendering’’, *Computer Graphics (Proc. SIGGRAPH)*, **22**, 4 (August 1988), 65-74.

12. Dubois, D. and Prade, H., *Fuzzy Sets and Systems: Theory and Applications*, Academic Press, 1980.

13. Dunne, S., Napel, S. and Rutt, R., "Fast Reprojection of Volume Data", *Proceedings of the First Conference on Visualization in Biomedical Computing*, Atlanta, GA, May 1990, 11-18.

14. Eastman, C. M., "Vector versus Raster: A Functional Comarison of Drawing Technologies", *IEEE Computer Graphics & Applications*, **10**, 5 (September 1990), 68-80.

15. Ebert, D. S. and Parent, R. E., "Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques", *Computer Graphics*, **24**, 4 (August 1990), 367-376.

16. Frieder, G., Gordon, D. and Reynolds, R. A., "Back-to-Front Display of Voxel-Based Objects", *IEEE Computer Graphics and Applications*, **5**, 1 (January 1985), 52-60.

17. Glassner, A. S., "Space Subdivision for Fast Ray Tracing", *IEEE Computer Graphics and Applications*, **4**, 10 (October 1984), 15-22.

18. Gordon, D. and Reynolds, R. A., "Image Space Shading of 3-Dimensional Objects", *Computer Vision, Graphics and Image Processing*, **29**, (1985), 361-376.

19. Hanrahan, P., Salzman, D. and Aupperle, L., "A Rapid Hierarchical Radiosity Algorithm", *Computer Graphics*, **25**, 4 (July 1991), 197-206.

20. Hart, J. C., Sandin, D. J. and Kauffman, L. H., "Ray Tracing Deterministic 3-D Fractals", *Computer Graphics*, **23**, 3 (July 1989), 289-296.

21. Herman, G. T. and Liu, H. K., "Three-Dimensional Display of Human Organs from Computed Tomograms", *Computer Graphics and Image Processing*, **9**, 1 (January 1979), 1-21.

22. Herman, G. T. and Udupa, J. K., "Display of Three Dimensional Discrete Surfaces", *Proceedings of the SPIE*, **283**, (1981), 90-97.

23. Hoehne, K. H. and Bernstein, R., "Shading 3D-Images from CT Using Gray-Level Gradients", *IEEE Transactions on Medical Imaging*, **MI-5**, 1 (March 1986), 45-47.

24. Hottel, H. C. and Sarofim, A. D., *Radiative Transfer*, McGraw-Hill, New York, New York, 1967.

25. Kajiya, J. T. and Kay, T. L., "Rendering Fur with Three Dimensional Textures", *Computer Graphics*, **23**, 3 (July 1989), 271-280.

26. Kaufman, A. and Shimony, E., "3D Scan-Conversion Algorithms for Voxel-Based Graphics", *Proc. ACM Workshop on Interactive 3D Graphics*, Chapel Hill, NC, October 1986, 45-76.

27. Kaufman, A., "An Algorithm for 3D Scan-Conversion of Polygons", *Proc. EUROGRAPHICS'87*, Amsterdam, Netherlands, August 1987, 197-208.

28. Kaufman, A., "Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes", *Computer Graphics*, **21**, 4 (July 1987), 171-179.

29. Kaufman, A., Yagel, R. and Cohen, D., "Intermixing Surface and Volume Rendering", in *3D Imaging in Medicine: Algorithms, Systems, Applications*, K. H. Hoehne, H. Fuchs and S. M. Pizer, (eds.), June 1990, 217-227.

30. Kaufman, A., *Volume Visualization*, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1991.

31. Kaufman, A., "The *voxblt* Engine: A Voxel Frame Buffer Processor", in *Advances in Graphics Hardware III*, A. A. M. Kuijk, (ed.), Springer-Verlag, Berlin, 1992, 85-102.

32. Kaufman, A., Cohen, D. and Yagel, R., "Volume Graphics", *IEEE Computer*, **26**, 7 (July 1993), 51-64.

33. Kaufman, A., Yagel, R. and Cohen, D., ‘‘Modeling in Volume Graphics’’, in *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunii, (eds.), Springer-Verlag, June 1993, 441-454.

34. Kaufman, A. and Sobierajski, L., ‘‘Continuum Volume Display’’, in *Engineering Visualization*, R. S. Gallagher, (ed.), CRC Press, 1994.

35. Kruger, W., ‘‘The Application of Transport Theory to Visualization of 3-D Scalar Data Fields’’, *Computers in Physics*, July/August 1991, 397-406.

36. Laur, D. and Hanrahan, P., ‘‘Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Renderi ng’’, *Computer Graphics*, **25**, 4 (July 1991), 285-288.

37. Lee, Y. T. and Requicha, A. A. G., ‘‘Algorithms for Computing the Volume and Other Integral Properties of Solids: I-Known Methods and Open Issues; II-A Family of Algorithms Based on Representation Conversion and Cellular Approximation’’, *Communications of the ACM*, **25**, 9 (September 1982), 635-650.

38. Levoy, M., ‘‘Display of Surfaces from Volume Data’’, *Computer Graphics and Applications*, **8**, 5 (May 1988), 29-37.

39. Levoy, M., ‘‘Volume Rendering using the Fourier Projection-Slice Theorem’’, *Graphics Interface ’92*, 1992, 61-69.

40. Lorensen, W. E. and Cline, H. E., ‘‘Marching Cubes: A High Resolution 3D Surface Construction Algorithm’’, *Computer Graphics*, **21**, 4 (July 1987), 163-169.

41. Malzbender, T. and Kitson, F., ‘‘A Fourier Technique for Volume Rendering’’, *Focus on Scientific Visualization*, 1991, 305-316.

42. Malzbender, T., ‘‘Fourier Volume Rendering’’, *ACM Transactions of Graphics*, **12**, 3 (July 1993), 233-250.

43. Mokrzycki, W., ‘‘Algorithms of Discretization of Algebraic Spatial Curves on Homogeneous Cubical Grids’’, *Computers & Graphics*, **12**, 3/4 (1988), 477-487.

44. Norton, V. A., ‘‘Generation and Rendering of Geometric Fractals in 3-D’’, *Computer Graphics*, **16**, 3 (1982), 61-67.

45. Perlin, K. and Hoffert, E. M., ‘‘Hypertexture’’, *Computer Graphics*, **23**, 3 (July 1989), 253-262.

46. Rushmeier, H. E. and Torrance, K. E., ‘‘The Zonal Method For Calculating Light Intensities in the Presence of a Participating Medium’’, *Computer Graphics*, **21**, 4 (July 1987), 293-306.

47. Sabella, P., ‘‘A Rendering Algorithm for Visualizing 3D Scalar Fields’’, *Computer Graphics (Proc. SIGGRAPH)*, **22**, 4 (August 1988), 160-165.

48. Snyder, J. M. and Barr, A. H., ‘‘Ray Tracing Complex Models Containing Surface Tessellations’’, *Computer Graphics*, **21**, 4 (July 1987), 119-128.

49. Sobierajski, L., Cohen, D., Kaufman, A., Yagel, R. and Acker, D., ‘‘A Fast Display Method for Volumemtric Data’’, *The Visual Computer*, **10**, 2 (1993), 116-124.

50. Sobierajski, L. and Kaufman, A., ‘‘Volumetric Ray Tracing’’, Technical Report 94.03.31, Computer Science, SUNY Stony Brook, March 1994.

51. Sobierajski, L. and Kaufman, A., ‘‘Volumetric Radiosity’’, Technical Report 94.01.05, Computer Science, SUNY Stony Brook, January 1994.

52. Tiede, U., Hoehne, K. H. and Riemer, M., ‘‘Comparison of Surface Rendering Techniques for 3D Tomographics Objects’’, *Proceedings NCGA ’88 Conference*, Berlin Heidelberg New York , 1987, 599-610 .

53. Tiede, U., Riemer, M., Bomans, M. and Hoehne, K. H., ‘‘Display Techniques for 3-D Tomographic Volume Data’’, *Proceedings of NCGA’88 Conference*, **III**, (March 1988), 188-197, National Computer Graphics Association.

54. Totsuka, T. and Levoy, M., ‘‘Frequency Domain Volume Rendering’’, *Computer Graphics* , 1993, 271-278.

55. Tuy, H. K. and Tuy, L. T., ‘‘Direct 2-D Display of 3-D Objects’’, *IEEE Computer Graphics & Applications*, **4**, 10 (November 1984), 29-33.

56. Upson, C. and Keeler, M., ‘‘V-BUFFER: Visible Volume Rendering’’, *Computer Graphics (Proc. SIGGRAPH)*, 1988, 59-64.

57. Vannier, M. W., Marsh, J. L. and Warren, J. O., ‘‘Three Dimensional Computer Graphics for Craniofacial Surgical Planning and Evaluation’’, *Computer Graphics*, **17**, 3 (July 1983), 263-273.

58. Wallace, G. K., ‘‘The JPEG Still Picture Compression Standard’’, *Communications of the ACM*, **34**, 4 (April 1991), 30-44.

59. Wang, S. and Kaufman, A., ‘‘Volume Sampled Voxelization of Geometric Primitives’’, *Visualization ’93 Proceedings*, San Jose, CA, October 1993, 78-84.

60. Wang, S. and Kaufman, A., ‘‘Volume-Sampled 3D Modeling’’, *IEEE Computer Graphics & Applications*, September 1994.

61. Wang, S. and Kaufman, A., ‘‘3D Antialiasing’’, Technical Report 94.01.03, Computer Science, SUNY Stony Brook, January 1994.

62. Westover, L., ‘‘Footprint Evaluation for Volume Rendering’’, *Computer Graphics (Proc. SIGGRAPH)*, **24**, 4 (August 1990), 144-153.

63. Whitted, T., ‘‘An Improved Illumination Model for Shaded Display’’, *Communications of the ACM*, **23**, 6 (June 1980), 343-349.

64. Wright, J. and Hsieh, J., ‘‘A Voxel-Based, Forward Projection Algorithm for Rendering Surface and Volumetric Data’’, *Proceedings Visualization ’92*, Boston, MA, October 1992, 340-348.

65. Yagel, R., ‘‘Efficient Methods for Computer Graphics’’, PhD Dissertation, SUNY at Stony Brook, December 1991.

66. Yagel, R., Kaufman, A. and Zhang, Q., ‘‘Realistic Volume Imaging’’, *Proceedings Visualization ’90*, San Diego, CA, October 1991, 226-231.

67. Yagel, R., Cohen, D. and Kaufman, A., ‘‘Normal Estimation in 3D Discrete Space’’, *The Visual Computer*, June 1992, 278-291.

68. Yagel, R., ‘‘High Quality Template-Based Volume Viewing’’, *OSU-CISRC-10/92-TR28*, Department of Computer and Information Science, The Ohio State University, October 1992.

69. Yagel, R. and Kaufman, A., ‘‘Template-Based Volume Viewing’’, *Computer Graphics Forum*, **11**, 3 (September 1992), 153-167.

70. Yagel, R., Cohen, D. and Kaufman, A., ‘‘Discrete Ray Tracing’’, *IEEE Computer Graphics and Applications*, **12**, 5 (September 1992), 19-28.