High Quality Template-Based Volume Rendering

Roni Yagel and Kim Ciula
Department of Computer and Information Science
The Ohio State University

Abstract

Volumetric representation of 3D scenes is becoming widely used in scientific visualization and computer graphics. Volume rendering confronts performance hurdles due to the voluminous nature of the data, and quality hurdles due to its discreteness. In this paper we present a three-phase template-based algorithm for expediting parallel viewing of volumetric datasets. Our approach exploits coherency between rays in parallel views by storing, in a ray-template, information that is common to all rays. From an axis-aligned plane, and within the projected region of the volume, parallel rays are cast into the volume by repeating the sequence of steps specified by the ray-template. We show that the template can also store information that is used to expedite screen-space and object-space supersampling. Finally, we utilize templates to support adaptive volume traversal.

1: Introduction

Visualization is the process of converting complex information to a format that is amenable to human understanding, while maintaining the integrity and accuracy of the information. One type of information that can benefit from visualization techniques is volumetric data which consist of information on three dimensional phenomenon. Volumes are commonly composed of a set of values sampled or computed at the integral points of a 3D Euclidean grid. The process of determining, for a given view point, which parts of the volume are visible is called *volume viewing*. Algorithms for volume viewing operate by scanning through the data points, called *voxels*, in a strict order designed to achieve hidden voxel removal.

Volume viewing algorithms can be classified according to the partial voxel occupancy they support. Many early algorithms [Herman and Liu 1979, Reynolds, Gordon, and Chen 1987, Tuy and Tuy 1984] assume *uniform (binary) occupancy*, that is, a voxel is either fully occupied by some object or it is devoid of any object presence. In contrast to uniform occupancy, methods based on *partial occupancy* assume that data values represent partial voxel occupancy by an object of homogeneous material. This provides a mechanism for the representation and display of objects that are smaller than the acquisition grid or that are not aligned with it[1]. Figure 1 demonstrates, in 2D, the difference between these approaches. In the binary approach ((a) in Figure 1) the voxel value in the center will be assigned to any sample (denoted by $*$) inside the grey frame – implying uniform value across this space unit. In the partial occupancy approach, the value of the sample will be determined by the weighted average of a neighborhood of values ((b), (c) in Figure 1).

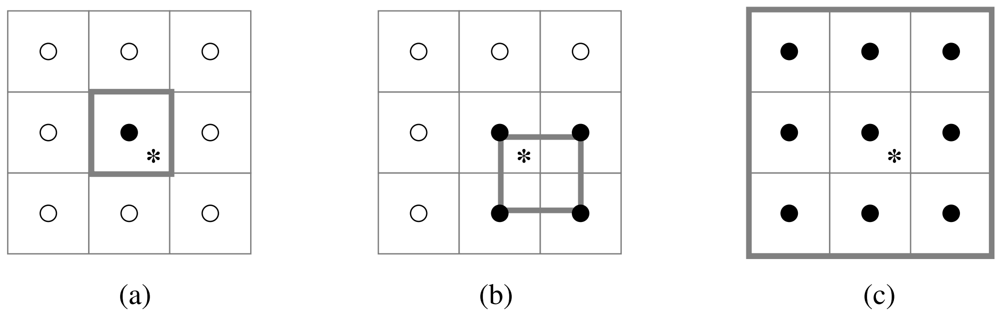


Figure 1: Grid sampling: (a) uniform - the same value for all samples within the grey frame, (b) 2×2 neighborhood variance - the four values influence samples within the grey frame, (c) 3×3 neighborhood variance (all nine values).

There are two main approaches to the implementation of volume viewing. The first one traverses the volume, regarding each voxel as a 3D point. Each voxel is transformed by the viewing matrix and then projected onto a Z-buffer or a compositing buffer [Drebin, Carpenter, and Hanrahan 1988, Westover 1989]. Alternatively, the approach we explore here is based on the *ray casting algorithm* which casts a ray from each pixel on the screen into the volume data along the view vector, accumulating the values of voxels the ray pierces [Bakalash and Kaufman 1989, Drebin, Carpenter, and Hanrahan 1988, Farrell, Yang, and Zappulla 1985, Tuy and Tuy 1984, Upson and Keeler 1988].

The ray-traversal approach has three major variations: parallel (orthographic) ray casting, perspective ray casting, and ray tracing. The first two are variations of ray casting, in which only primary rays (rays from the eye through the screen) are followed. In *parallel ray casting* the eye is at infinity and all rays are parallel to one viewing vector. This viewing scheme is used in applications that do not benefit from perspective distortion, such as biomedicine. Alternatively, ray casting can be implemented to support perspective viewing. Since ray casting follows only primary rays, it does not directly support the simulation of light phenomena such as reflection, shadows, and refraction. As an alternative, the *ray tracing* algorithm can recursively consider both primary and secondary rays and thus can create "photorealistic" images [Yagel, Cohen, and Kaufman 1992].

One disadvantage of the ray casting approach is aliasing caused by point sampling in screen space and object-space. The common solution is to perform supersampling, that is, to shoot several rays from a pixel

1. Partial volume occupancy is also assumed when occupancy fractions are estimated for each of a set of materials that might be present in a voxel [Drebin, Carpenter, and Hanrahan 1988] and when the grey-level gradient is used to compute the normal [Hoehne and Bernstein 1986]

(screen-space supersampling) and to take samples along the ray at smaller intervals (object-space supersampling). In this paper we describe methods for efficiently supersampling in both screen and object-spaces.

The major hurdle confronting any volume renderer is the vast amount of computation needed. For a moderate resolution (250^3) volume, over 15.6 million voxels have to be processed, while in the case of high resolution (500^3 - 1000^3), this number rises to 10^8-10^9 voxels! Before we describe our approach to the solution of this problem, we describe, in more detail, the general ray casting algorithm and survey existing methods for expediting it.

1.1: Volume Viewing by Ray Casting

The ray casting algorithm is rather simple: for each pixel on the screen, send a ray into the scene ((a) in Figure 2). Follow the ray, while sampling the voxels it pierces and accumulating the colors and opacities of those samples. Stop following the ray when it is known that additional samples will not significantly change the ray's final value, that is, when the ray has accumulated an opaque color, or when it is no longer inside the volume ((c) in Figure 2). This basic algorithm is presented in pseudo-code in Figure 3.

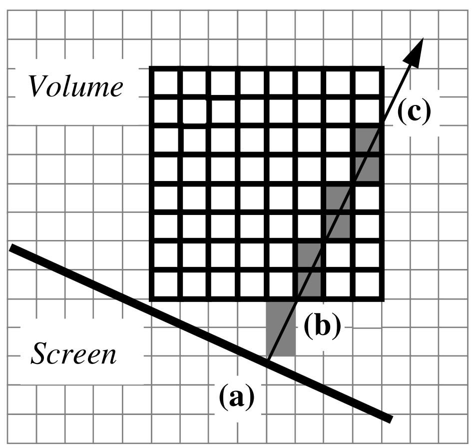


Figure 2: Principles of template-based ray casting, illustrated in 2D. The volume (emphasized grid) is embedded in object-space (dotted grid).

The variable *accumulator* serves as an accumulator of colors and opacities along the ray. The function *ray_-enter_volume* computes the point where the ray emitted at pixel *(i,j)*, ((a) in Figure 2), first meets the volume ((b) in Figure 2). The function *ray_exit_volume* computes the point where the ray leaves the volume ((c) in Figure 2). The function *initialize_line_algorithm* initializes a 3D line algorithm employed to step along the ray. The call to *next_voxel_along_ray* represents the computation needed to make one step along the ray.

The implementation of *initialize_line_algorithm* and *next_voxel_along_ray* is usually based on a 3D variation of the well known 2D incremental line generation algorithm [Cohen and Kaufman 1991]. There are two types of line algorithms that can be employed – *discrete* and *continuous*. The discrete approach uses an integer-based discrete line algorithm that can only determine which voxels are visited by the ray but can not provide information on where the ray enters a voxel and how far it travels inside it. Alternatively, rays can be generated by a floating-point, continuous line algorithm that allows multiple samples per voxel. The discrete approach, which is much faster, is employed when the data voxels are binary-defined, that is, when voxels are assumed to have uniform values across their extent. In this case, the benefit of sampling a voxel more than once is limited since all samples within a single voxel will yield the same value. However, when it is assumed that voxel values smoothly vary across their extent, image quality can be greatly improved by taking multiple samples per voxels (object-space supersampling). This can be supported only if a continuous line algorithm is employed.

```
set all pixels to BACKGROUND
for each screen sample point (i, j) do                          -- (a) in Figure 2.
    [u, v, w] := ray_enter_volume(i, j);                         -- (b) in Figure 2.
    [x, y, z] := ray_exit_volume(i, j);                          -- (c) in Figure 2.
    initialize_line_algorithm([u, v, w], [x, y, z]);
    accumulator := EMPTY;
    while (inside the volume and not opaque(accumulator)) do
        sample := sample_volume_at([u, v, w]);
        accumulator := compose_voxels(sample, accumulator);
        [u, v, w] := next_voxel_along_ray();
    end_while;
    accumulator := compose_voxels(BACKGROUND, accumulator);
    draw_pixel(i, j, accumulator);                               -- (a) in Figure 2.
end_for;
```

Figure 3: Basic ray casting algorithm.

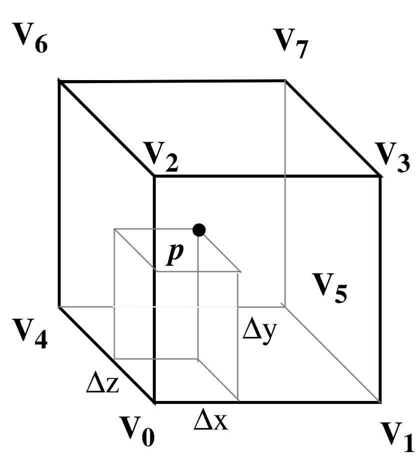


Figure 4: The sample value at point *p* is determined by a weighted sum of the voxel values V_0,..., V_7 in the neighborhood *N(p)* of *p*.

When generating a discrete 3D line we usually can either traverse a *6-connected* or a *26-connected* set of voxels. When the line can only step along one axis at a time the line is called 6-connected (consecutive voxels share one of their 6 faces). When the line can step along one, two, or three axes at a time the line is called 26-connected (consecutive voxels share one of their 26 faces, edges, or vertices).

The sampling of the volume (by the function *sample_volume_at*) can be implemented in various ways. Assume we sample at a point **p** located at (x,y,z). Let us denote the relative displacement of **p** from the voxel origin by $(\Delta x, \Delta y, \Delta z) = (x - \lfloor x \rfloor, y - \lfloor y \rfloor, z - \lfloor z \rfloor)$. A set of voxel values $\mathbf{V}_i$ in the neighborhood of **p**, denoted by N(**p**), is used in the sampling process. Commonly, N(**p**) consists of the eight voxel values at the grid points around **p**, as shown in Figure 4.

The value of the sample at **p**, denoted by S(**p**) is:

$$S(\boldsymbol{p}) = \sum_{i=0}^{\|N(\boldsymbol{p})\|-1} V_i \cdot \omega_i \qquad V_i \in N(\boldsymbol{p}) \tag{1}$$

where ω_i are the weights assigned to each voxel value. The discrete-line method finds the nearest voxel $\mathbf{V}_i$ to **p** and considers only its value by assigning $\omega_i = 1$ and setting $\omega_j = 0$ for all $j \neq i$. In contrast to this zero-order interpolation, the continuous line algorithm can use higher order interpolation (such as trilinear) to estimate the appropriate value [Levoy 1988, Sabella 1988, Upson and Keeler 1988, Westover 1989]. The trilinear interpolation scheme assigns:

$$\omega_0 = -\Delta x - \Delta y - \Delta z + \Delta x\Delta y + \Delta y\Delta z + \Delta x\Delta z - \Delta x\Delta y\Delta z + 1 \tag{2}$$

$$\omega_1 = \Delta x - \Delta x\Delta y - \Delta x\Delta z + \Delta x\Delta y\Delta z \tag{3}$$

$$\omega_2 = \Delta y - \Delta x\Delta y - \Delta y\Delta z + \Delta x\Delta y\Delta z \tag{4}$$

$$\omega_3 = \Delta x\Delta y - \Delta x\Delta y\Delta z \tag{5}$$

$$\omega_4 = \Delta z - \Delta y\Delta z - \Delta x\Delta z + \Delta x\Delta y\Delta z \tag{6}$$

$$\omega_5 = \Delta x\Delta z - \Delta x\Delta y\Delta z \tag{7}$$

$$\omega_6 = \Delta y\Delta z - \Delta x\Delta y\Delta z \tag{8}$$

$$\omega_7 = \Delta x\Delta y\Delta z \tag{9}$$

Alternatively, one could compute the Euclidean distance from V_i to **p**, denoted by d_i, and set $w_i = 1/d_i$, $w_i = 1/d_i^2$ or use some other model for value variation (e.g., [Wyvill and Wyvill 1987]).

Note that the exact implementation of *compose_voxels* may vary from the traditional *first opaque projection* that displays the first non-transparent voxel encountered by the ray [Kaufman and Bakalash 1988], to *maximum/minimum projection* which displays the maximum (minimum) value encountered by the ray.

Alternatively, *weighted additive projection* can be employed which produces an X-ray-like image by averaging the intensities the ray encountered. Assigning opacities to voxel values will enable a *semitransparent compositing projection* to simulate semitransparent volumes such as gases [Drebin, Carpenter, and Hanrahan 1988, Kaufman and Bakalash 1988, Levoy 1989, Sabella 1988, Upson and Keeler 1988]. The algorithm presented here does not assume any specific compositing scheme and can support any of them. As shown in Figure 5, the template algorithm, utilizing continuous rays and semitransparent compositing, was used to render a quantum mechanics calculation of a one-electron orbital of a four-iron, eight-sulfur cluster found in many natural proteins.

Figure 5: A 64^3 volume electron density simulation rendered by the template algorithm utilizing continuous rays and employing semitransparent composition.

1.2: Previous Methods for Expediting Ray Casting

Existing methods for speeding up the process of ray casting usually rely on one or more of the following principles: screen-space coherency, object-space coherency, space-leaping, inter-image coherency, or progressive image generation. We turn now to briefly describe each of those.

Screen-space coherency. There is often a high level of coherency between pixels in image-space. That is, it is highly probable that between two pixels having identical or similar color we will find another pixel having the same (or similar) color. Adaptive image supersampling exploits such screen-space coherency. First, rays are cast from only a subset of the screen pixels (e.g. every other pixel). “Empty pixels” between pixels with similar value are assigned an interpolated value. In areas of high gradient in the pixel colors, additional rays are cast to resolve ambiguities [Levoy 1990a].

Object-space coherency. The extension of screen-space coherency to 3D assumes that there often is coherency between voxels in object-space. Therefore, it should be possible to avoid excessive sampling in 3D regions having uniform or similar values. One method that exploits object-space coherency [van Walsum et al. 1991] starts sampling the volume at low frequency (i.e., large steps between sample points). If the ray encounters a large difference between two adjacent samples, additional samples are taken between them to resolve ambiguities in these high frequency regions. More recently, this basic idea was extended to efficiently lower the sampling rate in areas where only small contributions of opacities are made, or in regions where the volume is relatively homogeneous [Danskin and Hanrahan 1992].

Space-leaping. The passage of a ray through the volume is two-phased. In the first phase the ray advances through the empty space searching for an object. In the second phase the ray integrates colors and opacities as it penetrates the object (in the case of multiple or concave objects these two phases may repeat). Depending on the object's opacity, the second phase may make only a few steps. Since the passage of the ray through empty space does not contribute to the final image, skipping the empty space could provide significant speed-up without affecting image quality.

Several methods have been proposed that take advantage of this observation. One of these methods uses a hierarchical representation (e.g. octree) to decompose the volume into uniform regions that can be represented by nodes in a tree data structure. An adjusted ray traversal algorithm skips the (uniform) empty space by maneuvering through the hierarchical data structure [Levoy 1990b, Samet and Webber 1988].

The empty space does not have to be sampled – it has only to be crossed as fast as possible. Therefore, we have proposed [Yagel et al. 1991c, Yagel, Cohen, and Kaufman 1992] to utilize a fast line algorithm in the empty space (e.g. integer-based 26-connected discrete line algorithm) and another, slower but more accurate (e.g. 6-connected integer or floating point continuous line algorithm), in the vicinity and interior of objects.

The proximity-clouds method [Cohen and Shefer 1993, Zuiderveld, Koning, and Viergever 1992] is based on a preprocessing stage in which, for each empty voxel, the distance to the closest occupied voxel is computed. At rendering time, when a ray traverses the volume it can either encounter an occupied data voxel, to be handled as usual, or a "proximity voxel" containing the value L. This suggests that the ray can safely take a L-step leap forward and be assured that there is no object in the skipped span of voxels.

Inter-Image Coherency. Recently, Yagel and Shi [Yagel and Shi 1993] have developed a method for speeding up the process of volume rendering a sequence of images. It exploits coherency between consecutive images to shorten the path rays take through the volume. This is achieved by using the information in one image to provide each ray with the information needed to leap over most of the empty space in the next image and commence volume traversal in the vicinity of meaningful data.

Progressive Image Generation. One practical solution to the rendering time problem is the generation of partial images that are progressively refined as the user interacts with the crude image. This method starts by first sampling the screen in low resolution. The screen regions where no rays are emitted receive values interpolated from the pixels that were assigned rays. Later, more rays are cast and the interpolated values are replaced by the more accurate results [Levoy 1990a]. Additionally, rays that are intended to cover large screen areas can be traced in the lower-resolution components of a pyramid [Levoy and Whitaker 1990].

Figure 6: A 256^3 volume rendered by the adaptive template algorithm utilizing discrete and continuous rays and supersampling in screen-space (four rays/pixel) and object-space (two samples/voxel).

1.3: Overview

In this paper we describe a volume viewing algorithm for parallel ray casting. In the next section we present the basic algorithm and introduce the template principle. A three-phase algorithm that takes advantage of the template approach is described and is shown to guarantee uniform sampling of the volumetric space. In Section 3 we describe a variation of the basic algorithm that uses discrete rays for rendering volumes with uniform voxel occupancy and show how it supports screen supersampling. In Section 4 we describe another version of the template algorithm that uses continuous rays for the rendition of non-uniform voxels and how it supports object-space supersampling. In Section 5 we describe a third variation of the algorithm that can adaptively alternate between various ray types and sampling rates with essentially no performance penalty.

Section 6 lists performance measurements and provides explanation to the algorithm behavior. We conclude that the proposed algorithm is attractive, as it provides excellent performance when rendering discrete scenes as well as when continuous rays and super-sampling are required, without giving up accuracy. Figure 6 shows a 256^3 MRI dataset that was rendered with the adaptive ray casting method.

The template-based approach to parallel ray casting was also shown to have an efficient hardware implementation [Yagel 91] and was recently shown to have an elegant implementation on a massively parallel architecture (CM5) [Schroeder and Krueger 1993].

2: The Template-Based Approach to Ray Casting

In parallel viewing, where the observer is placed at infinity, all rays have exactly the same form. Therefore, there is no need to reactivate a line algorithm for each ray (as done in Figure 3). Instead, we compute the *form* of the ray once and store it in a data structure called a *ray-template*. All rays can then be generated by following the ray template. This approach has great advantages both in terms of performance and in terms of accuracy, as we show later.

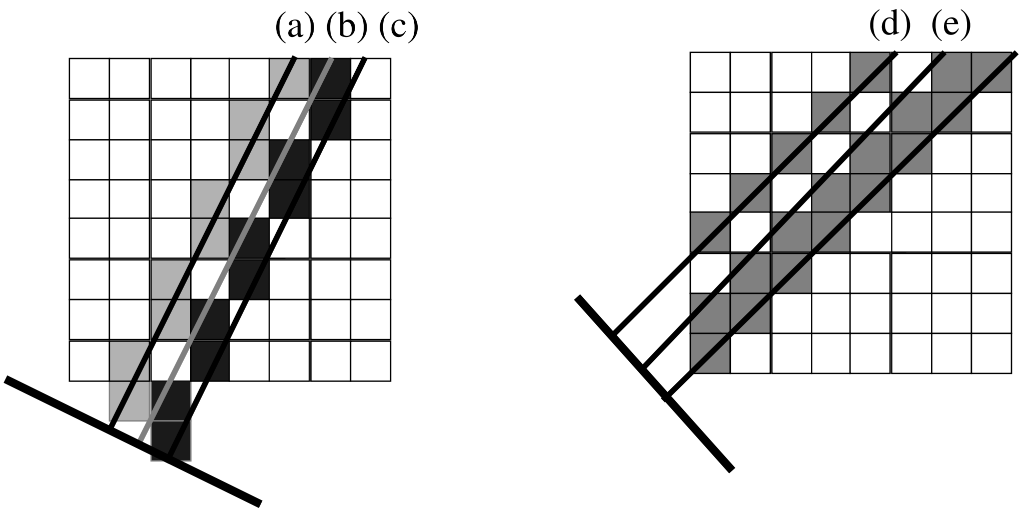


Figure 7: View dependent artifacts in volume traversal with a template.

The rays, however, differ in the appropriate portion of the template that should be repeated. We might be tempted to imitate the basic algorithm (shown in Figure 3), in which a line is started at each pixel, by always starting to follow the template from its first step. However, when using discrete rays this strategy causes significant difficulties. In the case of 6-connected discrete rays some voxels will be multiply traversed (all pixels pierced by two rays in Figure 7). In the case of 26-connected discrete rays this strategy causes the omission of some voxels, as depicted in Figure 7. What is even more disturbing in this phenomena is that its acuity is view dependent; the uneven sampling pattern is changing from one view point to another. Figure 7 illustrates this phenomena; in the left illustration some voxels, between the ray labeled (a) and the ray labeled (b), are skipped while others are visited by the two rays (b) and (c). In the right illustration in Figure 7 no voxel is visited twice; however, a whole ray of voxels is skipped between rays (d) and (e).

Our solution to the above difficulties is to choose a plane, parallel to one of the volume faces, to serve as a *base-plane* for the template placement. The image is computed by sliding the template along that plane, emitting a ray at each of its pixels. This placement guarantees, as can be seen in Figure 8, a complete and uniform tessellation of the volume by 26-rays.

The proposed algorithm is composed of three phases: initialization, ray casting, and 2D mapping. In the first phase the base-plane is determined, the extent of the projected image on the base-plane is computed, and a template in the viewing direction is constructed. In the second phase a ray is traversed from each pixel inside the image extent by repeating the sequence of steps stored in the ray template. The points where the ray enters and exits the volume are also extracted from the ray template by employing discrete arithmetic. In the last phase, the projected image is mapped from the base-plane onto the screen-plane by employing a 2D transformation. The proposed algorithm guarantees uniform sampling and is based on simple arithmetic that forms the basis for an extremely efficient implementation. We now turn to describe, in more detail, each of the algorithm's three phases.

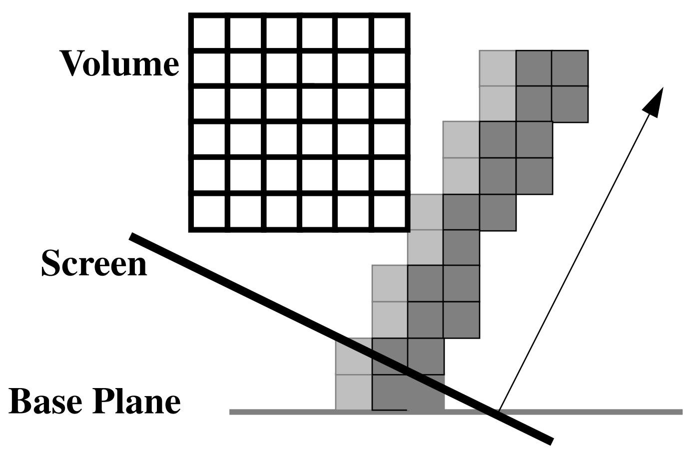


Figure 8: Complete volume tessellation via careful template placement - three copies of the template are placed side by side on the base plane here.

Initialization

In the initialization phase, the algorithm first determines which of the volume faces projects to the largest area on the screen-plane. The plane containing this face is called the *base-plane* and our algorithm will trace rays emanating from its pixels. Unless the view vector is parallel to one of the major axes, the base-plane is not the same as the screen-plane, as shown in Figure 8. However, the hidden voxel removal problem has the same solution in both planes; it is only a matter of image scaling that distinguishes one image from the other. Therefore, we can render the image on the base plane and then perform a 2D image scaling in order to produce the desired result - an image projected on the screen-plane.

The initialization phase next computes the extent of the projected image on the base-plane by projecting the vertices of the box defining the extent of the volume data, and drawing the 2D lines representing the projected edges of the volume faces. This process yields the silhouette of the projected image. In the next stage, rays will be traced from each pixel within this silhouette. Finally, the initialization phase computes the form of a ray and stores it in the ray-template. The simplest way to define the ray template is with an array containing the *U, V* and *W* increments for each step along the ray. That is, if after stepping *n* steps along the ray we are at coordinate *[u,v,w]*, then the voxel we should sample next is at *[u', v', w']* computed by:

$$[u', v', w'] := [u, v, w] \oplus Ray\_Template[n]$$

where $\oplus$ stands for vector addition.

Ray Casting

We are now ready to rewrite the ray casting algorithm (see Figure 3 and Figure 9). The function *ray_enter_-volume* computes (as will be described later) the volume coordinate where the ray emitted at pixel *(i,j)* enters the volume. The *ray_template_placement* computes the appropriate offset from which the current ray (emitted at pixel *(i,j)*) should start following the template.

```
set all pixels to BACKGROUND;
for each screen sample point (i,j) do
    [u,v,w] := ray_enter_volume(i,j);
    t_index := ray_template_placement(i,j);
    accumulator := EMPTY;
    while (inside the volume and not opaque(accumulator)) do
        sample := sample_volume_at([u,v,w]);
        accumulator := compose_voxels(sample, accumulator);
        [u,v,w] := [u,v,w] ⊕ Ray_Template[t_index];
        t_index := t_index + 1;
    end_while;
    accumulator := compose_voxels(BACKGROUND, accumulator);
    draw_pixel(i, j, accumulator);
end_for;
```

Figure 9: Template-driven ray casting algorithm.

2D Mapping

The result of the second phase of the algorithm is a projection of the volume on the base-plane. However, we actually need the projection of the volume on the screen-plane. Therefore, the third phase of the algorithms transforms the projected image from the base-plane to the screen-plane. Note that we were careful to ensure uniform sampling of the volume. In order to maintain this uniformity and avoid other sampling artifacts this transformation must be based on an accurate re-sampling method. The topic of accurate and efficient 2D raster transformations was dealt with extensively in the past. The interested reader is referred to the reference list [Hersch 1985, Heckbert 1986, Paeth 1986, Mitchel and Netravali 1988, Foley et al. 1990].

3: Template-Based Discrete Rays

We first make a note concerning the desired characteristics of the line algorithm that governs the steps taken by the ray. Unless these steps are determined solely by the line's slope, regardless of its length, we might end-up in a situation where, although all rays have the same slope, they might have different forms, depending on their length. This phenomenon is caused by the line algorithm's tendency to evenly distribute the error along the line causing variations in the head and tail of lines of equal slope and different length. Rounding errors are another source of inaccuracies; the entry point of the line is calculated in floating-point accuracy but a discrete line algorithm will be activated from the closest integer grid point. Both these difficulties are alleviated by generating all rays from one template.

We have chosen 26-connected rays (in which consecutive voxels have at least one vertex in common) to generate the template for two reasons. First, 26-rays consist of fewer voxels; a line from (x,y,z) to $(x+\Delta x, y+\Delta y, z+\Delta z)$ consists of $max(\Delta x, \Delta y, \Delta z)$ voxels while the equivalent 6-connected ray (where consecutive voxels share a face) consists of $\Delta x+\Delta y+\Delta z$ voxels. On average, a 6-line has almost twice as many voxels as a 26-line. Thus, a projection algorithm that is based on 6-rays can take almost twice the time it takes to project the same volume using 26-rays. The second reason to use 26-rays is that they can uniformly sample the volume. When tracing 26-rays from each pixel on a plane parallel to one of the major axis, every voxel is visited exactly once. This is not true in the case of 6-rays. Since a 6-ray consists of all voxels pierced by the continuous line, when two rays pierce the same voxel, it will be sampled by two discrete 6-rays.

3.1: Computing the Ray Entry and Exit Points

A special difficulty arises when computing the object-space coordinates where the discrete ray enters and exits the volume. This difficulty is due to the fact that, in 26-rays, the geometric entry and exit point do not necessarily correspond to the discrete entry and exit points (Figure 10). Note that, in contrast to 26-lines, an accurate 6-line algorithm such as the tripod algorithm [Cohen and Kaufman 1991] can guarantee that the discrete line consists of all voxels pierced by the continuous line. However, in the case of 26-rays, in a projection from an arbitrary view point, there may be some rays with discrete entry and exit point that agree with the geometric points and some rays in which one or two of these points disagree. In the case of line (a) in Figure 10 there is an agreement between the discrete and the continuous entry and exit points. However, in line (b) there is no agreement in either case while in line (c) the problem arises only at the exit point. This implies that the computation of the entry point as well as the computation of the number of steps the discrete line makes inside the volume (which is based on the computation of both entry and exit points) cannot be determined by continuous-space computation. Therefore, the object-space coordinates of these two points have to be computed in discrete space. As we show, all of the information needed to determine the discrete entry and exit points is already available in the ray-template itself. Note that the use of discrete arithmetic provides both efficiency and accuracy by performing computations regarding discrete objects in the space in which they reside.

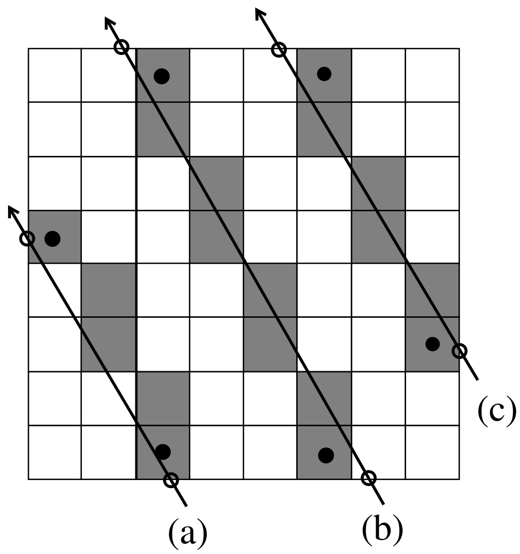


Figure 10: The difficulty in computing entry and exit points (a) discrete (filled circles) and continuous points (empty circles) agree, (b) do not agree, and (c) agree only on entry point.

For the sake of simplicity we explain our method in the 2D case. The 3D algorithm is an extension of the same principles. We assume that a buffer of $N \times N$ elements has to be projected onto a 1D screen using 2D discrete rays, as depicted in Figure 11. We denote by UV the coordinate system of the 2D buffer and assume that U is the base-line where the base-line is the 2D equivalent to the base-plane in 3D.

Recall that the ray template contains, in its i^{th} entry, the step we have to make to get to the next voxel along the ray. We define the table $\Sigma^{\Psi}(i)$ to contain, in its i^{th} entry ($i \geq 1$), the sum of the number of steps in the ψ direction in the first i-slots of the ray template, where $\psi \in \{U,V\}$. We also define the table $\Pi^{\Psi}(i)$ to contain, in its i^{th} entry ($i \geq 0$), the number of steps *along the template* we have to make in order to advance i steps in the ψ dimension. In other words, if j is the first index in which $\Sigma^{\Psi}(j) = i$ then $\Pi^{\Psi}(i) = j$. For example, if the first 10 values in the ray template, along the ψ dimension, are 0,1,0,1,0,1,0,1,0,1 then Σ^{Ψ} will contain the values 0,1,1,2,2,3,3,4,4,5, and Π^{Ψ} will contain the values 1,2,4,6,8,10.

Let us examine the ray emitted at screen coordinate U=$\bar{u}$ where $0 \le u < N$. Because our image is generated by scanning the base-line, the entry point of the rays emitted along the major face is known - $(\bar{u}, 0)$ ((a) in Figure 11). The exit point, in general, can be either at $V \equiv N$ ((b) in Figure 11) or at $U \equiv 0$ ((c) in Figure 11).

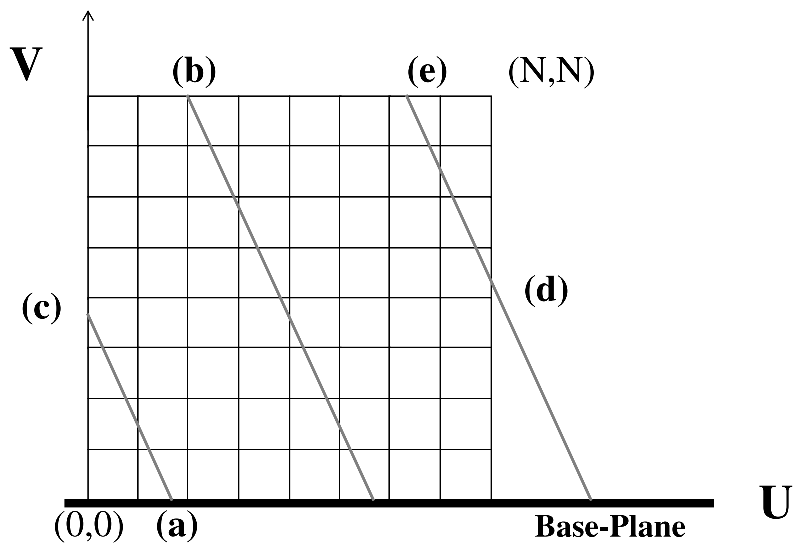
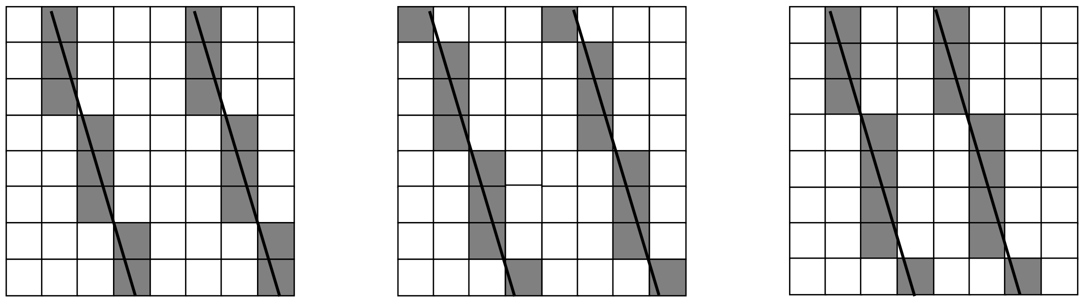


Figure 11: Computing entry and exit points using the template. The three cases for 2D rays depending on entry and exit faces. (a)-(c) enters at V=0 and exits at U=0. (b) enters at V=0 and exits at V=N. (d)-(e) enters at U=N and exits at V=N.

How many steps along the ray emitted at $(\bar{u}, 0)$ do we make until we first cross one of these lines? From the definition of $\Pi^U(i)$ it follows that $U \equiv 0$ is reached after $\Pi^U(\bar{u})$ steps. Since we use a V-major ray, the line $V \equiv N$ is reached after N steps. Thus, lines emitted at $(\bar{u},0)$ traverse $min(N, \Pi^U(\bar{u}))$ voxels.

A ray that is sent from $(\bar{u}, 0)$ (where $N \le \bar{u} < 2N$) to the second face of our 2D raster will enter the volume at $U \equiv N$ ((d) in Figure 11), that is, the ray will take $\Pi^U(\bar{u}-N)$ steps along the ray before entering the 2D buffer. This type of ray will always exit from the face $V \equiv N$ ((e) in Figure 11), that is, after N steps in the template (because the line is V-major). Therefore, the line has to take $S=N-\Pi^U(\bar{u}-N)$ steps inside the raster. Note that while scanning the base-line from $\bar{u}=N$ towards $2N$, the value of S becomes negative only when the ray does not hit the volume anymore, which indicates that we should stop the scan. The above considerations and logic can be applied to a 3D arbitrary viewing direction in order to compute, in discrete space, the entry and exit points of any ray.

3.2: Screen Supersampling



1/2 **1/4** **4/9**

Figure 12: Template driven traversal with origin displacement. The left grid contains rays centered at the origin pixels. The center image contains rays with 1/4 offset from the pixel side while in the right figure all rays are 4/9 from the pixel side.

So far, our algorithm assumed that only one ray is traced from each screen pixel. However, the algorithm can be slightly modified to support multiple rays emitted from the same pixel but from different sub-pixel addresses. Instead of having one type of ray (i.e., one template), we now have several templates, one for each relative position of the ray-origin in a pixel.

For example, if we want a supersampling rate of four rays per pixel, for the pixel at coordinate *(i,j)*, these four rays will originate from $(i \pm 1/4, j \pm 1/4)$. We observe that for all *(i,j)*, the rays $(i+1/4, j+1/4)$ have the same

(a) **(b)**

Figure 13: A 128^3 volume containing thirteen binary-defined spheres rendered with the discrete template algorithm. (a) one ray per pixel. (b) four rays per pixel.

form. Therefore we need only four different templates – one for each relative displacement from the pixel origin. This phenomena is illustrated in Figure 12 which shows three 2D rasters with rays emanating from different sub-pixel locations. Although different ray origins yield different forms, all rays with the same relative displacement of the origin have the same form. This observation can be used even for stochastic supersampling [Cook 1986, Dippe and Wold 1985] as long as the ray-origin is confined to a set of predefined sub-pixel addresses for which template are precomputed. The traditional line algorithms for ray generation can be easily modified by just altering the initialization phase in order to support non-integer (fixed-point) origin coordinates.

Figure 13 shows the difference in image quality when rendering the same dataset, containing thirteen binary-defined spheres, without screen supersampling (a) and with screen supersampling of four rays per pixel (b). In order to show the enhanced image quality contributed by supersampling, the spheres are binary defined (i.e, not antialiased in object-space) and the ray performs semitransparent compositing. Higher quality rendering of geometric objects can be achieved by generating antialiased objects [Wang and Kaufman 1993] and by supersampling in object-space, as we describe now.

4: Template-Based Continuous Rays

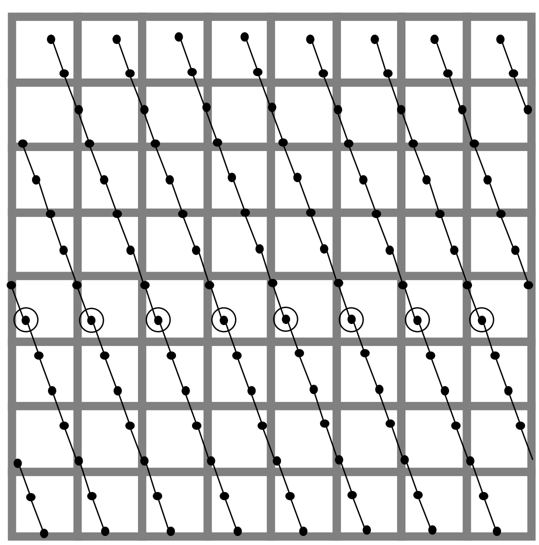


Figure 14: Template driven traversal with sub-voxel addresses for object-space supersampling. The seventh sample along each ray is circled.

The template algorithm can also support rendering of partially occupied voxels (e.g. trilinear value variation across the voxel extent). Under this assumption, the exact volume value at a specific point is computed by a weighted average of a neighborhood of voxels, as described in Section 1. Therefore, it makes sense to sample each voxel more than once in order to enhance image quality (i.e. supersampling along the ray). Usually a trilinear approach is taken that interpolates the sample value from its eight closest neighbors.

The template-based algorithm presented in the previous section can be modified to efficiently support multiple samples per voxel. First, let us adopt the parametric formulation of the line extending from the voxel v_0 to the voxel v_1. The line is denoted by $v_0 \oplus t\,\bar{v}$ where $t=0$ at v_0, $t=1$ at v_1, and $\bar{v}$ is the vector $(\Delta x, \Delta y, \Delta z)$. Assume that we would like to sample the ray at constant intervals Δt along the line, that is, we have to increment our sampling location by $\Delta v = \Delta t \cdot \bar{v}$. Let us take a closer look at the i^{th} sample point in each ray (see Figure 14).

If we assume that rays are emitted from the centers of the pixels in a plane that is parallel to one of the volume faces (such as the base-plane) then all these sample points are in the same relative position inside the voxel, that is, they have exactly the same set of distances to the eight corners of the voxel. Therefore, we can employ a continuous line algorithm to generate a template of floating point steps. For each sample point we can precompute the weight to be assigned to each of the voxel values participating in the sampling operation. This will save the need to compute, on the fly, the value of ω_i for each sample, as in Equations 2-9. Since the range of voxel values is limited, the precomputed weights can be represented in a fixed-point notation that serves as an index into a multiplication table that will further expedite the evaluation of Equation 1.

(a) **(b)**

Figure 15: A 128^3 volume containing thirteen binary defined spheres rendered with the continuous template algorithm. (a) one sample per voxel. (b) two samples per voxel.

The algorithm for ray casting continuous rays is very similar to the basic one described previously (see Figure 9). It performs integer-based stepping along the ray template. The only difference is that now, at the i^{th} step along the ray, we replace the access to volume at [u,v,w] with a call to a function that computes the sample value by utilizing the precomputed set of weights and the precomputed multiplication table.

In summary, the template based ray casting algorithm uses very few (≈8) integer additions per sample instead of the naive implementation of Equations 1-9 which requires tens of floating-point additions and multiplications. Figure 15 demonstrates the difference in image quality when rendering the same dataset, containing thirteen spheres, with one sample per voxel (a) and with ray supersampling of approximately two samples per voxel (b). The spheres are binary defined (i.e, not antialiased in object-space); however, the ray trilinearly interpolates an intermediate value of color and opacity and performs semitransparent compositing.

5: Adaptive Sampling

In several applications, rendering is performed by adaptively switching from one rendering method or sampling rate to another. Adaptive screen sampling is well known in traditional ray tracing [Glassner 1989] where the number of rays emitted from a given pixel is adapted to the color change in a small neighborhood.

One problem with 26-rays is that they may occasionally "miss" the surface, which is guaranteed to be detected by 6-rays because they always traverse all voxels pierced by the continuous ray. On the other hand, we prefer to employ 26-rays because they can be twice as fast as their 6-connected counterpart. One solution is to adaptively alternate between 26- and 6-rays by traversing the empty space with 26-rays and then switching to 6-rays in the vicinity of objects.

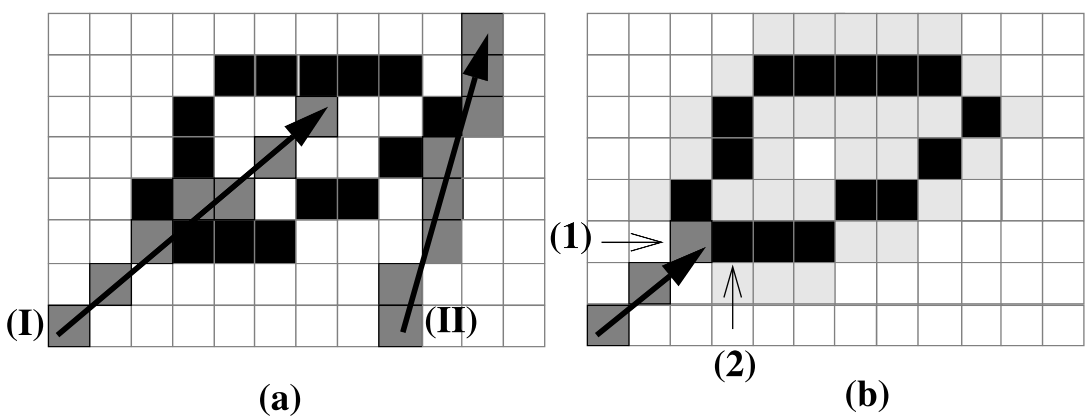


Figure 16: Adaptive 6/26 rays with proximity flags. (a) a 26-ray penetrates through a 26-connected surface (depicted by the black 8-curves). (b) the surface surrounded by vicinity flags (light grey pixels) that causes the ray to switch, at point (1), to 6-connectivity and hit the surface at point (2).

To implement this feature, one has to add (during a pre-processing stage) a proximity flag to all the voxels around the object surface to indicate that we are in the vicinity of an object (Figure 16.b). The traversal starts with a 26-ray which rapidly skips over the blank regions. Whenever a proximity flag is encountered ((1) in Figure 16.b), the 26-line algorithm adaptively "slows down" and takes 6-connected steps in order to avoid missing the object silhouette (ray II in Figure 16.a) and to avoid penetration of surfaces that are not "thick" enough to avoid penetration by 26-rays (ray I in Figure 16.a) but that can avoid penetration by 6-rays ((2) in Figure 16.b). In Figure 16.a we see an 8-ray (the 2D analogue to 26-ray) penetrating through a 8-connected curve, that is, a "thin" curve that can only avoid penetration by 4-rays. In Figure 16.b the object is surrounded by pixels that have their proximity flag turned on. When the ray reaches pixel (1) it encounters the proximity flag and switches to 4-connected traversal which causes it to detect the 4-curve at pixel (2).

In order to benefit from the adaptive approach, the line algorithm is required to efficiently perform the connectivity switch. In [Yagel et al. 1991a] we described a line algorithm that adaptively switches from 26-connectedness to 6-connectedness without noticeable time penalty. The extra proximity flags in the adaptive algorithm add only one test per step to the overall execution time of the algorithm. However, employing 26-rays has an overall speedup compared to 6-rays mainly in diagonal views when performing parallel projection, in all views when performing perspective projection, and in volumetric recursive ray tracing where secondary rays are spawned in arbitrary directions [Yagel, Kaufman, and Zhang 1991b, Yagel, Cohen, and Kaufman 1992].

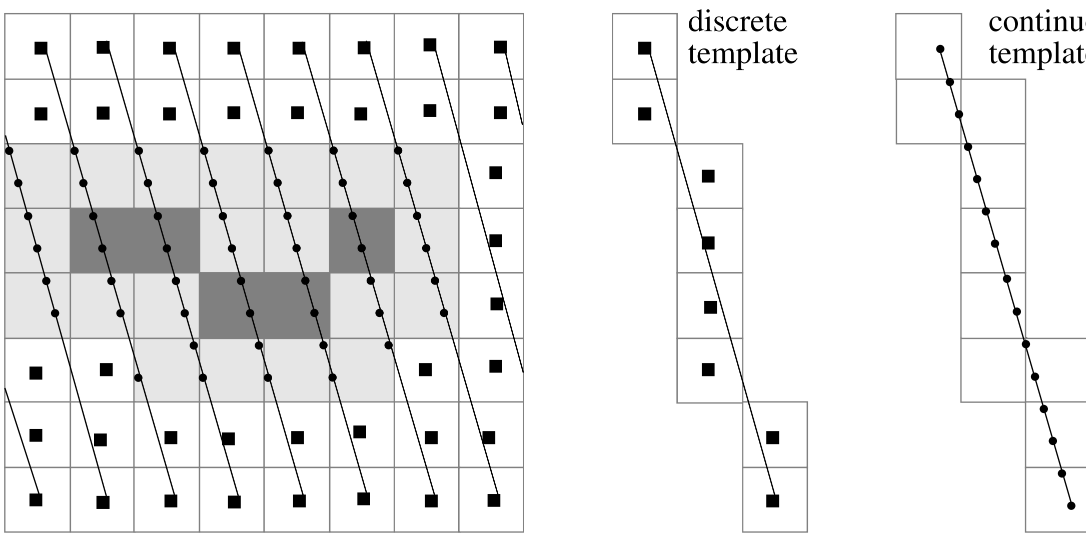


Figure 17: Adaptive discrete/continuous rays with proximity flags.

The sampling rate along a ray can also be changed in accordance to changes in the traversed density [van Walsum et al. 1991, Danskin and Hanrahan 1992]. Whenever there is little change in the field of values, the ray can make larger steps without affecting the sampling quality. However, when the value gradient is high, indicating high frequency changes in the value field, the ray will adaptively decrease its step size in order to sample the field more densely. We have also implemented an adaptive algorithm that, as depicted in Figure 17, switches between a discrete and a continuous line algorithm in such a way that the discrete algorithm is used to efficiently traverse the empty space while the continuous algorithm is activated in the vicinity of non-empty voxels, in order to achieve greater sampling accuracy.

Figure 18: A 128^3 volume of an electron density map rendered by the adaptive discrete/ continuous template algorithm.

All these adaptive paradigms can be efficiently implemented *without* the need for any special-purpose line algorithm by maintaining templates of all lines that we can switch between. In addition, for each pair of templates t_1 and t_2 we precompute two tables $m_{1\Rightarrow 2}$ and $m_{2\Rightarrow 1}$. The table $m_{1\Rightarrow 2}$, for example, holds in its i^{th} entry a number j which is to be used as the location to start following t_2 if a switch from t_1 to t_2 has occurred while performing the i^{th} step of t_1. The table $m_{2\Rightarrow 1}$ has the same function when switching from t_2 to t_1. These two tables are used to efficiently switch between a group of various space traversal algorithms (templates) which are employed adaptively.

Figure 18 shows an example of a electron density map of active site of superoxide dismutase [SOD] enzyme as determined by X-ray crystallography.

6: Results

To thoroughly assess the attributes of the template-based ray casting, we have conducted several experiments aimed at measuring rendering time as a function of:

- average data opacity
- volume resolution
- scene sparsity
- ray traversal algorithm

In order to better understand the reasons for performance fluctuations we also measured, for each image generated, the number of rays, the average number of samples, the number of discrete (continuous) steps that sampled empty space, and the number of discrete (continuous) steps that sampled and composited data. Reported numbers are the average of the results from multiple view points.The first experiment employed continuous rays which took approximately two samples per voxel. A 256^3 volume was completely filled with a solid object made of some material. The opacity of this material was changed from 0% (fully transparent) to 100% (fully opaque) and the results are reported in Table 1.

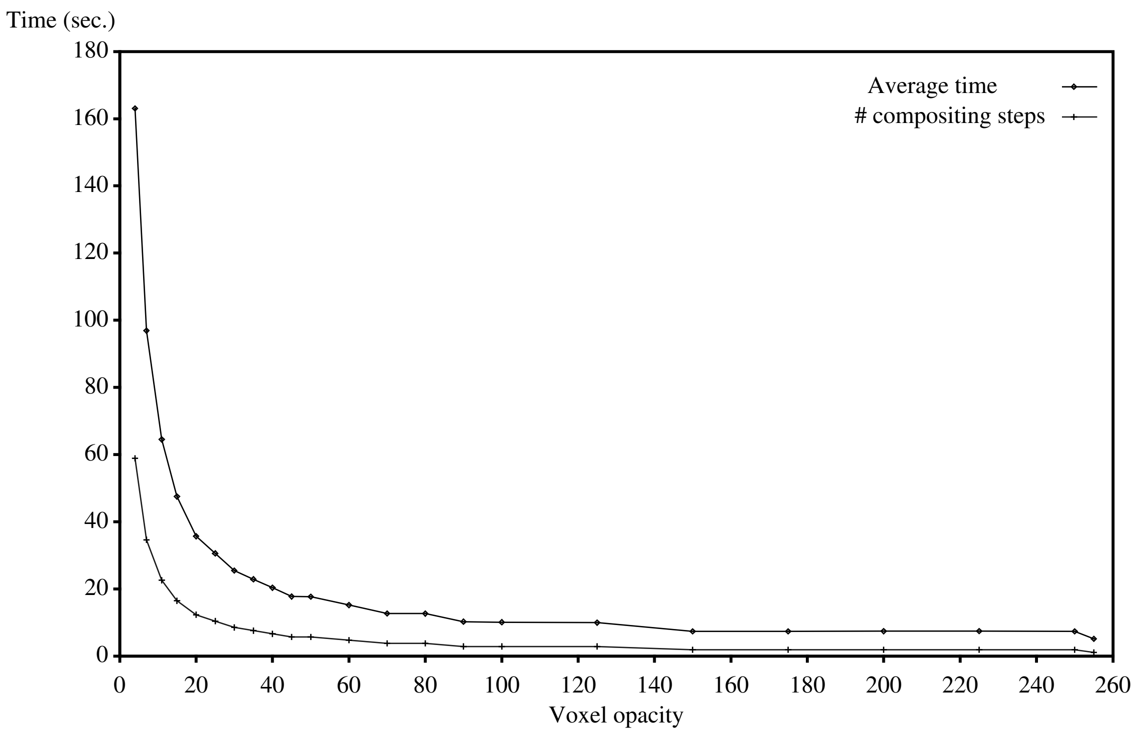


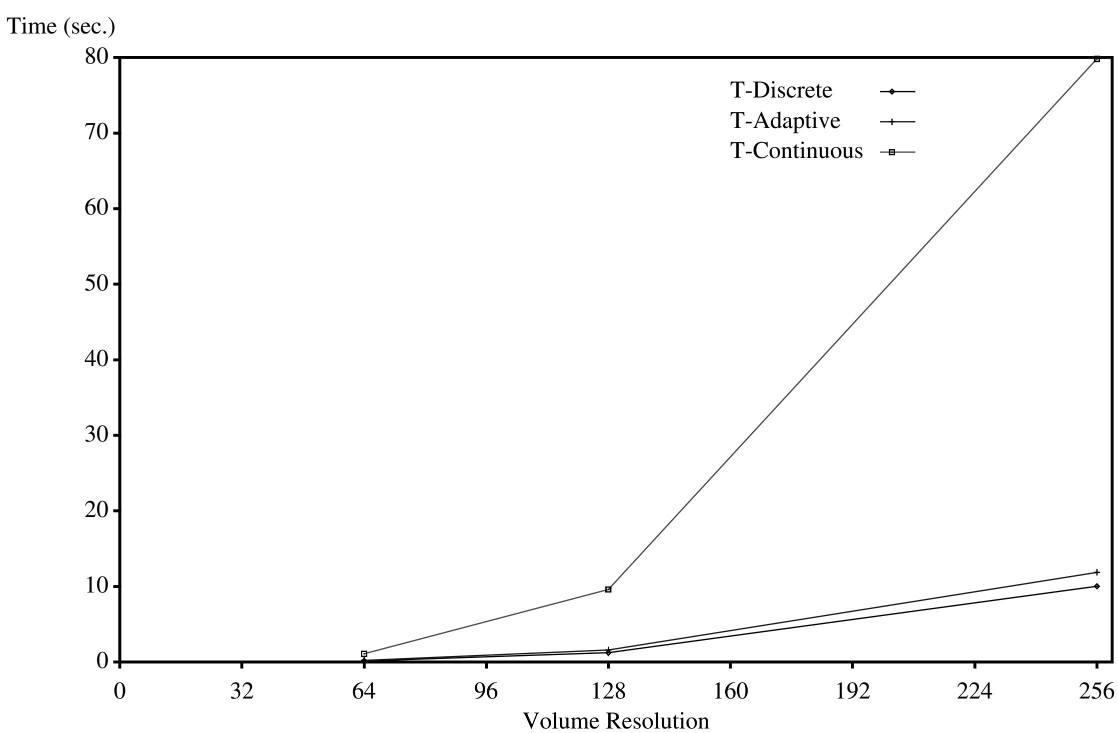
Figure 19: Graph showing the performance of the continuous template algorithms as a function of volume opacity.

In the case of 0% the ray does not composite but scans the whole volume without stopping (performing, on the average, 363.46 steps per ray). In the case of 0.8%, the ray performs a large number of compositing steps (112.98, on the average) before accumulating an opaque value. As the opacity of the material increases, fewer compositing steps are needed before the ray stops, the average ray length decreases, and therefore the performance improves rapidly. We also observe that compositing, even with the template-stored information, plays a crucial role in determining execution time. As can be seen in Figure 19, there is a strong correlation between the two.

Table 1: Rendering time as a function of opacity in a full volume.

Opacity	Time (seconds)	# of compositing samples per ray
0%	112.2	0.00
0.8%	310.4	112.98
10%	30.6	10.41
49%	10.0	2.85
78%	7.4	1.90
100%	5.1	1.10

The second experiment measured the performance of the various template-based algorithms (discrete (denoted by *T-Discrete* in Figure 20), continuous (denoted by *T-Continuous* in Figure 20), and adaptive (denoted by *T-Adaptive* in Figure 20)) as a function of volume size. Rays where traced in 64^3, 128^3, and 256^3 volumes consisting of thirteen opaque spheres (occupying 40% of the space). The continuous rays and adaptive rays took two samples per voxel. The results are shown in Figure 20.



The main reason for the gap between the continuous method and the other two is due to the test that the continuous ray performs to detect if a voxel is empty - by verifying that all eight values at the voxel's vertices are transparent. As the resolution increases, the number of such tests increases as a cubic function. Since the objects are opaque, rays will almost always stop immediately, once the object is encountered. Therefore, in this case the adaptive ray will behave very much like a discrete ray. Only when a ray grazes the silhouette of an object, will the continuous (and adaptive) not immediately accumulate complete opacity.

The third test was aimed at assessing the influence scene sparsity has on performance. We traced a 256^3 volume with discrete, continuous, and adaptive rays. The data consisted of a random distribution of opaque points. The number of points varied from 0% of the voxels in the volume up to 100%. The results are reported in Figure 21.

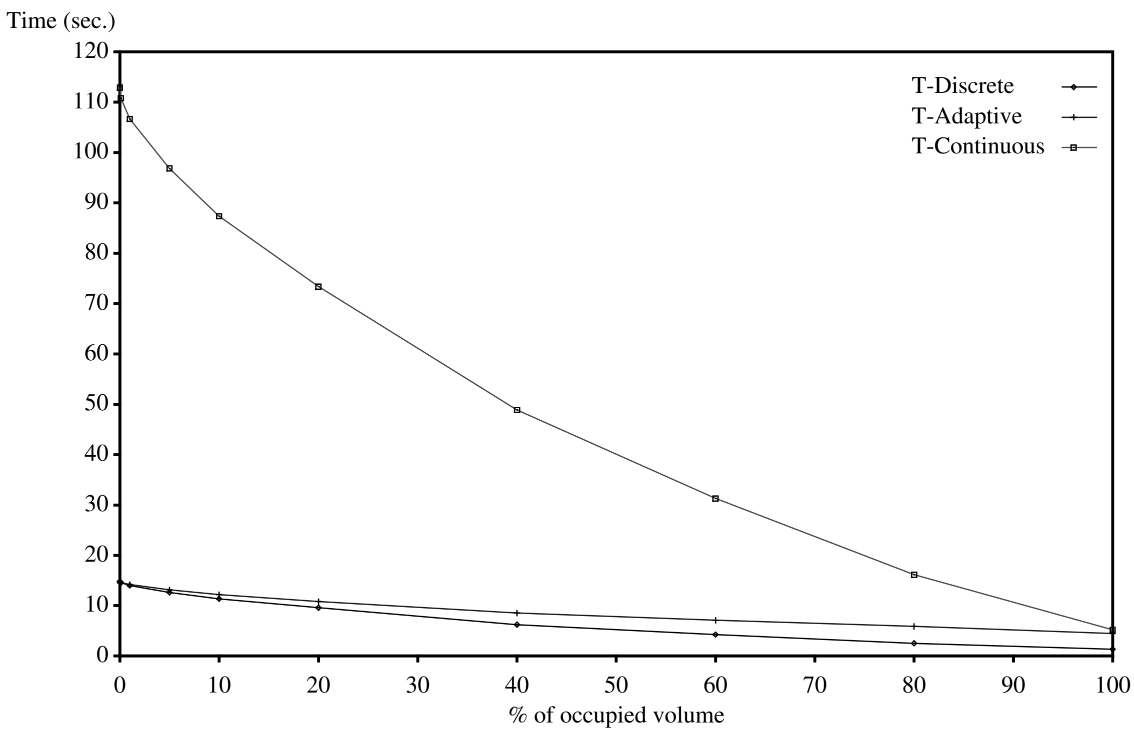


Figure 20: Graph showing the performance of the various template algorithms as a function of volume sparsity.

In a very sparse volume rays usually do not hit an object, so adaptive rays tend to behave exactly like discrete rays. The test for void cells performed by the continuous rays inflict a severe performance penalty in sparse spaces. As the volume becomes denser, the continuous rays tend to find voxels with data quickly. Since voxels in this experiment are opaque, the cost of compositing is not significant and continuous rays converge to the behavior of the adaptive method.

The fourth and last test compared the performance of the various ray traversal algorithms, namely: non-template discrete, template-based discrete, non-template continuous, template-based continuous, and the template-based adaptive traversal. Rays were traced in a 256^3 volume that consisted of thirteen opaque spheres. We have varied the sampling rate, which is denoted by *(i,j)*, where *i* is the number of rays per pixel (screen supersampling) and *j* is the number of samples per voxel (ray supersampling). The results are reported in Table 2.

The '*number of samples*' column in Table 2 refers to the non-adaptive methods. The number of samples the adaptive method performs (and therefore its performance) for a *(i, j)* image is the same as *(i, 1)*, if the object is opaque, but approaches the results of the continuous template algorithm when the volume is densely populated with very transparent material. In our sphere example the adaptive method, which provides the same image quality as the continuous algorithms, performs four times faster (in the case of (1,1)) and up to 17 times faster (in the (16, 4) case). Since the objects are opaque, the adaptive rays are insensitive to the sampling rate along the ray, while the continuous rays (template and non-template), which perform more steps, suffer a performance penalty.

Table 2: Rendering time in seconds as a function of the rendering algorithm.

Supersampling Method	# of samples (in millions)	Discrete Non-Template	Discrete Template	Continuous Template	Continuous Non-Template	Adaptive
(1, 1)	13.88	28.4	10.1	44.2	62.0	11.8
(1, 2)	27.45	-	-	82.1	109.9	11.7
(1, 4)	54.16	-	-	152.2	202.9	11.6
(4, 1)	55.128	111.0	42.0	176.7	246.4	47.8
(4, 2)	109.81	-	-	321.8	436.7	46.7
(4, 4)	216.66			602.9	809.1	48.0
(16, 1)	222.04	445.4	171.8	707.2	985.3	188.7
(16, 4)	439.0	-	-	1275.9	1750.6	191.7
(16, 16)	866.14	-	-	2414.3	3231.5	187.9

We have also used the results of this experiment to calculate the number of samples per second each algorithm performs. The results, in decreasing order of performance, are:

	Discrete Template	Adaptive	Discrete Non-Template	Continuous Template	Continuous Non-Template
Samples/Sec. (in thousands)	1214.6	1084.6	451.2	337.1	246.8

We observe that on the average, we gain a 2.7 speedup when employing discrete rays, as compared to the traditional discrete ray casting, and a speedup of 4.4 when employing the adaptive templates instead of the traditional continuous rays.

In summary, the adaptive template-assisted ray casting is by far more advantageous over existing methods. Its performance, per ray, depends on the number of integer steps it takes when skipping the empty space and the number of compositing steps it computes when penetrating the scene objects. The first is a function of volume resolution (Figure 20) and scene sparsity (Figure 21). The latter is a function of data translucency (Table 1 and Figure 19) and sampling rate along the ray (Table 2). The overall algorithm performance (given the per ray cost) is then a linear function of the number of rays (i.e. image size and screen supersampling rate) (Table 2).

7: Conclusion

We presented the template-based approach to parallel viewing and described a new three-phase algorithm that takes advantage of templates and their exact placement in order to provide accurate and efficient volume rendering. We have described several variations of the algorithm that support screen-space and object-space supersampling. We demonstrated the advantages of our approach by introducing the template assisted adaptive traversal and compared its performance with traditional methods. Our method delivers significant speedup over exiting volume ray casting methods while maintaining high fidelity sampling of the volume when casting either discrete or continuous rays.

Acknowledgments

This work has been supported by the National Science Foundation under grant CCR-9211288. The datasets shown in Figure 5, Figure 6, and Figure 18 are courtesy of UNC Chapel Hill. We thank Yair Kurzion, Raghu Machiraju, and Naeem Shareef for reading and correcting the manuscript.

8: References

Bakalash, R. and Kaufman, A., “A 3D Medical Imaging Architecture”, Computers & Graphics, 13, 2, (1989).

Cohen, D. and Kaufman, A., “3D Discrete Lines: Voxelization Algorithms and Connectivity Control”, TR 91.05.09, Computer Science, SUNY at Stony Brook, (May 1991).

Cohen, D. and Shefer, Z., “Proximity Clouds - An Acceleration Technique for 3D Grid Traversal”, Technical Report FC 93-01, Department of Mathematics and Computer Science, Ben Gurion University of the Negev, (February 1993).

Cook, R. L., “Stochastic Sampling in Computer Graphics”, ACM Transaction on Graphics, 5, 1, 51-72, (January 1986).

Danskin, J. and Hanrahan, P., “Fast Algorithms for Volume Ray Tracing”, Proceedings of 1992 Workshop on Volume Visualization, Boston, MA, 91-105, (October 1992).

Dippe, M. A. Z. and Wold, E. H., “Antialiasing Through Stochastic Sampling”, Computer Graphics, 19, 3, 69-78, (July 1985).

Drebin, R. A., Carpenter, L., and Hanrahan, P., “Volume Rendering”, Computer Graphics, 22, 4, 65-74, (August 1988).

Farrell, E. J., Yang, W. C., and Zappulla, R. A., “Animated 3D CT Imaging”, IEEE Computer Graphics and Applications, 5, 12, 26-32, (December 1985).

Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F., Computer Graphics Principles and Practice, Addison-Wesley, Reading, MA, 1990.

Glassner, A. S., (ed.), An Introduction to Ray Tracing, Academic Press, 1989.

Heckbert, P. S., “Survey of Texture Mapping”, IEEE Computer Graphics & Applications, 6, 11, 56-67, (November 1986).

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

Hersch, R. D., “Raster Rotation of Bilevel Bitmap Images”, Proceedings EUROGRAPHICS ‘85, Nice, France, 295-308, (September 1985).

Kaufman, A. and Bakalash, R., “Memory and Processing Architecture for 3-D Voxel-Based Imagery”, IEEE Computer Graphics & Applications, 8, 11, 10-23, (November 1988).

Levoy, M., “Display of Surfaces from Volume Data”, IEEE Computer Graphics and Applications, 8, 5, 29-37, (May 1988).

Levoy, M., “Display of Surfaces from Volume Data”, Ph.D. Dissertation, University of North Carolina at Chapel Hill, May 1989.

Levoy, M., “Volume Rendering by Adaptive Refinement”, The Visual Computer, 6, 1, 2-7, (February 1990).

Levoy, M., “Efficient Ray Tracing of Volume Data”, ACM Transactions on Graphics, 9, 3, 245-261, (July 1990).

Levoy, M. and Whitaker, R., “Gaze-Directed Volume Rendering”, Computer Graphics, 24, 2, 217-223, (March 1990).

Mitchel, D. P. and Netravali, A. N., “Reconstruction Filters in Computer Graphics”, Computer Graphics, 22, 4, 221-228, (August 1988).

Paeth, A. W., “A Fast Algorithm for General Raster Rotation”, Graphics Interface’86, 77-81, 1986.

Reynolds, R. A., Gordon, D., and Chen, L. S., “A Dynamic Screen Technique for Shaded Graphic Display of Slice-Represented Objects”, Computer Vision, Graphics and Image Processing, 38, 3, 275-298, (June 1987).

Sabella, P., “A Rendering Algorithm for Visualizing 3D Scalar Fields”, Computer Graphics, 22, 4, 51-58, (August 1988).

Samet, H. and Webber, R. E., “Hierarchical Data Structures and Algorithms for Computer Graphics, Part II: Applications”, IEEE Computer Graphics and Applications, 8, 7, 59-75, (July 1988).

Schroeder P. and Krueger, “Data Parallel Volume Rendering Algorithms for Interactive Visualization”, The Visual Computer, 9, 405-416, (1993).

Tuy, H. K. and Tuy, L. T., “Direct 2-D Display of 3-D Objects”, IEEE Computer Graphics & Applications, 4, 10, 29-33, (November 1984).

Upson, C. and Keeler, M., “V-BUFFER: Visible Volume Rendering”, Computer Graphics, 22, 4, 59-64, (August 1988).

van Walsum, T., Hin, A. J. S., Versloot, J., and Post, F. H., “Efficient Hybrid Rendering of Volume Data and Polygons”, Second Eurographics Workshop on Visualization in Scientific Computing, Delft, Netherlands, (April 1991).

Westover, L., “Interactive Volume Rendering”, Proceedings of the Chapel Hill Workshop on Volume Visualization, Chapel Hill, NC, 9-16, (May 1989).

Wyvill J. and Wyvill B. “Rendering Soft Objects”, Proceedings of EUROGRAPHICS 1987.

Yagel, R. “Efficient Methods for Volumetric Graphics”, Ph.D. Dissertation, State University of New York at Stony Brook, December 1991).

Yagel, R., Cohen, D., Kaufman, A., and Zhang, Q., “Volumetric Ray Tracing”, TR 91.01.09, Computer Science, SUNY at Stony Brook, (January 1991).

Yagel, R., Kaufman, A., and Zhang, Q., “Realistic Volume Imaging”, Proceedings of Visualization’91, San Diego, CA, 226-231, (October 1991).

Yagel, R., Cohen, D., and Kaufman, A., “Discrete Ray Tracing”, IEEE Computer Graphics & Applications, 12, 5, 19-28, (September 1992).

Yagel, R. and Shi, Z., “Accelerating Volume Animation by Space-Leaping”, Proceedings of Visualization’93, San Jose, CA, (October 1993).

Wang, S. and Kaufman, A., “Antialiased Voxelization”, Proceedings of Visualization’93, San Jose, CA, October 1993.

Zuiderveld, K., Koning, A. H. J., and Viergever, M. A., “Acceleration of Ray Casting Using 3D Distance Transforms”, Proceedings of Visualization in Biomedical Computing 1992, 1808, 324-335, (October 1992).

Color Images

Figure 5: A 64^3 volume electron density simulation rendered by the template algorithm utilizing continuous rays and employing semitransparent composition.

Figure 6: A 256^3 volume rendered by the adaptive template algorithm utilizing discrete and continuous rays and supersampling in screen-space (four rays/pixel) and object-space (two samples/voxel).

(a) **(b)**

Figure 13: A 128^3 volume containing thirteen binary-defined spheres rendered with the discrete template algorithm. (a) one ray per pixel. (b) four rays per pixel.

(a) **(b)**

Figure 15: A 128^3 volume containing thirteen binary defined spheres rendered with the continuous template algorithm. (a) one sample per voxel. (b) two samples per voxel.

Figure 18: A 128^3 volume of an electron density map rendered by the adaptive discrete/continuous template algorithm.