Sheared Interpolation and Gradient Estimation for Real-Time Volume Rendering

Hanspeter Pfister, Frank Wessels, and Arie Kaufman
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794–4400, U.S.A.

Abstract

In this paper we present a technique for the interactive control and display of static and dynamic 3D datasets. We describe novel ways of tri-linear interpolation and gradient estimation for a real-time volume rendering system, using coherency between rays. We show simulation results that compare the proposed methods to traditional algorithms and present them in the context of Cube-3, a special-purpose architecture capable of rendering 512^3 16-bit per voxel datasets at over 20 frames per second.

1. Introduction

Numerous scientific applications, including biomedical and geophysical analysis, computational fluid dynamics and finite element models, require the rapid display of dynamically acquired or computer generated 3D datasets. Real-time visualization of dynamic volume data, called *4D (spatial-temporal) visualization*, permits observation of 3D data changes, such as the study of fluid flow in rocks or the study of a beating heart. In order to reveal the internal structure of the data, direct volume rendering methods have to be employed that generate an image without pre-processing and allow for the interactive control of viewing parameters [6].

The massive computational resources necessary to achieve 4D visualization at high frame rates place hard to meet requirements on sequential implementations and general-purpose computers. Only parallelism among a dedicated set of processors can achieve the necessary high memory bandwidth and arithmetic performance [4, 7, 12, 15] [6, Chapter 6]. While relatively fast algorithms exist for the display of static datasets on massively parallel architectures [17, 18], very little attention has been paid to the real-time visualization of dynamically changing high-resolution 3D data. This is the main objective of Cube-3, a special-purpose architecture capable of rendering 512^3 16-bit per voxel datasets at over 20 frames per second [16].

Authors' email:
pfister@cs.sunysb.edu, frankw@cs.sunysb.edu, ari@cs.sunysb.edu

Cube-3 implements ray-casting, a powerful volume rendering technique that offers high image quality while allowing for algorithmic optimizations which significantly reduce image generation times [6, 13, 14]. Rays are cast from the viewing position into the volume data. At evenly spaced locations along each ray, the data is tri-linearly interpolated using values of surrounding voxels. Central differences of voxels around the sample point yield a gradient which is used as a surface normal approximation. Using the gradient and the interpolated sample value, a local shading model is applied and a sample opacity is assigned. Finally, ray samples along the ray are composited into pixel values to produce an image [11].

An important problem of ray-casting is the non-uniform mapping of samples onto voxels, since voxels may contain more than one ray sample or may be involved in multiple gradient calculations. This leads to redundant data accesses and irregular interprocessor communication that affect the performance. In Cube-3 we use a ray-casting approach that transforms the volume into an intermediate coordinate system for which there is a mapping of ray samples onto the volume that is one-to-one. This allows for efficient projections onto a face of the volume, and the distorted image is then warped (2D transformed and projected) onto the view plane.

Using a similar approach, Yagel and Kaufman [20] describe a template based ray-casting scheme to simplify path generation for rays through the volume, and Schröder and Stoll [17] have implemented this method on a Princeton Engine of 1024 processors and have achieved sub-second rendering times for a 128^3 dataset. Cameron and Underill [3] efficiently use an intermediate volume transformation to reduce data communication in a SIMD parallel processor. Lacroute and Levoy [10] recently reported on a fast implementation using a shear-warp transformation and were able to achieve interactive rendering times for 256^3 datasets on a graphics workstation. All these implementations require a pre-processing step to calculate the gradient field or to generate color and opacity volumes and are therefore not suitable for 4D visualization.

This paper presents two new methods that allow for real-time tri-linear interpolation and gradient estimation without pre-computation. They are suitable for 4D visualization and lead to an efficient implementation in hardware. Section 2 describes the underlying real-time ray-casting approach that transforms the volume into an intermediate sheared coordinate space. Section 3 discusses the problems associated with performing interpolation in this sheared space and introduces sheared tri-linear interpolation as an effective solution. We then present a new way of gradient approximation using coherency between rays in Section 4. Section 5 describes the main architectural features of Cube-3 and Section 6 gives results on the proposed interpolation and shading methods.

2. Real-Time Ray-Casting

Our real-time ray-casting algorithm assumes that the volume is sampled on a rectilinear grid. A distorted intermediate image is projected onto the volume face that is most perpendicular to the viewing direction. Using a term by Yagel and Kaufman [20] we call this face the *base-plane*. A 2D warp of the base-plane projection produces the final image.

The first step is to transform the volume into an intermediate coordinate system for which there is a simple mapping of voxels onto base-plane pixels. In a recent approach, Lacroute and Levoy [10] use a shear-warp factorization of the viewing transform and project the volume in a *slice-parallel* fashion onto the base-plane. The volume is treated as a set of 2D slices which are subject to a 2D shear-scale and resampling operation according to the viewing transform. Each slice is treated independently without computing individual rays, and the resulting base-plane image is warped onto the viewing plane.

Other approaches [20] operate in a *ray-parallel* fashion, where resampling and compositing operations take place on rays cast from each pixel of the base-plane. In both approaches the 3D volume is traversed only once per projection. The algorithms involve one resampling of the volume and an inexpensive 2D image warp. In Cube-3 we adopted the ray-parallel approach because it allows for efficient parallel implementations of compositing along rays.

Using a technique by Yagel and Kaufman [20], we generate lookup tables or templates to cast discrete rays from the base-plane into the volume. Figure 1 shows an example of a parallel and perspective projection. 26-connected discrete lines are pre-generated using a 3D variation [8] [6, pp. 280–301] of Bresenham's algorithm modified for non-integer endpoints. This algorithm guarantees constant stepping by a distance of one along the major axis (the Z-axis in Figure 1). The stepping along the two other axes (the X- and Y-axes in Figure 1) is stored in two templates. For parallel projections, where neighboring rays follow the exact same path through the volume, the templates store n positions for an n^3 volume. For perspective projections they are of size n^2 each (see Figure 1).

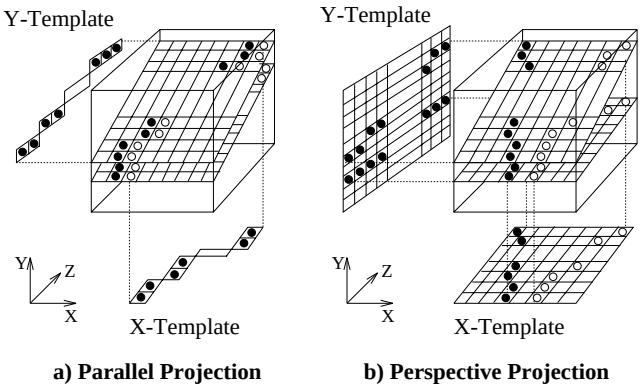


Figure 1: X/Y-Templates for Discrete Rays.

Figure 2 schematically shows how the algorithm proceeds. All the discrete rays belonging to the same scanline of the base-plane image reside on the same plane inside the volume, called the Projection Ray Plane (PRP). By fetching all voxels on a PRP and transforming them accordingly into a 2D buffer, all discrete rays can be aligned along a direction parallel to an axis, e.g. horizontal. If we define beams to be rays parallel to a main axis of the Cubic Frame Buffer (CFB), then for parallel projections this transformation is simply a shear of beams to the left or right (see Figure 2). For perspective projections each voxel belonging to a discrete ray has to be shifted by a different amount. We refer to this process as de-fanning, since diverging rays are stored adjacent to each other in the 2D buffer.

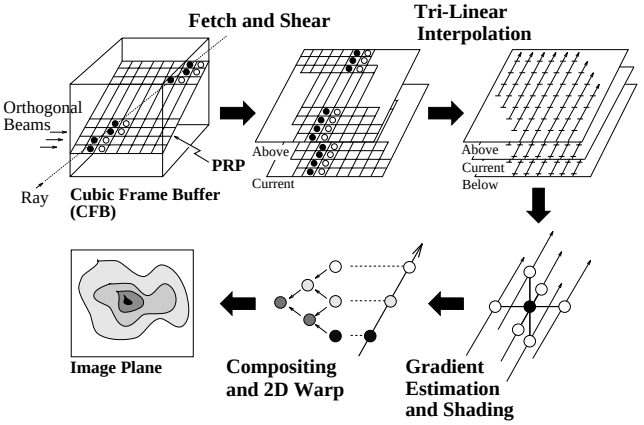


Figure 2: Real-Time Ray-Casting.

As soon as two PRPs are stored in two 2D buffers (referred to as the *above* and *current* buffers in Figure 2), a tri-linear interpolation is performed to generate

sample points on continuous rays using the voxels of four discrete rays as input data (see Section 3). The two 2D buffers generate one interpolated plane of continuous rays. Three such planes, *above*, *below* and *current*, are needed for local gradient approximations using neighboring rays (see Section 4).

The samples of the rays are shaded and opacities are assigned using a user controllable transfer function. The shaded rays are composited into a final pixel color using a parallel implementation of the front-to-back (or back-to-front) compositing:

$$\begin{aligned} C' &= C_L + (1-\alpha_L)C_R \\ \alpha' &= \alpha_L + (1-\alpha_L)\alpha_R \end{aligned} \quad (1)$$

Here the subscripts L and R indicate sample color C or opacity α from left or right children of the binary tree, respectively. Other parallel projection schemes such as first or last opaque projection, maximum or minimum voxel value and weighted summation can also be employed.

The next section discusses the issues of tri-linear interpolation between discrete rays to generate continuous rays, and Section 4 shows how to compute the local gradient at each continuous sample point.

3. Sheared Tri-Linear Interpolation

Tri-linear interpolation generates a value at non-integer locations by fetching the eight surrounding voxels and interpolating as follows:

$$\begin{aligned} P_{abc} = {} & P_{000}\,(1-a)(1-b)(1-c) + P_{100}\,a(1-b)(1-c) + \\ & P_{010}\,(1-a)b(1-c) + P_{001}\,(1-a)(1-b)c + \\ & P_{101}\,a(1-b)c + P_{011}\,(1-a)bc + \\ & P_{110}\,ab(1-c) + P_{111}\,abc. \end{aligned} \quad (2)$$

Here the relative 3D coordinate of a sample point within a cube with respect to the corner voxel closest to the origin is $\langle a, b, c\rangle$ and the data values associated with the corner voxels of the cube are P_{ijk}, where i, j, k = 0 or 1, and the interpolated data value associated with the sample point is P_{abc}. Different optimizations aim at reducing the arithmetic complexity of this operation [9, 16], but the arbitrary memory access to fetch eight neighboring voxels for each sample point makes this one of the most time consuming operations during volume rendering.

By transforming discrete rays from the PRP so that they are aligned and storing them in two 2D buffers (see Figure 2), we can greatly reduce this data access and communication cost. Instead of fetching the eight-neighborhood of each resampling location, four discrete rays are fetched from the buffer, two from each of the above and below planes. In parallel implementations, neighboring rays reside in adjacent interpolation modules, requiring only a local shift operation of one voxel unit between neighbors.

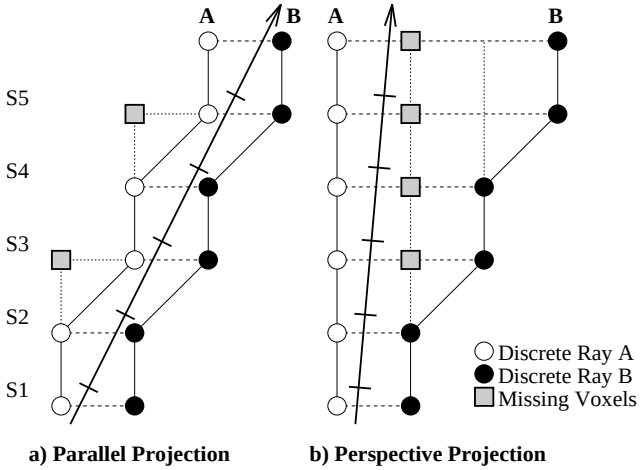


Figure 3: Problems with Discrete Ray Interpolation.

However, there is a problem intrinsic to interpolation between discrete rays. Figure 3 illustrates this in 2D. The samples on the continuous ray have to be interpolated using bi-linear interpolation between samples of the discrete rays A (white) and B (black). Sample S1 can be correctly interpolated using four voxels from A and B, since they form a rectangle, i.e., the rays do not make a discrete step to the left or right.

As soon as the discrete rays step to the left or right as is the case for samples S2 and S4, the neighboring voxels form a parallelogram, and a straightforward bi-linear interpolation would produce the wrong sample values. The grey shaded square voxels in Figure 3a would be needed to yield the correct result, but they reside on rays two units apart from ray B.

This problem is exacerbated for perspective projections (Figure 3b). The discrete rays diverge, and the correct neighboring voxels are not even stored in the 2D plane buffers. For example, only two voxels of ray A contribute to the correct interpolation at sample point S3. In the 3D case as many as six voxels may be missing in the immediate neighborhood of a sample point for perspective projections.

The solution is to perform a *sheared tri-linear interpolation* by factoring it into four linear and one bi-linear interpolation. Instead of specifying the sample location with respect to a corner voxel closest to the origin, each 3D coordinate along the ray consists of relative weights for linear interpolations along each axis in possibly sheared voxel neighborhoods. These weights can be pre-computed and stored in the X/Y-templates discussed in Section 2. Figure 4 shows the necessary interpolation steps in 3D.

First we perform four linear interpolations in direction of the major axis (the Z-axis in Figure 4) using

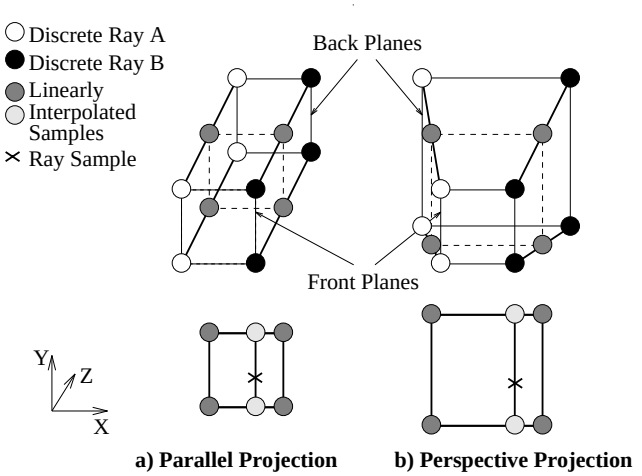


Figure 4: Sheared Tri-Linear Interpolation.

eight voxels of four neighboring discrete rays inside the 2D buffers. These eight voxels are the vertices of an oblique parallelepiped for parallel projections (see Figure 4a) or of a frustum of a pyramid for perspective projections (see Figure 4b). Four voxels each reside on two separate planes one unit apart, which we call the front or the back plane depending on when it is encountered during ray traversal in the direction of the major axis. Therefore, only one weight factor has to be stored, corresponding to the distance between the front plane and the position of the ray sample point. The resulting four interpolated values form a rectangle and can be bi-linearly interpolated to yield the final sample value. We split this bi-linear interpolation into two linear interpolations between the corner values and a final linear interpolation between the edge values. At the bottom of Figure 4 this is shown as two interpolations in X-direction followed by one interpolation in Y-direction.

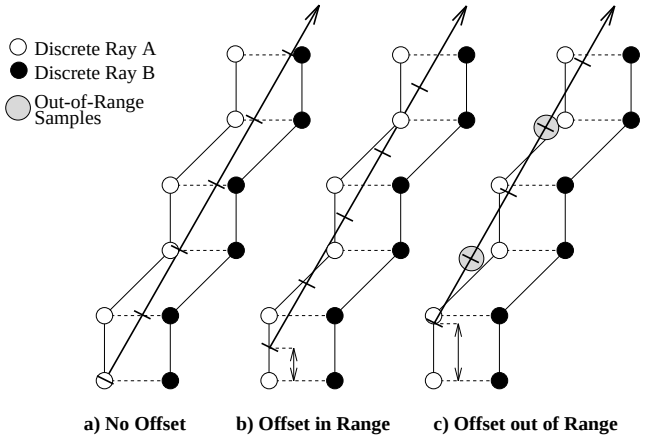


Figure 5: Variable Ray Offsets in Major Direction.

The sample points corresponding to the continuous rays have to be inside the polyhedron defined by the voxels on the four surrounding discrete rays. When constructing the discrete rays, all continuous rays start at integer positions of the base plane, i.e., they coincide with voxels of the first slice of the volume dataset. However, as Figure 5a shows, using these rays during ray-casting effectively reduces the tri-linear interpolation to a bi-linear interpolation, because all sample points along the ray fall onto the front planes of the parallelepipeds or pyramid frustum.

Using X and Y integer positions on the base-plane we can allow an offset from the base-plane in major direction as a degree of freedom and are able to perform sheared tri-linear interpolations (Figure 5b). But for offsets in major direction that are too big, as shown in Figure 5c), some of the samples along the rays may fall outside the bounding box defined by the discrete rays.

In order to get an upper bound for admissible offsets we have to understand how steps in non-major direction along discrete rays occur. Figure 6 shows the situation in 2D. The view vector is split into a dx component along the X-axis (dx and dy in 3D) and a unit vector in direction of the major axis (the Y-axis in Figure 6). Stepping in direction of the major axis, we add the viewing vector to the current sample position at S_n in order to get the new sample position at S_{n+1}.

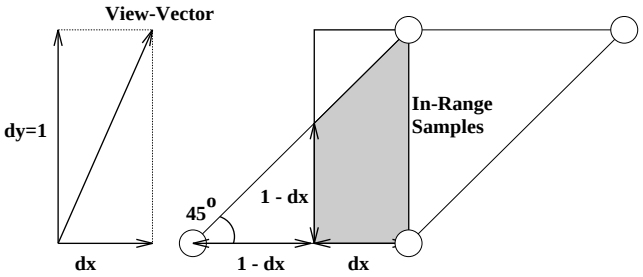


Figure 6: Maximum Offset Estimation.

Suppose that the addition of dx at point S_n leads to a step of the discrete rays in x direction. This step can only occur if S_n has a relative x offset with respect to the lower left corner voxel of more than $1 - dx$ for positive dx (or less than $1 + dx$ for negative dx). In other words, sample S_n was inside the rectangle of size dx by 1 shown in Figure 6. However, only the shaded region of this rectangle contains sample positions inside the parallelepiped defined by the corner voxels. Taking the smallest side in major axis as the worst-case, this means that in-range samples have a maximal relative y offset of no more than $1 - dx$ for positive dx (no less than $1 + dx$ for negative dx).

Since we step with a unit vector in the direction of the major axis, all relative offsets along the ray are determined by the offsets of the first ray samples from the base-plane. The above argument easily extends to 3D, making the maximum allowed offset in direction of the major axis:

$$min(1 - dx, 1 - dy), \qquad dx, dy \geq 0$$

$$
\begin{aligned}
&min(1+dx, 1-dy), && dx < 0, dy \geq 0 \\
&min(1-dx, 1+dy), && dx \geq 0, dy < 0 \\
&min(1+dx, 1+dy), && dx, dy < 0,
\end{aligned} \tag{3}
$$

where dx and dy are the components of the viewing vector in x and y direction, respectively. Notice that for 45^o viewing angle dx and dy are 1, yielding an offset of 0 and bi-linear interpolation as in Figure 5a. This fact will be of importance when discussing the results in Section 6.

In our implementation we cast a single ray from the origin of the image plane onto the base-plane using uniform distance between samples and choose the offset in major direction of the first sample after penetration of the base-plane. If necessary the offset is iteratively reduced until it satisfies the above condition. This leads to view dependent offsets in major direction and to varying resampling of the dataset. The variation of resampling points according to viewing direction is an advantage for interactive visualization, because more of the internal data structure can be revealed.

Each discrete ray consists of n voxels, independent of the viewing direction. Since the maximum viewing angle difference with the major axis is not more than 45 degrees, the volume sample rate is defined by the diagonal through the cube and is by a factor of $\sqrt{3}$ higher for orthographic viewing. We found that for ray-compositing this is not an important consideration due to the averaging nature of the compositing operator.

A more severe problem is the varying size of the sample neighborhood (see Figure 4). For parallel projections, the eight voxels surrounding the sample point either form a cube with sides of length one or an oblique parallelepiped as in Figure 4a. For perspective projections, however, the surrounding voxels may form the frustum of a pyramid with parallel front and back planes as in Figure 4b. Due to the divergence of rays towards the back of the dataset, the volume spanned by this frustum increases, thereby reducing the precision of the tri-linear interpolation. However, we found that the distance between neighboring discrete rays at the end of the volume never exceeded two voxels for a 256^3 dataset while still achieving a high amount of perspectivity. Furthermore, in typical datasets the samples at the back of the volume have little influence on the final pixel color due to compositing along the ray.

The center of projection C and the field-of-view (FOV) in perspective projections also influence the sampling rate (see Figure 7). The discrete line algorithm casts exactly one ray per pixel of the base-plane, or a maximum of $2n$ rays per scanline. In cases where the FOV extends across the the dataset (Figure 7a) this guarantees better sampling than regular image order ray-casting, which would cast n rays spanning the FOV and send wasteful rays that miss the dataset. However, for a small FOV the discrete line stepping yields undersampling in the active regions of the base-plane (Figure 7b). Figure 7c shows a case where two base-plane images contribute to the final view image. The worst case in 3D is the generation of three base-plane projections for a single perspective image.

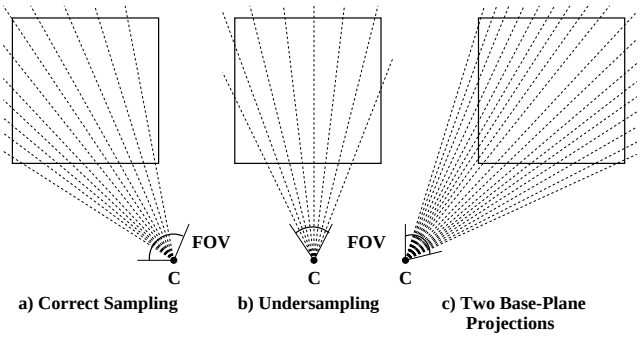


Figure 7: Sampling for Perspective Projections.

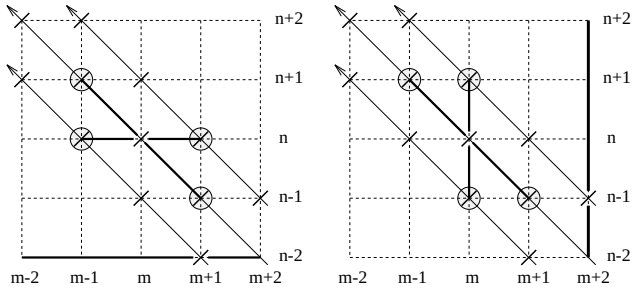
Section 6 presents comparisons between image order ray-casting using a view independent sampling rate along the rays, tri-linear interpolation employing equation 2 using the correct voxels, and the proposed sheared tri-linear interpolation among discrete rays. The next section describes methods for gradient estimation using samples on neighboring rays.

4. ABC Gradient Estimation

To approximate the surface normals necessary for shading and classification we use the gray-level gradient which is computed by the differences between the values of the current sample and its immediate neighbors [5]. In order to evaluate the gradient at a particular point, we form central differences between the tri-linearly interpolated values of rays on the immediate left, right, above and below, as well as the values of the current ray. Since this amounts to storing three consecutive planes of ray samples, we call this method *ABC gradient estimation* for the above, below, and current ray sample buffers.

The simplest approach, shown in Figure 8 for 2D, is to use the 6-neighborhood gradient, which uses the differences of neighboring sample values along the ray, $P_{(n,m+1)} - P_{(n,m-1)}$ in base-plane direction and $P_{(n+1,m-1)} - P_{(n-1,m+1)}$ in the ray direction. Although the left, right, above and below ray samples are in the same plane and orthogonal to each other, the samples in the ray direction may be slanted. A more critical problem occurs during a switch of base-plane. Figure 8a shows the situation for almost 45^o viewing direction, where an image is projected onto the horizontal base-plane. For any angle greater than 45^o a switch of base-planes occurs, and the values of $P_{(n+1,m)} - P_{(n-1,m)}$ are used instead to calculate the gradient in the base-plane direction. This leads to intolerable temporal aliasing.

We also simulated the use of a 26-neighborhood gradient (Figure 9). Instead of fetching sample values from



(a) 6-neighborhood, , Horizontal Base-Plane

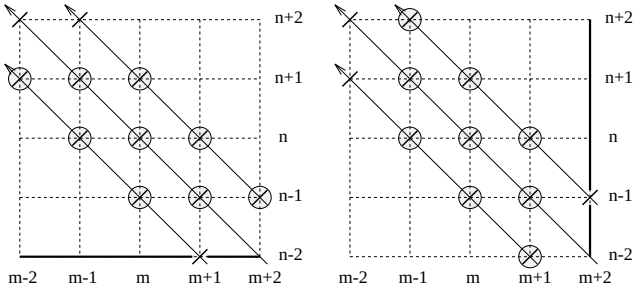
(b) 6-neighborhood, Vertical Base-Plane

Figure 8: 6-neighborhood Gradient.

four neighboring rays, 26 interpolated samples from 8 neighboring rays are fetched. Each sample is assigned a weight factor corresponding to the inverse Manhattan distance in the interpolated buffer to the center sample. For example, sample $P_{(n,m-1)}$ in Figure 9a has a weight of 1, whereas sample $P_{(n+1,m-2)}$ has a weight of $\frac{1}{2}$. In 3D we also get weight factors of $\frac{1}{3}$ for the corner samples of the 26-neighborhood. However, to simplify the arithmetic we use powers of 2, so that these samples are multiplied by a weight of $\frac{1}{4}$. The gradient is estimated by taking weighted sums of ray samples and differences between opposite sample planes. For the 2D example in Figure 9a this corresponds to:

$$\begin{aligned} G_{base} &= [\tfrac{1}{2}P_{(n+1,m)} + P_{(n,m+1)} + \tfrac{1}{2}P_{(n-1,m+2)}] - \\ &\quad [\tfrac{1}{2}P_{(n+1,m-2)} + P_{(n,m-1)} + \tfrac{1}{2}P_{(n-1,m)}] \\ G_{ray} &= [\tfrac{1}{2}P_{(n+1,m-2)} + P_{(n+1,m-1)} + \tfrac{1}{2}P_{(n+1,m)}] - \\ &\quad [\tfrac{1}{2}P_{(n-1,m)} + P_{(n-1,m+1)} + \tfrac{1}{2}P_{(n-1,m+2)}] \end{aligned} \quad (4)$$

This method leads to better overall image quality when compared to the 6-neighborhood gradient, but the switching of major axis is still noticeable (compare Figure 9a and 9b).

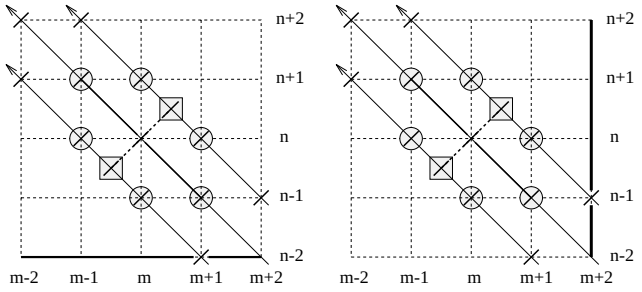


(a) 26-neighborhood, Horizontal Base-Plane

(b) 26-neighborhood, Vertical Base-Plane

Figure 9: 26-neighborhood Gradient.

To circumvent this problem we take a similar approach to the 6-neighborhood method but use an additional linear interpolation step to resample the rays on correct orthogonal positions. Figure 10 shows how the round samples on the left and right ray are used to linearly interpolate the correct square samples. We call this approach the 10-neighborhood gradient estimation for the 3D case, since 10 voxels participate in the computation. It adequately solves the problem of switching the major axis during object rotations and yields high image quality. The linear interpolation weights are constant along a ray and correspond to a shift of all samples in the viewing direction. Section 6 presents a direct comparison between the 6-, 10- and 26-neighborhood gradient methods.



(a) 10-neighborhood, Horizontal Base-Plane

(b) 10-neighborhood, Vertical Base-Plane

Figure 10: 10-neighborhood Gradient.

In the case of perspective projections, the front of each PRP is uniformly sampled with n rays one unit apart. As the rays diverge towards the back of the volume, the distance between rays increases, and the gradient estimation becomes less accurate. However, because of the usually small distance between rays and due to the averaging nature of shading, classification and compositing, these effects do not influence image quality for typical datasets.

With the gradient estimation and light vector directions, the sample intensity can be generated using a variety of shading methods (e.g., using lookup tables [10]). Opacity values for compositing are generated using a transfer function represented as a 2D lookup table indexed by sample density and gradient magnitude [11].

The next section shows how the presented sheared tri-linear interpolation and ABC gradient estimation are supported in the Cube-3 architecture in order to achieve real-time 4D visualization.

5. Cube-3 Architecture

Cube-3 is a special-purpose real-time volume visualization system that allows for the display of high-resolution 512^3 16-bit per voxel datasets at frames rates over 20 Hz. It contains a large CFB memory to hold the volumetric dataset and performs base-plane projections according to user controlled parameters. A host computer, connected to Cube-3 and containing the frame buffer for the final image display, runs the user inter-

face software and performs the final 2D image warp onto the viewing plane. Real-time acquisition devices such as a confocal microscope, microtomograph, ultrasound, or a computer running a simulation model are tightly coupled to the Cube-3 memory using high-bandwidth optical links for the input of dynamically changing 3D datasets.

The Cube-3 architecture is highly-parallel and pipelined [16]. Figure 11 shows a block diagram of the overall dataflow. The CFB is a 3D memory organized in n dual-access memory modules, each storing n^2 voxels. A special 3D skewed organization enables the conflict-free access to any beam of n voxels [7]. PRPs are fetched as a sequence of voxel beams and stored in consecutive 2D Skewed Buffers (2DSB). A high-bandwidth interconnection network, the Fast Bus, allows the alignment of the discrete rays on the PRP parallel to a main axis in the 2DSB modules.

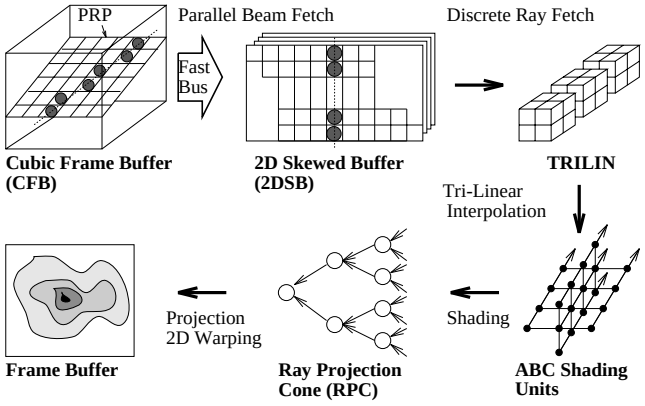


Figure 11: Cube-3 System Overview.

Three 2DSBs are used in a pipelined fashion to support sheared tri-linear interpolation. Aligned discrete rays from 2DSBs are fetched conflict-free and placed into special purpose Tri-Linear Interpolation (TRILIN) units. The resulting continuous projection rays are placed onto ABC Shading Units, where the gradients are estimated and each ray sample is converted into both an intensity and an associated opacity value according to lighting and data segmentation parameters. These intensity/opacity ray samples are fed into the leaves of a Ray Projection Cone (RPC). The RPC is a folded binary tree that generates in parallel and in a pipelined fashion the final pixel value using a variety of projection schemes on the cone nodes. The resulting base-plane pixel is transmitted to the host where it is post-processed (e.g., post-shaded or splatted) and 2D transformed (warped) onto the viewing plane. The result is stored in the 2D frame-buffer.

The parallel conflict-free memory architecture of Cube-3 reduces the memory access bottleneck from $O(n^3)$ per projection to $O(n^2)$ and allows for very high data throughput. For a dataset size of 512^3 16-bit voxels we estimate a performance of up to 30 frames per second. Such a system would require 8 boards and a custom fabricated backplane.

Cube-3 is a scalable and flexible architecture that allows the user to interactively control the following parameters: viewing angle from any parallel and perspective direction, control over shading and projection (e.g., first opaque, maximum value, x-ray, compositing), color segmentation and thresholding, control over translucency, sectioning and slicing. It will provide a rendering performance that is an order of magnitude higher than that of previously reported systems and thereby revolutionize the way scientists conduct their studies.

6. Results

We implemented the different interpolation and gradient estimation methods in software and conducted several experiments. The first program, `VolRen` implements traditional image order volume rendering. Rays are cast from the image plane into the volume and sampled at uniform steps. The tri-linear interpolation is performed according to Equation 2 using the correct 8-neighborhood around sample points. The gradient is estimated using central differences of tri-linear interpolated values in a 6-neighborhood around each sample.

The second program, `True3D`, uses our real-time discrete ray-casting method, but instead of performing sheared tri-linear interpolation it fetches the exact 8-neighborhood around each sample point. The last program, `Sheared3D`, implements the same algorithm but with the proposed sheared tri-linear interpolation. Both `True3D` and `Sheared3D` can use any of the 6-, 26- or 10-neighborhood gradient methods for comparison purposes. For the implementation of these algorithms we used the VolVis volume visualization system, developed at the State University of New York at Stony Brook [2, 1]. (The source code of VolVis is freely available by sending email to *volvis@cs.sunysb.edu.*)

6.1. Tri-Linear Interpolation Comparison

First we compare images resulting from `Sheared3D` to results obtained from `VolRen` and `True3D`. The gradient approximation method used for `Sheared3D` and `True3D` was the proposed 10-neighborhood gradient estimation.

The dataset, a CT study of a cadaver head of size $256 \times 256 \times 225$ voxels at 8-bit per voxel, was taken on a General Electric CT Scanner and provided courtesy of North Carolina Memorial Hospital. All programs use the same shading model and an opacity transfer function that maps voxel values below 80 to $\alpha = 0$, has a linear ramp for α from 0 to 0.75 for values between 80 and 100, and assigns $\alpha = 0.75$ to values above 100. We chose this particular transfer function to classify bone

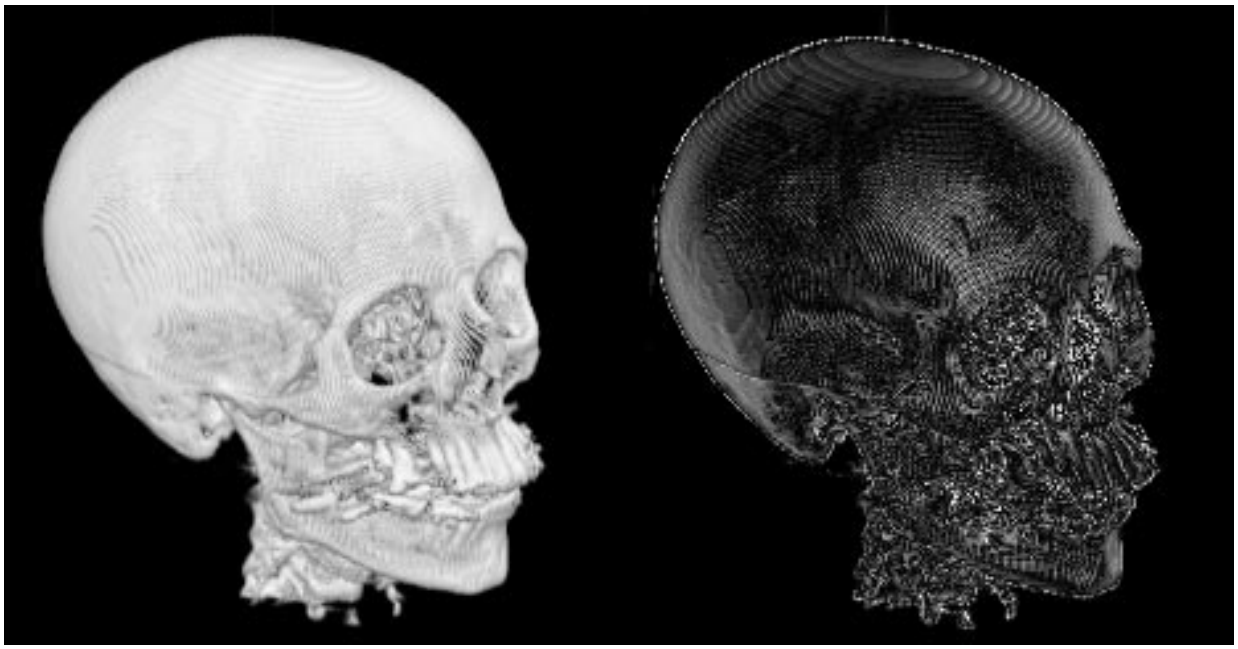


Figure 12: Dataset rendered with sheared tri-linear interpolation (left) and the difference image to traditional volume rendering (right) for 45^o rotation angle. This is the worst case for sheared tri-linear interpolation.

in the dataset as opaque in order to try to maximize the display of aliasing effects on the forehead of the CT skull.

For the experiments we rotated the dataset by 70^o around the horizontal axis with respect to the world coordinate system, and during animations we rotated it around a vertical axis between 0^o and 90^o in steps of 5^o. As error measure between the resulting images we use the average Euclidean distance of RGB values between corresponding pixels. Figure 12 shows the dataset rotated by 45^o around the vertical axis. The left image was generated using `Sheared3D` and the image on the right is the difference image, mapped to gray-scale, comparing the corresponding `Sheared3D` and `VolRen` images for this rotation angle.

Figure 13 shows the relative Euclidean error in percentage between images from `Sheared3D` and `VolRen` and between `Sheared3D` and `True3D`, respectively. The comparison with `VolRen` (top curve) shows how the error raises towards 45^o rotation angle and reaches a minimum at 0^o and 90^o. The peak at 45^o is due to the different sampling distance along the ray, which is by $\sqrt{3}$ bigger for discrete line stepping (see Section 3). Furthermore, due to the offset considerations explained in Section 3, our algorithm performs only bi-linear interpolation as opposed to the the tri-linear interpolation in `VolRen`.

The comparison to `True3D` shows zero error for 45^o because both algorithms perform bi-linear interpolation and use the same gradient estimation technique. The relative error in percent compared to `VolRen` stays below 1.3%, and compared to `True3D` it stays below 0.3%.

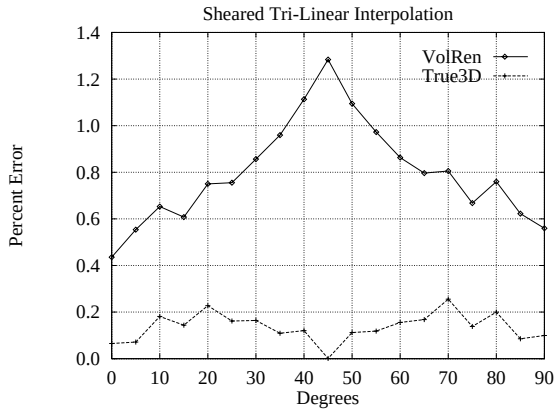


Figure 13: Sheared Interpolation Percentage Error.

6.2. ABC Gradient Estimation Comparison

For the comparison of the different ABC gradient estimation techniques we use a voxelized model of a sphere as dataset. The sphere is scan-converted using the volume sampling method described in [19]. The surface intersection points are obtained by thresholding, i.e., as soon as a certain voxel value is exceeded we calculate the gradient at that point. Each gradient is compared to the true geometric surface normal. As error measure we use the magnitude of angular difference between two vectors. All differences are accumulated and averaged over all surface intersection points.

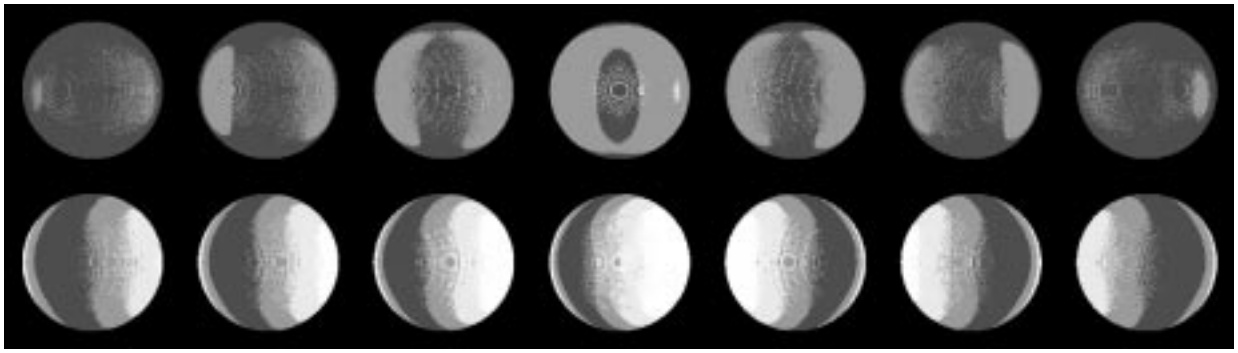


Figure 14: Error magnitude of comparing surface normals of 10- (Top) and 26-neighborhood gradients (Bottom) to the true analytic normal of the voxelized sphere. Notice the jump of regions of high error for the 26-neighborhood gradient between 45^o and 50^o rotation angle.
Dark: $0^o \leq |e| < 8.5^o$, Medium: $8.5^o \leq |e| < 20^o$, Light: $20^o \leq |e| < 31.5^o$, White: $|e| \geq 31.5^o$.
Rotation angles (left to right): $30^o, 35^o, 40^o, 45^o, 50^o, 55^o, 60^o$.

Figure 15 shows the results of rotating the sphere around a vertical axis between 0^o and 90^o in steps of 5^o. The top two curves compare the analytic normal with the 26- and the 6-neighborhood gradient, respectively. The error increases towards 45^o rotation angle due to the non-orthogonality of the gradient directions which reaches a maximum at 45^o. Although the 26-gradient shows a little higher error magnitude, the difference between these two methods is not significant.

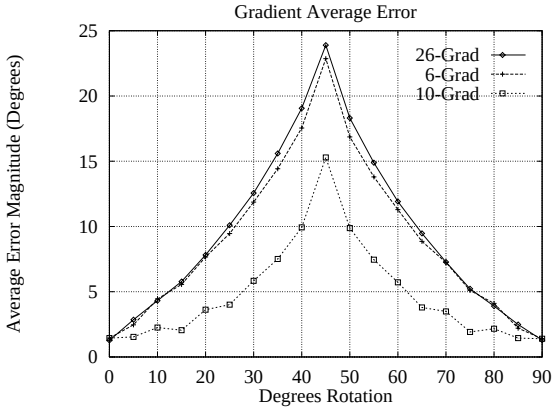


Figure 15: Average Error Magnitude for ABC Gradient Estimations Compared to the Analytic Normal.

The curve on the bottom in Figure 15 shows the comparison of the analytic normal with the 10-neighborhood gradient estimation. The error magnitude is significantly smaller than for the other gradient methods. The error also increases towards 45^o rotation angle. This is due to the different distances between samples that are used for the gradient calculations in the three orthogonal directions.

Figure 14 shows how the error propagates around the sphere for rotation angles from 30^o to 60^o in steps of 5^o. Dark shaded regions indicate regions of low error magnitude, light shaded regions indicate higher error magnitudes. The top row shows the 10-neighborhood gradient method with a fairly regular error transition from left to right during a switch of base-planes at 45^o (center sphere). The bottom row, depicting the 26-neighborhood gradient method, shows a generally larger error magnitude. Additionally, the region of largest error jumps from the right side of the sphere to the left during the switch of base-planes. This jump leads to noticeable changes in image intensity during object rotation, an effect that we described as temporal aliasing in Section 4.

7. Conclusions

In order to achieve the goal of real-time visualization of dynamic datasets we developed Cube-3, a scalable architecture that exploits parallelism and pipelining. In this paper we presented the underlying real-time ray-casting approach that allows for a mapping of ray-samples onto voxels that is one-to-one. Using templates and shearing/de-fanning of beams, we fetch 2D planes from the volume dataset and perform sheared tri-linear interpolation between discrete neighboring rays. Using the resulting interpolated ray samples from above, current and below planes, we described novel ways of gradient estimation using coherency between rays.

Using software simulations we compared the proposed methods to traditional image order ray-casting. The error of using sheared tri-linear interpolation instead of performing image order ray-casting is below 1.3% relative difference in Euclidean distance of the resulting image pixels. We showed that use of the proposed 10-neighborhood instead of a 6- or 26-neighborhood gradient approach reduces both the average error compared to analytically computed normals and the temporal aliasing that arises from switching base-planes during object rotations. We presented both

methods in the context of Cube-3, a special purpose architecture aimed at real-time 4D visualization of high-resolution volumetric datasets.

8. Acknowledgments

This work has been supported by the National Science Foundation under grant CCR-9205047. We would like to thank Lisa Sobierajski and Rick Avila for their helpful suggestions during the development of these methods. Sidney Wang provided us with the sphere dataset and helped with the generation of Figure 14. A discussion with Claudio Silva gave us the insight into the various error metrics we used. We also thank Patrick Tonra for helpful system administration during the more hectic moments in the development of this project.

References

1. Avila, R., He, T., Hong, L., Kaufman, A., Pfister, H., Silva, C., Sobierajski, L., and Wang, S. VolVis: A diversified system for volume visualization research and development. To appear in *Proceedings of Visualization '94* (Washington, DC, Oct. 1994).
2. Avila, R., Sobierajski, L., and Kaufman, A. Towards a comprehensive volume visualization system. In *Proceedings of Visualization '92* (Boston, MA, Oct. 1992), IEEE Computer Society Press, pp. 13–20.
3. Cameron, G., and Underill, P. E. Rendering volumetric medical image data on a SIMD architecture computer. In *Proceedings of Third Eurographics Workshop on Rendering* (May 1992).
4. Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics 23, No. 3* (July 1989), 79–88.
5. Höhne, K. H., and Bernstein, R. Shading 3D-images from CT using gray-level gradients. *IEEE Transactions on Medical Imaging MI-5*, 1 (Mar. 1986), 45–47.
6. Kaufman, A. *Volume Visualization.* IEEE CS Press Tutorial, Los Alamitos, CA, 1991.
7. Kaufman, A., and Bakalash, R. Memory and processing architecture for 3D voxel-based imagery. *IEEE Computer Graphics & Applications 8*, 6 (Nov. 1988), 10–23.
8. Kaufman, A., and Shimony, E. 3D scan-conversion algorithms for voxel-based graphics. In *ACM Workshop on Interactive 3D Graphics* (Chapel Hill, NC, Oct. 1986), pp. 45–76.
9. Knittel, G. VERVE: Voxel engine for real-time visualization and examination. In *Computer Graphics Forum* (September 1993), vol. 12, No. 3, pp. C–37 – C–48.
10. Lacroute, P., and Levoy, M. Fast volume rendering using a shear-warp factorization of the viewing transform. *Computer Graphics, Proceedings of SIGGRAPH '94* (July 1994), 451–457.
11. Levoy, M. Display of surfaces from volume data. *IEEE Computer Graphics & Applications 8*, 5 (May 1988), 29–37.
12. Levoy, M. Design for real-time high-quality volume rendering workstation. In *1989 Workshop on Volume Visualization* (Chapel Hill, NC, May 1989), pp. 85–90.
13. Levoy, M. Efficient ray tracing of volume data. *ACM Transactions on Graphics 9*, 3 (July 1990), 245–261.
14. Levoy, M. Volume rendering by adaptive refinement. *The Visual Computer* (July 1990), 2–7.
15. Molnar, S., Eyles, J., and Poulton, J. Pixelflow: High-speed rendering using image composition. *Computer Graphics 26*, 2 (July 1992), 231–240.
16. Pfister, H., Kaufman, A., and Chiueh, T. Cube-3: A Real-Time Architecture for High-Resolution Volume Visualization. To appear in *1994 Workshop on Volume Visualization* (Washington, DC, Oct. 1994).
17. Schröder, P., and Stoll, G. Data parallel volume rendering as line drawing. In *1992 Workshop on Volume Visualization* (Boston, MA, Oct. 1992), pp. 25–31.
18. Vézina, G., Fletcher, P., and Robertson, P. Volume rendering on the MasPar MP-1. In *1992 Workshop on Volume Visualization* (Boston, MA, Oct. 1992), pp. 3–8.
19. Wang, S., and Kaufman, A. Volume sampled voxelization of geometric primitives. In *Proceedings of Visualization '93* (San José, CA, Oct. 1993), IEEE Computer Society Press, pp. 78–84.
20. Yagel, R., and Kaufman, A. Template-based volume viewing. *Computer Graphics Forum 11*, 3 (Sept. 1992), 153–167.