Cube-3: A Real-Time Architecture for High-Resolution Volume Visualization

Hanspeter Pfister, Arie Kaufman, and Tzi-cker Chiueh
Department of Computer Science
State University of New York at Stony Brook

Abstract

This paper describes a high-performance special-purpose system, Cube-3, for displaying and manipulating high-resolution volumetric datasets in real-time. A primary goal of Cube-3 is to render 512^3, 16-bit per vozel, datasets at about 30 frames per second. Cube-3 implements a ray-casting algorithm in a highly-parallel and pipelined architecture, using a 3D skewed volume memory, a modular fast bus, 2D skewed buffers, 3D interpolation and shading units, and a ray projection cone. Cube-3 will allow users to interactively visualize and investigate in real-time static (3D) and dynamic (4D) high-resolution volumetric datasets.

1. Introduction

A volumetric dataset is typically represented as a 3D regular grid of voxels (volume elements) representing some uniform or piecewise property of an object or phenomenon. This 3D dataset is commonly stored in a regular cubic frame buffer (CFB), which is a large 3D array of voxels (e.g., 128M voxels for a 512^3 dataset) and is displayed on a raster screen using a direct volume rendering technique (see e.g., [14, 9]). Alternatively, the dataset may be represented as a sequence of cross-sections or as an irregular grid.

Applications of volume visualization include sampled, simulated, and modeled datasets in confocal microscopy, astro- and geophysical measurements, molecular structures, finite element models, computational fluid dynamics, and 3D reconstructed medical data, to name just a few (see [9] Chapter 7). As with other display methods of 3D objects, the provision of real-time data manipulation rates, typically defined to be more than 10 and preferably 30 frames per second, is essential for the investigation of 3D *static* datasets. Furthermore, in many *dynamic* applications, 4D (spatial-temporal) real-time visualization is a necessary component of an integrated acquisition-visualization system. Examples are the real-time analysis of an in-vivo specimen under a confocal microscope or the real-time study of in-situ fluid flow or crack formation in rocks under Computed Microtomography (CMT).

High-bandwidth memory access and high arithmetic performance are key elements of real-time volume rendering and can be met by exploiting parallelism among a set of dedicated processors [5, 16, 15, 10] [9, Chapter 6]. Sub-second rendering times for a 128^3 dataset have been reported by Schröder and Stoll on a Princeton Engine of 1024 processors [18] and by Vézina et al. on a 16k-PE MasPar MP-1 computer [22]. For higher resolution datasets, however, the number of processors and their interconnection bandwidths must increase, placing hard-to-meet requirements on existing general-purpose workstations or supercomputers.

The main objective of the Cube-3 architecture is to develop a special-purpose real-time volume visualization system for high-resolution 512^3 16-bit per voxel datasets that achieves frame rates over 20 Hz. This rendering performance is orders of magnitude higher than that of previously reported systems, while requiring only moderate hardware complexity. Furthermore, Cube-3 allows interactive control over a wide variety of rendering and segmentation parameters. The availability of such a system will revolutionize the way scientists and engineers conduct their studies.

2. System Overview

Figure 1 shows the overall organization of two real-time volume visualization environments. The host computer controls the entire environment and runs non time-critical parts of the Cube-3 software. It also contains a frame buffer for the final image display on a color monitor. The acquisition device is a sampling device such as a confocal microscope, microtomograph, ultrasound, MRI, or CT scanner. Alternatively, the acquisition device is a computer running either a simulation model (e.g., computational fluid dynamics) or synthesizing (voxelizing) a voxel-based geometric model from a display list (e.g., CAD). The sampled, simulated, or modeled dataset is either a sequence of cross sections, a regular 3D reconstructed volume, or an irregular data that has been converted into a regular volume. All these

Authors' Address: Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400
email pfister@cs.sunysb.edu,ari@cs.sunysb.edu,chiueh@cs.sunysb.edu

0-8186-7067-3/95 $4.00 © 1995 IEEE

formats can be maintained and archived by the filing module of the host software.

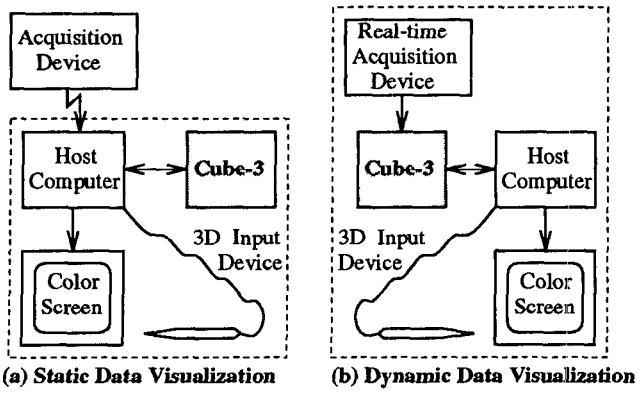


Figure 1: Real-Time Volume Visualization Systems.

Figure 1 (a) shows an environment in which the acquisition and reconstruction are performed in several seconds to several minutes on the acquisition device, while the visualization and manipulation are running on Cube-3 in real-time. Such a system fulfills the needs for *static visualization*, where the nature of the acquired data is static and Cube-3 allows for interactive control of viewing parameters, clipping planes, shading parameters, and data segmentation. Figure 1 (b) shows the ultimate environment, in which Cube-3 is tightly-coupled with the real-time acquisition device to create an integrated acquisition-visualization system that would allow the real-time 4D (spatial-temporal) visualization of dynamic systems. The need for this *dynamic visualization* clearly will arise, since the data rates of emerging acquisition devices such as microtomographs will reach several complete 3D high-resolution datasets per second well before the end of the decade. In addition to controlling the Cube-3 volume visualization engine on a device driver level, the host also runs a volume visualization software system and user interface called VolVis [2], which has been developed at SUNY Stony Brook and complements the Cube-3 hardware.

The next section describes aspects of the Cube-3 system in detail. Sections 4 and 5 provide estimated performance and hardware real-estate.

3. Cube-3 Architecture

Cube-3 implements a ray-casting algorithm, a flexible and frequently used technique for direct volume rendering [14]. Figure 2 shows a block diagram of the overall dataflow. In order to meet the high performance requirements of real-time ray-casting, the Cube-3 architecture is highly-parallel and pipelined. The *Cubic Frame Buffer (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 (i.e., a ray parallel to a main axis) of n voxels (see Section 3.1). All the rays belonging to the same scan line of a parallel or perspective projection reside on a slanted plane inside the CFB, termed the *Projection Ray Plane (PRP)*. In order to support arbitrary viewing, each PRP is fetched as a sequence of voxel beams and stored in consecutive *2D Skewed Buffers (2DSB)* (see Section 3.2). 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 (see Section 3.3).

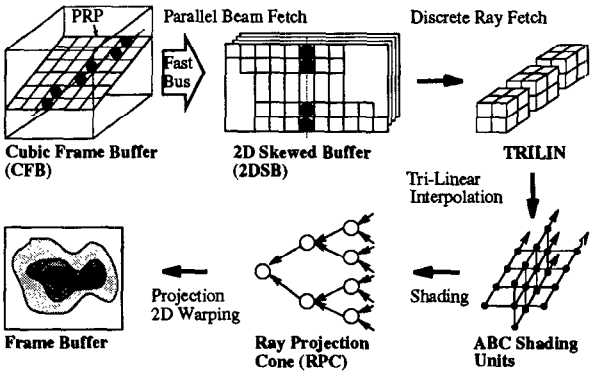


Figure 2: Cube-3 System Overview.

Several 2D Skewed Buffers are used in a pipelined fashion to support tri-linear interpolation and gray-level shading. Aligned discrete rays from 2DSBs are fetched conflict-free and placed into special purpose *Tri-Linear Interpolation (TRILIN)* units (see Section 3.4). The resulting continuous projection rays are placed onto *ABC Shading Units*, where each ray sample is converted into both an intensity and an associated opacity value according to lighting and data segmentation parameters (see Section 3.5). These intensity/opacity ray samples are fetched into the leaves of the *Ray Projection Cone (RPC)*. The RPC is a folded binary tree that generates in parallel and in a pipelined fashion the final ray-pixel value using a variety of projection schemes on the cone nodes (see Section 3.6). The resulting pixel is post-processed (e.g., post-shaded or splatted), 2D transformed, and stored in the 2D frame-buffer.

3.1. Parallel Cubic Frame Buffer Organization

A special 3D skewed organization of the n^3 voxel CFB enables conflict-free access to any beam of n voxels [10]. The CFB consists of n memory modules, each with n^2 voxels and its own independent access and addressing unit. A voxel with space coordinates (x, y, z) is being mapped onto the k-th module by:

$$k = (x + y + z) \bmod n \qquad 0 \leq k, x, y, z \leq n - 1.$$

Since two coordinates are always constant along any beam, the third coordinate guarantees that one and only one voxel from the beam resides in any one of the

modules. The internal mapping (i, j) within the module is given by: $i = x$, $j = y$.

This skewing scheme has successfully been employed in Cube-1 [10] and Cube-2 [3], first and second generation 16^3 prototype architectures built at SUNY Stony Brook. They use a sequence of n processing units, which team up to generate the first opaque projection along a beam of n voxels in $O(\log n)$ time, using a voxel multiple-write bus [6, 10]. Consequently, the time necessary to generate an orthographic projection of n^2 pixels is only $O(n^2 \log n)$, rather than the conventional $O(n^3)$ time. However, in this technique projections can be generated only from a finite number of predetermined directions [4]. The next section describes the enhanced architecture used in Cube-3 for arbitrary parallel and perspective viewing.

3.2. Architecture for Arbitrary Viewing

All the rays belonging to the same scan line of the 2D frame-buffer reside on the same plane, the PRP (see Figure 3). For every parallel and perspective projection, all the PRPs can be made parallel to one major axis by fixing a degree of freedom in specifying the projection parameters. For example, in parallel projection the projection plane can be rotated about the viewing axis, which can be reversed after projection in the 2D frame-buffer. Since there is no direct way to fetch arbitrary discrete rays from the CFB conflict free, a whole PRP of beams (which are now parallel to an axis) is instead fetched in n cycles, beam after beam, and stored in a 2D temporary buffer, the 2DSB.

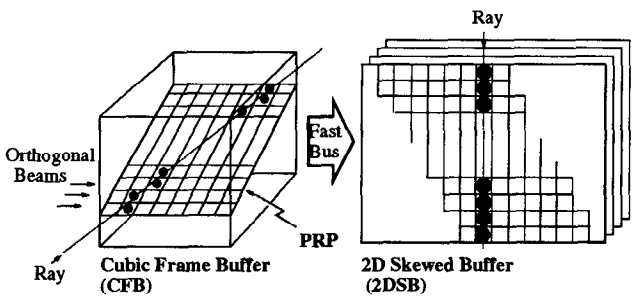


Figure 3: Arbitrary Viewing Mechanism.

The direction of the viewing ray within the original PRP depends on the observer's viewing direction. When a PRP is moved from the CFB to the 2DSB, it undergoes a 2D shearing (either to the left or to the right) to align all the viewing rays into beams along a direction parallel to a 2D axis (e.g., vertical). This step is a de-skewing step that is accomplished by the high-bandwidth Fast Bus that interconnects the CFB and 2DSB memory modules. Once the discrete viewing rays are aligned vertically within the 2D memory, they can be individually fetched and placed at the leaves of the ray projection mechanism for interpolation and shading. Since there may be up to $2n - 1$ parallel viewing rays entering the PRP, the 2DSB size is $2n \times n$ voxels.

The 2DSB thus supports conflict-free storage of *horizontal* beams coming from the CFB and conflict-free retrieval of *vertical* discrete rays. The 2DSB is divided into n memory modules, each storing $2n$ voxels, and is skewed such that any module appears exactly once in every row and every column. To achieve this, the (i, j) voxel is mapped onto module $(i + j) \bmod n$ in location $i + j$ (see also [13], which is a hardware solution for 90^o rotation and mirroring of bitmaps).

As an example, assume that a parallel projection is performed without loss of generality approximately from the $+z$ direction; namely, for each projection ray, z grows faster than x and y in absolute value. The 26-connected discrete ray representation is pre-generated on the host computer using a 3D variation [11] [9, pp. 280–301] of Bresenham's algorithm modified for non-integer endpoints [7]. First, the representation of the projection of the ray along the fetch axis is generated. This representation determines which beams to fetch from the CFB for each PRP. These viewing parameters are broadcast to the CFB addressing units. Second, the ray parameters within the PRP are calculated, determining how much to shear each beam. For parallel projections all rays have the same slope and thus the generalized Bresenham's steps in both directions have to be calculated only once (cf. [23]). These pre-calculated slope templates are down-loaded from the host into the Fast Bus control units (see Section 3.3). Each PRP contributes to all pixels of one scanline in the final image up to a 2D transformation.

A perspective projection can be generated in a way similar to the parallel projection. However, for perspective projections several beams are averaged into a single beam that is stored in the 2DSB. The number of beams averaged depends on the divergence of the perspective rays. Voxels in a given beam are not only averaged but also scaled and sheared between the Fast Bus and 2DSB. This is equivalent to casting a fan of rays, where larger portions of the volume are sampled as the fan diverges. Note that the assumption that for each projection ray z grows faster than x and y in absolute value is not always true in perspective projection, which may require the separate processing of z-major, x-major, and y-major rays.

3.3. Modular Fast Bus

The Fast Bus is an interconnection network that allows the high-bandwidth transfer of data from the n CFB modules to the n 2DSB modules. It enables the arbitrary shearing and averaging necessary for parallel and perspective projections. A set of fast multiplexers and transceivers with associated control units and a multi-channel bus are used to accomplish the data transfer speeds necessary for real-time rendering.

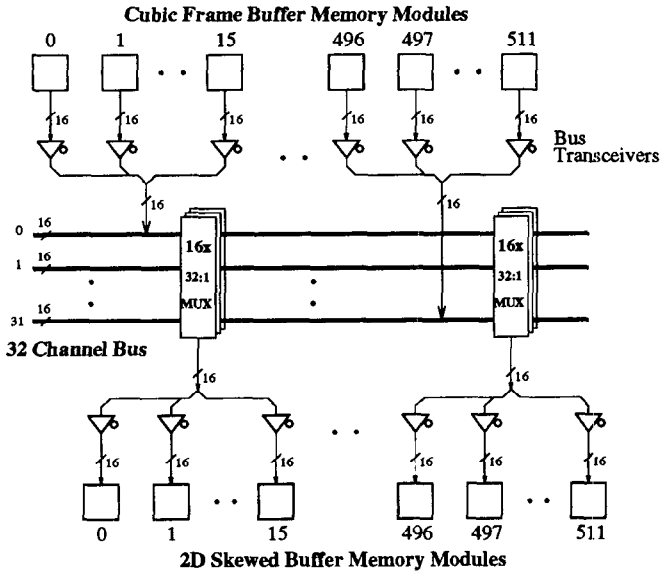


Figure 4: Fast Bus Configuration for $n = 512$.

Figure 4 shows the Fast Bus configuration with $n = 512$ CFB memory modules and 32 bus channels. The CFB modules are first divided into 32 groups of 16 memory modules each. The data from the 16 modules of each group are time-multiplexed onto the designated 16-bit Fast Bus channel for that group. The data now appears on the Fast Bus as illustrated in Table 1. The multiplexing is achieved with the bus transceivers associated with each memory module.

	Time Slice				
Channel	00	01	···	14	15
00	000	001	···	014	015
01	016	017	···	030	031
⋮	⋮	⋮	⋮	⋮	⋮
30	480	481	···	494	495
31	496	497	···	510	511

Table 1: Memory Module Data Time-Multiplexed on the Fast Bus.

The 2DSB modules are likewise divided into 32 groups of 16 modules each. For each group of 2DSB modules, data from the 32 channels of the Fast Bus are multiplexed into the group of 16 2DSB modules. The multiplexers are placed on the backplane of the Fast Bus, and de-multiplexing is implemented with the associated bus transceivers. Hardware necessary for the averaging operation between beams for perspective projections can be incorporated between the bus receivers and the 2DSB modules.

Operation of the multiplexers/transceivers is controlled by lookup tables, called bus channel maps. The maps are pre-computed for each projection and downloaded from the host. A change of viewing parameters requires re-computation of these maps, but their limited size allows for real-time update rates. A requirement of the system is that the data intended to reach the 16 2DSB modules of a channel group are not transmitted during the same time step. Note that this is trivially satisfied, as all voxels of the beam are kept in a sequence and are moving the same amount either left or right.

We investigated the use of alternative structures such as multistage cube/shuffle-exchange networks [20] or expanded delta networks with packet routing [1]. Although these networks typically require less hardware, their routing overhead severely limits their performance. Furthermore, the Fast Bus requires only readily available, off-the-shelf hardware components.

3.4. Fast 3D Interpolation

When sampling in non-grid locations along the ray for compositing [14], the sampled value is tri-linearly interpolated from the values of the eight voxels (called a cube) around the sample point. Note that this kind of sampling does not necessarily require a regular isotropic dataset, and slice data can be accommodated as well. In Cube-3 this interpolation is performed at the leaves of the Ray Projection Cone with data from the 2DSB.

Instead of fetching the eight-neighborhood of each resampling location, four discrete rays are fetched from the 2DSBs, two from each of two consecutive buffer planes. The four rays are subdivided into voxel cubes and fed into the tri-linear interpolation units. Because of the skewing of the 2DSB, neighboring rays reside in adjacent memory modules, requiring a local shift operation of voxels between neighboring units. The pipelined operation of the tri-linear interpolation accounts for this additional latency.

Due to the discrete line-stepping algorithm and the hardware organization into n memory modules, exactly n voxels per discrete ray are fetched, independent of the viewing direction. Since the maximum viewing angle difference along a major axis is not more than 45 degrees up to a 2D rotation, the volume sample rate is defined by the diagonal through the cube and is by a factor of $\sqrt{3}$ higher for orthographic viewing. For ray-compositing, this is of no consideration due to the averaging nature of the compositing operator. High-quality surface rendering, however, requires the adaptation of the stepping distance along the continuous ray according to the view direction.

TRILIN, the 3D interpolation unit, computes the interpolated data values of the samples on the projection ray as it traverses through the volume data. Suppose 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, then the interpolated data value associated with the sample point, P_{abc}, is computed through a tri-linear

interpolation process as follows:

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

A brute-force implementation of this formula requires 13 multiplications and 20 additions for *each* sample point that is not a voxel. We solve this problem by making the observation that a tri-linear interpolation is actually equivalent to a linear interpolation followed by two bi-linear interpolations, and by replacing time-consuming arithmetic operations with a table look-up (see Figure 5).

From the above equation it is clear that the only part that allows pre-computation is the intermediate values involving a, b, and c. With a 16-bit data path and $n = 512$, the number of bits left for fractionals, i.e., relative coordinates within a cube, is seven. With a seven-bit resolution, the number of possible combinations for $\langle a\ b\ c \rangle$ triples becomes $2^7 2^7 2^7 = 2^{21}$. For each triple, there are eight intermediate products, each being 8-bit wide. Thus, the total size of the look-up table of partial products would be 16 MBytes. Such a look-up table is needed for the simultaneous computation of each interpolated data value. Therefore, it cannot be shared and needs to be replicated n times. Simply because of the required memory size, this design is clearly too expensive and potentially slow.

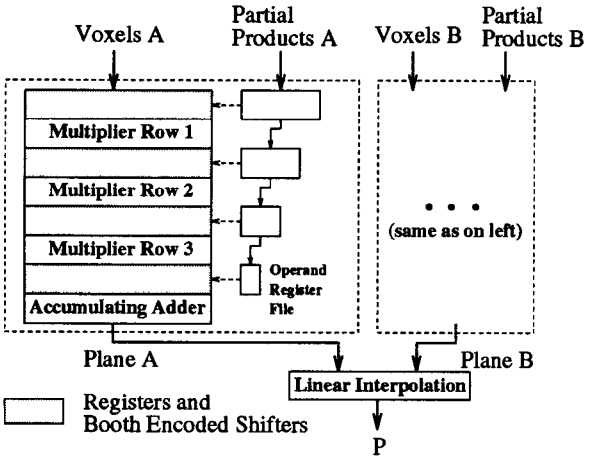


Figure 5: TRILIN: Tri-Linear Interpolation Unit.

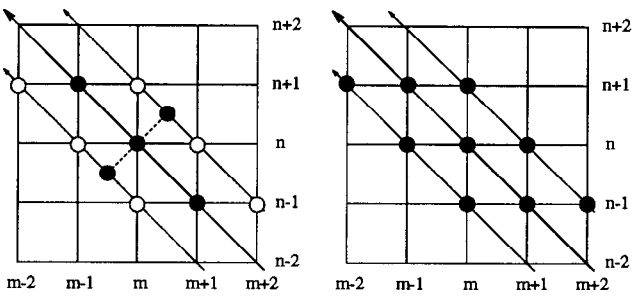
By substituting two bi-linear interpolations followed by a linear interpolation for a tri-linear interpolation, the look-up table size is only 64 KBytes. The price we pay for this design decision is that two more multiplications are needed than in the above equation. Fortunately, the performance overhead associated with these additional multiplications can be minimized by exploiting parallelism and pipelining.

The second key idea in the fast 3D interpolation unit design is to exploit the internal structure of a parallel multiplier. To a first approximation, a parallel multiplier is actually a 2D array of single-bit carry-save adders. Therefore, it is possible to integrate a multiplication and an addition operation by inserting an extra row of carry-save adders. Moreover, to reduce the hardware cost, one can pipeline multiple multiply-add operations through such an augmented parallel multiplier. Consequently, it becomes feasible to implement the entire 3D-interpolation function in one chip.

3.5. Volumetric Shading Mechanisms

A prominent object-based volumetric shading method is gray-level gradient shading [8]. It uses the gradient of the data values as a measure of surface inclination. The gradient is approximated by the differences between the values of the current sample and its immediate neighbors.

In Cube-3 we use the tri-linearly interpolated values of neighboring rays to evaluate the gradient field inside the dataset. In order to evaluate the gradient at a particular point, we need the rays on the immediate left, right, above and below, as well as the values in the current ray. The left and right point sample values can be fetched from neighboring shading units, and the above and below samples arrive from the consecutive processing of PRPs. Since we need to store complete rays, we call the shading units *ABC Shaders* for their above, below, and current ray sample buffers.



(a) Corrected 10-neighborhood Gradient Estimation

(b) 26-neighborhood Gradient Estimation

Figure 6: Gradient Estimation Schemes.

Figure 6 illustrates the different gradient estimation schemes (using a 2D drawing). The simplest approach is the 6-neighborhood gradient, which uses the differences of neighboring sample values along the ray, $P_{(n,m+1)} - P_{(n,m-1)}$ in x and $P_{(n+1,m+1)} - P_{(n-1,m-1)}$ in the ray direction (Figure 6 (a)). 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 are not. More importantly, when a change in the viewing direction causes a change in the major axis from m to n, the values of $P_{(n+1,m)} - P_{(n-1,m)}$ are

used to calculate the gradient in the x direction. This leads to noticeable motion aliasing.

To circumvent this problem we use an additional linear interpolation step to resample the rays on correct orthogonal positions (Figure 6 (a), black samples). We call this approach the *10-neighborhood gradient* estimation, and it adequately solves the problem of switching the major axis during object rotations. The linear interpolation weights are constant along a ray and correspond to a constant shift of the complete ray in the viewing direction.

We also simulated the use of a 26-neighborhood gradient (Figure 6 (b)). Instead of fetching sample values from four neighboring rays, 26 interpolated samples from 8 neighboring rays are fetched and the gradient is estimated by taking weighted sums inside and differences between adjacent planes. This method leads to better overall image quality, but the switching of major axis is still noticeable, although less than with the 6-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 averaged values described above are used instead.

With the gradient estimation and a light vector lookup table, the sample intensity is generated using a variety of shading methods (e.g., using an integrated Phong Shader [12]). Opacity values for compositing are generated using a transfer function represented as a 2D lookup table indexed by sample density.

3.6. Ray Projection Mechanism

The pipelined hardware mechanism for ray projection is the RPC, which can generate one projected pixel value per clock cycle using a rich variety of projection schemes. The cone is a folded (circular) cross-linked binary tree with n leaves, which can be dynamically mapped onto a tree with its leftmost leaf at any arbitrary end-node on the cone (see Figure 7). This allows the processing of a ray of voxels starting from any leaf of the cone. This in turn allows the cone to be hardwired to the outputs of the 2DSB modules containing the voxels. Such a configuration eliminates the need for a set of n n-to-1 switching units or a barrel shifter for de-skewing of the 2DSB. The leaves of the cone contain the TRILIN interpolation and the ABC Shading units.

The cone accepts as input a set of n samples along the viewing ray and produces the final value for the corresponding pixel. The cone is a hierarchical pipeline of $n-1$ primitive computation nodes called Voxel Combination Units (VCU). Each VCU accepts two consecutive intensity/opacity pairs as input and combines them into an output value. At any given snapshot the cone is processing $\log n$ rays simultaneously in a pipelined fashion,

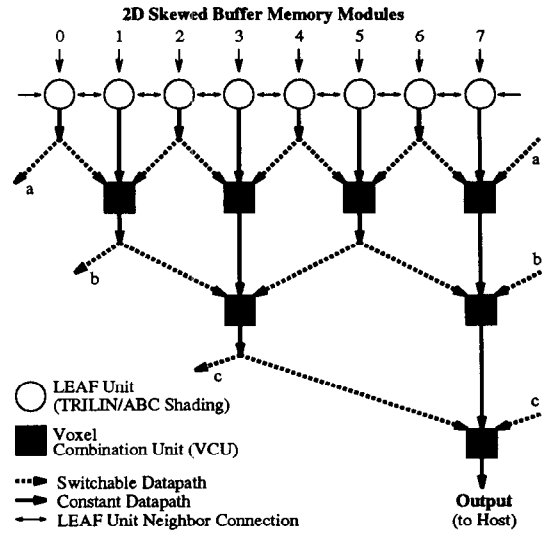


Figure 7: Folded Binary Cone Tree ($n = 8$ Leaf Nodes).

producing a new pixel color every clock cycle. Sectioning and clipping can be implemented on the RPC by discarding voxels according to user specified clip-planes.

Each VCU is capable of combining its two input samples in a variety of ways in order to implement viewing schemes such as first or last opaque projection, maximum or minimum voxel value, weighted summation, and α-compositing. Accordingly, each VCU selects as input the left and center or center and right datapaths, each one consisting of color C and opacity α of the current ray sample.

The opacity of the voxel is either pre-stored with every voxel or provided through a look-up table of a transfer function inside the ABC Shading Unit at the leaves of the cone. The VCU produces an output voxel V' by performing one of the following operations:

First opaque:	if (α_L is opaque)	$V' = V_L$
	else	$V' = V_R$
Maximum value:	if ($C_L < C_R$)	$V' = V_R$
	else	$V' = V_L$
Weighted sum:	$C' = C_L + W_k C_R$	

where W is the weighting factor and k is the cone level. W_k is pre-computed and pre-loaded into the VCUs. Weighted sum is useful for depth cueing, bright field, and x-ray projections.

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

where the first level VCUs compute $C_I = C_I \alpha_I$, assuming the values are gray-levels or RGB. This is actually a parallel implementation of the front-to-back (or back-to-front) compositing. The pixel output is transmitted,

for example, to the host, where post-processing, such as post-shading, splatting, and 2D transformation or warping, is performed. A frame buffer outputs the final image to a color display.

4. Performance Estimation

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. Due to the highly pipelined architecture, the frame rate is limited only by the data-transfer rate on the Fast Bus. If we assume a dataset size of n^3, one resample location per volume element, and a final screen resolution of n^2 pixels, we need to transfer a discrete ray of n voxels on the Fast Bus in $\frac{1}{n^2 f}$ seconds. f is the frame rate in updates per seconds. Since the Fast Bus operates in a time-multiplexed fashion with m time-slices, we need $\frac{1}{n^2 fm}$ seconds per transfer or a clock speed on the bus of $n^2 fm$.

Dataset $n \times n \times n$	Frame Rate f	Bus Frequency
128 x 128 x 128	30 Hz	8 MHz
256 x 256 x 256	30 Hz	33 MHz
512 x 512 x 512	15 Hz	66 MHz
512 x 512 x 512	30 Hz	125 MHz

Table 2: Fast-Bus Performance Requirements ($m = 16$).

Table 2 gives some examples of required bus performance for a multiplexing rate of $m = 16$. High-bandwidth buses have been used by other researchers [16], and technologies and driving chip sets are readily available for most bus speed requirements [19, 21]. We believe that a high-resolution compositing projection of a 512^3 dataset can be generated in Cube-3 with about 30 frames per second.

5. Hardware Estimation

Figure 8 shows the overall hardware structure of Cube-3. It is a modular design that is scalable to higher resolution datasets. The CFB boards contain several CFB modules, each consisting of a memory module, an addressing and bus control unit, and a bus transceiver. Using off-the-shelf components such as SIMMs (Single Inline Memory Modules) and FPGAs (Field Programmable Gate Arrays), it is possible to fit up to 128 CFB modules together with I/O hardware and I/O bus access logic on a single board. The CFB modules on each board can be connected to the acquisition device by high-speed parallel input channels.

Each 2DSB consists of a Fast Bus transceiver, a memory module, and a special purpose LEAF chip. This chip contains the addressing and bus control units, the TRILIN interpolator, and the ABC shading unit. A special purpose VCU chip contains several VCUs of the

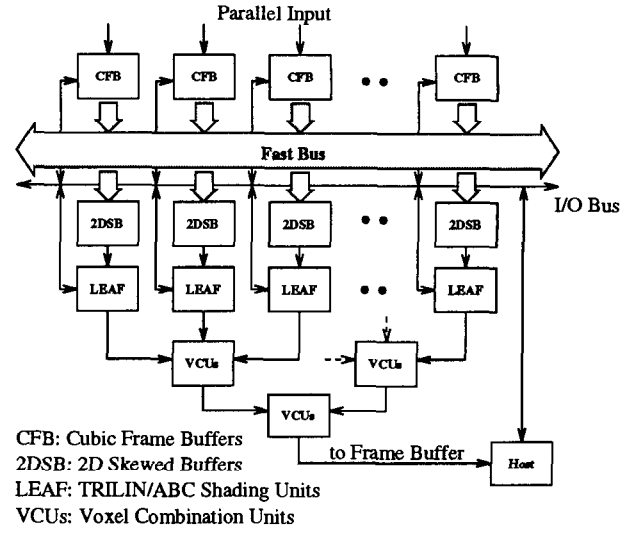


Figure 8: Cube-3 Hardware Architecture.

RPC. Each individual VCU has only modest complexity, so that the number of VCUs per chip is determined by the width of the I/O datapath. Assuming an I/O pin count of 260 pins and 16-bit datapaths, it is feasible to put 8 VCU units per chip. Sixteen VCU chips together with 128 2DSB/LEAF units fit on a single board.

The CFB and 2DSB boards are connected through the high-bandwidth Fast Bus on the backplane, which is the main technological challenge in Cube-3. Assuming a voxel resolution of 16-bit and a 512^3 dataset, the backplane contains a 512-bit wide bus at clock-speeds possibly over 100MHz. Furthermore, the backplane contains a separate I/O bus for LEAF node and host connections.

With the above board estimations, a Cube-3 system for 512^3 16-bit per voxel datasets would require 8 boards and a custom fabricated backplane. This board layout and chip count may change depending on off-the-shelf chip availability, pin count, package size, and bus interface technology.

6. Conclusions

Cube-3 is a scalable, high-resolution volume visualization architecture that exploits parallelism and pipelining to achieve real-time performance. It will provide the following capabilities to the scientist and researcher: viewing from any parallel and perspective direction, control of shading and projection parameters (e.g., first opaque, max value, x-ray, compositing), programmable color segmentation and thresholding, and control over translucency, sectioning, and slicing.

We have simulated the Cube-3 architecture in C and in Verilog, and are designing the general layout of a 512^3 16-bit per voxel prototype implementation. We are currently simulating the effects of the 10-neighborhood gradient estimation for perspective projections. Future

For each triangle, the color and opacity are interpolated linearly from the three vertex values to the interior, usually along the edges and then across scan lines, as in Gouraud shading. Then the interpolated color C_i and opacity α_i are composited over the old pixel color C_{old} to give the new color C_{new}, by the formula: $C_{new} = \alpha_i C_i + (1 - \alpha_i) C_{old}$. Often the linear interpolation and compositing steps can be performed by special purpose hardware available in the rendering engines of a graphics workstation.

The projected tetrahedra algorithm has several artifacts which produce incorrect colors, or Mach bands revealing the subdivision into tetrahedra. The first artifact comes from the linear interpolation of the color and opacity across the tetrahedra. This interpolation is not C^1 across the faces, and can produce Mach bands, particularly at faces which are parallel to the viewing direction and project to lines. The only cure is higher order interpolation, which is not available in hardware on most workstations.

However, there is a more serious problem with the algorithm, which occurs even when the color C and extinction coefficient τ are constant. The problem is easiest to understand when the color is zero, so that the image shows an opacity cloud hiding the background, and in 2-D, where the tetrahedra become triangles. Consider a strip of triangles T_0, T_1, T_2 and T_3 of a constant thickness l as shown in Figure 2(a), projected vertically to a scan line. In triangle T_1, C is the "thick" vertex, where the opacity $\alpha = 1 - \exp(-\tau l)$, and α=0 at B and D. Figure 2(b) is a graph of the transparency $t_1(x) = 1 - \alpha_1(x)$ along the scan line, which is used to multiply the background color during compositing of triangle T_1. It is piecewise linear, because the opacity $\alpha(x)$ has been linearly interpolated across the scan line segments BC and CD. Similarly, Figure 2(c) shows the transparency $t_2(x)$ from triangle T_2. The final transparency along the segment CD, resulting from compositing both triangles on top of the background is the product $t(x)=t_1(x)t_2(x)$, shown as the quadratic polynomial segment above CD in Figure 2(d).

To derive the form of this quadratic polynomial, let $x=s$D+(1-s)C be the point a fraction s of the way from C to D.

Then

$$\begin{aligned} t_1(x) &= 1 - \alpha_1(x) \\ &= s + (1-s) - [s \cdot 0 + (1-s)(1 - \exp(-\tau l))] \\ &= s + (1-s)\exp(-\tau l) \end{aligned}$$

and similarly

$$\begin{aligned} t_2(x) &= 1 - \alpha_2(x) \\ &= s + (1-s) - [s \cdot (1 - \exp(-\tau l) + (1-s) \cdot 0]. \\ &= s \cdot \exp(-\tau l) + (1-s) \end{aligned}$$

Thus

$$t(x) = \exp(-\tau l) + s \cdot (1-s)[1 - \exp(-\tau l)]^2$$

The transparency should actually be $\exp(-\tau l)$, so the quadratic term $s(1-s)[1-\exp(-\tau l)]^2$ represents the error due to approximating $t_1(x)$ and $t_2(x)$ linearly.

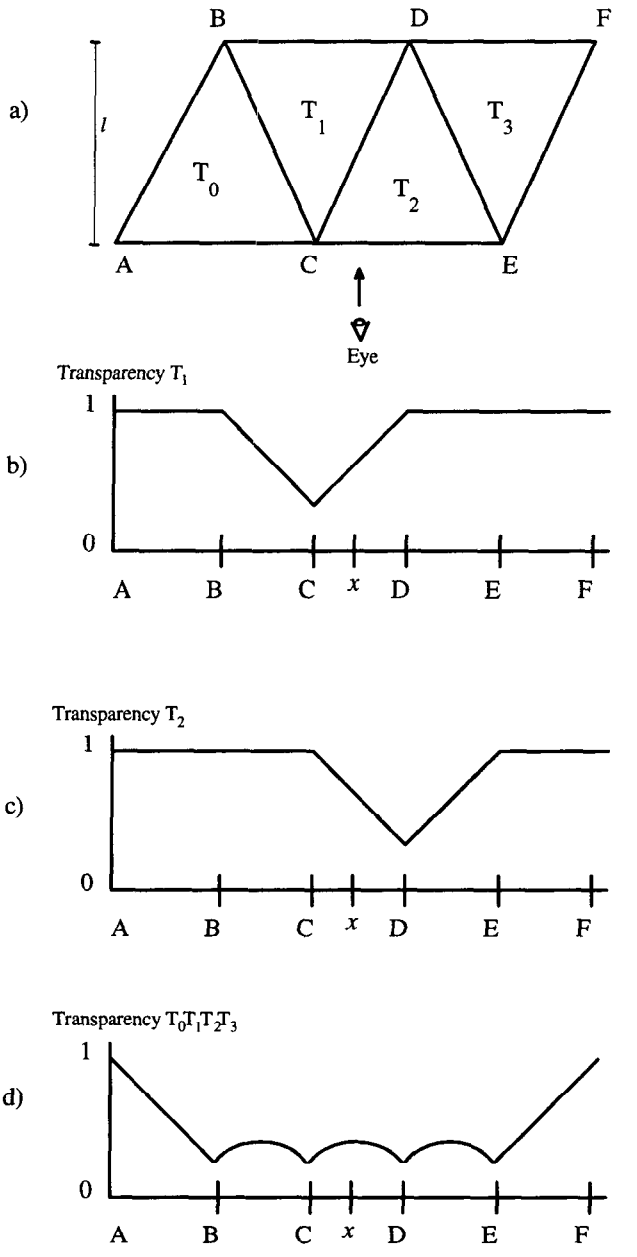


Figure 2

Other similar quadratic segments come from other projected diagonal edges, and the final intensity, proportional to the transparency if the background is uniform, is not C^1. In three dimensions, the corresponding effect produces Mach bands along the projections of edges of the tetrahedra.

The solution to this problem is to define $\alpha_1(x)$ correctly as $1 - \exp(-s\tau l)$. This requires a linear interpolation of the quantity

τl, and then an exponential per pixel, which is not commonly available in hardware. Instead, we have used a texture map table on our SGI Onyx™ system. For the case of constant τ per tetrahedron, as in our flow volume application, we put the quantity $1-\exp(-u)$ in a one dimensional texture table, indexed by u as described in [3]. The texture coordinate u was set to zero at the thin vertices of each triangle, and to τl at the thick vertex, and was interpolated by the shading hardware before being used as an address to the texture table.

If τ varies linearly within each tetrahedron the product τl varies quadratically inside each triangle. Quadratic interpolation of texture coordinates was implemented in hardware on the Apollo DN10000VS [2], but was not available on our Onyx™. Therefore we used a 2-D texture table, with coordinates τ and l, and put $1-\exp(-\tau l)$ in the table. At places, like the edges of a rotated cube where the derivative of the thickness changes suddenly, Mach bands will remain when using our table-based exponential-per-pixel method, but there they are physically appropriate and give useful cues about the object shape.

Now consider the case when the color also varies linearly across the tetrahedron. The Shirley-Tuchman approximation $(C_0+C_1)/2$ for the color of the thick vertex is not precise; it weighs the two colors equally. The frontmost color should have greater weight, because the opacity along the ray segment hides the rear color more than the front one. Williams and Max [10] have found an exact formula for the color in this case, which they implement with the aid of table lookups. However, the supplementary arithmetic required goes far beyond what is practical in hardware computation at each pixel. As a compromise, we have used the exact analytic form of the color of the thick vertex, and then used the hardware to interpolate the color across each triangle. The colors of the thin vertices come from the original color specification, and the opacity is determined, as above, from a texture table. This compromise can be implemented entirely in hardware, and gives a fairly smooth color variation that seems to move appropriately when a colored volume density rotates.

Figures 7(a) and (b), 8(b), and 9(a) and (b) all use texture mapping for the opacity. Figures 7(a) and (b) show a triangular prism divided into three tetrahedra. Figure 7(a) uses the average color $(C_0+C_1)/2$ at the thick vertices, while Figure 7(b) uses the more accurate color integration of Williams and Max [10]. Note that in Figure 7(b) the color of the yellow-orange vertex closest to the viewer is more prominent because this colored region partially hides the differently colored regions behind it. The blue "band" in Figure 7(b) is due to the linear interpolation of the colors from the "thick" vertex to the other vertices Figure 8(a) shows a 2x2x2 array of cubes, each divided into five tetrahedra, and rendered by linearly interpolated opacities. Notice the Mach Bands predicted in Figure 2. Figure 8(b) shows the same volume using the texture mapping for more accurate opacities, and is much improved. Figure 9 illustrates an irregular tetrahedral mesh using the improved color interpolation and hardware texture mapping. Figure 10 shows the turbulence behind a simulated submarine fairwater. This data set is an irregular brick mesh and uses the improved color interpolation with texture mapping.

The Sorting Algorithm

Most volume rendering algorithms use point sampling methods to calculate the color and intensity. Because the Shirley-Tuchman algorithm allows us to scan convert entire polyhedra very quickly, we needed to devise an efficient algorithm that would sort unstructured meshed elements in a back to front order. Our implementation will correctly sort arbitrarily shaped convex elements in a back to front order as long as there are no cycles or intersections in the data set. Each polyhedron can then be subdivided into a set of tetrahedra for rendering. If a convex mesh is structured so that cells meet on common faces, and this topological information is stored in an adjacency graph, then the adjacency graph can be used to produce a back-to-front sort (see [4] or [11]). However, we wanted to handle unstructured meshes where this information is unavailable. Such examples are sliding interfaces, where cells meet on only part of their faces, and non-convex meshes, such as those with cavities. Figure 3 shows such features in a mesh of a piston inside a cylinder. We extended the Newell, Newell and Sancha sort for polygons to correctly handle convex polyhedra. The sort will not perform subdivisions in the case of intersecting polyhedra or cycles, instead it will render them in an arbitrary order discussed later.

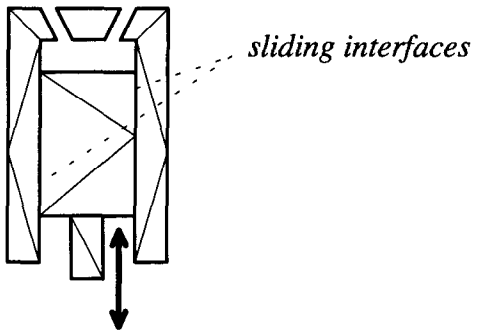


Figure 3.

This algorithm is a three dimensional extension of the Newell, Newell and Sancha painter's algorithm [1,5,6,7] and operates on the volumes after having performed all of the perspective transformation operations. Once the elements have been sorted in back-to-front order, they can be fed to the volume renderer for scan conversion and compositing, using the techniques described in the previous sections.

There are three stages to the sorting process. The first applies all viewing transformations on the vertices to obtain the screen coordinates with a perspective corrected Z. The second obtains a preliminary sorting of the polyhedra based on the rearmost Z component of each element. Since we have applied the viewing transformation to all vertices and have scaled Z so as to correct for perspective, we would like to sort by increasing Z (the eye looks down the Z axis towards negative infinity in a right-handed coordinate system). In our implementation, this preliminary sort was obtained through an $O(n\log n)$ QuickSort. The third stage, or "fine tuning" of the sort, is a bit more complicated. However, like the painter's algorithm approach, it is also broken down into multiple steps with each one increasing in computational complexity, in hopes that a majority of the polyhedra will pass the earlier and less expensive tests.

The goal of the fine tuning is to find a separating plane between two polyhedra, P and Q, in order to determine whether or not P can be safely drawn before Q. The fine tuning process is broken down into five steps in order to efficiently find this separating plane. Given a list H of polyhedra exactly sorted in back to front order by increasing Z coordinate of the farthest vertex (called $Z_{rearmost}$), let polyhedron P be at the head of the list. P can be safely rendered if, for all polyhedra Q in the list H whose $Z_{rearmost}$ is less than (behind) P's $Z_{frontmost}$, the following function returns a value of True:

```
Test_Polyhedra(P,Q)
{
(1) if (P and Q do not have
        overlapping X extents) return True
(2)    else if (P and Q do not have
           overlapping Y extents)
           return True
(3)       else if (P is behind a
              back-plane of Q) return True
(4)          else if (Q is in front of a
                 front-plane of P) return True
                 else if
(5)                 (Q!=EdgeIntersection(P,Q))
                  return True
                    else return False
}
```

The function `EdgeIntersection(P,Q)` returns the polyhedron which it determines to be in back. It makes this decision by looking for intersections between the edges of P's screen projection and the edges of Q's screen projection. For each projected edge in one polyhedra we test all of the projected edges in the other polyhedra for intersections. If one is found, it finds the Z component of that intersection point for P's edge and for Q's edge, and returns the polyhedron whose $Z_{intersection}$ is farther from the eye. `EdgeIntersection()` will not test the remaining edges. In the case that they are both equal, then we continue searching for intersections looking for an inequality between the two $Z_{intersection}$ components.

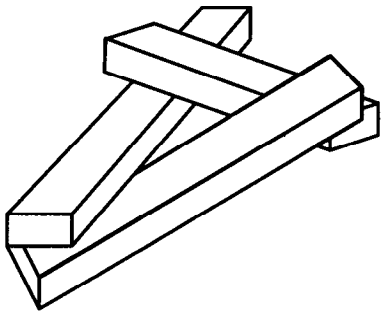


Figure 4.

If `Test_Polyhedra()` returns False, then polyhedra P and Q are considered to be in the wrong order and Q should be moved to the head of the list and the tests should be repeated with Q becoming the new P. It is possible that the list H could contain a cycle. For instance, if polyhedron A obscures B, and B obscures C, and C, in turn, obscures A, then there is no correct ordering for the polyhedra involved. Figure 4 illustrates a cycle for three polyhedra. The existence of a cycle is easily determined by tagging polyhedron Q before inserting it at the head of the list after the `Test_Polyhedra()` function fails. If Q has already been tagged, then a cycle exists and it will need to be addressed.

If polyhedron P passes the tests for all polyhedra Q where $Q_{rearmost}$ is less than $P_{frontmost}$, then polyhedron P is free to be rendered; the tests have determined that P will not obscure any polyhedra which are considered to be in front of it. P is then shipped to the renderer and the next polyhedron in the list is chosen for the new P.

The first two tests check the bounding boxes of the two polyhedra in the X and Y plane. The main thrust of the third and fourth tests is to find a separating plane between P and Q. If such a plane exists, then P can safely be considered to lie behind Q. To simplify the third and fourth tests, we can mark each face of every polyhedron as being either a front-facing polygon (it faces the eye) or a back-facing polygon. This is easily determined because the algorithm stores an outward pointing normal for each face. Therefore, a simple query as to the sign of the Z component of a face's normal is enough to determine whether the face is front facing or not. A positive Z, in a right-handed coordinate system, is front facing. Otherwise it is back-facing. This pre-processing is all performed while reading in the meshed topology.

The third test then simplifies to testing whether all of P's vertices lie behind a plane defined by any one of Q's back-facing polygons. If this is true, then the face under consideration forms a separating plane between P and Q and therefore we can conclude that P is behind Q. Performing this test is a matter of making sure that for at least one back-facing polygon of Q, the sign of $f(x_j, y_j, z_j)$ for all vertices j in P is non-negative for that particular face of Q. The plane equation, f, is based on the outward pointing normals for that face. If this test fails, then the algorithm will proceed to the fourth test and try to determine whether the plane specified by a front-facing polygon belonging to P separates P from Q.

This fourth test is very similar to the third test. In determining whether Q lies entirely in front of P, one must make sure that for at least one front-facing polygon of P, $f(x_j, y_j, z_j)$ is positive for all vertices j in polyhedron Q. This time, f is the plane equation for a front-facing polygon of P, again based on outward pointing normals. If this test passes, then Q lies entirely in front of at least one of the front-facing polygons of P and it can be concluded that P lies behind Q.

The fifth test, `EdgeIntersection()`, returns either the index of the polyhedra which is in back, or an error condition if it cannot detect any intersecting edges. The two cases where this test can fail are shown in Figure 5. As we will see, this does not jeopardize the correctness of our algorithm.

The illustrations (a) and (b) in Figure 5, which both represent screen projections, both fail the `EdgeIntersection()` function because neither have intersecting edges in their projections. However, in case (a) the order in which the two tetrahedra are rendered makes no difference since they are

completely disjoint in the screen projection and therefore an error condition can correctly be treated as if polyhedron P were in front of polyhedron Q. On the other hand, this is not necessarily the situation in case (b). We can rest assured that this [case (b)] will never cause a sorting glitch because the front face of the brick (assuming the tetrahedron is in front of the brick) is a front-facing separating plane and would have been caught in the fourth test. This fifth test is a more efficient alternative to the linear programming method proposed by Newell [6]. If the fifth test fails, then polygon Q should be moved to the front of the list and the whole process should be repeated.

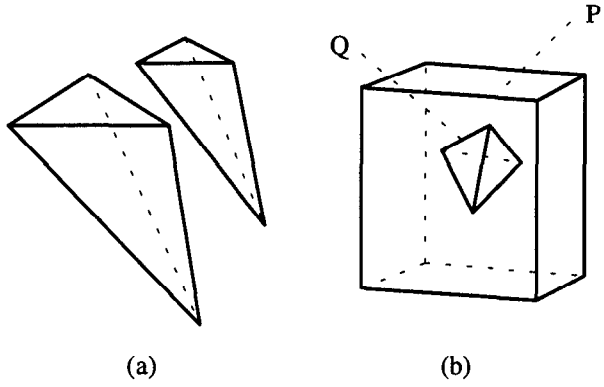


Figure 5.

With the exception of the fifth, these tests are very easy to perform. When reading in the topological data-set, one must store the plane equation coefficients, with respect to an outward pointing normal, in the polyhedral database. From these pre-computed coefficients, determining which side of a face a point j lies is as simple as finding the sign of $ax_j + by_j + cz_j + d$. We have not addressed the issue of degenerate faces.

In the case that all of the tests fail and we have a cycle, the program will render first whichever of P and Q has a $Z_{frontmost}$ further from the eye.

Non-planar Faces

The algorithm described works correctly for convex polyhedra with planar faces and no cycles or intersections. Unfortunately, it is quite possible, in finite element codes, for the faces to skew slightly yielding non-planar faces. Fortunately, the faces will be mostly planar because highly non-planar faces can lead to instabilities in the modeling code. Figure 6 illustrates an exaggeration of what could possibly happen. Even if the face were mildly non-planar, it is still enough to cause the tests to fail. To accommodate slightly non-planar faces, we have introduced an error tolerance δ.

In order to sort convex polyhedra with non-planar faces as shown in Figure 4, the algorithm first calculates an average outward pointing normal, (a,b,c), for each face. This is done using Newell's method as follows [7,9]:

$$a = \sum_{i=1}^{n} (y_i - y_j)(z_i + z_j)$$

$$b = \sum_{i=1}^{n} (z_i - z_j)(x_i + x_j)$$

$$c = \sum_{i=1}^{n} (x_i - x_j)(y_i + y_j)$$

where: $j=(i+1) \bmod n$
and n is the number of vertices per polygon

The last coefficient of the plane equation, d, can be calculated by picking some point on the average plane. We chose the center of gravity of the face for this point as follows:

$$\frac{1}{n}\left(\sum_{i=1}^{n} x_i, \sum_{i=1}^{n} y_i, \sum_{i=1}^{n} z_i\right)$$

To determine on which side of a plane a point lies, an error tolerance is used. This is needed because with non-planar faces the algorithm could return vertex a of polyhedron Q as being contained inside of P, which would ultimately result in a cycle (see Figure 6). This is not the case. In fact, if vertex a were actually touching a plane of polyhedron P, machine round-off might place a on the wrong side of that face which, again, would result in a cycle. Therefore a tolerance, δ, is used to represent an

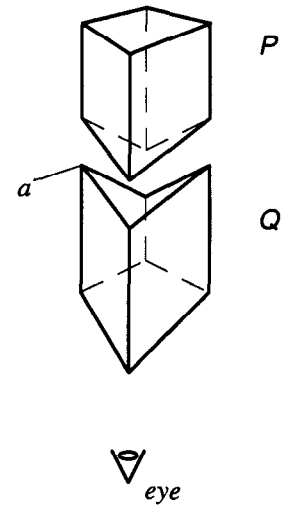


Figure 6.

acceptable distance from a vertex to a face. In other words, the third and fourth tests should consider vertex a to be on the outside of a face (the plane equation evaluated at point a should yield a non-negative value) if point a is within δ units away from the plane under consideration, regardless of which side of the face point a actually lies. We can rationalize the existence of this δ tolerance as follows: if a corner of polyhedron Q happens to intersect a planar face of polyhedron P by the amount δ, for a suitably small δ, the visual impact will be minimal, if perceptible

at all. Our implementation uses a unique δ for each face, based on the maximum deviation of a vertex from its corresponding average plane.

Discussion

The fine-tuning sort described runs in $O(n^2)$with respect to the number of polyhedra sorted. However, this quadratic running time is an upper bound and would only be found in the most pathological cases where all polyhedra have overlapping Z extents. The average running times for normal data sets should be lower. While the first and second tests run in constant time assuming the bounding boxes are known ahead of time, the third and fourth tests run in $O(F_iE_j)$ and $O(E_iF_j)$ time where E_i and E_j correspond to the number of edges for polyhedra i and j, respectively, and F_i and F_j are the number of faces. The fifth test runs in $O(E_iE_j)$ Again, this is a worst case running time and it should be substantially better in practice since the function terminates once a suitable intersection in the two projections is found.

The algorithm was implemented in C++ and has been used to sort those primitives found in the SGI Explorer pyramid type. The volume primitives are all subclasses of a general *primitive* C++ class. These subclasses are as follows: the tetrahedra, pyramid, prism, wedge and brick. We can easily extend the system to include others.

Table 1 shows some timing data using the complete sort on an SGI Indigo$^{2\text{TM}}$. As contrast, the QuickSort can sort 24,000 elements in 4 seconds, and 157,000 elements in 27 seconds.

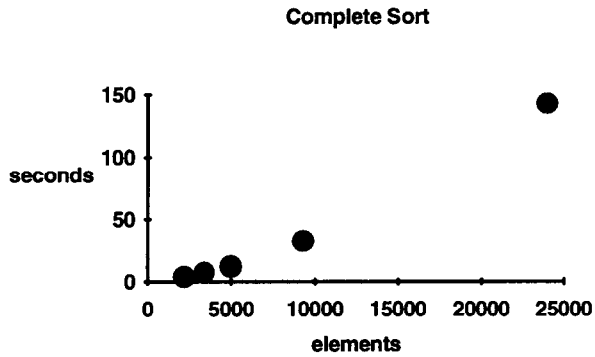


Table 1.

We present no new approaches to cycle breaking. If a cycle is detected during the sorting, then the polyhedron with the vertex farthest from the eye would be removed from the list and rendered. The most common form of a cycle the algorithm would detect in a data set would probably be two non-planar faced polyhedra "intersecting" each other. However, the δ overlapping tolerance should eliminate most of these situations. The traditional, but slower method for removing cycles, such as the type illustrated in Figure 4, would be to pass one or more cutting planes through the offending polyhedra. [11] describes methods for breaking cycles by re-triangulating with a Delaunay triangulation.

Conclusion

This paper presents extensions to the Shirley-Tuchman algorithm for compositing semi-transparent and colored elements with hardware assisted texture mapping. We have also presented extensions to the Newell, Newell, and Sancha sort for use with unstructured data. For quick interaction or still frames, QuickSorting alone is adequate. For a final animation, the full sort is necessary because popping will become apparent if the rendering order suddenly becomes incorrect.

Acknowledgments

We would like to thank Roger Crawfis and Peter Williams for all of their help and suggestions. We would also like to thank the people at Silicon Graphics for their advice and help in creating our Explorer modules. The data set in Figure 10 was provided by Mark Christon from the Lawrence Livermore National Laboratory, whom we would also like to thank.

This work was performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

References

1. Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics Principles and Practice, 2nd Edition,* Addison-Wesley, Reading, Massachusetts, 1990.

2. Kirk, David and Douglas Voorhies, "The Rendering Architecture of the DN10000VS", Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6-10, 1990). In *Computer Graphics* 24, 4 (August 1990), 299-307.

3. Max, Nelson, Barry Becker, and Roger Crawfis, "Flow Volumes for Interactive Vector Field Visualization", *Proceedings of Visualization '93,* (October 1993), pp. 19-25.

4. Max, N., P. Hanrahan, and R. Crawfis, "Area and Volume Coherence for Efficient Visualization of 3-D Scalar Functions", *Computer Graphics*, Vol. 24, No. 5 (November 1990), pp. 27-33.

5. Newell, M. E., R. G. Newell, and T. L. Sancha, "A Solution to the Hidden Surface Problem." *Proceedings of the ACM National Conference 1972*, pp. 443–450.

6. Newell, M. E."The Utilization of Procedure Models in Digital Image Synthesis", Ph.D. Thesis, University of Utah, 1974 (UTEC-CSc-76-218 and NTIS AD/A 039 008/LL).

7. Rogers, David F., *Procedural Elements for Computer Graphics,* McGraw-Hill, New York. 1985.

8. Shirley, Peter and Allan Tuchman, "A Polygonal Approach to Direct Scalar Volume Rendering", *Computer Graphics,* Vol. 24, No. 5 (November 1990), pp 63-70.

9. Sutherland, I. E., R. F. Sproull, and R. A. Schumaker, "A Characterization of Ten Hidden-Surface Algorithms", *Computing Surveys*, Vol. 6, 1974, pp. 1-55.

10. Williams, Peter and Nelson Max, "A Volume Density Optical Model", Proceedings of the 1992 Workshop on Volume Visualization (Boston, MA, October 19-20, 1992), special issue of *Computer Graphics*, ACM SIGGRAPH, New York, 1992.

11. Williams, Peter "Visibility Ordering of Meshed Polyhedra", *ACM Transactions on Graphics*, Vol. 11, No. 2, (April 1992), pp.103-126.