

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU



HARVARD LIBRARY
Office for Scholarly Communication

Cube-4 - A Scalable Architecture for Real-Time Volume Rendering

Citation

Pfister, Hanspeter, and Arie Kaufman. 1996. Cube-4 - A scalable architecture for real-time volume rendering. In Proceedings of the 1996 symposium on Volume visualization: October 28-29, 1996, San Francisco, C.A., ed. R. Crawfis and C. Hansen, 47-54. New York, N.Y.: Association for Computing Machinery.

Published Version

http://portal.acm.org/citation.cfm?id=236232

Permanent link

http://nrs.harvard.edu/urn-3:HUL.InstRepos:4238981

Terms of Use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

To appear in the 1996 Symposium on Volume Visualization proceedings

Cube-4 – A Scalable Architecture for Real-Time Volume Rendering

Hanspeter Pfister and Arie Kaufman

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

Abstract

We present Cube-4, a special-purpose volume rendering architecture that is capable of rendering high-resolution (e.g., 1024^3) datasets at 30 frames per second. The underlying algorithm, called slice-parallel ray-casting, uses tri-linear interpolation of samples between data slices for parallel and perspective projections. The architecture uses a distributed interleaved memory, several parallel processing pipelines, and an innovative parallel dataflow scheme that requires no global communication, except at the pixel level. This leads to local, fixed bandwidth interconnections and has the benefits of high memory bandwidth, real-time data input, modularity, and scalability. We have simulated the architecture and have implemented a working prototype of the complete hardware on a configurable custom hardware machine. Our results indicate true real-time performance for high-resolution datasets and linear scalability of performance with the number of processing pipelines.

1 Introduction

Volume rendering is a key technology for the interpretation of the large amounts of 3D scalar data generated by acquisition devices such as biomedical scanners, by supercomputer simulations, or by voxelizing geometric models. Especially important for the exploration and understanding of the data are sub-second display rates and instantaneous visual feedback during the change of rendering parameters. To create the illusion of smooth dynamics, the image must be updated in true real-time. In this paper we describe Cube-4, a scalable architecture for volume rendering that achieves 30 projections per second for up to 1024^3 16-bit voxel datasets.

The high computational requirements of traditional computer graphics led to the development of special-purpose graphics engines, primarily for polygon rendering. Similarly, the special needs of volume rendering, where an image must be computed rapidly and repeatedly from a volume dataset, lends itself to the development of special-purpose volume rendering architectures. A dedicated accelerator, which separates volume rendering from general-purpose computing, seems to be best suited to provide true real-time volume rendering on standard deskside or desktop computers. Volume rendering hardware may also be used to directly view changes of the 3D data over time for 4D (spatial-temporal) visualization, such as in real-time 3D ultrasonography, micro-tomography, or confocal microscopy. This may lead to the direct integration of volume visualization hardware with real-time acquisition devices, in much the same way as fast signal processing hardware became part of today's scanning devices.

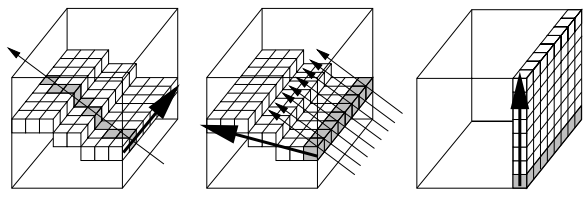
Consequently, research has been conducted towards the development of dedicated real-time volume rendering architectures (see [6] Chapter 6). Among the more recent approaches is VIRIM [5]. However, even a large 16 board VIRIM system achieves only 10 frames per second for low-resolution $256 \times 256 \times 128$ datasets. A more modular approach is taken by VOGUE [9]. A 256^3 dataset can be rendered at high quality with 0.6 frames/sec using a single board and at 4 frames/sec using 8 boards and a 640 MB/sec global bus. Our earlier Cube-3 architecture has been estimated to render a medium-resolution 512^3 dataset at 30 frames/sec [12]. However, such an implementation would require 8 boards interconnected by a 3 GB/sec global bus. At this time, no volume rendering architecture is capable of achieving real-time frame rates at an acceptable hardware cost, and none is modular and scalable in performance.

The Cube-4 architecture, presented in this paper, performs arbitrary parallel and perspective projections of high-resolution datasets at true real-time frame rates. The performance is data and classification independent and can be achieved at a fraction of the cost of a multiprocessor computer. Cube-4 uses accurate 3D interpolation and high-quality surface normal estimation without any precomputation or data duplication. Consequently, Cube-4 is also appropriate for 4D visualization as an embedded volume visualization hardware system in emerging real-time acquisition devices. The Cube-4 architecture performance grows proportionally with increasing number of memory and processing units, ultimately limited by memory speeds.

In the following sections, we first present the underlying algorithm of the Cube-4 system. In Section 5, we present the Cube-4 dataflow, a main contribution of this research. It leads to localized, near-neighbor datapaths for the Cube-4 architecture, described in Section 6. In Section 7, we show results from simulations and a prototype implementation of Cube-4 on the Teramac, a configurable custom hardware machine developed by HP Labs. Finally, in Section 8, we analyze the theoretical achievable performance.

2 Parallel Ray-Casting

Our research focuses on ray-casting of regular datasets. Ray-casting offers room for algorithmic improvements while still allowing for high image quality. We modified the original ray-casting algorithm to make it better suited for a parallel hardware implementation. Figure 1 shows three possible approaches to parallelizing ray-casting. According to the form of parallelism that is exploited, we call these algorithms *ray-*, *beam-*, or *slice-parallel*.



a) Ray-Parallel **b) Beam-Parallel** **c) Slice-Parallel**

Figure 1: *Three different approaches to parallelizing ray-casting. Shaded voxels are processed simultaneously. The thick arrows indicate the direction in which the algorithm proceeds.*

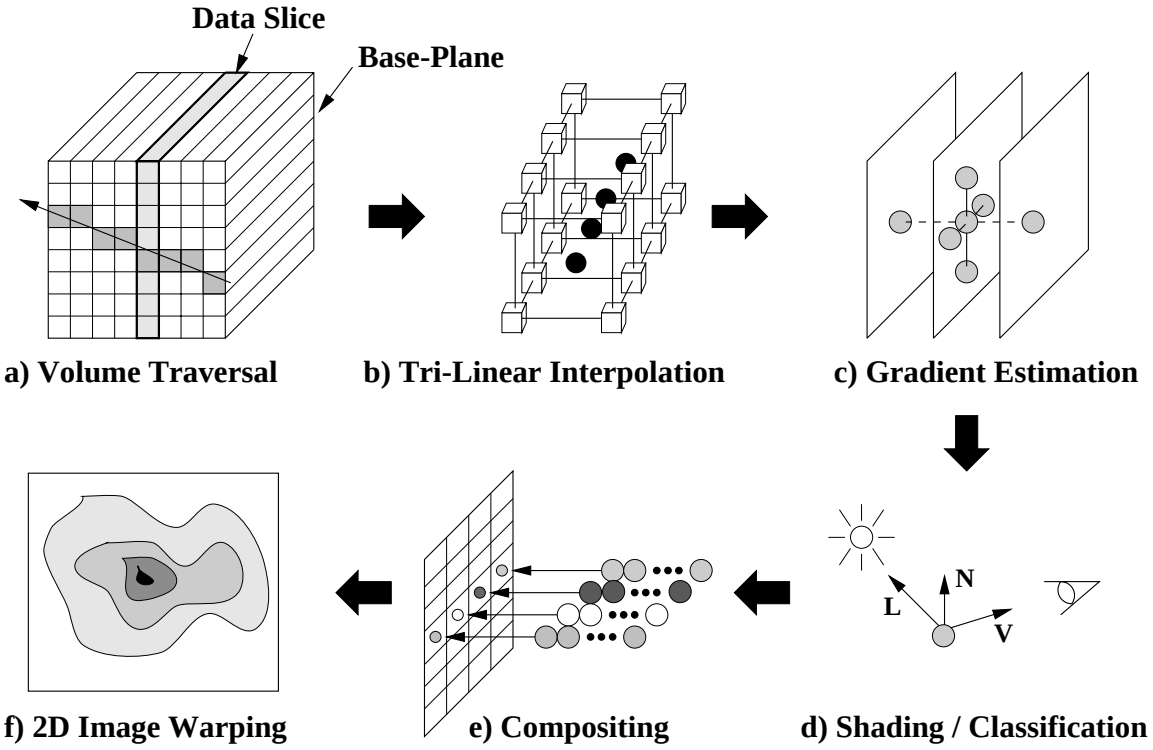


Figure 2: *Pipeline stages of the slice-parallel ray-casting algorithm.*

In the ray-parallel approach, all voxels along a ray are processed simultaneously (the shaded voxels in Figure 1a). The algorithm proceeds ray by ray in scanline order (the thick arrow in Figure 1a). Our earlier Cube-3 architecture [12] is a highly-parallel implementation of this approach. However, the simultaneous access to all voxels along a ray requires global communication between the volume memory and the processing units. This ultimately limits the performance and the scalability of the architecture because of the very high bandwidth requirements.

An alternative to operating on all samples of a single ray is to simultaneously operate on samples of several neighboring rays. Depending on how the algorithm proceeds, we call these approaches beam-parallel (see Figure 1b) or slice-parallel (see Figure 1c). A *beam* is a scanline of voxels that is parallel to a principal axis of the dataset. The beam-parallel ray-casting approach follows a group of rays by fetching consecutive beams in the major viewing direction. We presented a preliminary proposal towards a beam-parallel ray-casting architecture in [13]. However, the stepping along slanted planes of rays requires complicated addressing mechanisms and leads to non-uniform processor communication.

The slice-parallel approach processes consecutive data slices that are parallel to a face of the volume dataset. This processing order appears similar to multipass resampling [4] or object order compositing algorithms [16]. However, in addition to the object-order data traversal we incorporate advantages of ray-casting into the algorithm. Slice-parallel algorithms have been used in various forms by other researchers. Reynolds et al. [14] and Lacroute and Levoy [10] use a shear-warp factorization of the viewing transformation to project the volume in a slice-parallel fashion onto the base-plane. Cameron and Underill [3] and Schröder and Stoll [15] have used slice-parallel approaches on massively-parallel SIMD machines.

Our hardware implementation of the slice-parallel ray-casting algorithm improves on these previous results in several ways. Shear-warp algorithms use linear 2D resampling filters [10], while the Cube-4 architecture implements accurate 3D resampling using tri-linear interpolation between data slices. Furthermore, Cube-4 does not use any pre-computations and stores only one copy of the dataset, allowing for real-time data input. The focus and the primary contribution of this paper is the Cube-4 architecture, an efficient and scalable implementation of pipelined slice-parallel ray-casting in hardware.

3 Slice-Parallel Ray-Casting

In this section we present a fully pipelined version of slice-parallel ray-casting that accesses each voxel of the dataset exactly once per projection. Figure 2 gives an overview of how the data flows through a sequence of stages in a pipelined fashion.

The volumetric dataset is stored as a 3D regular grid of voxels (Figure 2a). The face of the volume memory that is most perpendicular to the major component of the viewing direction is called the *base-plane*. Consecutive data slices parallel to the base-plane are traversed in scanline order. Beams of two adjacent data slices of voxels are processed simultaneously to compute a new slice of interpolated sample values inbetween these two slices. In the following section we present a distributed memory system that allows conflict-free access to beams from all three principal axes.

The orthogonal voxel neighborhoods between data slices allow for accurate 3D resampling using tri-linear interpolation (Figure 2b). In order to reduce the computation of resampling weights, we use a lookup-table based ray-casting technique that was first introduced by Yagel and Kaufman [17] and that we used in the Cube-3 architecture [12]. Correct 3D resampling along rays may lead to multiple samples inbetween data slices. Consequently, we may compute more than one interpolated data slice inbetween voxel slices.

To approximate the surface normals necessary for shading and classification (Figure 2c) and to avoid any further access to the volume memory after tri-linear interpolation we use the interpolated

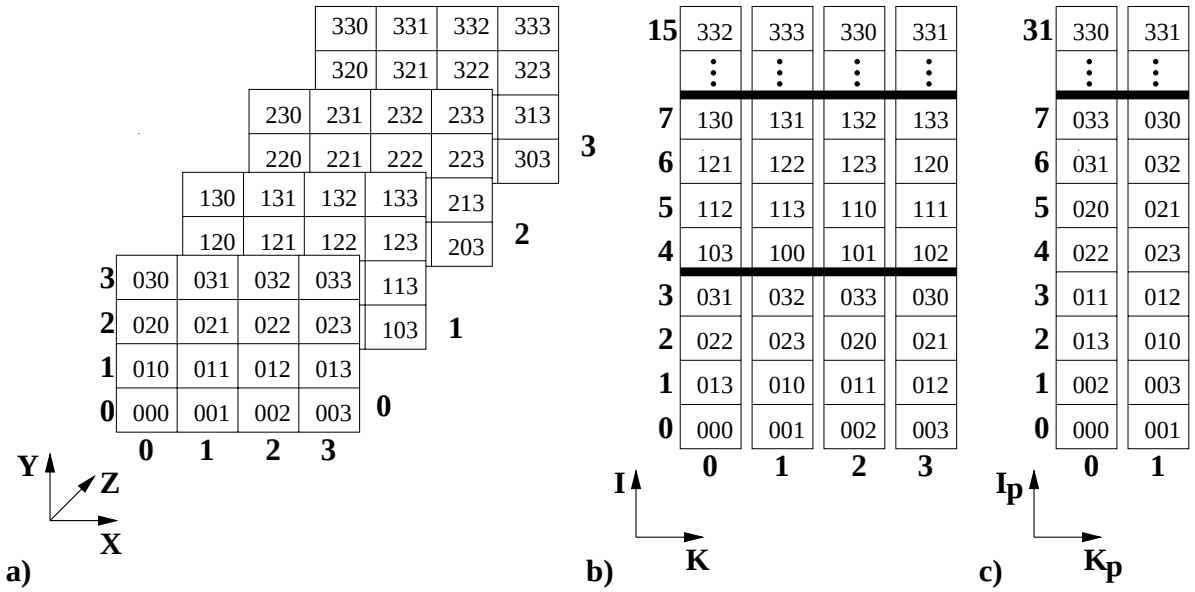


Figure 4: *3D skewed memory organization for $n = 4$. a) Assignment of voxel addresses $[zyx]$ in volume space. b) Dataset stored in $n = m = 4$ memory modules. c) Dataset stored in $m = 2$ memory modules. Thick lines indicate slice boundaries inside the memory.*

sample values to estimate the gradients on each sample position (cf. [13]). Figure 3 illustrates the technique for parallel projections for major viewing direction *Z*. The interpolated data slices from

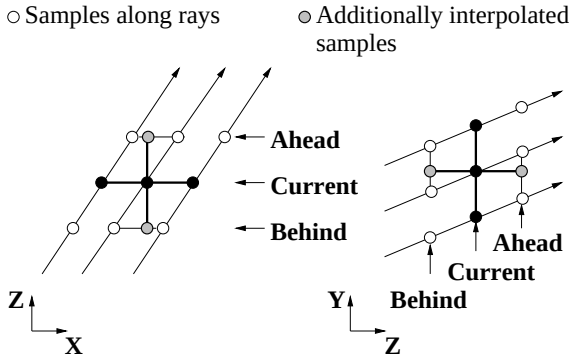


Figure 3: *Gradient estimation using interpolated samples from the ahead, behind, and current (ABC) slices. The example shows parallel projection with major viewing direction* Z*.*

the tri-linear interpolation stage are stored in the so-called *ABC buffers*. The current buffer stores the samples that are currently being shaded. The ahead and behind buffers store the samples one slice ahead and one slice behind in major viewing direction, respectively.

As Figure 3 shows, the gradients in non-major direction (*X* and *Y*) can be computed by taking central differences of neighboring samples (shown in black) inside the current buffer. In the major direction *Z*, because of the possibly slanted rays, we need to interpolate two additional samples (shown in grey). These samples can be computed using two additional bi-linear interpolations between samples of the ahead and behind buffers. This method is called the 12-neighborhood ABC gradient estimation because a total of 12 samples participate in the computation.

Using the gradient as a surface-normal approximation, each sample is shaded and classified by an opacity transfer function (Figure 2d). Compositing of samples along rays onto the base-plane (Figure 2e) is performed using any of the well-known methods in the literature [11]. The distorted intermediate base-plane image is then 2D transformed (warped) onto the viewing plane to produce the final image (Figure 2f).

Perspective projection is nearly identical to parallel projection, except that the interpolation stage also needs to compute averages of larger neighborhoods for slices further away from the base-plane (cf. [10]). The first slice of data is uniformly sampled and scaled by a factor of one, which corresponds to shooting one ray per pixel of the base-plane. In all subsequent slices, the slices are scaled according to the viewing transformation, and a larger portion of the slice is sampled. This averaging of larger neighborhoods can be implemented in hardware using additional interpolation stages that perform a simple box-filtering of slices. The maximum extent of this box filter, needed for the slice furthest away from the base-plane, is $1 + 2 \tan \alpha$, where α is half of the field-of-view angle. For $\alpha < 45^0$, or any field-of-view less than 90^0, this corresponds to a maximum extent of 3 voxels. Therefore, averaging of samples for perspective projections can be implemented using three additional interpolation stages. After samples from averaged slices have been computed, the subsequent algorithm remains the same as for parallel projections.

4 Memory Organization

In this section, we present a memory interleaving technique based on a linear skewing of the address space that allows for conflict-free access to beams of voxels from all three principal viewing axes. Kaufman and Bakalash [7] have used a simplified version of this memory organization. The volume dataset is stored only once without data duplication.

Figure 4a shows a $4 \times 4 \times 4$ dataset in its local coordinate system. Each voxel in the figure is represented by its address which is $a = [zyx]$, the tuple with the local coordinates of the voxel. We refer to this standard arrangement of voxels as *volume space*. A regular volumetric dataset with $n \times n \times n$ voxels in volume space is stored in m physical memory modules, each containing w words of either 8 or 16 bits, using a *skewing function* $\sigma : [z, y, x] \rightarrow [k, i]$, which

maps a voxel with local coordinates $[z, y, x]$ into memory module number k at index i as follows:

$$\begin{array}{ll} k = (x + y + z) \bmod n & 0 \le k, x, y, z < n, \\ i = y + zn & 0 \le i < n^2. \end{array} \quad (1)$$

Adjacent voxels of beams in *X* direction are placed in the same relative locations of adjacent memory modules (i.e., rows across the memory). This choice of storage is arbitrary. If the number of memory modules m is smaller than n, we apply a re-mapping of the skewed memory space by a *partitioning function* $\phi : [k, i] \rightarrow [k_p, i_p]$, where:

$$\begin{array}{ll} k_p = k \bmod m & 0 \le k_p < m, \\ i_p = i\frac{n}{m} + \lfloor \frac{k}{m} \rfloor & 0 \le i_p < \frac{n^3}{m}. \end{array} \quad (2)$$

Figure 4b shows the resulting assignment of voxel addresses $[zyx]$ to memory modules, for $n = m = 4$. Notice that we can access beams in *X*, *Y*, or *Z* direction conflict-free from the four memory modules. Figure 4c shows the partitioned memory space for $n = 4$ and $m = 2$. It is important to notice that this skewing and partitioning of the memory space works for any n and m as long as n is a multiple of m. In general, the computation of $(x+y+z) \bmod n$ or $k \bmod m$ involves a division operation. If n and m are powers of two, it degenerates to a masking operation with the low order bits of the operand.

5 Slice-Parallel Dataflow

The *skewing distance* s is the distance by which two beams have been shifted ($\bmod\ m$) relative to each other. For example, Figure 4b shows that each beam of a slice (in volume space) has been shifted by $s = 1$ (in memory space) with respect to the beam below it. This means that, in general, beams can not be accessed from memory in the same order they have in volume space.

One solution to the problem is to permute fetched beams by an intermediate interconnection network between the memory and the processing units. This permutation of beams is called *unskewing*, because it reduces the skewing distance between consecutively fetched beams to zero. This approach has been used in the Cube-3 architecture [12]. However, the hardware complexity of such a global interconnection is high and becomes prohibitive for large m, limiting the scalability. In Cube-4 we take a very different approach that does not require any global communication except at the pixel level.

We now explain the datapaths and the resulting dataflow in detail using signal flow graphs (SFGs). A SFG is a directed graph with non-negative edge and node weights. A node stands for an arithmetic or logic function performed with zero delay and an edge stands for data transport. The order of operations is represented as directed edges emanating from the node that is to be executed first. The weight of the edge indicates by how many clock cycles the first operation must precede the second operation. We do not show edge weights of 0. An edge may also be viewed as a datapath from one operation to another and its weight as indicating the number of registers included in that datapath. The width of all datapaths is assumed to be constant. To simplify the discussion, we first restrict our attention to the case of $m = n$. Later, we discuss the generalization of these results to the case of $m < n$.

Tri-Linear Interpolation

Tri-linear interpolation requires 8 voxels arranged in a $2 \times 2 \times 2$ orthogonal voxel neighborhood. This is equivalent to two 2×2 voxel neighborhoods from consecutive data slices. Figure 5 shows one slice of a $4 \times 4 \times 4$ dataset in volume space and in skewed memory

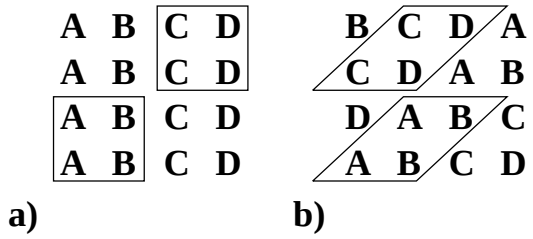


Figure 5: *Bi-linear neighborhoods.*

space. For simplicity we have indicated increasing voxel addresses along rows with consecutive letters. The neighborhoods required for the bi-linear interpolation inside the slice are surrounded by a box. Notice how the orthogonal neighborhoods are shifted and sheared in memory space due to the skewing difference between beams.

Assume that we fetch consecutive beams in positive *Y* direction from the dataset. This corresponds to fetching consecutive rows in column direction in Figure 5b. The SFG in Figure 6 shows how the data is moved between pipeline stages. Dashed edges that leave

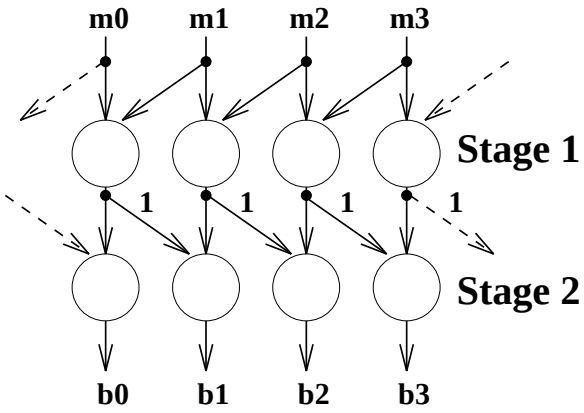


Figure 6: *SFG for bi-linear interpolation.*

on one side of the figure are connected to corresponding edges on the other side in a wrap around fashion. Each node in the graph performs a linear interpolation of its two inputs. The first stage performs a linear interpolation between neighboring voxels of one beam using w_x, the interpolation weight in X direction. The second stage performs a linear interpolation between the linearly interpolated samples of two consecutive beams using w_y, the interpolation weight in Y direction.

Looking at the SFG we notice some important patterns. The datapath between memory and stage 1 is used to join two (spatially) adjacent voxels from a beam at a time. This is easily achieved by a *merger* of adjacent voxels at the processing nodes. The datapath between stage 1 and stage 2 is used to join data of two (temporally) subsequent beams. Because the two beams are output in consecutive clock periods, this can be achieved by a *shift and delay*. Although the skewing difference between input beams has been corrected, the results are still skewed.

To perform a tri-linear interpolation, we need voxel data from two subsequent slices. Figure 7 shows the complete SFG for tri-linear interpolation using the SFG of Figure 6. Because voxels from the second slice are output n clock periods later, we need to delay data from the previous slice by n cycles. Furthermore, because of the skewing difference between beams of subsequent slices, we need to shift the non-delayed output from the memory by one po-

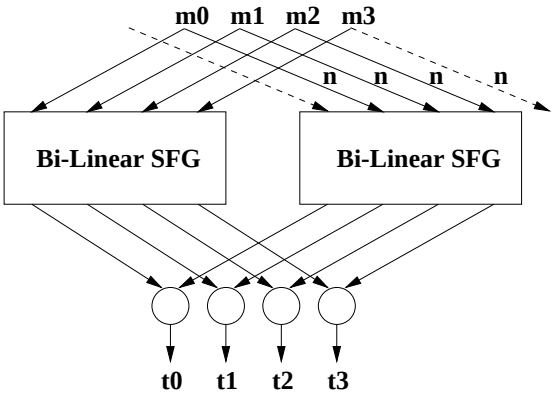


Figure 7: *SFG for tri-linear interpolation.*

sition. For example, compare the beams at index $i = 0$ and $i = 4$ in Figure 4b. The last stage of the SFG in Figure 7 performs the linear interpolation between the bi-linearly interpolated samples of the two slices using w_z, the interpolation weight in Z direction.

ABC Gradient Estimation

ABC gradient estimation is similar to tri-linear interpolation. It requires the collection of a $3 \times 3 \times 3$ neighborhood of interpolated samples between the three ABC sample slices. The ray-samples are output each clock cycle by the tri-linear interpolation stage as skewed beams. To compute any additional samples required for orthogonal gradients (as shown in Figure 3) we use a similar dataflow approach as for bi-linear interpolation.

To compute the gradients we need to collect the data from the three consecutive sample slices. Figure 8 shows the corresponding SFG. The samples currently output by the tri-linear interpolation

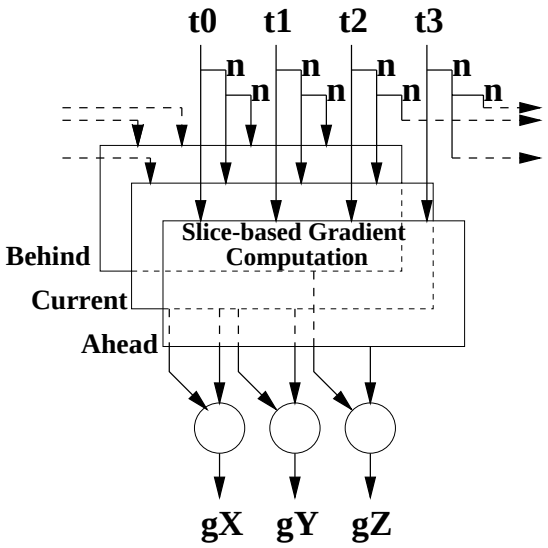


Figure 8: *Top-level SFG for ABC gradient estimation.*

stage are input without delay as ahead samples. The ahead samples are delayed by n cycles and input as the current samples. A delay of the current samples by n cycles produces the behind samples. As in the case of tri-linear interpolation, the delayed samples are shifted according to their skewing distance. The last stage in the SFG computes the central differences between interpolated samples and outputs the three gradient components.

Shading and Classification

Using this gradient, each sample is shaded using any of the standard local illumination models. For maximum performance, we need to perfectly pipeline the shading calculations. Other researchers have proposed fully pipelined Phong shading and vector normalization architectures [8]. For our prototype implementation, we use a small, lookup-table based reflectance map shader [2]. It allows to implement any higher-order shading model without expensive square root units. Classification is performed based on sample value and possibly gradient magnitude using a lookup-table opacity map.

Compositing

The shading stage produces consecutive beams of color intensity values within slices. In the slice-parallel dataflow, the compositing stage accumulates these intensity values to pixels stored in the base-plane. The total size of this base-plane buffer is $(2n)^2$, the maximum size of a base-plane [15]. However, this buffer is distributed among m compositing units. The difficulty is how to forward the intensity values along a ray to the compositing unit that stores the intermediate base-plane pixel corresponding to that ray. Or, alternatively, how to forward the intermediate base-plane pixel value to the compositing unit that receives the next intensity value along the ray.

Consider a partially composited base-plane pixel that was produced after compositing slice S. We have to forward this pixel to the compositing unit which receives the next intensity value along the ray from the shader. Because all rays are 26-connected in discrete space, the next sample along the ray must come from a 3×3 neighborhood inside the next slice $S + 1$. Using the discrete ray-templates of the template-based ray-casting algorithm [17], we can determine the position inside this neighborhood of the next intensity value along the ray. Using Figure 9, we can determine the forwarding pattern for all possible cases. The figure assumes that the major viewing direction is *Z* and that the dataset is stored along beams in *X* direction.

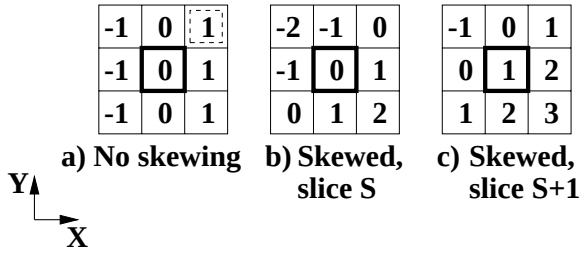


Figure 9: *Compositing neighborhood.*

Figure 9a shows the 3×3 neighborhood in case of no skewing. The center position, surrounded by a box, indicates the current position of the intermediate base-plane pixel. The numbers indicate the relative distance in *X* to the compositing unit that receives the next sample along the ray. For example, if the ray-templates indicate that the ray in discrete space makes a step in positive *X* and *Y* directions, the next sample is forwarded to the compositing unit one position in positive *X* direction (shown by a dashed box in the figure). Because of the skewing difference between beams inside slices, this forwarding distance is altered as shown in Figure 9b. Finally, Figure 9c shows the forwarding distances if we take the skewing between slices S and $S + 1$ into account. Because of the forwarding distances, each compositing unit has to be connected to three units in positive and one unit in negative *X* direction. Figure 10 shows the corresponding SFG for compositing. Notice that,

due to the maximum skewing differences of -1 and $+3$ shown in Figure 9c, a minimum of five rendering pipelines is required.

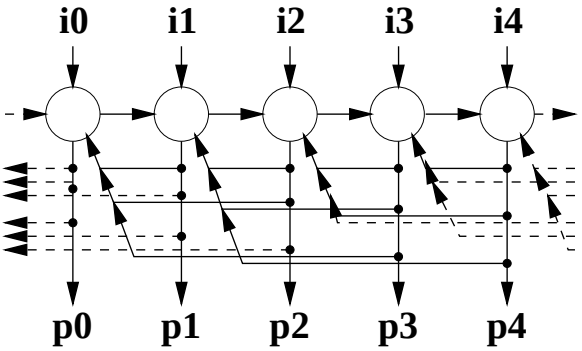


Figure 10: *Compositing SFG.*

The resulting pixels of the base-plane are still generated in a skewed order. However, pixel scanlines can easily be unskewed by a simple address-permutation inside or when stored into the frame-buffer.

Extensions for $m < n$

If $m < n$, we have to add two minor changes to the dataflow presented so far. Instead of complete beams we forward partial beams with $\frac{n}{m}$ samples each. The order of partial beam access is along beams. To fetch the data of a complete beam requires m cycles instead of one cycle. Consequently, all delay operations on edges in the SFGs, which are needed to gather data from consecutive beams, need to be changed from 1 to m.

The second change is required because of border cases between partial beams. For example, the tri-linear interpolation units at rightmost position m require voxels from the partial beam that will be fetched one cycle later. Figure 11 shows how to deal with these border cases using a technique we call *beam-extension*. The partial

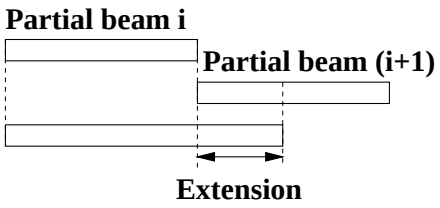


Figure 11: *Beam extension.*

beam i, is delayed by one cycle, until the next partial beam $(i+1)$ arrives, and the overlap necessary for the border cases is available as an extension to beam i. Notice that we need to extend beams only in the direction of partial beam access. The amount of extension depends on the processing stage and varies between 3 and 4 data samples.

6 Cube-4 Architecture

Figure 12 shows the complete top-level diagram of the Cube-4 architecture with five rendering pipelines. Due to the skewing difference for pixel exchange in the compositing stage this is the minimal configuration. The dataset is stored in the multiple cubic frame buffer (CFB) memory modules. Each rendering pipeline contains four types of processing units: CFB memory and address generation, tri-linear interpolation (TRILIN), ABC gradient estimation and shading (Shader), and compositing (Compos). All datapaths have constant width, corresponding to the word-width of a voxel (e.g., 8 or 16 bits). The delay of data required for tri-linear interpolation and for the ABC gradient estimation is achieved by first-in first-out (FIFO) memories.

Control of Cube-4 is very simple and can be part of the dataflow. The host downloads the viewing vector into the CFB address-generation units. The ray templates are generated in hardware by adding the viewing vector to the current sample location and computing the resampling weights. From there on, all necessary control signals travel with the data through the machine, making centralized control unnecessary.

7 Simulations and Prototyping

We have extensively simulated the algorithm and architecture in C and a high-level hardware description language (VHDL). Table 7 shows results from the VHDL simulation. The table shows rendering performance in frames per second versus the number of rendering pipelines for three different dataset resolutions. To translate the number of simulation cycles into frames per second, we assumed a relatively low processing frequency of 33 MHz.

Dataset	# Pipelines	Cycles/frame	Frames/sec
64^3	8	32,814	1,006
	16	16,422	2,009
	32	8,226	4,012
128^3	8	262,206	126
	16	131,118	252
	32	65,574	503
256^3	8	2,097,246	16
	16	1,048,638	31
	32	524,334	63
	64	262,182	126
	128	133,106	248

Table 1: *VHDL simulation results: Rendering performance as a function of the number of rendering pipelines.*

As a proof of concept we implemented a Cube-4 prototype on the Teramac, a configurable custom hardware machine developed at Hewlett-Packard Laboratories [1]. Figure 13a (in the color section of the proceedings) shows a picture of a 4-board Teramac system. Teramac can execute synchronous logic designs of up to one million gates at rates up to 1 MHz. The system has been built from custom field-programmable logic arrays (FPGAs) packaged in large multichip modules (MCMs). Figure 13b (in the color section) shows a picture of a single MCM, which carries 27 FPGAs. Each MCM measures 6.13×7.4 inches, weighs approximately 3 pounds, and has over 3000 pins. The Teramac system we used for our Cube-4 implementation includes 8 boards, 250 MB of RAM, 32 MCMs and 864 FPGAs.

Our prototype of Cube-4 on Teramac implements the design shown in Figure 12 with five rendering pipelines. The implementation is capable of producing parallel color projections of 128^3 8-bit per voxel datasets from arbitrary directions. Inside the shader units, we use a lookup-table based reflectance map shading. The total logic complexity for all five rendering pipelines is 330K gates. Compilation of the complete design onto Teramac takes less than one hour without user intervention.

The Cube-4 prototype generated an image of any of the 128^3 datasets in 1.5 seconds at 0.25 MHz, independent of dataset com-

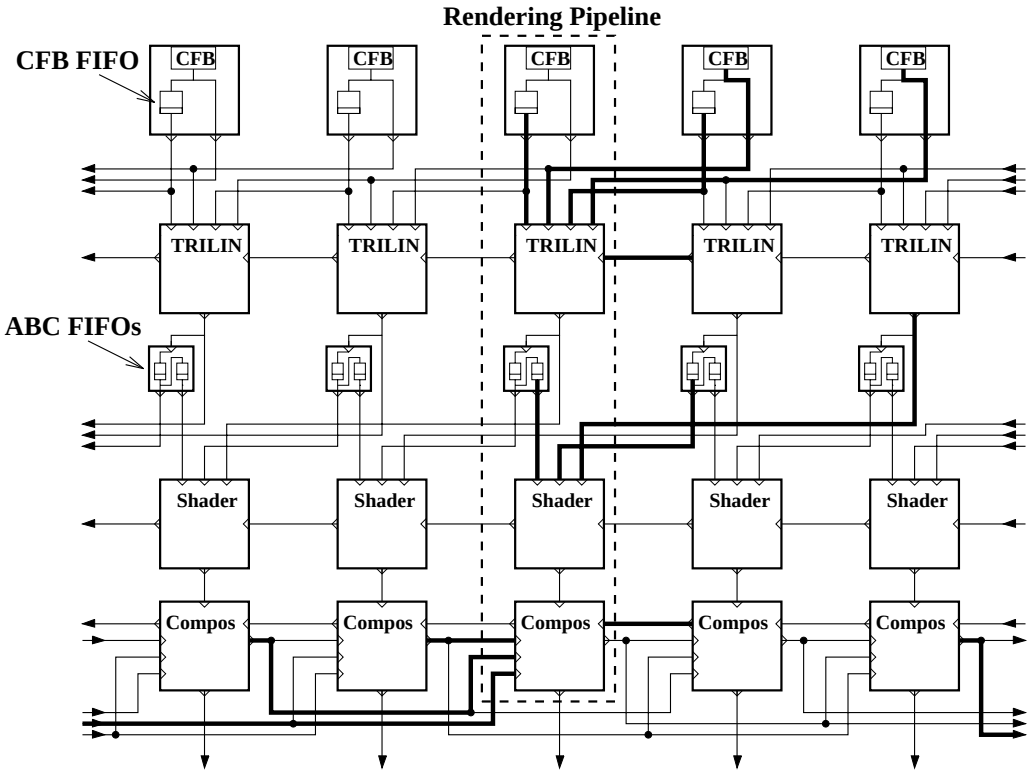


Figure 12: *The Cube-4 slice-parallel architecture. Bold lines indicate all data connections of the rendering pipeline in the center. (CFB = Cubic Frame Buffer, TRILIN = Tri-Linear Interpolation Unit, Compos = Compositing Unit.)*

plexity, transfer function, or viewing parameters. The maximum processing frequency of Cube-4 on Teramac is 0.96 MHz without any performance optimizations, although higher speeds could be achieved by careful insertion of additional pipeline stages. Figure 13c (in the color section) shows volume renderings of a CT lobster dataset and Figure 14 (in the color section) shows volume renderings of several other datasets. The use of different opacity and color transfer functions reveals different aspects of the data.

8 Performance Analysis

The results we presented in the previous section indicate linear scalability of performance with increasing number of rendering pipelines. In this section, we look at the theoretical maximum performance of Cube-4. Assuming perfect pipelining of interpolation, shading, and compositing, we can continually enter data at the maximum possible rate, and the theoretical performance of Cube-4 is thus limited by the access speed of the memories.

If n is the dimension of the dataset, p the number of rendering pipelines, and f_p the processing frequency of the machine, the theoretical rendering rate f_r in frames per second is $\frac{pf_p}{n^3}$. Figure 15 the frame rate f_r as a function of the number of rendering pipelines p for three different dataset sizes. We show graphs for two different processing frequencies f_p. The solid lines shows graphs for f_p = 33 MHz, corresponding to the cycle time of SDRAM, the fastest currently available DRAM memory technology. The dashed lines show performance assuming 100 MHz processing frequency. Because current DRAM memory can not output data at this rate it has to be additionally interleaved per rendering pipeline. This additional interleaving is a standard memory bank arrangement as used in current general-purpose processors.

In order to allow for a compact implementation, we are currently developing an application-specific integrated circuit (ASIC) containing several of the Cube-4 rendering pipelines. We have a contract with a company that will fabricate such an ASIC. Preliminary estimates indicate that an ASIC containing 4 rendering pipelines requires less than 300 pins, including power and ground. Each ASIC requires only 400K gates, and internal memory for the ABC FIFO buffers of 40 K, assuming a total of 32 rendering pipelines.

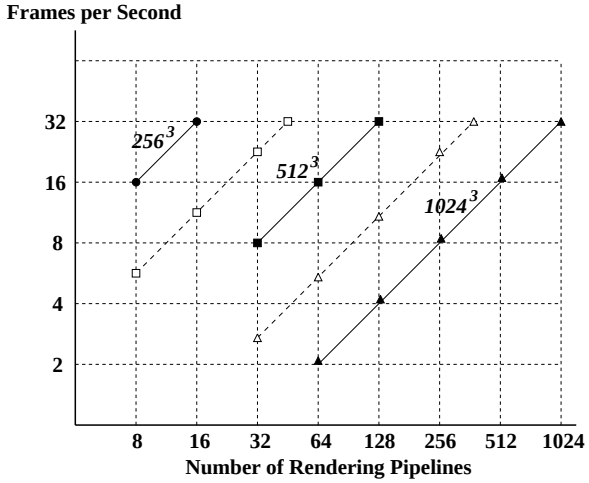


Figure 15: *Theoretical rendering performance of Cube-4 as a function of the number of rendering pipelines. We show graphs for different dataset sizes ($\bigcirc = 256^3, \square = 512^3, \triangle = 1024^3$). Solid lines indicate 33 MHz processing frequency, and dashed lines indicate 100 MHz processing frequency.*

We are designing a long PCI card system with 32 rendering pipelines or 8 Cube-4 ASICs, 32 SRAM chips, and a PCI host interface. Such a card would cost a few thousand dollars and provide 30 projections per second for 256^3 datasets. Larger systems for higher resolution datasets supporting 30 projections per second, 16-bit per voxel, can be built, such as a workstation board (e.g., VME size) for 512^3 datasets, and multiple boards for 1024^3 datasets.

9 Conclusions

We have introduced Cube-4, a scalable architecture for true real-time ray-casting of large volumetric datasets. The unique features of Cube-4 are a high bandwidth skewed memory organization, localized and near-neighbor datapaths, and multiple, parallel rendering-pipelines with simple processing units. System performance scales linearly with the number of rendering pipelines, limited only by memory access speed. The Cube-4 architecture, viewed as a near-neighbor array of simple processors, is extremely well-suited for very large scale integration (VLSI). Due to its modularity, it is feasible to build a Cube-4 VLSI chip containing several rendering pipelines. Such a chip allows the construction of modular and cost-effective small to medium size volume rendering systems with true real-time performance for low- to high-resolution datasets – far above the performance of current systems.

Finally, the choice of whether one adopts a general-purpose or a special-purpose solution to volume rendering depends upon the circumstances. If maximum flexibility is required, general-purpose appears to be the best way to proceed. However, an important feature of graphics accelerators is that they are integrated into a much larger environment where software can shape the form of input and output data, thereby providing the additional flexibility that is needed. A good example is the relationship between the needs of conventional computer graphics and special-purpose graphics hardware. Nobody would dispute the necessity for polygon graphics acceleration despite its obvious limitations. We are making the exact same argument for our Cube-4 volume rendering architecture.

Acknowledgements

This work has been supported by the National Science Foundation under grant MIP-9527694, Japan Radio Corporation, and Hewlett Packard. Datasets for Figures 13 and 14 are courtesy of Siemens, Scripps Clinic, AVS, UNC, Howard Hughes Medical Institute, and the Visualization Laboratory at Stony Brook. The authors would like to thank all the members of the Cube-4 team that contributed to this research, especially Frank Wessels, Urs Kanus, Michael Meissner, Ingmar Bitter, and Pat Tonra. Special thanks for support from Richard J. Carter and the Teramac design team of W. Bruce Culbertson, Philip Kuekes, Greg Snider, and Rick Amerson at HP Labs.

References

[1] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac – configurable custom computing. In *Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines*, pages 32–38, Napa, CA, April 1995.

[2] M. Bosma and J. van Scheltinga. Efficient super resolution volume rendering. Master's thesis, University of Twente, Faculty of Electrical Engineering, Enschede, The Netherlands, August 1995.

[3] G. G. Cameron and P. E. Underill. Rendering volumetric medical image data on a SIMD-architecture computer. In *Proceedings of Third Eurographics Workshop on Rendering*, pages 135–145, Bristol, UK, May 1992.

[4] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, August 1988.

[5] T. Günther, C. Poliwoda, C. Reinhard, J. Hesser, R. Männer, H.-P. Meinzer, and H.-J. Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. In *Proceedings of the 9th Eurographics Hardware Workshop*, pages 103–108, Oslo, Norway, September 1994.

[6] A. Kaufman. *Volume Visualization*. IEEE CS Press Tutorial, Los Alamitos, CA, 1991.

[7] A. Kaufman and R. Bakalash. Memory and processing architecture for 3D voxel-based imagery. *IEEE Computer Graphics & Applications*, 8(6):10–23, November 1988.

[8] G. Knittel. VERVE: Voxel engine for real-time visualization and examination. In *Computer Graphics Forum*, volume 12, No. 3, pages 37–48, September 1993.

[9] G. Knittel and W. Strasser. A compact volume rendering accelerator. In *1994 Workshop on Volume Visualization*, pages 67–74, Washington, DC, October 1994.

[10] P. Lacroute and M. Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 451–458. ACM Press, July 1994.

[11] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[12] H. Pfister, A. Kaufman, and T. Chiueh. Cube-3: A Real-Time Architecture for High-Resolution Volume Visualization. In *Volume Visualization Symp. Proc.*, pages 75–83, Washington, DC, October 1994.

[13] H. Pfister, A. Kaufman, and F. Wessels. Towards a scalable architecture for real-time volume rendering. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, pages 123–130, Maastricht, The Netherlands, August 1995.

[14] R. A. Reynolds, D. Gordon, and L.-S. Chen. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38(3):275–298, 1987.

[15] P. Schröder and G. Stoll. Data parallel volume rendering as line drawing. In *1992 Workshop on Volume Visualization*, pages 25–31, Boston, MA, October 1992.

[16] L. Westover. Footprint evaluation for volume rendering. In *Computer Graphics Proceedings*, volume 24/4, pages 367–376. ACM Siggraph, August 1990.

[17] R. Yagel and A. Kaufman. Template-based volume viewing. *Computer Graphics Forum*, 11(3):153–167, September 1992.