EM-Cube: An Architecture for Low-Cost Real-Time Volume Rendering

EM-Cube: An Architecture for Low-Cost Real-Time Volume Rendering

Rändy Osborne* Hanspeter Pfister Hugh Lauer Neil McKenzie Sarah Gibson Wally Hiatt TakaHide Ohkami

MERL – A Mitsubishi Electric Research Lab

Abstract

EM-Cube is a VLSI architecture for low-cost, high quality volume rendering at full video frame rates. Derived from the Cube-4 architecture developed at SUNY at Stony Brook, EM-Cube computes sample points and gradients on-the-fly to project 3-dimensional volume data onto 2-dimensional images with realistic lighting and shading. A modest rendering system based on EM-Cube consists of a PCI card with four rendering chips (ASICs), four 64Mbit SDRAMs to hold the volume data, and four SRAMs to capture the rendered image. The performance target for this configuration is to render images from a $256^3 \times 16$ bit data set at 30 frames/sec. The EM-Cube architecture can be scaled to larger volume data-sets and/or higher frame rates by adding additional ASICs, SDRAMs, and SRAMs.

This paper addresses three major challenges encountered developing EM-Cube into a practical product: exploiting the bandwidth inherent in the SDRAMs containing the volume data, keeping the pin-count between adjacent ASICs at a tractable level, and reducing the on-chip storage required to hold the intermediate results of rendering.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors B.3.2 [Memory Structures]: Design Styles—Interleaved Memories

1 Introduction

Real-time volume rendering is an enabling technology for medical applications including diagnosis, surgical training, and surgical simulation [6]. The large computational and memory requirements of real-time volume rendering place it beyond the capabilities of single processor PCs and workstations without dedicated hardware. While high performance graphics systems can perform volume rendering in real-time (e.g. the SGI InfiniteReality Engine), such systems are very expensive.

Our goal is to develop a family of products that provide real-time volume rendering at affordable prices — i.e., within reach of personal computer budgets. This family is intended to address medical applications where volume rendering is an obvious requirement, but also to provide a foundation for the development of *interactive volume graphics* — that is, the graphics of 3-D sampled images and their manipulation at interactive speeds. We expect that as systems for real-time volume rendering become cheaper and more commonplace, a broader class of applications — e.g. scientific visualization, industrial design and analysis, virtual sculpture, and games — will begin to use volume graphical methods. Eventually, we envision that the mechanisms of volume graphics and conventional polygon-based graphics will converge, so that both kinds of rendering will be supported by the same kind of hardware.

This paper describes the architecture of the first member of this family, a volume rendering chip currently under development. The architecture is a scalable systolic array based on Cube-4, developed at SUNY at Stony Brook [16]. The performance target is a chipset that fits onto a single PCI card and renders volume data sets of size $256^3 \times 16$ bit voxels, at 30 frames/sec. The cost of such an accelerator will be on the order of a low-cost PC. In subsequent generations the cost will decrease as the underlying implementation technology improves.

Cube-4, though scalable to larger volumes by adding more ASICs and memory modules, is impractical for low-cost ASIC implementation. The key challenges are delivering the required bandwidth with as few chips as possible, reducing the inter-chip communication to keep the pin count reasonable, and reducing the on-chip storage required for intermediate results. Our EM-Cube (Enhanced Memory Cube-4) architecture meets the first two challenges by using a block skewed memory, which exploits inherent SDRAM burst bandwidth, and meets the third challenge by subdividing the volume in a technique we call sectioning.

The organization of this paper is as follows. Section 2 describes related work. Sections 3 and 4 describe Cube-4 and introduce the three implementation challenges. Sections 5 and 6 introduce block skewed memory and show how it meets the first and second challenges respectively. Section 7 discusses the on-chip storage problem and our solution via sectioning. Section 8 presents the overall architecture. Finally, Sections 9 to 11 discuss features needed for a commercial product, such as support for multiple voxel formats.

2 Related Work

Several approaches have been taken to achieve interactive volume rendering rates. Software implementations use acceleration techniques which require pre-computation, additional data storage, or trade-off image quality for speed. Shear-warp rendering, the currently fastest software algorithm, achieves one projection in a few seconds on a regular workstation [11]. Many researchers have implemented volume rendering algorithms on large general-purpose multiprocessors [2, 5, 14, 15]. However, this approach requires expensive, typically network-shared machines to achieve acceptable frame rates, and the lack of direct frame-buffer access prohibits real-time output rates. Another approach is to use existing polygon graphics hardware for volume rendering [18, 8, 13]. Interactive ren-

*osborne@merl.com, 201 Broadway, Cambridge, MA 02139, Phone: (617) 621-7524, FAX: (617) 621-7550

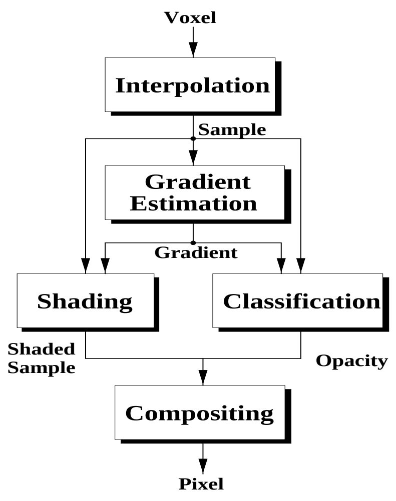

Figure 1: *Rendering pipeline*

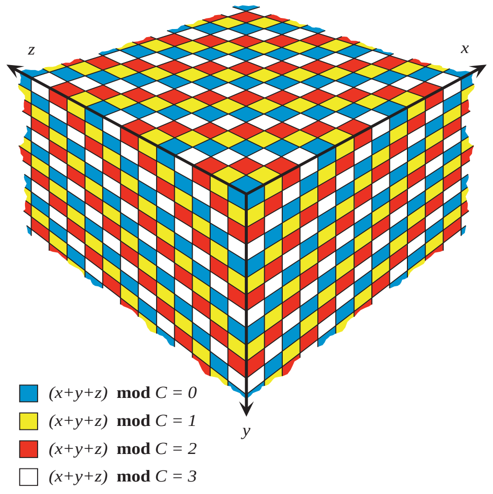

Figure 2: *Skewed voxel memory*

dering rates have been achieved on the SGI Reality Engine using 3D texturing hardware [3, 1]. However, current 3D texturing hardware is expensive and does not support estimation of gradients that is required for high-quality shading and classification. Furthermore, the best volume rendering performance on large general-purpose supercomputers or special-purpose texture mapping hardware is still below 15 frames/sec for 256^3 volumes.

In view of these limitations, it is not surprising that a number of researchers have undertaken the development of special-purpose hardware for volume rendering. VOGUE, one of the most concrete proposals, is a compact ray-casting unit which provides interactive rendering speeds at moderate hardware costs [10]. A single board consisting of eight-way interleaved volume memory and four VLSI chips provides 2.5 frames/sec for 256^3 volumes. Near real-time rates of 20 frames/sec can be achieved by connecting several modules over a ring-connected cubic network [9]. VIRIM, an object-order volume rendering engine, is one of the few research proposals that has been built and tested [7]. The machine consists of four VME boards with special-purpose geometry processors for data resampling and programmable ray-casting processors for the final image generation. VIRIM achieves 2.5 frames/sec for 256^3 datasets.

3 Cube-4 Architecture

Cube-4, developed at SUNY Stony Brook, is a scalable systolic array of rendering pipelines, each connected to its own memory module [16]. Figure 1 shows the major functions in each rendering pipeline. Cube-4 uses a modified ray casting algorithm. Instead of processing along each ray in depth-first fashion, Cube-4 processes rays in parallel in a breadth-first fashion. In particular, all the sample points contained in an entire plane of voxels are processed in parallel, thereby avoiding the need to re-read neighboring voxels from memory. Such a voxel plane, called a slice, is always perpendicular to one of the three axes of the volume data cube. Cube-4 chooses the direction for the slice such that the slice normal subtends the smallest angle with the actual viewing direction.[1]

Since a slice has too many voxels to be processed at once, Cube-4 scans each slice a beam (i.e. a row) at a time. Beams are further divided into partial beams of p voxels. Each voxel of a partial beam is processed by a separate rendering pipeline capable of fetching a new voxel from an associated memory module every clock cycle. Thus a Cube-4 system with p pipelines can process a beam in N/p cycles, a slice in N^2/p cycles, and a volume in N^3/p cycles, where N is the size of a cubic dataset in any dimension.

A key feature of the Cube-4 architecture is that rendering pipelines communicate only locally with associated memories and neighboring pipelines up to three away. Thus the Cube-4 architecture is highly scalable.

3.1 Cube-4 Skewed Memory

A fundamental challenge in Cube-4 is arranging data amongst memory modules so that the processing chips can concurrently fetch all p voxels in a partial beam regardless of the viewing direction. To meet this challenge, Cube-4 uses 3D skewed memory. A voxel at position (x, y, z) in unskewed voxel space is mapped to position (i, r, s) in skewed voxel space where $i = (x + y + z) \bmod N$, $r = y$, and $s = z$. Given C memory modules, where N is a multiple of C, a voxel (i, r, s) in skewed voxel space is mapped to module number $i \bmod C$ and to an address within that memory module of $\lfloor i/C \rfloor + r * N/C + s * N^2/C$.

The layout of voxels in the volume memory is illustrated in Figure 2 which shows a set of voxels near the origin in each of the three dimensions for $C = 4$. Voxels are represented by small cubes, with the shading illustrating their assignment to memory modules. The ordering of the assignments of colors to voxels is identical for each of the three visible faces. Throughout the volume, adjacent voxels within a beam are stored in adjacent memory modules, and thus regardless of the view direction, a partial beam of $p = C$ voxels can be fetched concurrently from the C separate memory modules.

The 3D skewing introduces a lateral shifting in voxels between adjacent beams within a slice and also between adjacent beams in a row plane perpendicular to a slice. As discussed in Section 6, this shifting must be undone in order to process each voxel (e.g. see Figure 6), and it leads to significant communication between adjacent rendering pipelines.

4 Implementation Issues

To achieve a low-cost system, the number of rendering chips and associated memory chips must be as small as possible. The rendering chips must have a reasonable die size and must be compatible with

[1]The algorithm chooses arbitrarily amongst view normals having equally small subtended angles.

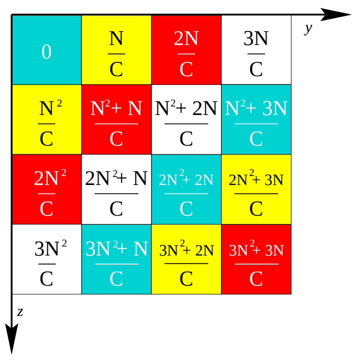

Figure 3: *Memory location assignments of YZ face*

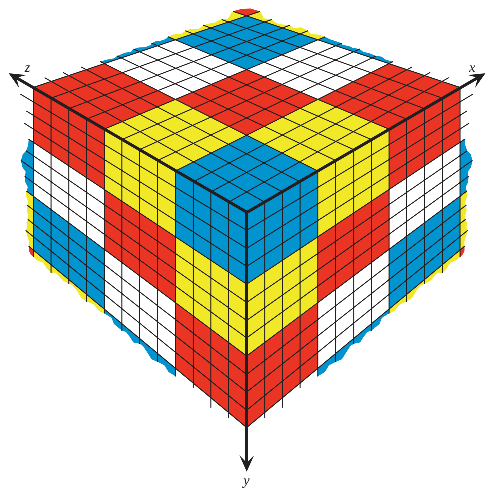

Figure 4: *Blocked skewed memory* $(b = 4)$

current packaging technology. The Cube-4 architecture described in Section 3 does not meet these goals. It requires too many memory modules (about 20), too many pins per rendering chip (on the order of 512 signal pins), and too much on-chip storage, resulting in an excessively large die (in excess of 100mm^2 for storage alone). Subsequent sections describe each of these points in more detail and describe our modifications to Cube-4 to attain a feasible design for VLSI implementation.

5 Voxel Bandwidth

To meet our performance targets, the voxel memory must have a capacity of 32Mbytes and must deliver a sustained bandwidth of 1Gbyte/sec independent of view direction.

5.1 Cube-4 memory access patterns

The Cube-4 skewed memory organization has view-dependent memory access strides which exceed common DRAM page sizes for some view directions. This precludes the use of fast page (i.e. column) mode access in DRAMs in such view directions, reducing achievable memory performance to random (i.e. row) access levels. View dependence forces the entire memory system design to handle this worst case.

In particular, for a N^3 dataset with C memory modules, the memory access stride is 1, N/C, or N^2/C if the view normal direction is parallel to the Z, X, or Y axes respectively. Figure 3 shows the assignment of memory locations of voxels on the YZ face for a view direction parallel to the X axis. A stride of N/C is required to access successive voxels in successive partial beams parallel to the Y axis. Moreover, there is an anomaly in this stride at the beginning of each beam. Therefore, except for small N and/or large C, only a few successive accesses will fall on the same DRAM page, making little benefit of fast page mode access. Likewise, on the ZX face (not shown), a stride of N^2/C is required to access successive voxels of successive partial beams, also with an anomaly at the beginning of each beam. For small C and reasonable values of N, this N^2/C stride is larger than typical DRAM pages, completely precluding the use of fast page mode.

5.2 Memory Technology

64Mbit synchronous DRAMs (SDRAMs) will be the mainstream DRAM in the next 1-2 year period. Such SDRAMs meet our 32Mbyte capacity requirement, and 4Mx16 versions at 125MHz deliver 1Gbyte/sec with just 4 chips. 64Mbit Rambus(TM) will ramp up during the same period but its higher clock speed requires a more complicated interface.

Unfortunately, Cube-4's large memory strides prevent getting anywhere near the maximum 1Gbyte/sec bandwidth with 4 memory chips. For Mitsubishi Electric's 64Mbit 125MHz SDRAM, the cycle time for a row access is $t_{RC} = 80$nsec. In practical operation, at most two banks can be overlapped in t_{RC}, thus limiting the maximum performance to 2 accesses per 80nsec, or 50Mbytes/sec per SDRAM (at 16bits/voxel). Thus 20 SDRAMs are needed to obtain 1Gbyte/sec. The situation is similar for Rambus since it is also block oriented. This number is unreasonable for a low cost design.

To significantly reduce the row access time, the DRAM banks must be smaller, and as a side effect usually less dense. Examples are 16Mbit Enhanced SDRAM (30nsec row access time) and MoSys's 1Mbyte multibank MDRAM (20nsec row access). However, these devices are too slow (a 20nsec row access time implies 10 chips) or not dense enough. The performance of various cache+DRAM combinations, such as 16Mbit cached DRAM (CDRAM) and Enhanced SDRAM, degrades to the row access time for strides greater than a DRAM page.

5.3 Block Skewed Memory

To take advantage of the high bandwidth of SDRAM in fast page mode, we organize the volume memory into subcubes or blocks of $b \times b \times b$ voxels in such a way that all of the voxels of a block are stored linearly in the same DRAM page. The memory is still skewed to support rendering independent of view direction, but it is now skewed at the block granularity rather than voxel granularity as in Cube-4. Each rendering chip processes a block and maintains a block-sized reordering buffer so that the voxels in a block can be read out in the order appropriate for the view direction. Figure 4 illustrates the block skewed memory for $b = 4$.

In this new organization, a row of blocks comprises a block-beam and a two-dimensional array comprises a block-slice. At the block granularity the processing algorithm is the same as the Cube-4 algorithm, except that partial block-beams replace partial beams. Each block is processed internally on a voxel granularity using the Cube-4 algorithm.

There are several design points for b.

PageBlock: b can be as large as possible while still allowing the b^3 block to fit into a single DRAM page. Thus the burst transfer size can be as large as a page size, which easily permits sustaining full bandwidth from the SDRAMs. One disadvantage of this scheme is the block size depends on the voxel size. The 512 byte pages in

64Mbit SDRAMs support $b = 8$ for 8 bit voxels and $b = 4$ for 16 or 32 bit voxels. Another disadvantage is that it requires a page-sized buffer on-chip.

MiniBlock: Alternatively, b can be as small as possible. This eliminates the sensitivity to voxel size. Blocks with $b = 2$ are large enough to completely overlap the row access overhead of the SDRAM module with data transfer. Assuming 16 bit voxels and Mitsubishi Electric's 4Mx16 SDRAM at 125MHz, the single burst access time for a 2x2x2 block is 112nsec, i.e. 8 accesses in 14 clocks. Two of the four banks in the SDRAM can be interleaved to achieve 8 accesses in 8 clocks, i.e. full bandwidth.[2] A disadvantage of $b = 2$ is the large inter-chip communication.

Hierarchical Blocks: A compromise yielding the advantages of both large and small block sizes can be achieved by tiling blocks of size b with miniblocks. The blocks themselves are skewed across memory modules, but the miniblocks within them are not. This hierarchical blocking permits efficient implementation of larger blocks e.g. PageBlocks. Instead of fetching the entire block at once, which requires a b^3 voxel buffer, miniblocks can be fetched on a row by row basis on demand. This capability ensures minimal overhead for the sectioning described in Section 7.1.

The maximum block size is $b \leq N/C$ since blocks must be skewed over C chips so that a block-beam can be fetched without conflict for any view direction.

A hierarchical blocking scheme is also described in [12]. The data volume is divided into subcubes and subcubes are divided into 2x2x2 "supervoxels". However, while the hierarchical division is the same as above, the actual memory blocking is different. In [12] the eight voxels in a supervoxel are distributed across eight memory modules, i.e. supervoxels are the unit of interleaved memory access. In our blocking, all the voxels comprising a block are located in the same memory and miniblocks are the unit of pipelined burst access. In addition, all the blocks are skewed.

6 Inter-chip Communication

Figure 5 shows the EM-Cube architecture in a generic way independent of b. Voxel blocks are distributed across the set of SDRAM volume memories at the top. Each rendering chip connects to a SDRAM memory module, a pixel memory chip (SRAM or DRAM) for output, and neighboring rendering chips for transfer of intermediate values.[3] Such inter-chip communication is required for resampling (intermediate trilinear interpolation results and possibly voxels), gradient estimation (intermediate results and trilin results), and compositing (partial pixels).

Each voxel block is processed by a single rendering chip. Within a block, intermediate values are communicated on-chip. The only inter-chip communication results from processing voxels near the faces of each block. Since the area of a block face is b^2, the inter-chip communication grows as b^2. On the other hand, the number of voxels processed per block grows as b^3. Therefore, on a per voxel basis, the interchip communication scales as $1/b$. Thus a design with $b = 4$ requires up to 4 times less inter-chip communication bandwidth[4] than Cube-4. Table 1 summarizes the inter-chip communication requirements for several architectural variations. The compositing communication depends on the view direction.

	Trilin	Grad est	Compos
Unskewed	1	2	1
Cube-4	3	3	1
EM-Cube	$\frac{3}{b}$	$\frac{3}{b}$	$\frac{1}{b}$ to $\frac{3b^2-3b+1}{b^3}$

Table 1: *Summary of inter-chip communication bandwidth (in "values"/clock)*

The inter-chip communication for resampling has an interesting geometric interpretation. The left side of Figure 6 shows, in unskewed voxel space, the eight voxel neighborhood for trilinear interpolation. Here we assume $b = 1$ to simplify the picture, and thus there is one memory module and one rendering chip for each column i. It suffices to communicate the bilinear interpolation of the four side face voxels (e.g. 2, 4, 6, and 8) to the left neighbor. Skewing the volume transforms the eight voxel neighborhood cube into the slanted parallelepiped in the right of Figure 6. The transformation is the same as pulling vertices 4 and 5 of the unskewed voxel cube laterally to the right and left, respectively. Such pulling spreads the eight voxel cube over four columns. To perform the trilinear interpolation, we first undo the skewing by shifting voxels 5 and 6 to the right by 1 and likewise shifting voxels 3 and 4 to the left by 1. This lateral communication can be pipelined, with all front bottom voxels moving one to the right and all top rear voxels moving one to the left on each clock. The four side face voxels are then bilinearly interpolated and the result sent laterally to the left neighbor to compute the final trilinear interpolation result. The total communication is thus 3 values per clock. For $b > 1$ each vertex becomes a b^3 block of voxels and b^2 face voxels move to the left and another b^2 move to the right each time step.

For compositing, the inter-chip communication is equal to the number of rays exiting a block. The best case shown in Table 1 occurs for a viewing direction parallel to an axis and the worst case occurs for a ray direction 45 degrees from two axes. The worst case communication scales as $1/b$ in all three dimensions. Thus b must be fairly big, e.g. 8, before there is a significant reduction in total compositing communication from the $b = 1$ case.

Comparing the entries in Table 1 for Cube-4 (skewed volume) and the unskewed volume reveals that skewing significantly increases the inter-chip communication. However, the unskewed volume is not practical because either the view direction must be restricted or there must be a copy of the entire dataset for each axis direction.

The blocked architecture permits a tradeoff between signal frequency and the pin count for inter-chip communication. The inter-chip bandwidth decreases by b allowing fewer pins and/or lower frequency. For example, if the resampling stage uses 16bit voxels, the inter-chip communication can be any combination of $(16/w)$bits wide every $(b/w) * 8$nsec where $w = 1, 2, 4, 8$, and 16 and $w < b$.

For $b = 8$ we estimate a rendering chip will have 267 signal pins. This is feasible for today's packaging technology. Only 20 of these pins need to run at 125MHz, the remainder at 62MHz or less. All the inter-chip signals use quarter-width paths, i.e. the pins are multiplexed over four 62MHz clocks. The unskewed volume variation has 72 fewer pins. Thus skewing costs 72 pins for $b = 8$ (the cost increases for smaller b).

7 On-chip Storage

As depicted in Figure 5, each rendering chip needs buffer storage for buffering blocks, voxels for interpolation, values on the slice ahead and slice behind for gradient estimation, and partially composited

[2]Provided that every row is accessed at least once within every 64msec, no additional overhead is necessary for refresh. Rendering the entire 256^3 dataset of 16 bit voxels accesses every row of four 64Mbit SDRAMs every 32msec. For a smaller volume or smaller voxel size, rendering might not access every row every 32msec. However, we do not need full 250Mbytes/sec bandwidth in such cases and thus we can slip in auto-refresh cycles without degrading the bandwidth.

[3]Because of the one-to-one correspondence of memory modules and rendering chips, we use C interchangeably for either.

[4]Exactly 4 less except for compositing which is $37/64$ less. See Table 1.

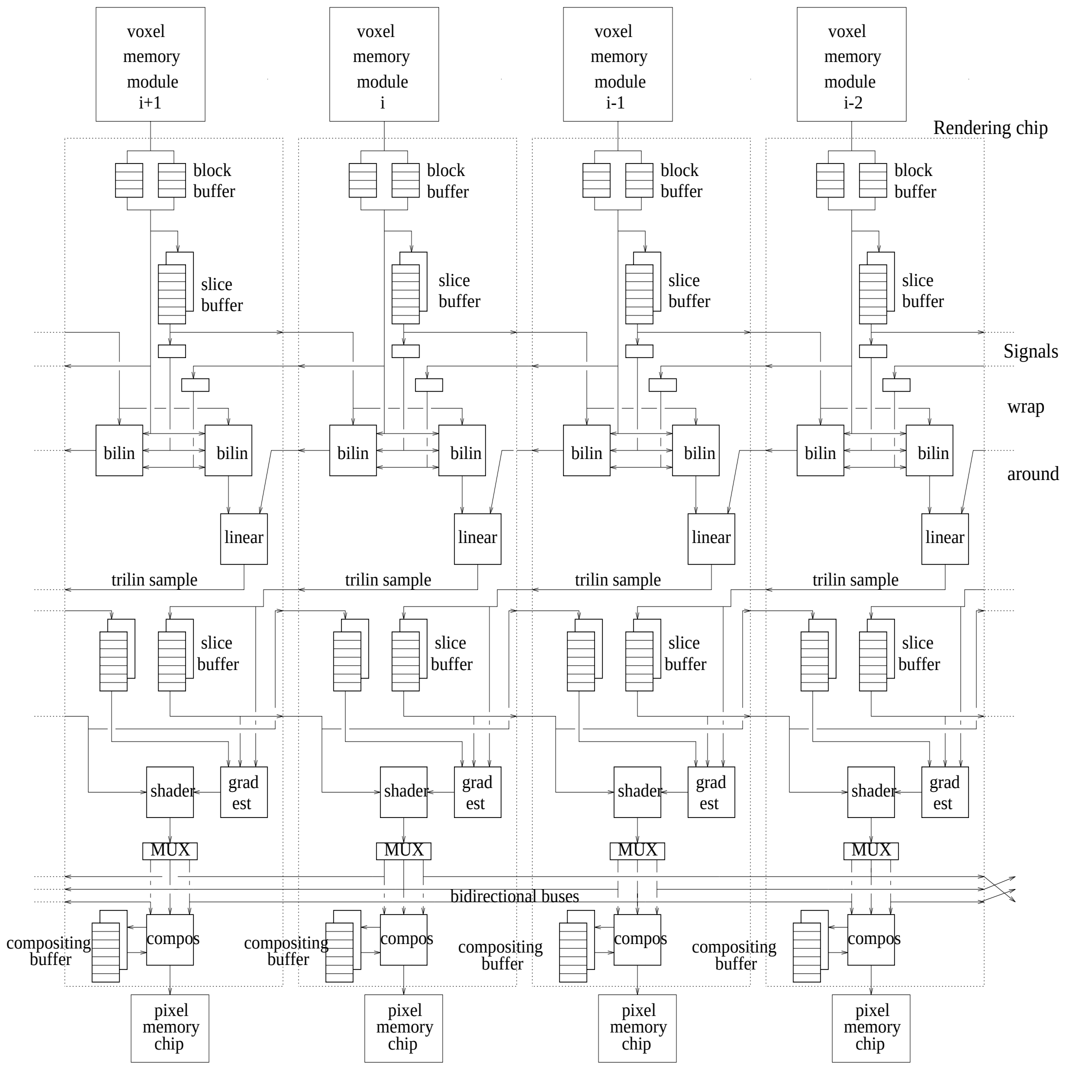

Figure 5: *EM-Cube architecture (4 rendering chips shown)*

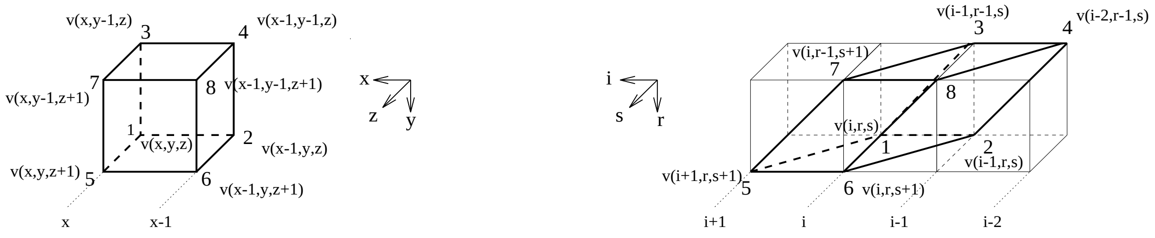

Figure 6: *Unskewed (left) and skewed (right) voxel cubes*

	6 bytes/pixel	3 bytes/pixel
Block buffer	3.1	3.1
Interpolation	1053/C	1053/C
Grad est	2097/C	2097/C
Compos	3146/C	1573/C
Lookup	36.9	36.9
Total	6296/C + 40	4723/C + 40

# chips	6 bytes/pixel	3 bytes/pixel
4	1614	1221
8	827	630
16	433	335
32	237	187

Table 2: *On-chip buffer storage for $b = 8$ (Kbits/chip where C is the number of chips)*

pixels. Each chip also needs lookup tables for opacity values, color values, and shading (not shown in Figure 5).

The blocked architectures require a reordering buffer of b^3 voxels. For uninterrupted supply of voxels, the block buffer must be double buffered with $2b^3$ voxel storage per rendering chip. However, for hierarchical blocking the storage drops to $3b^2$ voxels ($b > 2$).

Trilinear interpolation requires voxels in two adjacent slices. Thus voxels must be buffered from one slice to the next. This storage is independent of the architecture (e.g. Cube-4 or EM-Cube) and depends solely on the number of rendering chips, C. The slice storage required per rendering chip is N^2/C voxels. However, interpolation also requires voxels in the previous row, thus the total interpolation storage per rendering chip is $(N^2 + N)/C$ voxels.

To compute a central difference for gradient estimation requires samples from a slice ahead and a slice behind. This requires two slice buffers and thus the gradient estimate storage per rendering chip is $2N^2/C$ samples.

Shading produces partial pixels. As these partial pixels are generated slice by slice, they are composited into a "running" pixel buffer. All the partial pixels along the same ray (i.e. sharing the same screen pixel location) are composited into the same location in the running pixel buffer. Final pixels corresponding to a ray emerging on an exit face are immediately written to pixel memory off-chip. Consequently, only the N^2 running pixels of the slice cross-section of the volume need to be stored. Thus the compositing storage per rendering chip is N^2/C running pixels. We allow 3 to 6 bytes per pixel to cover a number of possible pixel formats, e.g. containing an alpha value (for front-to-back compositing).

For lookup tables, we assume a two-tiered table opacity lookup with two 512byte tables and one 512 entry table per color component (3x512 bytes total). Shading is not yet finalized. One possibility is the lookup table method of [17] which uses a reflectance map (one 512 byte table per axis direction, for 3x512bytes total) and an arctangent table (one 512 byte table).[5] The total for all lookup tables is 9x512bytes.

Table 2 lists the total on-chip storage required for $N = 256, b = 8$ with hierarchical blocks, and 16 bit voxels. With present embedded SRAM densities, the buffer storage per chip must be less than roughly 200Kbits to ensure a cost-effective core area of about 100mm^2, reserving half the core for logic. Thus 32 chips are required. This is far too many chips for a cost effective solution.

[5]This produces grey level shading; full color shading requires one reflectance map per color component.

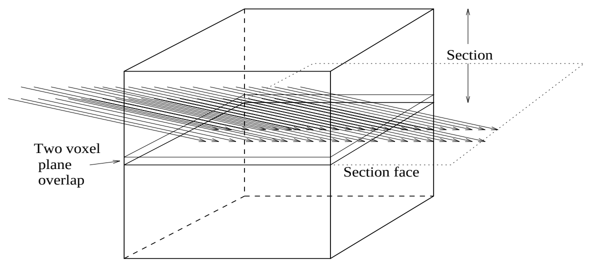

Figure 7: *Sectioning of volume memory*

7.1 Sectioning – A Solution for the On-Chip Buffer Size Problem

To reduce the on-chip buffer area to a feasible amount, we use the same approach as in [4]: we divide the volume into L horizontal sections as shown in Figure 7. We process each section in turn using the EM-Cube algorithm and then combine the results. This sectioning reduces the slice face area and hence the size of slice buffers: L sections reduce the size of on-chip slice buffers by $1/L$. For $C = 4$ chips, $L = 8$ is a feasible design.

Sectioning does not come for free. We are performing a space-time tradeoff: we re-read voxels from volume memory and move some intermediate results back and forth from external pixel memory.

7.1.1 Voxel bandwidth

Interpolation requires the voxels in the previous row while gradient interpolation requires the voxels in the two previous rows. Consequently, after the first section all subsequent sections require re-reading the bottom two rows of the previous voxel plane as depicted in Figure 7. If there are L sections, this means re-reading $2(L-1)N^2$ voxels per frame, and thus the total bandwidth overhead is $2(L-1)N^2/N^3 = 2(L-1)/N$. This is less than 5% of the total bandwidth if $L \leq 8$. For blocks with $b > 2$, tiling with miniblocks eliminates any excess overhead in re-reading the two voxel plane.

However, one consequence is that the SDRAM clock and rendering chip pipelines must run slightly faster to deliver the additional bandwidth. For $L = 8$, the SDRAM clock and rendering chip pipelines must run 5% faster, i.e. at 132MHz, or at 5% slower frame rate, i.e. 28frames/sec.

7.1.2 Pixel memory re-read

While processing a section, we only need on-chip storage for the compositing buffer proportional to the size N^2/L of the slice face area. All running pixels for rays emerging on a section face can be written to off-chip pixel memory as "interim" pixels.

However, interim pixels written to off-chip pixel memory for rays exiting a section face must be combined/composited with values for rays continuing into the adjoining section. We deal with this problem by reading interim pixels from off-chip pixel memory into the on-chip compositing buffer before processing the next section. There are up to N^2 interim pixels to read per section (the number is as few as 0 for rays parallel to a voxel row). The worst case can be handled by reading one beam of interim pixels from off-chip pixel memory per slice. In fact, the latency for reading these interim pixels can be hidden by the time to reload the additional two voxels per slice from voxel memory.

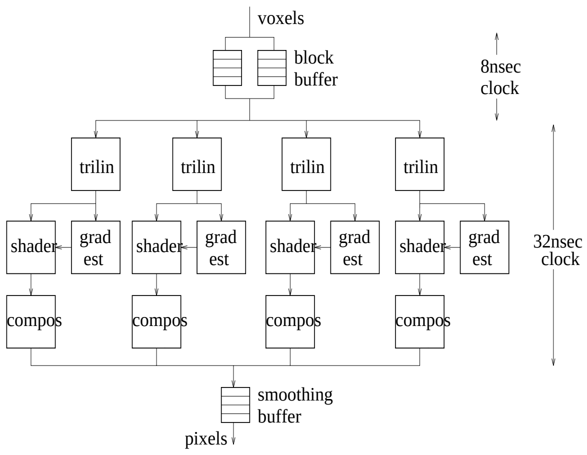

Figure 8: *Rendering chip pipelines*

intensity			8 bits
intensity	index		
intensity	index		
intensity			
intensity			
index			
intensity	index	grad.	
intensity	index		
grad. coeff	opacity		
rgb index	grad.	opacity	RGB table index
R	G		direct RGB
B	opacity/intensity		

Table 3: *Example voxel formats*

8 Rendering Chip Structure

Figure 5 shows the overall architecture. Each rendering chip has buffers and datapaths built-in for a nominal design such that 4 rendering chips, 4 SDRAMs, and 4 pixel memories achieve 28-30 frames/sec with $256^3 \times 16$ bit voxels. To reduce inter-chip communication cost, and hence the pin count, to manageable levels, we plan to use a block size of $b = 8$ hierarchically tiled with miniblocks. Each rendering chip processes 16bit voxels at 125MHz[6] and has slice buffers of size $256 \times 256 \times 16$bit/32 (4Kbytes). Currently we plan to have four pipelines on-chip, as shown in Figure 8, each 16 bits wide clocked at 32nsec. Larger voxels are treated as a sequence of 16 bit values with proportional reduction in frame rate.

9 Voxel Formats

Flexibility in voxel formats is important. Accordingly, the EM-Cube architecture allows the user to fashion the voxel format appropriately. Voxels are either 8 bits or a sequence of one or more 16 bit fields. We distinguish the format of voxels in memory (“memory voxels”) and the format of voxels in EM-Cube pipelines (“pipeline voxels”). In the simplest case, pipeline voxels are the same as memory voxels. In general, a pipeline voxel can be a simple transformation, e.g. a table lookup, on some or all fields of memory voxels. A memory voxel has the following *conceptual* components:

1. Intensity field: 8, 12, or 16 bits to indicate intensity or to index a RGB table.
2. Index field: 4, 8, (maybe 12), or 16 bits for color lookup and material type indicator.
3. Gradient coefficient: 8 bits (may increase later).
4. Opacity field: 8 bit value or index to opacity table.
5. Arbitrary user fields (size unrestricted as long as user pads overall voxel size out to a multiple of 16 bits).

Not all fields need be present; some fields may not exist and some may overlap with other fields. Table 3 shows examples of some of the voxel formats.

[6]Or slightly more due to sectioning overhead.

10 Scaling

It is important that EM-Cube scale to accommodate larger volumes and larger voxel sizes. Given C rendering chips each having the nominal design described in Section 8 and a volume dataset of N columns, M rows, S slices and $16v$ bits/voxel ($v = .5$, 1, 2, 4), we have the following constraints:

Memory capacity: $2vNMS/C \leq 8m$ Mbytes where there are m 64Mbit SDRAMs per rendering chip.

Frame rate: $\leq C/(2vNMS) * 250$M f/sec, determined by the rendering chip processing rate.[7]

Slice buffer: $2vNM/LC \leq 4096$ bytes

10.1 Voxel Scaling

The above constraints define the options if the voxel size v changes. For example, if v doubles and if $NMS = 256^3$ and $NM = 64$Kbytes, then we can half the volume size by halving N or M (halving S does not help because of the slice buffer constraint); or we can double the number of rendering chips C, SDRAMs, and pixel memories; or we can double the number of sections L, double the amount of voxel memory per rendering chip, and half the frame rate.

10.2 Volume Scaling

To handle a data set of size NMS larger than the nominal design of $N \times M \times S = 256 \times 256 \times 256$ supported in the four chip nominal design, we extend sectioning to three dimensions to divide the volume into smaller volumes. Thus we virtualize the voxel and pixel memories by paging them to the host memory system. As in Section 7.1, volume sections must overlap by two voxel planes requiring re-reading part of a section.

This 3D sectioning also allows us to handle reasonable volume sizes with just a single rendering chip, albeit with proportional reduction in performance.

[7]Frame rate degradation due to sectioning is ignored (typically only 5%, depending on L).

11 Other Issues

Several important issues such as supersampling, subvolumes, and perspective projections are unaddressed in this paper. We are investigating these issues as we refine our architecture. We anticipate that supersampling will be easy to work into the pipelines while subvolumes will be moderately more difficult.

12 Summary

We presented the outline of a feasible architecture for a low-cost, real-time volume rendering system suitable for PCI cards in PCs. Processing $256^3 \times 16$ bit voxels at 30frames/sec requires four sets of rendering chips and associated voxel and pixel memories.

A major innovation of the architecture is block-skewed memory. Blocking achieves maximum bandwidth from a small number of SDRAMs. While skewing eliminates memory access conflicts to provide view independence without duplicating voxel data, it increases inter-chip bandwidth. Blocking counteracts this problem, reducing the inter-chip bandwidth and thus the pin count. The block size b parameterizes the architecture. The larger b, the lower the communication overhead paid for skewing, and the more the data access pattern resembles that for an unskewed voxel memory.

A second key aspect of the architecture is sectioning. This reduces the on-chip storage requirements to achieve a feasible chip area for implementation.

Other features of the architecture are flexible voxel formats and scalability. As in Cube-4, one can always add more chips and memories for scalability. Alternatively, given a fixed amount of hardware, one can use sectioning in multiple dimensions to scale to larger volumes. We are investigating adding additional features such as supersampling, subvolumes, and perspective projection.

Architectural simulations of EM-Cube are underway. We plan to freeze the architecture in early summer and expect chips and a PCI reference board in the second half of 1998.

References

[1] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Workshop on Volume Visualization*, pages 91–98, 1994.

[2] B. Corrie and P. Mackerras. Parallel volume rendering and data coherence. In *Proc. Parallel Rendering Symposium*, pages 23–26, 1993.

[3] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture mapping hardware. Technical Report TR93-027, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, 1993.

[4] M. de Boer, A. Gropl, J. Hesser, and R. Manner. Latency- and hazard-free volume memory architecture for direct volume rendering. In *Proc. 11th Eurographics Hardware Workshop*, pages 109–118, 1996.

[5] K. Ma *et al.* A data distributed parallel algorithm for ray-traced volume rendering. In *Proc. Parallel Rendering Symposium*, pages 15–22. ACM Press, 1993.

[6] S. Gibson *et al.* Simulating arthroscopic knee surgery using volumetric object representations, real-time volume rendering and haptic feedback. In *First Joint Conference on Computer Vision, Virtual Reality, and Robotics in Medicine and Medical Robotics and Computer Assisted Surgery*, pages 369–378. Springer-Verlag, 1997.

[7] T. Guenther *et al.* VIRIM: A massively parallel processor for real-time volume visualization in medicine. In *Proc. 9th Eurographics Hardware Workshop*, pages 103–108, 1994.

[8] H. Fuchs and J. Poulton. Pixel-planes: A VLSI-oriented design for a graphics engine. *VLSI Design*, 2(3):20–28, 1981.

[9] G. Knittel. A scalable architecture for volume rendering. In *Proc. 9th Eurographics Hardware Workshop*, pages 58–69, 1994.

[10] G. Knittel and W. Strasser. A compact volume rendering accelerator. In *Proc. Volume Visualization Symposium*, pages 67–74. ACM Press, 1994.

[11] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. In *Proc. SIGGRAPH*, pages 451–457, 1994.

[12] J. Lichtermann. Design of a fast voxel processor for parallel volume visualization. In *Proc. 10th Eurographics Hardware Workshop*, pages 83–92, 1995.

[13] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. *Computer Graphics*, 26(2):231–240, July 1992.

[14] C. Montani, R. Perego, and R. Scopigno. Parallel volume visualization on a hypercube architecture. *Workshop on Volume Visualization*, pages 9–16, October 1992.

[15] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Proc. Parallel Rendering Symposium Proceedings*, pages 97–104, 1993.

[16] H. Pfister and A. Kaufman. Cube-4 – A scalable architecture for real-time volume rendering. In *ACM/IEEE Sympos. on Volume Visualization*, pages 47–54, 1996.

[17] J. Scheltinga, J. Smit, and M. Bosma. Design of an on-chip reflectance map. In *Proc. 10th Eurographics Hardware Workshop*, pages 51–55, 1995.

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