Hybrid Volume and Polygon Rendering with Cube Hardware

*Kevin Kreeger and Arie Kaufman**

Center for Visual Computing (CVC)
and Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

We present two methods which connect today's polygon graphics hardware accelerators to Cube-5 volume rendering hardware, the successor to Cube-4. The proposed methods allow mixing of both opaque and translucent polygons with volumes on PC class machines, while ensuring the correct compositing order of all objects. Both implementations connect the two hardware acceleration subsystems at the frame buffer. One shares a common DRAM buffer and one run-length encodes images of thin slabs of polygonal data and then combines them in the Cube composite buffer. In both realizations, we take advantage of the predictable ordered access to frame buffer storage that is utilized by Cube-5 and the rest of the family of volume rendering accelerators based on the Cube design.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism;

Keywords: Mixing polygons and volumes, Volume rendering, Ray casting, Run-length-encoding, Cube architecture

Figure 1: ***A flight simulation scene mixing a texture-mapped polygonal terrain, an opaque plane (with 4420 polygons), a translucent cockpit, and a volumetric cloud (also in color plate).***

* {kkreeger,ari}@cs.sunysb.edu

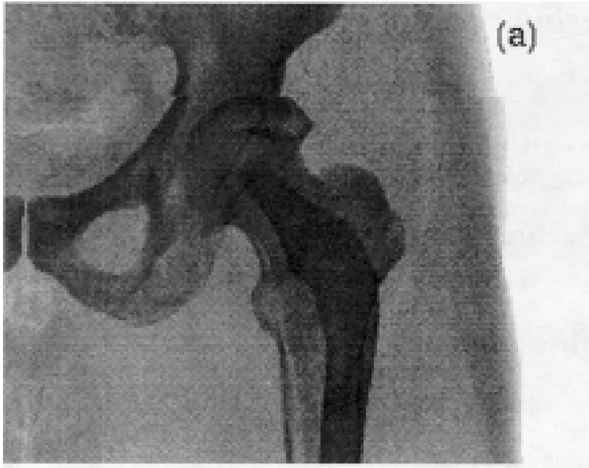
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
1999 Eurographics LosAngeles CA USA
Copyright ACM 1999 1-58113-170-4/99/08...$5.00

1 Introduction

Direct volume rendering accelerators will become commercially available this year, from Mitsubishi Electric in mid 1999 as a low-cost plug-in PCI card called VolumePro [10], and from Japan Radio Co. in late 1999 as the special purpose U-Cube ultrasound visualization system. Since these systems are based on Cube-4 [11] we refer to them as the Cube family of volume rendering accelerators. Unfortunately, these accelerators work independently from current geometry rendering hardware. Therefore, it is impossible to render mixtures of volumetric and polygonal data, unless none of the objects intersect. Unfortunately, many applications require objects to intersect. For example, in a flight simulation scene, a polygonal plane may fly through a volumetric cloud over a textured polygonal terrain viewed from within a translucent cockpit, as shown in Figure 1. In this paper, we begin expanding the Cube design to accelerate rendering of more than just volumetric data, creating the fifth generation of Cube architectures, the Cube-5.

We propose two methods to connect current geometry pipelines to the Cube family of volume rendering accelerators on PC class machines. Both allow mixing of opaque and/or translucent polygons with volumetric data. Translucent polygons complicate the situation because all fragments (both translucent polygon fragments and volume samples) must be drawn in topologically depth sorted order. This is required because compositing with the *over* operator [12] is not commutative. Interesting visual effects can be created with translucent polygons as shown in the translucent tank spinning in the desert kicking up a volumetric cloud of dust in Figure 2. Virtual environments, such as surgical simulation, require real-time or

Figure 2: ***Volumetric dust cloud kicked up by a spinning translucent tank (with 5082 Polygons) in a desert (also in color plate).***



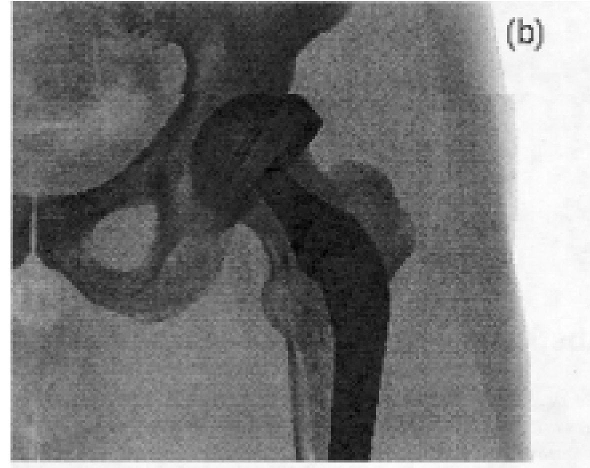


Figure 3: ***A prosthesis (containing 3758 polygons) being fit to a 256^3 CT scan of a hip: (a) translucent polygons reveal the bony structure behind the prosthesis, (b) opaque polygons obscure incorrect alignment (also in color plate).***

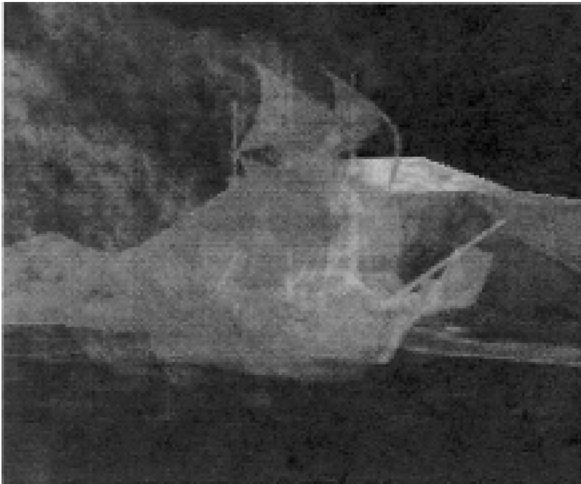


Figure 4: ***A translucent ghost ship (containing 4715 polygons) sailing out of a volumetric fog bank in front of an opaque texture-mapped island (also in color plate).***

at least interactive feedback. Figure 3 shows a fitting of a polygonal model of a prosthesis to a CT scan of a patient. The use of translucent polygons enables the surgeon to see the details of the volumetric bone behind the polygonal object. Finally, Figure 4 shows a translucent ghost ship sailing out of a volumetric fog bank.

Software systems, developed for production applications, allow mixing of volumetric and polygonally modeled objects. Ray tracing is commonly utilized by these systems to render mixtures of volumes and polygons [7, 15]. The flexibility of these systems comes at the price of an extremely slow frame generation rate. Therefore, these systems would not perform well for applications requiring interactivity.

Volume Graphics GmbH produces a product which mixes opaque polygons with volumes by ray casting. They achieve interactivity by utilizing reduced resolution rendering while parameters are being changed followed by slower high quality rendering whenever the viewing parameters become constant [13]. No hardware acceleration is utilized, so the performance is limited, and no translucent polygons can be used.

The OpenGL extensions proposed in the HP *Voxelator* [8] attempted to create a standard software interface to mix volumetric data with polygonal objects. However, no mention of hardware acceleration was given and only opaque objects were discussed, neglecting translucent polygons and volumes.

Three dimensional texture map based volume rendering [2] allows the mixing of volumes and polygons. Unfortunately, 3D texture map based volume rendering is only available on high end graphics workstations and is neither scalable nor capable of real shading. Additionally, these systems require very expensive equipment to achieve even interactive frame rates. Our solution, on the other hand, provides better frame rates, real Phong shading, and an order of magnitude lower cost in addition to being designed for a PC class machine.

All of the Cube family (Cube-4/Cube-5/VolumePro/U-Cube) of volume rendering accelerators utilize slice-order ray casting, an object order technique (e.g., [6, 16]). This means that input data is processed in a regular predefined order. This exploitation of memory coherence is critical to realizing the real-time 30Hz frame rates of the Cube family. We also take advantage of the repeatable access pattern in both of our designs, specifically to the frame buffer.

In Section 2, we describe our method to mix translucent polygons with volumes in volume object order. In Section 3, we discuss the differences between a DRAM frame buffer on current PC graphics cards and the SRAM composite buffer in previous Cube designs. Sections 4 and 5 present our two methods to create a hybrid volume and polygon rendering system using the Cube-5 architecture and relatively low cost (compared to high end workstations) PC graphics pipelines. Section 6 presents results and a performance comparison.

2 Mixing Polygons into Volume Rendering Systems

Volume rendering is the more difficult of the two rendering modalities. Also, the only methods to achieve interactive or real-time frame rates for volume rendering utilize object order. Therefore, in our method presented here, we adapt polygon rendering to slice order volume ray casting (used in the Cube architectures), and organize the overall rendering process on a volume slice-by-slice basis rather than a polygon-by-polygon or pixel-by-pixel basis.

In the Cube family of accelerators, the slices of the volume are processed in depth order. Thus, to correctly order translucent poly-

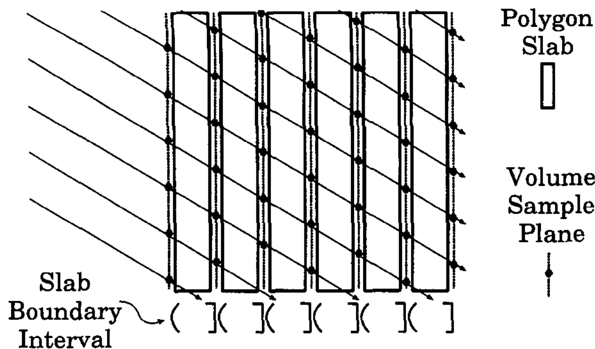


Figure 5: *Side view of dove-tailing translucent polygons and volume data. (The gaps between polygon slabs are shown for clarity. In reality, there is no gap or overlap as shown by the boundary intervals.)*

gon fragments with volume samples, thin slabs of polygons are rendered and composited in between slices of the volume, as shown in Figure 5. The polygonal slabs represent all of the translucent objects which lay between two consecutive slices of the volume data. The boundaries are created such that the union of all the slabs neither miss nor duplicate anything, such as less-than the current slice and greater-than-or-equal-to the next slice (see slab boundaries in Figure 5). The data from the volume slices and translucent polygonal slabs are dove-tailed together in an alternating fashion. In this way the correct depth ordering of all contributing entities is preserved, and use of the over operator to composite them creates correct colors in the final image pixels.

The Cube architectures take advantage of the storage order of voxels by processing slices orthogonal to one of the three major axis. The image produced by this method is aligned with the face of the volume most perpendicular to the view direction (called the baseplane). The 2D baseplane image is then warped onto the final image plane. Since volume slices are always orthogonal to one of the axes, the polygons should also align this way. Therefore, special handling of the projection and viewing matrices are used. While the following methods are extendible to perspective projection, we show examples for parallel projection, for simplicity and since that is what VolumePro currently supports.

The opaque polygons should be rendered such that, after projection through the volume dataset, warping creates the correct footprint on the final image. Also, the Z-depth values should be aligned along the processing axis, so that the volume slice index can be used for the Z-depth check. First, the object space is transformed by a permutation matrix so that the Z-component is the largest value in the view vector (i.e., the major viewing direction is along the Z-axis). The permutation is created by swapping the elements of the view vector, leaving the relative sizes unchanged. Then, the eye-point is moved to a position along the permuted Z-axis by rotating the vector between the look-at-point and the eye-point by some angle we call α around the X-axis and β around the Y-axis. Notice that α and β are always between -45 and 45 degrees, otherwise we would choose a different baseplane. We then apply an "X and Y according to Z" shear (also known as a Z-slice shear along X and Y [3]) to the viewing matrix as follows:

$$\begin{bmatrix} 1 & 0 & \tan\alpha & 0 \\ 0 & 1 & \tan\beta & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This can be seen in Figure 6. With this geometry, when the opaque polygons are drawn, the polygon footprints are "pre-warped" so

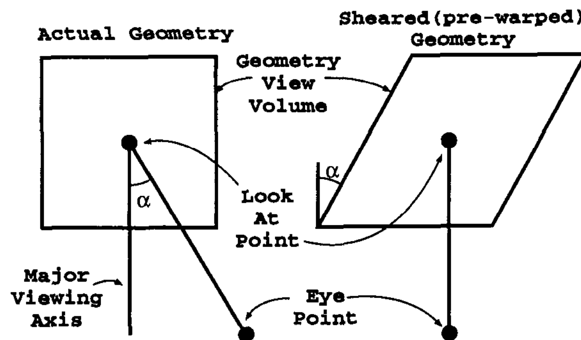


Figure 6: *Top view of creating sheared viewing geometry so that polygon footprints are "pre-warped" and Z-depths represent distance along the volume processing direction.*

that the warping operation at the end of Cube rendering creates correct polygons in the final image. Also, the Z-depths computed are proportional to the distances along the processing axis. It is sometimes possible (if all opaque geometry fits within the volume extents) to set the hither and yon clipping planes to the edges of the volume and, if the precision of the depth buffer is the same, the depths computed are exactly the volume slice indices for depth checking. Otherwise, a simple scaling must be applied when they are utilized by the volume rendering system. Light positions must be considered when using this method, as the shearing may not move the lights correctly.

The thin slices of translucent polygons should align geometrically with their 3D position in space. We begin by aligning the eye-point as before. Then, to keep the objects from projecting all the way to the final image plane, we translate the geometry so that the center of the current thin slab is at the $Z = 0$ plane before shearing. Clipping planes allow only the current thin slab to be rendered. The projection plane is set to be within the two volume slices which border that region with glOrtho (glFrustum for Perspective). In a recent paper [5], we presented an algorithm to accelerate the OpenGL rendering of the thin slices of translucent polygons.

3 Frame Buffers versus Composite Buffers

It is important to understand the organization of frame buffer design compared to composite buffer design. The previous Cube volume rendering accelerators utilize a tightly coupled on-chip SRAM buffer to hold the partially composited rays as a volume is processed in slice order (see Figure 7), called the composite buffer. Cube exploits the regular processing sequence inherent in slice order rendering. Specifically, each slice is processed in the same order as the previous, left-most voxel to right-most voxel of each row, and bottom-most row to top-most row of each slice (possibly with some skewing). In this way the SRAM composite buffer becomes a simple FIFO queue of length equal to the size of a slice. The SRAM queue is 32 or 48 bits wide to hold 8-bit or 12-bit fixed point RGBα values (called coxels for composite-buffer element). Each pipeline reads a coxel from the front of the queue and writes a coxel to the rear of the queue for each clock cycle. With this approach, each Cube pipeline can process 1 sample per clock, or over 500 million samples per second fill rate with 4 pipelines at 133MHz (current VolumePro configuration), sufficient for real-time volume rendering of 256^3 datasets.

Common PC class geometry pipelines, on the other hand, utilize an external DRAM frame buffer, where the RGBα color values and

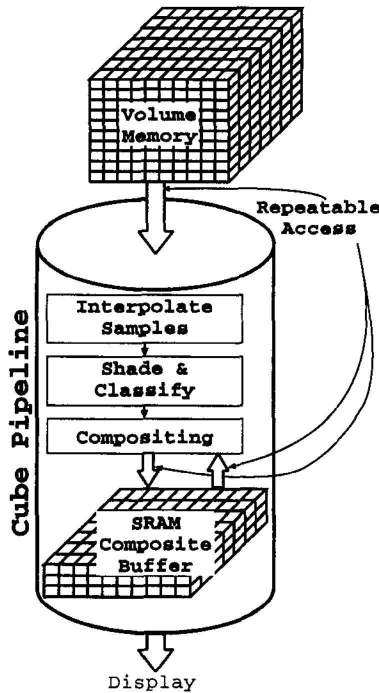


Figure 7: *Previous Cube pipeline showing the on-chip SRAM buffer used to store partially composited rays.*

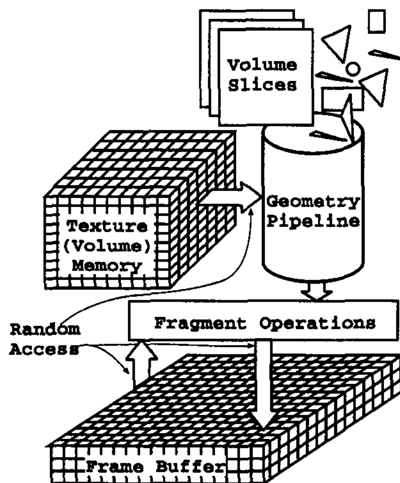


Figure 8: *Graphics accelerator solution, where volume slices compete with polygons for limited resources at the texture memory and frame buffer memory interfaces.*

Z-depth values for each pixel are stored (see Figure 8). This buffer must support random access since polygon rendering does not enjoy the regular access ordering inherent in slice-order volume rendering. Normal polygon rendering produces triangles on the screen averaging between 10 and 50 pixels. Therefore, the DRAM memory is organized to maximize access to areas of the screen of this size. For example, the Digital Neon chip achieves a maximum fill rate of 100 million fragments per second without blending [9], by interleaving pixels across parallel memory interfaces and chunking the frame buffer into tiles the size of a DRAM page. If the entire chunk is not utilized, burst mode access will also not be fully

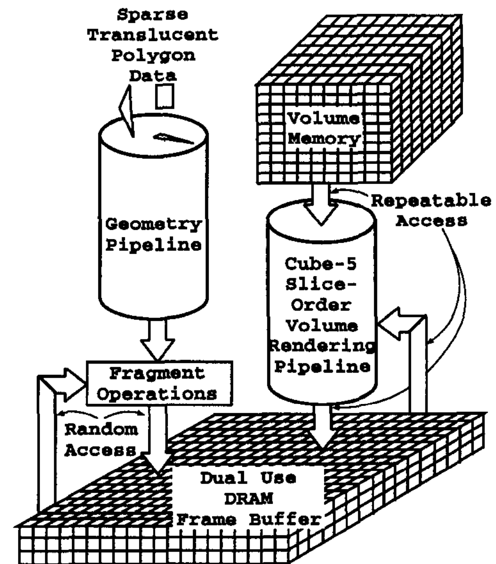


Figure 9: *Dual use DRAM frame buffer connecting a commodity surface graphics pipeline with a Cube-5 volume rendering pipeline.*

utilized, resulting in decreased bandwidth due to lack of latency hiding.

When the 3D texture mapping solution for volume rendering is implemented on geometry pipelines, volume slices perpendicular to the screen are texture mapped through the volume. The per-vertex geometry calculations for the slices are easily achievable with any level graphics hardware. However, the requirement to support random access to both the texture memory and frame buffer limits the performance of this approach to the fill rate achievable with a current DRAM frame buffer not optimized for repeatable access patterns as occur in slice order volume rendering.

Very high end surface graphics systems utilize massive parallelism in the fragment processing section of the polygon pipeline. This, coupled with a highly distributed frame buffer, allow increased fill rate performance. For example, an Infinite Reality graphics engine with 4 raster manager boards can place 710 million 16-bit textured, depth buffered fragments per second into the frame buffer. Yet, with only one board (a common configuration since it is the most expensive part), the fill rate quickly drops to 177 million fragments per second. In our tests we were only able to achieve up to 90 million fragments per second fill rate, below the published numbers, due to the blending required for volume rendering.

4 Mixing with a Dual Use DRAM Frame Buffer

We aim to create a low-cost system which is capable of rendering mixtures of polygons and volumes. Therefore, we propose to remove the SRAM composite buffer from inside the Cube-5 pipeline and replace it with an external DRAM frame buffer. The frame buffer is also accessible from a 3D graphics pipeline to allow mixing of polygonal data with volumes. Instead of a typical DRAM buffer such as in polygon engines, we organize the memory in our buffer so that it is optimized for volume rendering. Due to the higher performance requirements of volume rendering, the polygon performance will be equal to or better than current polygon DRAM frame buffers, but will require increased VLSI. Figure 9 shows how the dual use frame buffer connects the two pipelines. Only the frame buffer storage is currently shared. To minimize the

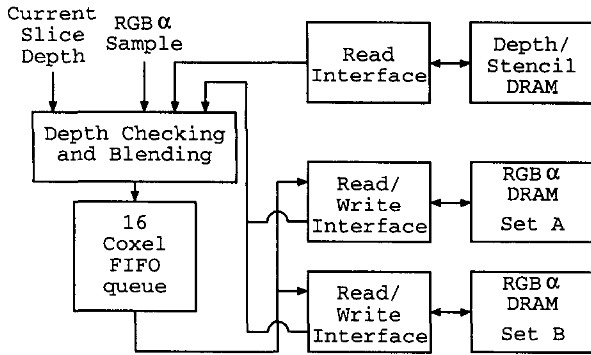


Figure 10: *Memory interfaces for each Cube-5 pipeline including coxel FIFO queue to align burst mode access and two RGBα DRAM sets to allow concurrent reading/writing.*

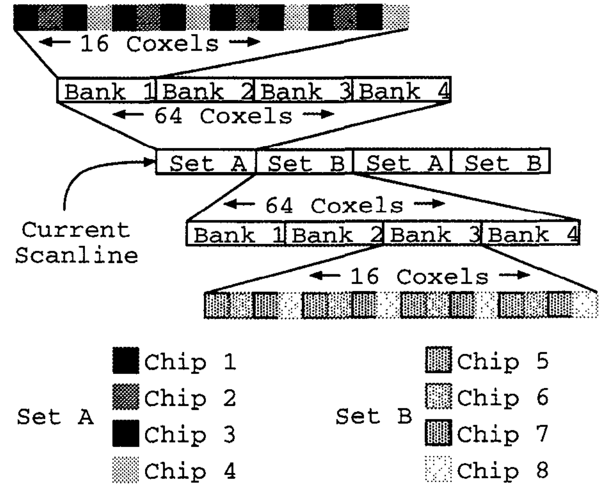


Figure 11: *RGBα coxel layout on 8 DRAM chips (also in color plate).*

impact on current hardware, all polygon fragment operations are still performed in the polygon pipeline, while all volume sample compositing occur in the volume pipeline.

To render a scene with both opaque and translucent polygons and also volume data, the geometry pipeline first renders all opaque polygons with Z-depth. The volume slices and thin slabs of translucent polygons are then rendered in an alternating fashion – volume slices by the Cube-5 pipeline and translucent polygons by the graphics pipeline (opaque polygons could also be handled with the same dovetailing algorithm, but with increased demand on the graphics pipeline). Z-depth checking is utilized to insure correct hidden object removal and blending is set in both pipelines to correctly composite the samples and fragments. Finally, the geometry engine performs the baseplane warp onto the image plane required by Cube. At any given point in time, either the geometry sub-system or the Cube-5 sub-system is stalled while the other is rendering to the common frame buffer.

The design of the DRAM buffer is critical to achieve the 503 million samples per second required for 30Hz rendering of 256^3 volume datasets. Therefore, we first look at creating a DRAM buffer just for the Cube-5 pipeline by itself, then look at connecting it to a graphics pipeline. Cube based volume rendering designs consist of multiple pipelines, such as the one in Figure 7. In each pipeline, at every clock cycle, a coxel (composite-buffer element consisting of RGBα) is read from the SRAM composite buffer FIFO, blended with an appropriate compositing equation and then the new coxel is placed at the rear of the FIFO. We change the structure of a coxel to contain 64 bits: 32 bits of color, 8 for each RGBα, and 32 bits of Z-depth information, 24 + 8-bit stencil. This is required to handle Z-depth checking in the compositing stage. If we assume that opaque polygon rendering is completed before any volume rendering begins, the 32 bits of Z-depth/stencil information is read, but not re-written. Therefore, for every clock cycle, each Cube pipeline needs to read 8 bytes of coxel data and write back 4 bytes.

We would like to utilize commodity DRAM chips to keep the price affordable to the PC market. SDRAM provides information synchronized to the pipeline clock and provides burst mode access to obtain the maximum bandwidth possible if the memory can be organized correctly. Commonly available chips today typically utilize 4 internal banks which must be accessed in succession with bursts of at least 8 words per burst to be able to saturate the bandwidth between the chip and the memory controller.

We propose to utilize memory chips with a word size of 16 bits. Therefore, four words must be read by each pipeline on each cycle and two words must be written. This means we would need six 16-bit memory interfaces per pipeline. An emerging technology in SDRAM chips is that of double data rate (DDR) which reads/writes data at both the rising and falling edges of the clock. Using DDR SDRAMs we can utilize two 16-bit memory interfaces for reading 64 bits per clock and one 16-bit memory interface for writing 32 bits per clock for a total of three 16-bit memory interfaces per pipeline.

Since we must read and write every clock cycle to keep the pipeline full, we read from one set of frame buffer chips and write to another. We keep two sets of chips, A and B. We could read from set A and write to set B for a complete slice of the volume, and then switch for the next slice. However, this way, each set would have to be large enough to hold the complete frame buffer, and the polygon engine would have to be told which set was current. Therefore, we alternate reading and writing between sets A and B within a slice and buffer the processed coxels from the read set until it becomes the write set. Since every memory access must be a burst, each one really lasts 4 clock cycles and reads/writes 4 coxels (8 words) with 16-bit DDR DRAM chips. We need to cycle through all 4 banks to keep the memory bandwidth saturated before writing the new RBGα values back. For this reason there is a 16 coxel FIFO queue (4 coxels for each of 4 banks) that the newly composited RBGα portions of the coxels are stored in, as shown in Figure 10.

There are many different possible configurations for the number of pipelines in a Cube system. We present an example for a case of 4 parallel pipelines creating 12 total memory interfaces. Each pipeline contains one read interface to the Z-depth/stencil portion of the frame buffer and two read/write interfaces to sets A and B of the RGBα portion of the frame buffer. To render a 256^3 volume at 30Hz, each of the 4 pipelines process 125 million voxels per second. Therefore, we utilize a 133MHz clock for the chip and the SDRAM. The mapping of the frame buffer pixels onto the memory chips is critical to performance. It must match exactly the processing order of the Cube pipelines and the parallel access by 4 pipelines at once. We assume the skewed memory access of the Cube architecture is "un-skewed" (as in the VolumePro implementation) so that the volume samples are in order from left to right across each scanline in groups of 4 since it is easier to follow in the explanations. The design can be extended to skewed memory, although the geometry pipeline and screen refresh system must be aware of the additional skewing.

Figure 11 shows the layout of the RGBα portion of the coxels in the frame buffer. For a given scanline there is a group of pixels which reside in set A followed by a group of pixels which reside in set B, repeated across the entire scanline. The length of each set is 64 pixels due to the fact that each set must contain pixels which are

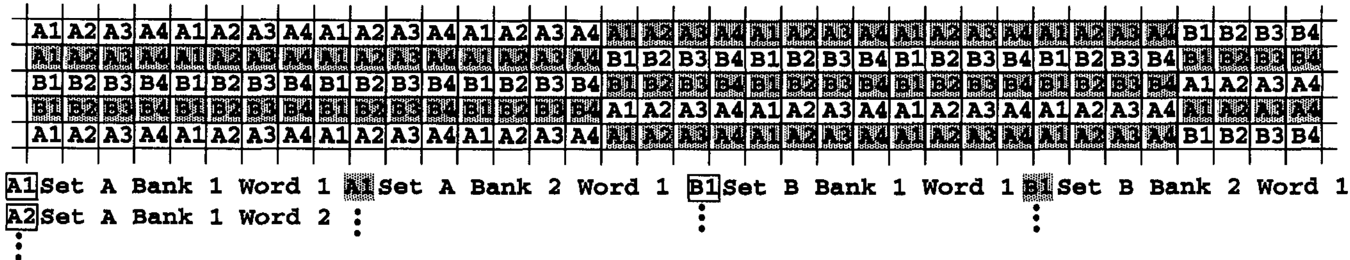


Figure 12: *Pixel layout in the frame buffer. Clear pixels are in bank 1 shaded pixels are in bank 2. Set A and Set B shown with 4 parallel chips per set. In reality there are 4 interleaved banks shifted across successive scanlines, but only 2 are shown so that it fits across the page.*

read from 4 different banks inside each chip, each bank consisting of 4 RGBα values from 4 parallel chips/pipelines. Thus, the pixel data in the frame buffer is interleaved across 8 chips, but in fine detail, it is really interleaved across only 4 chips.

This provides us with an interface which reads

$$4 \text{ pipelines} \times (1 \text{ RGB}\alpha \text{ chip} + 1 \text{ depth chip}) \times 16 \text{ bits} \\ \times 133\text{MHz} \times 2 \text{ data rate} = 34\text{Gbits} = 4.15\text{Gbytes}$$

of data per second. This surpasses the required

$$256^3 \times 30Hz \times 8 \text{ bytes} = 3.75 \text{ GBytes per second}$$

where 8 bytes are 4 bytes RGBα + 4 bytes Z-depth/stencil. Additionally, the frame buffer sub-system is capable of writing

$$4 \text{ pipelines} \times 1 \text{ RGB}\alpha \text{ chip} \times 16 \text{ bits} \times 133\text{MHz} \\ \times 2 \text{ data rate} = 17\text{Gbits} = 2.1\text{Gbytes}$$

once again handling the

$$256^3 \times 30Hz \times 4 \text{ bytes} = 1.8 \text{ GBytes per second}$$

required for real time 30Hz rendering of 256^3 volumes.

This extra bandwidth is not sitting idle. The screen must be refreshed from the data in the frame buffer. If we assume a 1280x1024 screen resolution with 60Hz refresh rate and that all 4 RGBα bytes are read from the frame buffer (since our burst mode access retrieves them anyway), then

$$1280 \times 1024 \times 60\text{Hz} \times 4 \text{ bytes} = 300\text{Mbytes}$$

are read from the frame buffer per second. Only the RGBα portion of the frame buffer is required for refresh. Therefore, the refresh data is read from 8 chips. To read the 300MB per second for screen refresh, it is sufficient to perform 10 data burst reads/writes (depending on set A or B) to each chip followed by 1 read of data for refresh. This distribution of memory accesses provides the refresh hardware with a consistent (although bursty) stream of data. The 10-1 ratio also provides enough bandwidth to the volume rendering pipelines to still allow 30Hz rendering of 256^3 datasets. Cube pipelines based on 133 MHz clocks, like the current VolumePro configuration, also contain the same percentage of excess cycles.

The dual use DRAM frame buffer, built out of 12 SDRAM chips, must also work for polygon rendering without affecting performance, or it would not be a desirable solution. The Neon chip [9] reports that they can achieve 45% of the maximum memory bandwidth from their memory sub-system. The amount of usable bandwidth depends upon the pattern of the interleaving of pixels to memory controllers. Our frame buffer organization utilizes a one dimensional interleaving. While this method is optimal for reading the pixels for volume rendering and screen refresh, McCormack et al. [9] show that if an entire column of the screen is mapped to one memory chip, poor load balancing can result in scenes such as architectural walkthroughs where polygons are all aligned vertically.

McCormack et al. go on to propose a 1D interleaved method that is shifted from one scanline to the next. We can do this in our buffer without affecting volume rendering performance. In fact, we set up the shifting from one scanline to the next so that the banks form a checkerboard pattern similar to Neon to further increase memory performance (see Figure 12) by allowing spatially coherent memory access to different banks so that latency is better hidden. Our system, with the two sets of memory chips allows additional separation and possibility to hide latency between memory accesses at the set boundaries. Unfortunately, it affects the hardware VLSI costs by increasing the number of memory controllers, but this is required to achieve the bandwidth for volume rendering. We predict that we should be able to achieve a similar percentage performance as Neon. Since we have a higher base bandwidth, we should be able to achieve even higher fill rate performance. Even using the 45% estimate from the Neon paper, we achieve 357 million 64-bit pixels per second fill rate from our 6.35GByte combined bandwidth to all 12 DDR 133MHz 16-bit SDRAM chips. Of course the performance of a particular memory layout to polygon fill rates depends upon the rasterization order of the pipeline. Neon utilizes a square chunking fragment generation ordering. For a different fragment scheme, a different pixel assignment may be more optimal, however, it must also consider the volume rendering requirements we discussed earlier.

Since our frame buffer is spread across 12 memory interfaces, we need to hook up only one 64Mbit SDRAM to each interface and have 96MBytes of frame buffer storage. This is enough storage to allocate a double buffered, 2500^2 pixel frame buffer with 8 bytes per pixel.

5 Mixing into the SRAM Composite Buffer

The first frame buffer we presented utilized commodity components and, to be realized, required minimal alterations to current hardware. Yet, it created a bottleneck at the frame buffer where the two sub-systems competed for the same resources. For comparison, we present an alternative approach to connecting a graphics pipeline to a volume rendering pipeline that keeps both working at all times and merges the data in the SRAM composite buffer inside the Cube chip. At any given time, the volume pipeline is compositing the current volume slice with the previous thin slab of polygon data over the composite buffer, and the graphics pipeline is rendering the next thin slab of translucent polygons.

We still utilize the dovetailing approach of volume slices and thin slabs of translucent polygonal data, described in Section 2. We first project all opaque polygons onto a Z-buffer coincident with the baseplane (e.g., the volume face most parallel to the screen). Secondly, the projected RGBαZ image is loaded into the composite

buffer of the volume rendering pipeline. Subsequently, the volume is rendered with a Z-comparison enabled in the compositing stage. The thin slabs of translucent polygons are rendered by the geometry pipeline, and their RGBα data is sent to the volume pipeline to be blended into the SRAM composite buffer within the volume pipeline.

We modify the compositing stage of the volume rendering accelerator to composite two layers (one volume and one translucent polygon) per step, thus not delaying the volume rendering process. This requires the addition of some extra logic. The straightforward formula for performing a double composition of a volume sample v over a translucent pixel fragment p over the old coxel c would require 4 additions and 4 multiplies in 5 stages:

$$C_c' = C_v\alpha_v + [C_p\alpha_p + C_c(1-\alpha_p)](1-\alpha_v)$$

However, simple math allows the double composition to be calculated with 4 additions and 2 multiplies in 6 stages with the following formula (some of the calculations are re-used):

$$C_c' = (C_c + (C_p - C_c)\alpha_p) + [C_v - (C_c + (C_p - C_c)\alpha_p)]\alpha_v$$

The hardware designer would choose the option more desirable for a given implementation: less logic and more stages, or fewer stages and more logic.

It seems simple enough to render a thin slab of translucent polygons in the geometry pipeline and then transfer this "slab image" to the volume rendering pipeline to be composited. However, consider the amount of data transfered for a 256^3 volume. There are 255 slabs plus one buffer in front of the volume and one behind. Each of these 257 slabs contains 256KB (256^2 pixels of RGBα) of data. This equates to 64MB to be read from the polygon frame buffer and transferred between the two sub-systems each frame. To achieve 30Hz would require a bandwidth of 1.9GB per second. While this much data could be transferred with wide enough channels, it must also be read from the frame buffer. Without changing the organization of the current DRAM polygon frame buffers, it is impossible to read this much data. Additionally, the frame buffer must be cleared 257 times per frame.

To solve this bandwidth challenge we propose to run-length-encode (RLE) the blank pixels. Each scanline in the polygon frame buffer is encoded separately, and a "run-of-zeros" is encoded as four 0's (RGBα) followed by the length of the run. We notice that the translucent polygon slabs are very sparse, since typically only a small percentage of the polygons in a scene are translucent. For example, out of our four test sequences, only an average of 91 pixels contain color information out of 64K pixels per "slab image". Run-length-encoding just the blank pixels in these thin slabs results in over 99% reduction in the required bandwidth. Lacroute and Levoy [6] utilized RLE to take advantage of sparse volume data on a slice-by-slice basis. They gained a rendering frame rate advantage by only processing the visible voxels. Here, we utilize RLE on 2D images of sparse translucent polygons to save on bandwidth.

This method requires hardware in the volume rendering pipeline that can decode the RLE input stream and create RGBα fragments. However, since these fragments are utilized by the volume pipeline in a regular order, it is simple to decode the input stream [1] using a double buffer to synchronize the two pipelines. Every clock cycle a value is output from the decoding hardware. If the volume rendering machine has multiple pipelines (as most current designs do) we replicate the decoding hardware for each pipeline, so that they can keep up with pixel demand.

Likewise, RLE hardware at the originating end connected to the geometry pipeline could encode the data in real-time before sending it to the volume pipeline. However, we would still need 1.9 GB per second access to the frame buffer to read all the thin slabs of translucent polygons and the 257 clears. Therefore, we implement

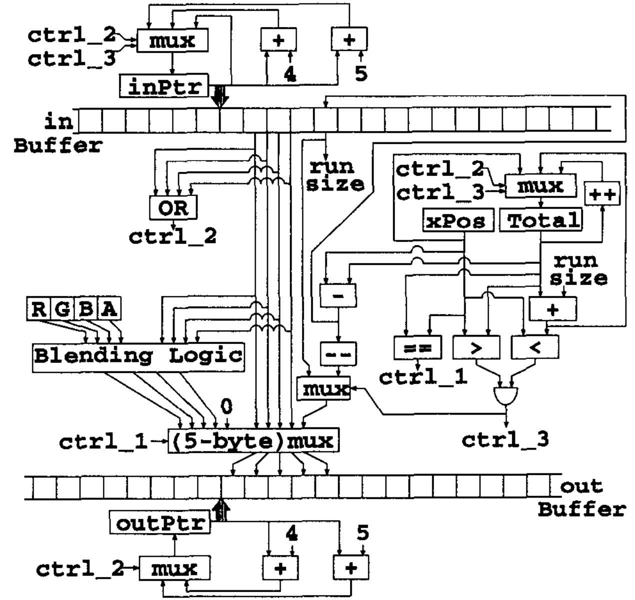


Figure 13: *An embedded DRAM chip implementation of run-length encoding frame buffer hardware. Every clock cycle a flit (either a run-of-zeros or a pixel) is copied from the in buffer to the out buffer for the scanline.*

a separate frame buffer which stores the data directly in RLE format. Since the thin slabs of translucent data are very sparse, more time is spent clearing and reading than rasterizing. An RLE buffer, while not efficient for rasterization, is better suited for both clearing and reading the data. For example, to clear an RLE frame buffer requires merely storing a single run of zeros (in 5 bytes) for each scanline instead of writing an entire 256^2 frame buffer.

To minimize the impact on the current geometry pipelines we propose implementing the RLE frame buffer using the emerging technology of embedded DRAM [14] and connecting it parallel to the normal frame buffer. Previous encoding algorithms assumed that the data was given in physical order. Triangle rasterization, however, does not guarantee any ordering of the fragments. Therefore, we must be able to randomly insert an RGBα value into an RLE scanline of data.

Figure 13 shows a diagram of our RLE insert. For each fragment, the encoded scanline is copied from one buffer to another, inserting the new RGBα value. Every clock cycle, a single flit (either an RGBα pixel, or run-of-zeros) is processed. The entire scanline is processed flit by flit. In Figure 13, "in Buffer" is the current encoded scanline and "out Buffer" is the newly encoded scanline with the new RGBα fragment inserted. The choice of what to insert at each cycle is performed by the 5 byte multiplexor in the center of the diagram. Pointers to the current flit of both the in ("inPtr") and out ("outPtr") buffers are located at the top and bottom. The right side calculates how much has been processed ("total") and two of the control points. The other mux control point is calculated by 'or'-ing together all of the RGBα values (the flag for run-of-zeros). "xPos" is the x position of the fragment. We implemented a lookup table of the current buffer's location in memory for each y value. Thus, the buffer can be moved while inserting new pixels and the table is simply updated. This is seen in the RLE_AddFragment routine in Algorithm 1. The RLE_AddPixelToScanline function demonstrates the processing that occurs in the hardware of Figure 13.

By utilizing an embedded DRAM we take advantage of the extremely high bandwidth available when processing occurs on the

```
RLE_AddFragment(xPos, yPos, RGBA) {
  tmp = nextFreeScanline();
  RLE_AddPixelToScanline(data[yPos],xPos,RGBA,tmp);
  freeScanLine(data[yPos]);
  data[yPos] = tmp;
}

RLE_AddPixelToScanline(in, xPos, RGBA, out) {
  total = 0;
  inPtr = 0;
  outPtr = 0;
  while(total < lineWidth) {
    if(total == xPos) {
      out[outPtr::outPtr+3]=Blend(RGBA,in[inPtr::inPtr+3]);
      outPtr += 4;
      total++;
      if(in[inPtr::inPtr+3] == 0)
        in[inPtr+4]--;
      else
        inPtr += 4;
    }

    out[outPtr::outPtr+3] = in[inPtr::inPtr+3];
    if(in[inPtr::inPtr+3] == 0) {
      if(total < xPos && total+in[inPtr+4] > xPos) {
        out[outPtr+4] = xPos-total-1;
        outPtr +=5;
        in[inPtr+4] -= xPos-total;
        total = xPos;
      } else {
        out[outPtr+4] = in[inPtr+4];
        total += in[inPtr+4];
        outPtr += 5;
        inPtr  += 5;
      }
    } else {
      total++;
      outPtr += 4;
      inPtr  += 4;
    } // endif run-of-zeroes
  } // endwhile still within scanline
}
```

Algorithm 1: *Pseudo-code showing processing occurring in RLE hardware.*

memory chip [4]. The processing is simple enough to be implemented in the DRAM manufacturing process (one of the drawbacks to eDRAM so far is that logic gates are not easily placed on DRAM manufacturing/testing process). For a 1280x1024 frame buffer, the maximum amount of memory required is 50Mbits. This fits onto eDRAM dies with room for over 3 million gates for the encoding hardware [14]. We estimate that our RLE frame buffer runs at a target clock rate of at least 200MHz. At 30Hz frame rate with 256 slices of volume data, that would equate to 27,300 cycles (or flits accessed) per slice.

Using a 200MHz clock and the flit count per slab, we calculate how long it takes to render a frame as follows

$$T = \sum_{s=0}^{256} MAX\left(\frac{flitCount_s}{200MHz}, 130\mu sec\right)$$

since a volume rendering pipeline spends $130\mu sec$ on each slice for a 256^3 volume at 30Hz. An advantage of the encoding algorithm is that the frame rate slips only slightly when the flit processing count for a thin slab exceeds its allotted amount.

The graph in Figure 14 represents the number of flits processed for each thin slab of translucent polygons for one frame from each sequence. We notice that the flit count is normally well below the 27,300 cycles that it takes for the volume pipeline to render one

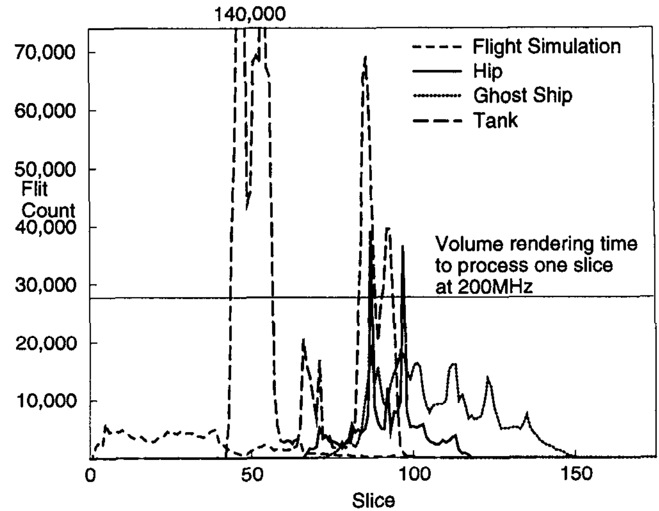


Figure 14: *Number of flits processed per thin slab for one frame from each test sequence. Assuming a 256^3 rendering geometry, objects are placed within the 256 slices of the volume data.*

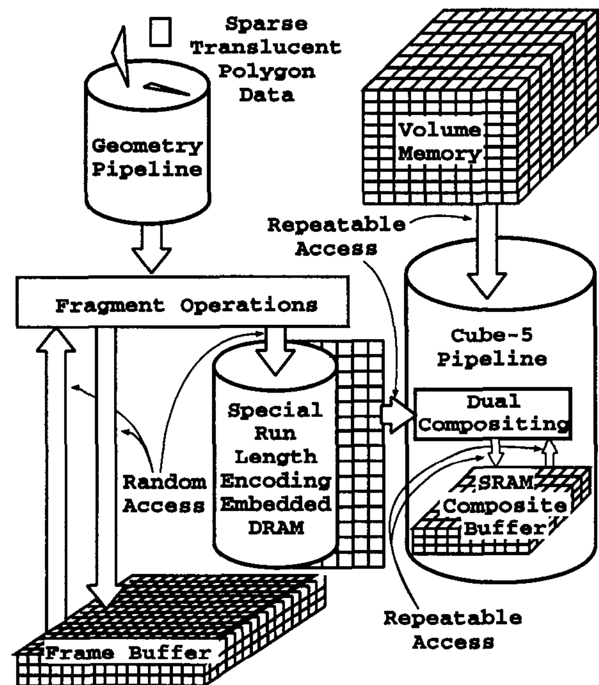


Figure 15: *RLE frame buffer connecting a geometry pipeline to the SRAM composite buffer in the Cube-5 pipeline.*

slice of data. The few times it exceeds this is when there are numerous polygons in a single slab, oriented parallel to the volume slices. For example, in the slabs which contain the sides of the tank, the flit count grows enormously. Similarly, in the hip, the alloted flit count is exceeded for the back and front face of the long spike which extends down the femur.

Figure 15 shows how a polygon pipeline and Cube-5 pipeline are connected through the RLE frame buffer, which is double-buffered to allow rendering during transmission of data. The auxiliary frame buffer is connected at the same place as the existing one by simply

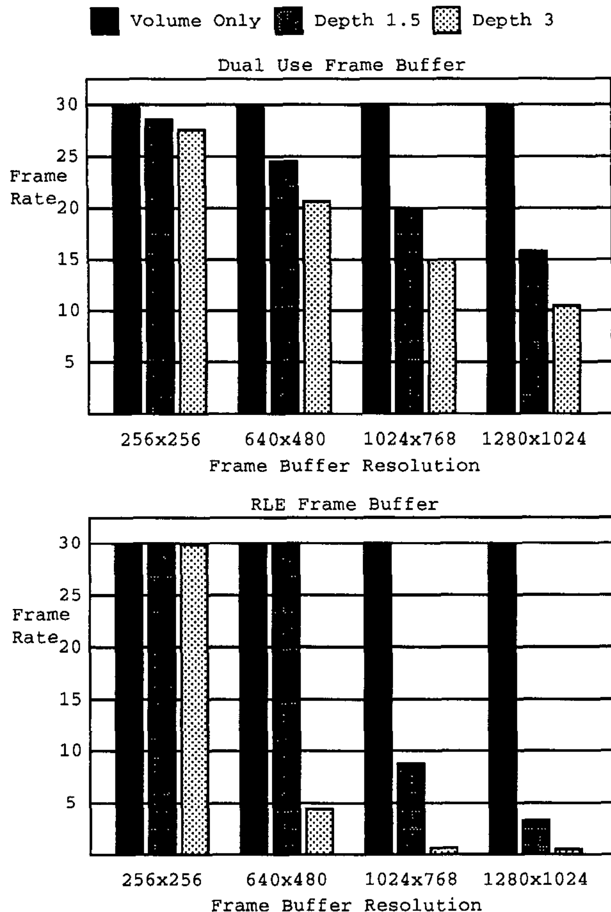


Figure 16: *Frame rates achievable for the two methods when rendering a* 256^3 *volume with a Cube-5 architecture (with 4 pipelines at 133MHz), for mixing with polygons at different depth complexities.*

duplicating the fragments, thus not affecting the remainder of the geometry pipeline. The volume pipeline also double buffers to allow receiving of data while blending the previous slab. Note how volume rendering does not conflict with polygon rendering. Since the volume pipeline always accesses its memory in a repeatable ordered fashion, it achieves the sample fill rate into the frame buffer at a sufficient speed to achieve 30Hz volume rendering. We utilize the graphics pipeline to render the opaque polygons before rendering the volume. This can normally be accomplished concurrently with the rendering of the volume for the previous frame. Even if the polygon engine must render translucent polygons mixed in with the volume, there is usually enough time to render the opaque polygons before the volume finishes due to the small number of translucent polygons in normal scenes.

This design represents a typical implementation; while the actual hardware may change some details, the efficiency of using an RLE frame buffer for the sparse translucent polygons in each thin slab can be analyzed. Run-length encoding for translucent polygons has a great impact on the amount of data transferred between the pipelines and read from the frame buffer. For example, non-RLE rendering of our 256^2 images with 256 volume slices requires 67MB of data to be transferred between the two pipelines per frame. RLE of the thin slabs of translucent polygon data, on the other hand, reduces this below 900KB for the ghost ship, 550KB for the tank, 470KB for the hip, and 420KB for the flight simulation.

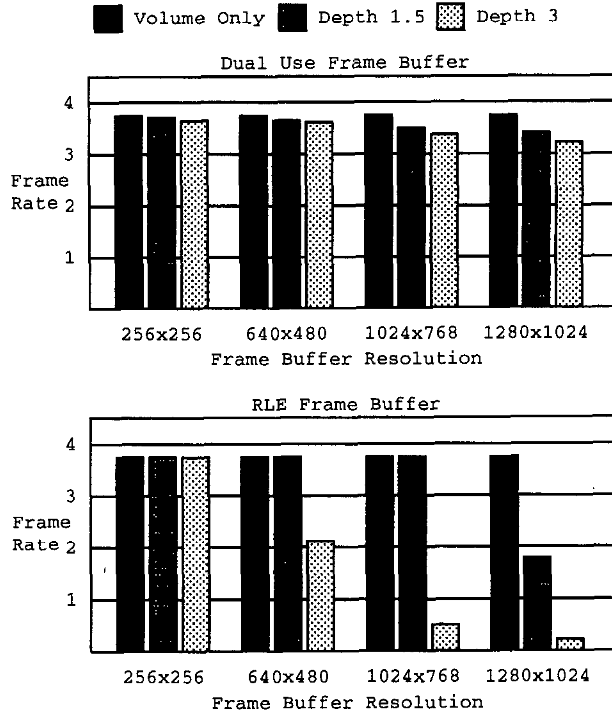


Figure 17: *Frame rates achievable for the two methods when rendering a* 512^3 *volume with a Cube-5 architecture (with 4 pipelines at 133MHz), for mixing with polygons at different depth complexities.*

6 Performance

We have simulated both the DRAM frame buffer and the RLE frame buffer in C++. Our simulation provides shaded volume samples as if they came through the Cube-5 pipeline. We implemented a triangle rasterization into a software simulation of our RLE frame buffer. Since we do not have an exact model of a 3D graphics card, we estimate a maximum pixel fill rate of 180 million pixels per second and up to 6 million triangles per second (e.g., current high-end PC graphics cards such as the RIVA TNT from *n*VIDIA). We believe that the real bandwidth to the dual use DRAM frame buffer is 357 million pixels per second, as shown in Section 4, but this number is a conservative estimate. Usually, the percentage of translucent polygons is small and thus triangle count is not a problem. With these assumptions, we analyzed the performance of both methods.

Figure 16 shows the frame rates achievable when rendering a 256^3 volume. Various frame buffer resolutions are shown, from 256×256 (size of the volume face) up to 1280×1024. When rendering the volume only, we always achieve 30Hz on both systems. The other two bars represent mixing with polygon rendering. We reference the amount of polygon rendering by the per pixel average depth complexity (number of objects in front of each other). For example, rendering 18,432 50-pixel triangles to a 640x480 screen, draws an average of 3 fragments per pixel for a depth complexity of 3. We can see in the 256×256 size that the RLE frame buffer retains the 30Hz rendering rate while the dual use DRAM buffer slowed down slightly. This is because of the contention for the shared frame buffer. This also occurs for the 640×480 resolution with low depth complexity. However, everywhere else the dual use DRAM buffer performs better than the RLE frame buffer. This is because of the inefficiency of the rasterization into the RLE buffer. For high pixel fill cases, the RLE frame buffer degrades very quickly while the dual use DRAM frame buffer degrades more

gracefully. Additionally, when the RLE buffer does perform better, it is only by a small amount.

Similarly, Figure 17 shows the frame rates achievable when rendering a 512^3 volume. The volume only frame rates drop to under 4Hz due to the performance of 4 pipeline Cube-5 system. However, since there is more time to render polygons per volume slice, the RLE frame buffer out performs the dual use DRAM buffer for higher pixel fill rates than the 256^3 case. Once again though, the dual use DRAM buffer degrades more gracefully than the RLE frame buffer. For Cube-5 systems that are capable of 30Hz 512^3 or larger volume rendering (e.g., with more than 4 pipelines), as long as the frame buffer size is scaled accordingly, the performance appears more like Figure 16 than Figure 17.

Sequences of 90 frames for each of our 4 test scenes from Figures 1, 2, 3, and 4 were generated. We measured an average depth complexity of approximately 1.5 when rendering to a 256×256 frame buffer with a 256^3 volume. The frame rates from our simulations match the analysis, with one exception. In the tank sequence, when all of the polygons along the side of the tank fell within one thin slab, the polygon rendering time was much longer and lowered the RLE frame rate to 27Hz instead of the achievable 30Hz.

7 Concluding Remarks

We showed a method of using a shared frame buffer for mixing volume and polygon rendering which required minimal changes to either pipeline. Unfortunately, this creates a bottleneck where the two sub-systems compete for the shared resource. Therefore, we also devised a method without such a bottleneck by transmitting data from the polygon pipeline to the volume pipeline. We propose an RLE solution to the bandwidth explosion, but it only works for very sparse polygon datasets. The RLE insert procedure could be more optimized for the rasterization. Instead of inserting one fragment at a time, it could insert a whole scanline of the current primitive. However, this would require radical changes to the 3D graphics pipeline.

Our analysis of the two systems show that the shared frame buffer performs better than transmitting data between the two pipelines for almost every case. Only when there are very few translucent polygons does the RLE frame buffer keep up with the volume rendering pipeline. In this case, we lose little time in frame buffer contention and the dual use DRAM frame buffer performs insignificantly worse than the RLE buffer. Since the shared frame buffer solution is so much simpler and represents fewer changes to current hardware, we believe that this method is the better one to implement.

The two solutions presented apply to the PC class of machines. They are not only is an order of magnitude cheaper than high-end graphics systems with 3D texture mapping, but provide higher frame rates and full Phong shading of the volume samples.

We believe that volume rendering is a more difficult task that polygon rendering (even with texture mapping). Therefore, to merge the two systems, it makes more sense to identify the similar parts of the pipelines and create a merged system designed around the requirements and current features of the volume rendering pipeline. So far, we have shown that for the frame buffer, a DRAM buffer capable of keeping up with the volume rendering fill rate is more than sufficient for polygon rendering. In the future Cube-5 design work, we plan to investigate other areas where the two pipelines can be merged (e.g., compositing and fragment blending operations are obvious candidates, but texture mapping and volume sampling could possibly also be merged). Hopefully a single pipeline can then accelerate rendering of both continuous polygonal and discrete volumetric data. The possibility is for multiple pipelines working in parallel to provide a scalable solution to universal rendering.

Acknowledgments

This work was supported by the National Science Foundation under grant MIP9527694 and Office of Naval Research under grant N000149710402.

References

[1] J. Barenholtz and C. Dewhurst. A Run Length Encoding Scheme for Real Time Video Animation. In *Proceedings of Northwest '76*, pages 63–69, Seattle, Washington, June 1976.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Symposium on Volume Visualization*, pages 91–98, Oct. 1994.

[3] B. Chen and A. Kaufman. 3D Volume Rotation Using Shear Transformation. Technical Report TR10.24.98, submitted for publication, State University of New York at Stony Brook, Oct. 1998.

[4] D. Elliott, W. Snelgrove, and M. Stumm. Computational Ram: A Memory-SIMD Hybrid and its Application to DSP. In *Custom Integrated Circuits Conference*, pages 30.6.1–30.6.4, May 1992.

[5] K. Kreeger and A. Kaufman. Mixing Translucent Polygons with Volumes. Technical Report TR.99.03.31, State University of New York at Stony Brook, Computer Science Department, Stony Brook, NY 11794-4400, Mar. 1999. Submitted for publication.

[6] P. Lacroute and M. Levoy. Fast Volume Rendering using a Shear-warp Factorization ot the Viewing Transform. In *Computer Graphics, SIGGRAPH 94*, pages 451–457, July 1994.

[7] M. Levoy. A Hybrid Ray Tracer for Rendering Polygon and Volume Data. *IEEE Computer Graphics & Applications*, 10(2):33–40, Mar. 1990.

[8] B. Lichtenbelt. Design of A High Performance Volume Visualization System. In *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 111–120, Aug. 1997.

[9] J. McCormack, R. McNamara, C. Gianos, N. Jouppi, T. Dutton, J. Zurawski, L. Seiler, and K. Correll. Implementing Neon: A 256-Bit Graphics Accelerator. *IEEE Micro*, 19(2), Mar. 1999.

[10] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro Real-Time Ray-Casting System. In *To Appear in Proceedings of SIGGRAPH 1999*, Aug. 1999.

[11] H. Pfister and A. Kaufman. Cube-4 - A Scalable Architecture for Real-Time Volume Visualization. In *Symposium on Volume Visualization*, pages 47–54, Oct. 1996.

[12] T. Porter and T. Duff. Compositing Digital Images. In *Computer Graphics, SIGGRAPH 84*, pages 253–259, July 1984.

[13] C. Reinhart. Welcome to Volume Graphics, May 1999. http://www.volumegraphics.com/index.html.

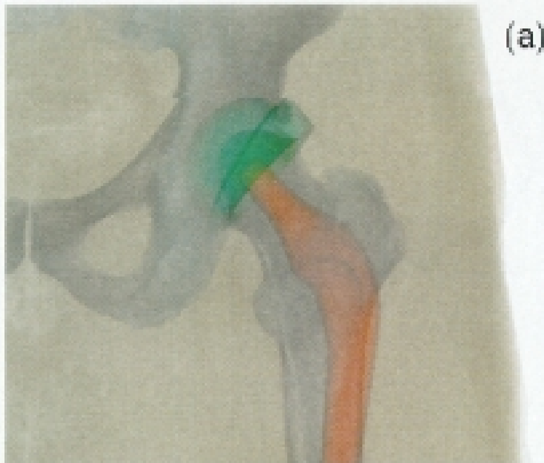
[14] Siemens Semiconductors. Embedded DRAM: Innovations that fit. http://www.siemens.de/semiconductor/edram, 1998.

[15] L. Sobierajski and A. Kaufman. Volumetric ray tracing. In *Symposium on Volume Visualization*, pages 11–19, Oct. 1994.

[16] L. Westover. Footprint Evaluation for Volume Rendering. In *Computer Graphics, SIGGRAPH 90*, pages 367–376, July 1990.

Figure 1: ***A flight simulation scene mixing a texture-mapped polygonal terrain, an opaque plane (with 4420 polygons), a translucent cockpit, and a volumetric cloud.***

Figure 2: ***Volumetric dust cloud kicked up by a spinning translucent tank (with 5082 Polygons) in a desert.***



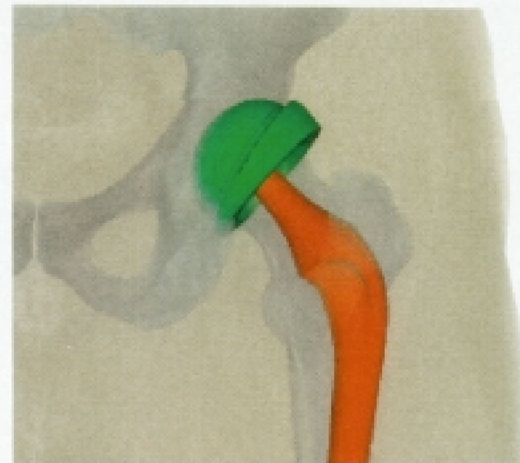


Figure 3: ***A prosthesis (containing 3758 polygons) being fit to a 256^3 CT scan of a hip: (a) translucent polygons reveal the bony structure behind the prosthesis, (b) opaque polygons obscure incorrect alignment.***

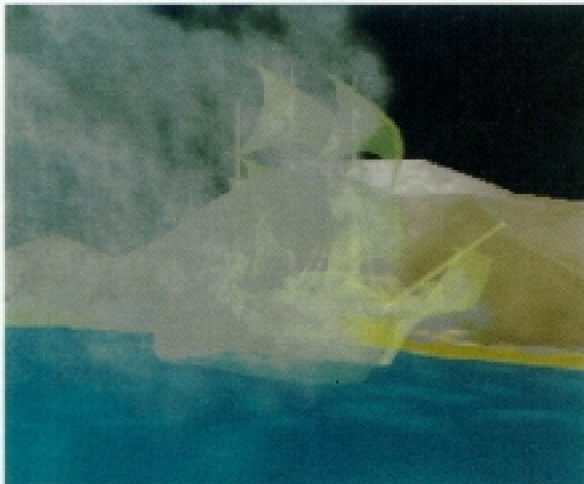


Figure 4: ***A translucent ghost ship (containing 4715 polygons) sailing out of a volumetric fog bank in front of an opaque texture-mapped island.***

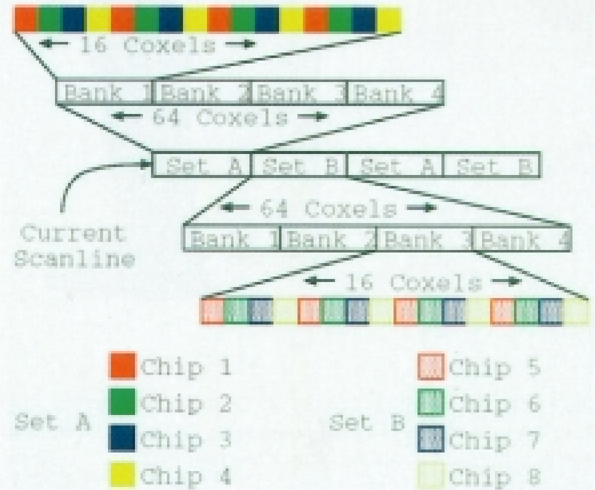


Figure 11: *RGBα coxel layout on 8 DRAM chips.*

Hybrid Volume and Polygon Rendering with Cube Hardware

Kevin Kreeger, Arie Kaufman