The VolumePro Real-Time Ray-Casting System

Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, Larry Seiler

TR99-19   April 1999

Abstract

This paper describes VolumePro, the worldś first single-chip real-time volume rendering system for PC class computers. VolumePro implements object-space ray-casting with parallel slice-by-slice processing. Our discussion of the architecture focuses on the rendering pipeline, the memory organization, and sectioning, which is a data partitioning scheme to reduce on-chip buffer requirements. We describe several advanced features of VolumePro, such as gradient magnitude modulation of opacity and illumination, supersampling, supervolumes, and crop and cut planes. The system renders more than 500 million interpolated, Phong illuminated, composited samples per second. This is sufficient to render a 256 x 256 x 256 volume at 30 frames per second.

SIGGRAPH 9́9

The VolumePro Real-Time Ray-Casting System

Hanspeter Pfister Jan Hardenbergh Jim Knittel
Hugh Lauer Larry Seiler

TR-99-19 April 1999

Abstract

This paper describes VolumePro, the world's first single-chip real-time volume rendering system for PC class computers. VolumePro implements object-space ray-casting with parallel slice-by-slice processing. Our discussion of the architecture focuses on the rendering pipeline, the memory organization, and sectioning, which is a data partitioning scheme to reduce on-chip buffer requirements. We describe several advanced features of VolumePro, such as gradient magnitude modulation of opacity and illumination, supersampling, supervolumes, and crop and cut planes. The system renders more than 500 million interpolated, Phong illuminated, composited samples per second. This is sufficient to render a 256^3 volume at 30 frames per second.

Submitted to SIGGRAPH '99

1. First printing, TR99-19, January 1999
2. Accepted to SIGGRAPH ’99, April 1999

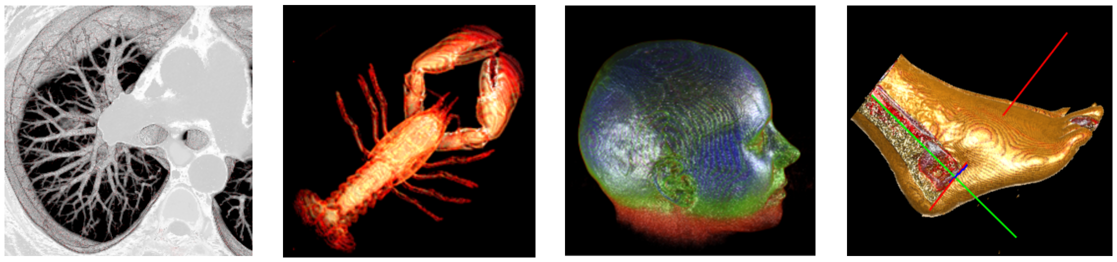


Figure 1: Several volumes rendered at 30 frames per second on VolumePro.

1 Introduction

Over the last decade, direct volume rendering has become an invaluable visualization technique for a wide variety of applications. Examples include visualization of 3D sampled medical data (CT, MRI), seismic data from oil and gas exploration, or computed finite element models. While volume rendering is a very popular visualization technique, the lack of interactive frame-rates has limited its widespread use. To render one frame typically takes several seconds due to the tremendous storage and processing requirements.

Highly optimized software techniques for volume rendering try to address this problem by using pre-processing to compute, for example, object illumination or regions of the volume that can be skipped during rendering [6, 8, 14]. However, pre-processing prohibits immediate visual feedback during parameter changes. Each time rendering parameters such as voxel transparency or illumination are changed, the lengthy pre-processing must be repeated before the new image is displayed. Furthermore, the pre-computed values typically increase the storage requirements by a factor of three to four.

Another approach to render volume data uses texture mapping hardware, a common feature of modern 3D polygon graphics accelerators. It can be used for volume rendering by applying a method called planar texture resampling [1, 4]. The volume is stored in 3D texture memory and resampled during rendering by extracting textured planes parallel to the image plane. However, current texture hardware does not support gradient estimation and per-sample illumination for real-time shading. In addition, although recent PC graphics cards support 3D texture mapping, interactive performance still requires expensive high-end graphics workstations.

To overcome these limitations, we have developed VolumePro, a special-purpose hardware system for real-time volume rendering on PC class computers. VolumePro produces 500 million interpolated, Phong illuminated, composited samples per second. That is sufficient to render datasets with up to 16 million voxels (e.g., 256^3) at 30 frames per second. A VolumePro system consists of the VolumePro PCI card, a companion 3D graphics card, and software. The VolumePro PCI card contains up to 128 MB of volume memory and the VoxelBlaster vg500 rendering chip. The VoxelBlaster Interface (VLI) is a collection of C++ objects and provides the application programming interface to the VolumePro features.

VolumePro is based on the Cube-4 volume rendering architecture developed by SUNY at Stony Brook [10]. It draws liberally from Cube-4 and also from another published system, Enhanced Memory Cube-4 (EM-Cube) [9], but it makes major enhancements to both. This paper is the first detailed description of the VolumePro architecture and system, including a complete description of features and rendering results.

The next section describes the ray-casting algorithm and the slice-by-slice processing order that are at the heart of VolumePro. In Section 3 we take an in-depth look at the VolumePro rendering pipeline. We describe novel features such as per-sample modulation of illumination by the gradient magnitude. Section 4 describes several of the advanced features of VolumePro. In Section 5 we discuss the architecture of the vg500 rendering chip, which contains four rendering pipelines and uses a novel high-bandwidth memory organization for volume data. Section 6 gives an overview of the VolumePro PCI card and Section 7 briefly discusses the main classes of the VLIapplication interface.

2 Rendering Algorithm

VolumePro implements ray-casting [7], one of the most commonly used image-order volume rendering algorithms. The current version of VolumePro supports parallel projections of rectilinear isotropic and anisotropic volume data sets. Ray-casting offers high image quality and is easy to parallelize.

To achieve uniform data access and to get a one-to-one mapping of ray-samples onto voxels we use a ray-casting technique with object-order data traversal similar to shear-warp rendering [6]. Instead of casting rays in image space, rays are sent into the dataset from each pixel on the base plane, which is the face of the volume data that is most parallel to the viewing plane. The resulting base plane image is then projected onto the image plane using a 3D transformation and a 2D image re-sampling operation. VolumePro uses 2D texture mapping on a companion graphics card for this final image warp (see Section 6).

The main advantage of the shear-warp algorithm is that voxels can be read and processed in planes of voxels, called slices, that are parallel to the base plane (see Figure 2). Within a slice, voxels are read from

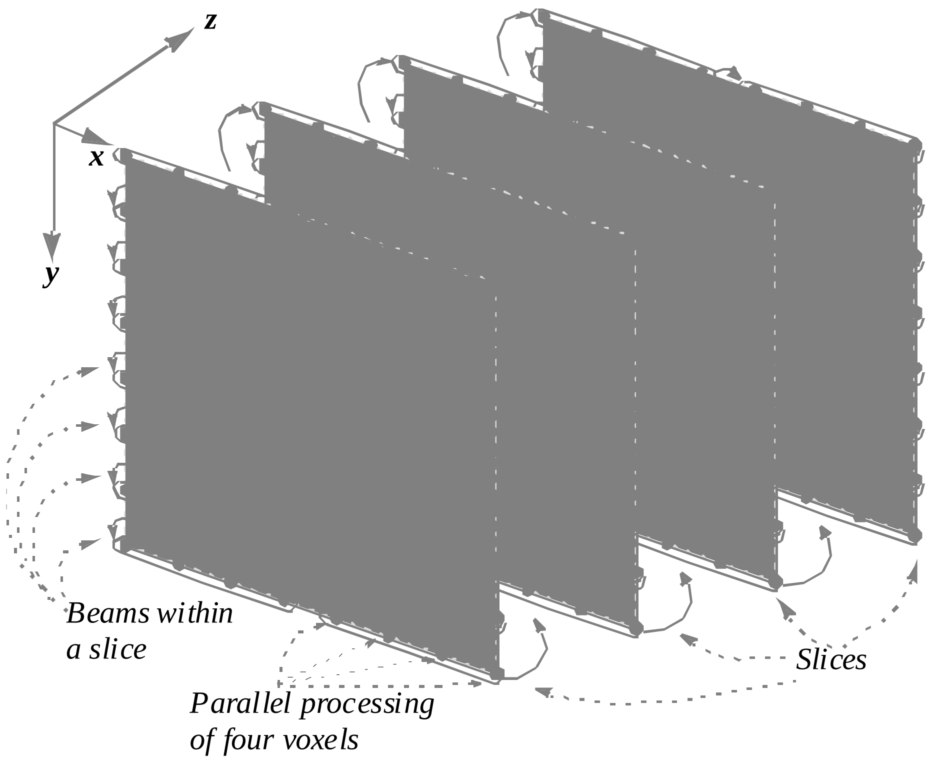


Figure 2: *Slice-by-slice processing.*

memory a scanline of voxels (called a voxel beam) at a time, in top to bottom order. This leads to regular, object-order data access. In addition, it allows parallelism by having multiple rendering pipelines work on several voxels in a voxel scanline at the same time. This concept is explained further in Section 5.

The next section describes the pipelined approach to ray casting in VolumePro. Our explanation deliberately and temporarily omits some important aspects of the VolumePro architecture. For example, we describe only one pipeline, even though VolumePro uses four parallel rendering pipelines. This is rectified later in Section 5 when the vg500 chip architecture is discussed.

3 The Ray-Casting Pipeline

A characteristic of VolumePro is that each voxel is read from memory exactly once per frame. Therefore, voxel values must be recirculated through the processing stages of the VolumePro pipeline so that they become available for calculations precisely when needed. This recirculation is at the heart of the VolumePro architecture and is described in this section.

A second characteristic of VolumePro is that the reading and processing of voxel data and calculation of pixel values on rays are highly pipelined. One VolumePro processing pipeline can accept a new voxel every cycle, and can forward intermediate results to subsequent pipeline stages every cycle, without any pipeline delays. The net effect is that VolumePro can render a volume data set at the speed of reading voxels.

Figure 3 illustrates a flow diagram of an idealized version of the VolumePro processing pipeline. In this

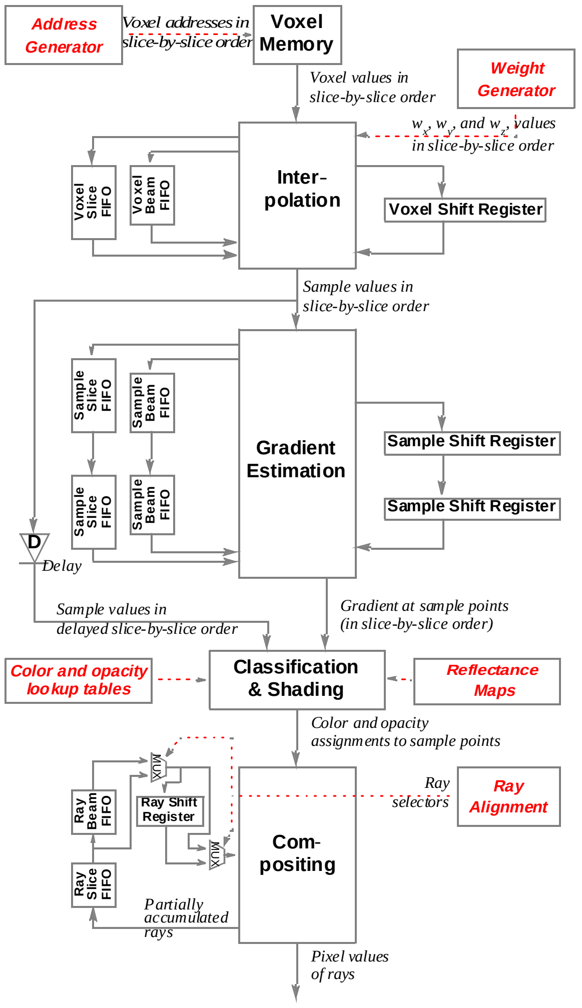
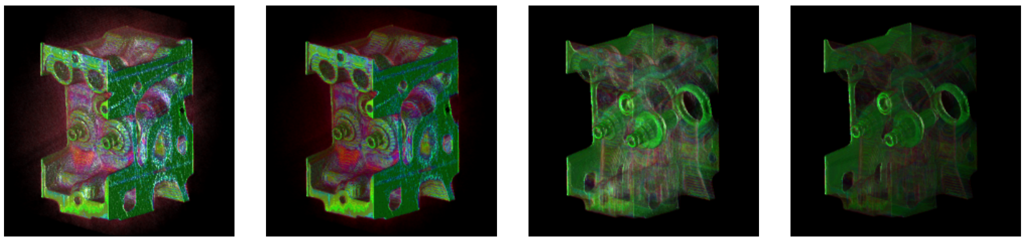


Figure 3: *Conceptual ray-casting pipeline.*

figure, the flow and processing of voxel data is shown in black, and control information is shown in red. The figure shows interpolation followed by gradient estimation. In actual practice, estimation of z-gradients happens before interpolation (see Section 3.2). This would complicate the flow diagram and so has been omitted from this figure.



(a) [0, 0, 0] (b) [0, 0, 1] (c) [1, 0, 0] (d) [1, 1, 1]

Figure 4: Images rendered with different values of [Gmom, $Gmim_d$, $Gmim_s$]. Notice how the appearance of surfaces changes.

3.1 Interpolating Voxel Values

At the top of Figure 3 is Volume Memory, also called Voxel Memory. The current generation of VolumePro supports 8- and 12-bit scalar voxels. All internal voxel datapaths of the vg500 chip are 12-bits wide. Voxels are read from the memory and are presented to the Interpolation unit in the slice-by-slice, beam-by-beam order described in Section 2, one voxel per cycle. The purpose of the Interpolation unit is to convert the stream of voxel values into a stream of sample values, also in slice-by-slice, beam-by-beam order, at the rate of one sample value per cycle.

Each sample value is derived from its eight nearest neighboring voxels by trilinear interpolation. Therefore, the Interpolation Unit must have on hand not only the current voxel from the input stream, but also all other voxels from the same tri-linear neighborhood. It does this by storing input voxels and other intermediate values in the Voxel Slice FIFO, Voxel Beam FIFO, and Voxel Shift Register. Each of these is a data storage element sized so that when a newly calculated value is inserted, it emerges again at precisely the time needed for a subsequent calculation.

Trilinear interpolation also requires a set of weights wx, wy, and wz. These are generated by the Weight Generator in Figure 3. This generator is based on a digital differential analyzer (DDA) algorithm [3] to calculate each new set of weights from the previous weights. Since all rays are parallel to each other and have exactly the same spacing as the pixel positions on the base plane (and hence, as the voxels of each slice), a single set of weights is sufficient for all of the samples of a slice. The Weight Generator, therefore, computes a new wx, wy, and wz prior to the beginning of each slice, based on the angle of the rays to the base plane and the initial offset between rays and voxels on the first slice. The interpolated sample values are rounded to 12-bit precision.

3.2 Gradient Estimation

In our idealized model, the output of the Interpolation unit is a stream of sample values, one per cycle in slice-by-slice, beam-by-beam order. This stream is presented to the Gradient Estimation unit for estimating the x-, y-, and z-components of the gradient on each sample using a central difference gradient filter [5].

In principal, gradients can be estimated either of two ways with equal results:

- Interpolate voxels to produce samples, then take central differences or some other (linear) convolution to estimate gradients on those samples; or
- Estimate gradients on voxels, then interpolate those gradients to obtain the gradients at the sample points.

VolumePro divides the estimation of gradients. The z-components of gradients (simply called the z-gradient in the rest of this document) are estimated from voxel values and interpolated to obtain z-gradients at sample points. The x- and y-gradients are estimated from interpolated sample points themselves.

The gradient at a particular sample point depends not only on data in the current slice but also on data in the previous and next slice. The next slice, of course, has not been read when the current sample value arrives at the input. Therefore, the Gradient Estimation unit operates one voxel, one beam, and one slice behind the Interpolation unit. That is, when sample $S_{(x+1),(y+1),(z+1)}$ arrives, the Gradient Estimation Unit finally has enough information to complete the estimation of the gradient at sample point $S_{x,y,z}$.

It does this by maintaining two full slices of buffering, plus two full beams of buffering, plus two extra samples. These are stored in the Sample Slice FIFOs, Sample Beam FIFOs, and Sample Shift Registers, respectively, which serve roughly the same functions as the Voxel Slice FIFO, Voxel Beam FIFO, and Voxel Shift Register. Each central difference gradient component has signed 8-bit precision, meaning it can represent numbers in the range $[-127 \cdots + 127]$.

3.3 Gradient Magnitude Approximation

The magnitude of the gradient can be used as an indication of a surface [7]. It is used in VolumePro to optionally modulate the opacity and illumination of a sample (see Section 3.6). VolumePro derives a sample's gradient magnitude in two phases. First, the square of the gradient magnitude is computed by taking the sum of the squares of the gradient components. Then a Newton-Raphson iteration [11] is used to compute the square-root of this value resulting in an 8-bit approximation of the gradient magnitude in the range of $[0 \cdots 220]$.

3.4 Gradient Magnitude Modulation

In VolumePro, the opacity and illumination values of samples can be optionally multiplied by the gradient magnitude. This modulation can be used to emphasize surface boundaries, to reduce the reflection from non-surfaces, i.e., samples where the gradient is very small, or to minimize the visual impact of noisy data.

The modulation is enabled based on the user-specified value of three registers, Gradient Magnitude Illumination Modulation for diffuse and specular illumination (Gmim_d and Gmim_s) and Gradient Magnitude Opacity Modulation (Gmom). Figure 4 shows four renderings of a CT scan of an engine block (256 x 256 x 110) with different values of Gmim_d, Gmim_s, and Gmom. Figures 4(a) and (b) show the difference between no modulation and gradient magnitude modulation of specular illumination. Regions of small gradient magnitude, such as noise, are de-emphasized in Figure 4(b). Figure 4(c) shows gradient magnitude modulation of opacity. Homogeneous regions with small gradient magnitude, such as the interior of walls, are more transparent compared to Figure 4(a). Figure 4(d) shows the combined effect of opacity and illumination modulation. Homogenous regions are more transparent and the overall illumination is dimmer because fewer samples have large gradient magnitude.

To highlight samples with a particular gradient magnitude value and to optionally attenuate the modulation effect, the gradient magnitude is mapped into the range $[0 \cdots 255]$ by a user-specified piece-wise linear function. The modulation function is stored in the Gradient Magnitude Lookup Table (GmLUT) indexed by the gradient magnitude. Eight table entries specify points of a piece-wise linear function at equal intervals along the gradient magnitude axis. A linear interpolation between the table values computes the 8-bit repeating-fraction value. It represents a value in the range $[0 \cdots 1]$ that is then optionally multiplied with the opacity and illumination values.

3.5 Assigning Color and Opacity

The output of the Gradient Estimation unit is a stream of gradient vectors and their gradient magnitude modulation values, one per cycle, in the same order as the sample values. These need to be paired with their respective sample values for presentation to the Classification and Shading unit. The sample values themselves come from the Interpolation unit. However, since the gradients emerge quite a bit later than the samples, the sample stream needs to be delayed in order for the two to be synchronized. This is accomplished by the Delay unit D in Figure 3.

The actual classification (i.e., assignment of RGBA values) at each individual sample point is a straightforward table lookup of the 12-bit sample value into a 4096×36 bit classification lookup table (LUT). This table is pre-computed and loaded into VolumePro prior to rendering the volume data set. The opacities are automatically corrected in software to adjust for non-unit sample spacing. The output of the lookup table is a 36-bit RGBα value. Values of R, G, B, and α are in the range [0..1] and are represented as 8-bit (for RGB) or 12-bit (for α) repeating-fraction numbers.

As mentioned in Section 3.4, the value of α may be optionally modulated (multiplied) by the magnitude of the gradient depending on the value of Gmom. Thus, the assignment of α is summarized by:

$$\alpha = \begin{cases} \mathrm{GmLUT}[|\mathrm{Gradient}|] \cdot \mathrm{opacityLUT}[\mathrm{sampleValue}] \\ \qquad\qquad\qquad\qquad \text{if Gmom = 1, and} \\ \mathrm{opacityLUT}[\mathrm{sampleValue}] \qquad \text{if Gmom = 0} \end{cases} \tag{1}$$

3.6 Sample Illumination

A VolumePro rendering pipeline implements Phong illumination at each sample point at the rate of one illuminated sample per clock cycle. The illumination of the sample point is calculated by adding its emissive, diffuse, and specular contributions [3]. The emissive or glowing contribution is a result of color classification of the sample value scaled by an emissive coefficient k_e. That is, each of the RGB values from the classification LUT is multiplied by k_e to produce the emissive color of the object at that sample point.

The diffuse contribution is obtained by multiplying a user-defined diffuse coefficient k_d with a diffuse illumination value I_d and the color previously obtained from the color lookup table. The diffuse illumination value I_d is looked up in a reflectance map indexed by the unnormalized gradient at that sample point. The reflectance map is a pre-computed table that defines the amount of diffuse reflection due to the sum of all of the light sources of the scene. Reflectance values are mapped onto six sides of a cube, and each face of the cube is indexed by the gradient vector [13]. Using bi-linear interpolation among reflectance values keeps the table size small without incurring noticeable visual artifacts [2, 12].

The specular contribution is obtained by multiplying a user-defined specular color by the product of a specular coefficient k_s and a specular illumination value I_s. The specular illumination value I_s is looked up in a reflectance map indexed by the unnormalized reflection vector computed from the gradient and the eye vector. The specular reflectance map is implemented identically to the diffuse reflectance map.

The reflection map implementation supports an unlimited number of light sources. The VLI software loads all reflectance map tables into VolumePro prior to rendering a volume data set. Because the reflection vector is computed in hardware, the reflectance map tables do not have to be reloaded when the eye vector changes. However, any change in the number or positions of the light sources requires reloading the tables.

The entire shading calculation can be summarized as:

$$\begin{aligned} \mathrm{sampleColor} &= ((k_e + (I_d \cdot G_d \cdot k_d)) \cdot \\ &\quad \mathrm{colorLUT}[\mathrm{sampleValue}]) + \\ &\quad (I_s \cdot G_s \cdot k_s \cdot \mathrm{specularColor}), \end{aligned} \tag{2}$$

where:

$$\begin{aligned} I_d &= \mathrm{diffuseReflectanceMap}[\mathrm{gradientVector}], \\ I_s &= \mathrm{specularReflectanceMap}[\mathrm{reflectionVector}], \\ G_{[d,s]} &= \begin{cases} \mathrm{GmLUT}[|\mathrm{Gradient}|] & \text{if } \mathrm{Gmim}_{[d,s]} = 1 \\ 1 & \text{if } \mathrm{Gmim}_{[d,s]} = 0 \end{cases} \end{aligned}$$

and where k_e, k_d, k_s, and specularColor are registers of VolumePro. The values of Gmim_d and Gmim_s enable or disable Gradient Magnitude Illumination Modulation for diffuse or specular illumination, respectively.

3.7 Accumulating Color Values along Rays

The output of the Classification and Shading unit is a stream of color and opacity values at sample points in slice-by-slice, beam-by-beam order. This stream is fed into the Compositing unit for accumulation into the pixel values of the rays. Because the samples are not presented in ray order, the pixel value of each ray must be accumulated one sample point at a time. Then it must be buffered until the color and opacity values of its next sample point arrive in the stream.

The Ray Slice FIFO, Ray Beam FIFO, and Ray Shift Register hold the values of the partially accumulated rays for this purpose. Although these look a little bit like the Voxel Slice FIFO, Voxel Beam FIFO, and Voxel Shift Register, their functions are different because of the cyclical nature of compositing. To understand this, observe that the compositing operation for sample $S_{x,y,z}$ requires as input the result of compositing one of $S_{x,y,(z-1)}, S_{(x-1),y,(z-1)}, S_{x,(y-1),(z-1)}$, or $S_{(x-1),(y-1),(z-1)}$, depending upon the view direction and the value of z. That is, the predecessor value required as input to a particular compositing operation is the result of one of four compositing operations of the previous slice of samples, depending on view direction. The selection of which particular one falls under the control of the Ray Alignment unit near the bottom of Figure 3, which drives two multiplexers (labeled MUX in the figure).

The Ray Slice FIFO stores partially accumulated pixel values of rays and makes them available for compositing with color and opacity values from the next slice. The Ray Beam FIFO stores the same values for a further beam time, so that in case the rays angle downward, input values can be obtained from the beam above and before the current sample. Likewise, the Ray Shift Register stores the same value one additional cycle, in case the input value needs to be obtained from a sample to the left of the current one.

In addition to alpha blending, VolumePro supports two additional blending modes that are summarized in Table 1. In the table, C_{acc} and α_{acc} are the accumulated color and opacity, respectively. For correct α-blending in large volumes, VolumePro keeps 12 bits of precision for $\alpha, C_{\mathrm{acc}}, \alpha_{\mathrm{acc}}$, and all intermediate compositing values.

Blend Mode	**Functions**
Front-to-back α-blending	$C_{\mathrm{acc}} += (1-\alpha_{\mathrm{acc}}) \times (\alpha_{\mathrm{sample}} C_{\mathrm{sample}})$ $\alpha_{\mathrm{acc}} += (1-\alpha_{\mathrm{acc}}) \times \alpha_{\mathrm{sample}}$
Minimum Intensity	if (sampleValue < minValue): $C_{\mathrm{acc}} = C_{\mathrm{sample}};$ minValue = sampleValue;
Maximum Intensity	if (sampleValue > maxValue): $C_{\mathrm{acc}} = C_{\mathrm{sample}};$ maxValue = sampleValue;

Table 1: Blending modes of VolumePro.

Finally, after the color and opacity values of all of the sample points on an individual ray have been accumulated, the resulting pixel value of that ray is output from VolumePro. This may occur when a ray passes through the back of the volume data set – i.e., with a maximum value of z – or when it passes through a side face of the data set. Base plane pixel values are written to Pixel Memory and then transfered to a companion 3D graphics card for the final image warp.

4 Advanced Features of VolumePro

In this section, several additional features are described that have some impact on the architecture of VolumePro. These include supersampling, supervolumes (volumes larger than 256 voxels in any dimension), partial volume updates, cut planes and cropping, and 3D line or plane cursors.

4.1 Supersampling

Supersampling is a technique for improving the quality of the rendered image. Sample points are defined in the volume data set at a higher spatial frequency than the voxel spacing. In the case of supersampling in the x- and y- directions, this would result in more samples per beam and more beams per slice, respectively. In the z-direction, it results in more sample slices per volume.

This implementation of VolumePro supports supersampling in hardware only in the z-direction. Additional slices of samples are interpolated between existing slices of voxels, so that the total number of slices of samples exceeds the number of slices of voxels in the volume data set. Figure 5 shows a foot (152 x 261 x 200) of the visible man CT dataset rendered with no supersampling (left) and supersampling by 2 (right). Notice the reduced artifacts in the supersampled image. The maximum supersampling factor in z is eight.

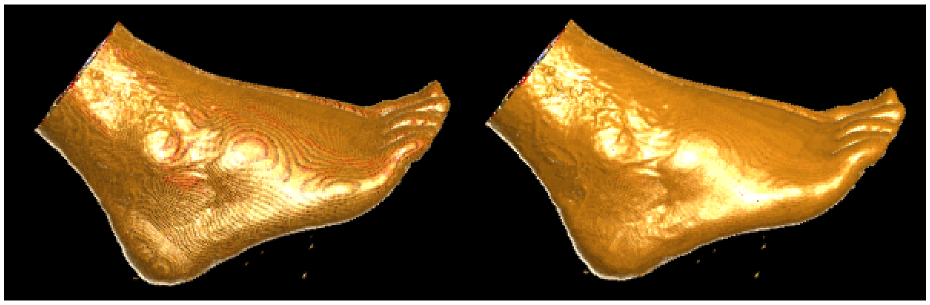


Figure 5: *No supersampling (left) and supersampling in z by 2 (right).*

The impact of supersampling on the VolumePro processing pipelines is minimal. Since it is necessary to have voxels from two slices on hand to interpolate in the z-direction, it is sufficient to do several of these interpolations from the same pair of slices before moving on to another slice. Since samples pass from the Interpolation unit in slice-by-slice order, the only impact is the number of slices. The net effect is that the voxel memory stages of the pipeline must stall and wait for the additional, supersampled slices to clear. Thus, if we supersample by a factor of k, the rendering rate of a 256^3 data set is reduced to $\frac{30}{k}$ frames per second.

With the current implementation of VolumePro, supersampling in the x- and y-directions can be implemented by repeated rendering a volume with slightly different offsets of the base plane in the x- and y-dimensions. The resulting images are then blended in software to create a smoother result. The VolumePro hardware supports initial offsets of the base plane at sub-pixel accuracy. The VLI software provides the necessary support to automatically render several frames and to blend them into a final supersampled image.

4.2 Supervolumes

Because of limited on-chip buffer size for the slice FIFOs, the VolumePro hardware can only render volume data sets with a maximum of 256 voxels on each edge. In order to render a larger volume, software must first partition the volume data set into blocks with a maximum size not exceeding 256 in any dimension. Each block is then rendered independently, and their resulting images are then combined in software to produce a rendering of the whole volume.

In order to maintain correct interpolation and gradients at the edges of the individual blocks of the supervolume, three voxels are duplicated in each of the three dimensions of adjacent blocks. If a ray exits the volume exactly on a border between two blocks it may happen that a sample is blended into the image twice, once from each block. In this case, VolumePro automatically excludes duplicated samples.

VolumePro allows setting sub-pixel accurate initial offsets in the x and y dimensions to align the rays correctly when passing from one supervolume block to another. In addition, an initial z offset maintains correct supersampling in the z-direction between blocks of a supervolume. VolumePro renders the supervolume blocks and the VLI software composits each resulting base plane from back-to-front into a single base plane. The resulting blended base plane is then warped to the final image.

The VolumePro driver software automatically partitions supervolumes, takes care of the data duplication between blocks, and blends intermediate base planes into the final image. Blocks are automatically swapped to and from host memory if a supervolume does not fit into the 128 MB of volume memory on the VolumePro PCI card. There is no limit on the size of a supervolume, although, of course, rendering time increases due to the limited PCI download bandwidth.

4.3 Subvolumes and Partial Volume Updates

VolumePro's memory controller can be programmed to transfer any subvolume from host memory to Voxel Memory and from Voxel Memory to host memory. Subvolumes can have as little as one voxel on each edge, which allows reading and writing single voxels, slices, or any rectangular slab to and from Voxel Memory. Subvolumes can be updated in-between frames. This allows dynamic and partial updates of data to achieve 4D animation effects. It also allows sections of a larger supervolume to be loaded piece-wise, allowing the user to effectively pan through a portion of the supervolume without having to reload the entire area of interest. Additionally, any subvolume of a volume loaded into Voxel Memory may be selected for rendering. This allows reduced rendering times on an area of interest.

4.4 Cropping and Cut Planes

VolumePro provides two features for clipping the volume data set during rendering. These make it possible to visualize slices, cross-sections, or other portions of the volume, thus providing the user an opportunity to see inside in creative ways.

The first feature is cropping. This simply cuts off the volume along planes parallel to its faces. An example image is shown in Figure 1 on the right. Figure 6 shows a conceptual diagram. Three slabs, one parallel to

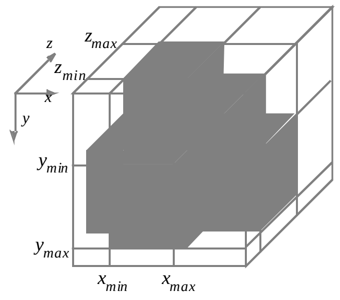


Figure 6: *Cropping of a volume dataset.*

each of the three axes of the volume data set, are defined by six registers $x_{\min}, x_{\max}, y_{\min}, y_{\max}, z_{\min}$, and $z_{\max}$. Slab S_x is the set of all points (x, y, z) such that $x_{\min} \le x \le x_{\max}$, slab S_y is the set of all points such that $y_{\min} \le y \le y_{\max}$, and slab S_z is the set of all points such that $z_{\min} \le z \le z_{\max}$. Slabs may be combined by taking intersections, unions, and inverses to define regions of visibility of the volume data set.

A sample at position (x, y, z) is visible if and only if it falls in this region of visibility. Thus in Figure 6, the shaded part of the volume remains visible and the remainder is invisible. Alternatively, the same slabs could be combined in a different way so that only the intersection of the three slabs is visible. Examples are illustrated in Figure 7. Note that the cropping planes defining the slabs may fall at arbitrary voxel positions.

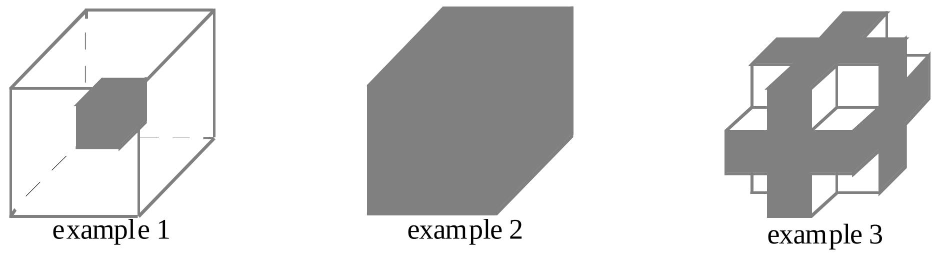


Figure 7: *Additional cropping examples.*

A side effect is that cropping is an easy way of specifying a rectilinear region of interest for rendering.

A second form of clipping is the cut plane. This is a planar slice of some thickness through the volume data set at an arbitrary angle. Samples are visible only if they lie between the two parallel surfaces of the cut plane; otherwise, their opacities are set to zero, just as with cropping. Alternatively, samples can be made visible only if they are outside of the cutplane. In addition, the opacities assigned to samples near the edges of the cut plane can be made to fall off gradually to provide a smooth looking edge. The current version of VolumePro supports a single cut plane.

Figure 8 illustrates a cut plane. The two parallel faces of the cut plane are given by the plane equations:

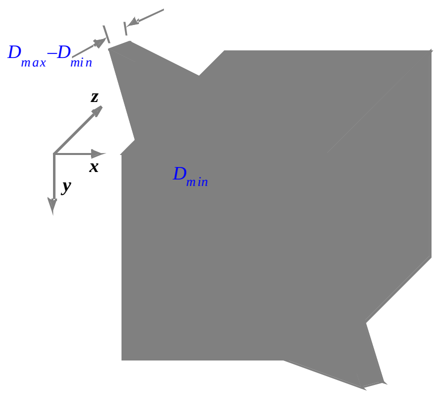


Figure 8: *Cut-plane through a volume dataset.*

$$\begin{aligned} Ax + By + Cz - D_{\mathrm{min}} &= 0 \\ Ax + By + Cz - D_{\mathrm{max}} &= 0, \end{aligned} \tag{3}$$

where $D_{\mathrm{min}} \leq D_{\mathrm{max}}$. That is, D_{min} measures the distance from the origin to one face of the cut plane, and D_{max} measures the distance from the origin to the other face. The thickness of the cut plane is $D_{\mathrm{max}} - D_{\mathrm{min}}$.

To allow for smooth cuts, a falloff parameter (D_{falloff}) specifies the width of the transition from full opacity to none. This is illustrated in Figure 9. Outside the cut plane, α is forced to zero. In the transition regions, α is multiplied by a correction factor. This correction factor increases linearly from zero to one. In the interior of the cut plane, α is simply the value from the shading stage. Figure 10a) shows an example of a cut plane with D_{falloff} set to five voxels.

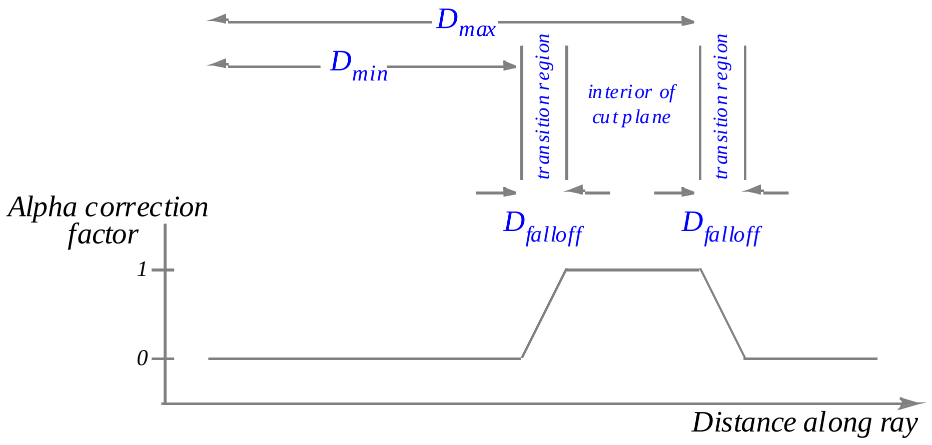


Figure 9: *Transition regions of a cut plane.*

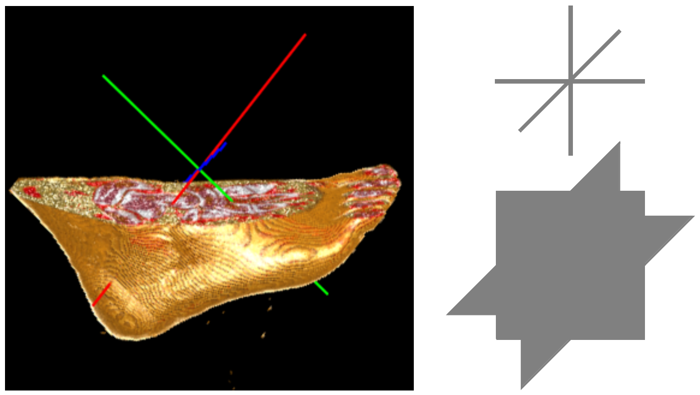


Figure 10: *(a) Cut plane with* $D_{\mathrm{falloff}} = 5$. *(b) Cross-hair and cross-plane 3D cursors.*

4.5 3D Cursor

VolumePro has a 3D cursor feature that inserts a hardware generated, software controlled, cursor into the volume data set being rendered. The 3D cursor allows users to explore and identify spatial relationships within the volume data. Figure 10b) shows the cross-hair and cross-plane 3D cursors supported by VolumePro. The right image in Figure 1 and Figure 10a) show examples of images with inserted cross-hair cursor. The cursors have user definable width and length in each direction. Each of the x, y and z cursor axis components can be independently enabled and has independent color and thickness. The cursors are blended into the volume data by α-blending.

5 vg500 Chip Architecture

The rendering engine of the VolumePro system is the vg500 chip with four parallel rendering pipelines. It is an application specific integrated circuit (ASIC) with approximately 3.2 million random logic transistors and 2 Mbits of on-chip SRAM. It is fabricated in 0.35 μ technology and runs at 133 MHz clock frequency. Figure 11 shows the layout of the vg500 ASIC. In this section we discuss a number of practical considerations

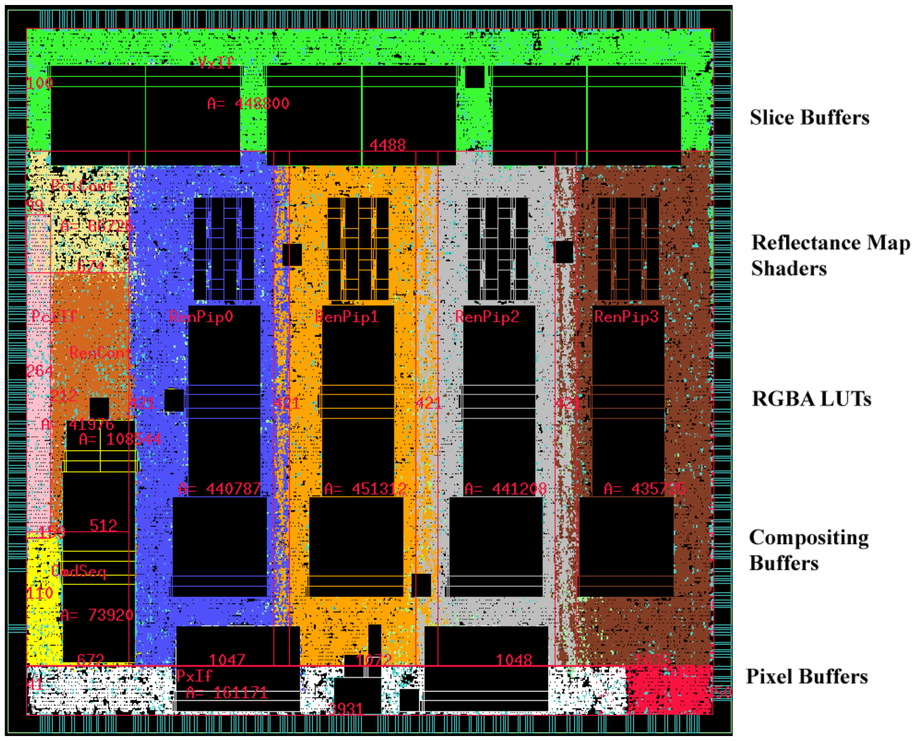


Figure 11: *Layout of the vg500 ASIC with four vertical rendering pipelines. Captions on right refer to SRAM blocks in the image.*

that affect the architecture of the vg500 ASIC and that lead to modifications of the idealized pipeline of Section 3.

5.1 Parallel Pipelines

To render a data set of 256^3 voxels at 30 frames per second, VolumePro must be able to read $256^3 \times 30$ voxels per second – that is, approximately 503 million voxels per second. In the current implementation of VolumePro, each pipeline operates at 133 MHz and can accept a new voxel every cycle. Thus, achieving real-time rendering rates requires four pipelines operating in parallel. These can process $4 \times 133 \cdot 10^6$ or approximately 532 million voxels per second.

The arrangement of the four pipelines in the vg500 chip architecture is illustrated in Figure 12. They work on adjacent voxel or sample values in the x-direction. Each pipeline is connected to its neighbor by means of the shift registers in each of the major units. Whereas in Figure 3 the shift registers recirculate values that are needed in the next cycle in the x-direction, in Figure 12 they provide values to their neighbors in the same cycle. The exceptions are the shift registers of the rightmost pipeline, each of which is marked with an asterisk. These act as in Figure 3 to recirculate values needed in the next cycle, but they send those values to the leftmost pipeline. An off-chip Voxel Memory supplies data to all of the pipelines, and they all write pixel values to an off-chip Pixel Memory. The additional datapaths on the right and the off-chip Section Memory will be discussed in Section 5.4.

5.2 Voxel Memory Organization

The vg500 chip has four 16-bit memory interfaces to Voxel Memory. A typical Voxel Memory configuration consists of four Synchronous Dynamic Random Access Memory (SDRAM) modules. The current implementation of VolumePro uses 64-megabit SDRAMs that provide burst mode access at 133 MHz. Voxel Memory can be extended to four SDRAMs per memory interface. Thus, sixteen SDRAMs provide 1024 megabits (i.e., 128 megabytes) of voxel storage. This is sufficient to hold a volume data set with 128 million 8-bit voxels or 64 million 12-bit voxels. Four 133 MHz memory interfaces can read Voxel Memory at a burst rate of 4×133 million (i.e., 532 million) voxels per second, provided that voxels can be organized appropriately in memory.

There are three challenges. First, voxels have to be organized so that data is read from Voxel Memory in bursts of eight or more voxels with consecutive addresses. This is done by arranging voxels in miniblocks. A miniblock is a $2 \times 2 \times 2$ array of voxels stored linearly in a single memory module at consecutive memory addresses. All data is read in bursts of the size of a miniblock.

Second, miniblocks themselves have to be distributed across memory modules in such a way that groups of four adjacent miniblocks in any direction are always stored in separate memory modules, avoiding memory access conflicts. This is done by skewing using the method described in [9]. In VolumePro, this skewing technique is applied to miniblocks, ensuring that four adjacent miniblocks, i.e., miniblocks parallel to any axis, are always in different memory modules, no matter what the view direction is.

Third, miniblocks within a memory module must be further skewed so that adjacent miniblocks never fall into the same memory bank of an SDRAM chip. 64-megabit SDRAMs have four internal memory banks. $4 \times 4 \times 4$ cubes of miniblocks are skewed across the four memory banks, thus allowing back-to-back accesses to miniblocks in any traversal order with no pipeline delays. This is illustrated in Figure 13. A voxel with coordinates (u, v, w) has the miniblock coordinates $(u_m, v_m, w_m) = (u/2, v/2, w/2)$ which is mapped to one of the memory modules and memory banks as follows:

$$\begin{aligned} \text{Module} &= (u_m + v_m + w_m) \bmod 4 \\ \text{Bank} &= ((u_m \div 4) + (v_m \div 4) + (w_m \div 4)) \bmod 4, \end{aligned} \quad (4)$$

where $\bmod$ denotes the modulus operator and $\div$ the integer division operator. In order to ensure that miniblocks are properly aligned for optimum SDRAM performance the Voxel Memory must be allocated such that all dimensions of the volume object appear to be multiples of $2 \times 4 \times 4 = 32$ voxels. Arbitrary volumes in host memory are stored in Voxel Memory with dimensions that are multiples of 32 voxels and then automatically cropped to their original size during rendering. The vg500 ASIC contains all necessary

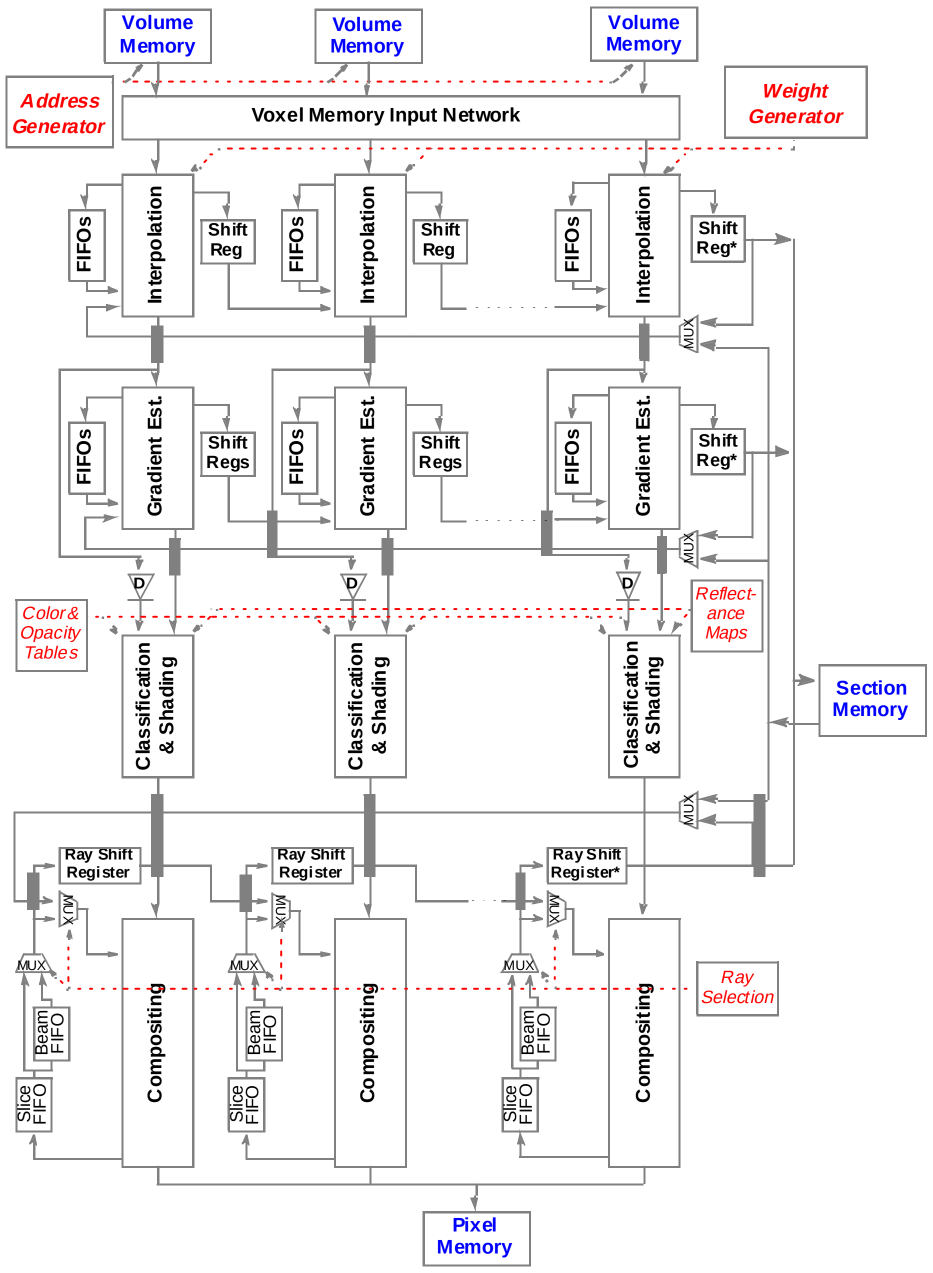


Figure 12: *The four parallel pipelines of the vg500 ASIC. Modules shown in blue are off-chip memory.*

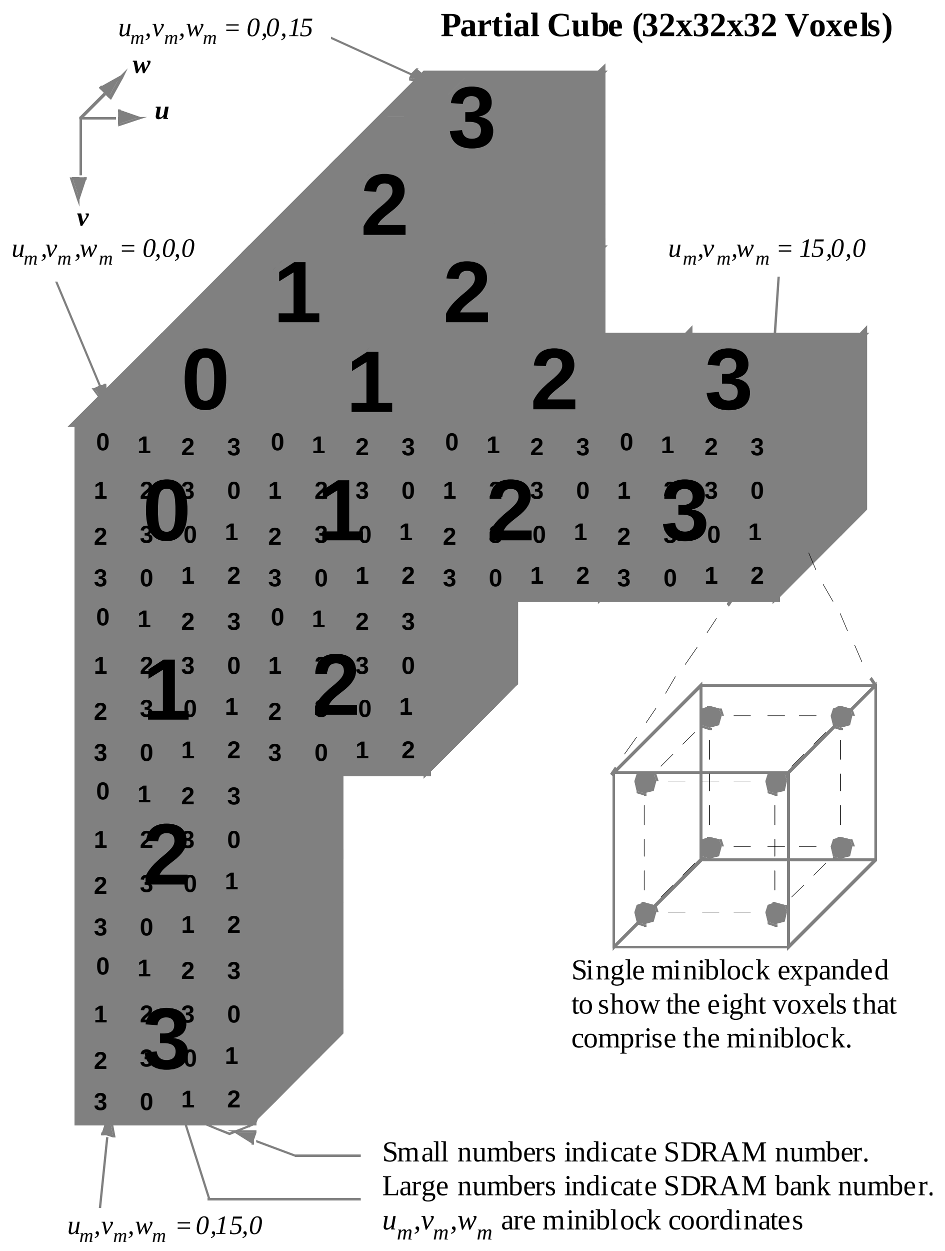

Partial Cube (32x32x32 Voxels)
u_m,v_m,w_m = 0,0,15
w
u
v
u_m,v_m,w_m = 0,0,0
u_m,v_m,w_m = 15,0,0
u_m,v_m,w_m = 0,15,0
Single miniblock expanded to show the eight voxels that comprise the miniblock.
Small numbers indicate SDRAM number.
Large numbers indicate SDRAM bank number.
u_m,v_m,w_m are miniblock coordinates

Figure 13: *Organization of miniblocks in Voxel Memory.*

hardware to arrange voxels into miniblocks and to skew and deskew miniblocks during data transfers to and from Voxel Memory.

5.3 Voxel Input To Rendering Pipelines

The description of the rendering pipeline in Section 3 assumes that the Interpolation and Gradient Estimation units read voxels in the slice-by-slice, beam-by-beam order described in Section 2. The Voxel Memory Input Network, shown in Figure 12, distributes data from the four Voxel Memory interfaces to the slice buffers. From there it can be read slice-by-slice and beam-by-beam by the four rendering pipelines.

Voxels in a miniblock are always read out of voxel memory in the same order because they are stored sequentially to take advantage of the burst capabilities of the SDRAMs. The miniblocks need to be re-arranged, based upon the current view direction, so that the positions of the voxels correspond to the canonical positions assumed for tri-linear interpolation. This is achieved using simple miniblock reorder logic in each memory interface.

Because of the skewing of miniblocks in Voxel Memory, the four miniblocks output by the Voxel Memory controllers must be de-skewed and potentially re-shuffled based upon the chosen view direction. This is required so that the left-most voxel along the x axis flows down the leftmost pipeline and adjacent voxels in x flow down in neighboring pipelines. This is achieved with a global de-skewing network that connects the four memory interfaces to the four rendering pipelines.

After the miniblock de-skewing network, the four miniblocks are written into the voxel slice buffers of the four rendering pipelines. Since data is read from voxel memory in miniblocks and processed by the pipelines as slices, a method must exist to convert from one to another. The Voxel Memory interface actually reads two slices of voxels at a time because of the memory organization in miniblocks There is also the requirement to store three slices worth of data so that the z gradients can be computed using central differences. After reading two slices of miniblocks and writing that data into four slice buffers all necessary data exists to compute interpolated samples and gradients.

5.4 Sectioning

To keep the amount of on-chip memory for the various FIFOs within the practical limit of current semiconductor processes, the volume data set is partitioned into sections and some intermediate values are off-loaded to external memory between the processing of sections. It is possible to partition the volume data set into sections in the x-, y-, and/or z-directions. The current version of VolumePro implements sectioning only in the x-direction. Each section is 32 voxels wide, corresponding to the memory organization outlined in Section 5.2. Additional background on sectioning is given in [9].

The impact of sectioning on the pipeline architecture is illustrated in Figure 12. In the right-hand pipeline of Figure 12 the shift registers are modified to optionally write values to Section Memory at the end of a section. Likewise, the left-hand pipeline either reads values from Section Memory (at the beginning of a section) or accepts them from the right-hand pipeline. In effect, the Section Memory becomes a big, external FIFO capable of holding values that need to be recirculated in the x-direction. Exactly the same values are written to, then read from, Section Memory as would have been passed to the next pipeline. Consequently, no approximation is made and no visual artifacts are introduced at section boundaries.

The voxel traversal order is therefore modified. In particular, voxels are read in slice-by-slice order within a single section. Likewise, within slices, voxels are read in beam-by-beam order, but with beams spanning only the width of a section. The full section is processed front-to-back, and the intermediate values needed for rendering near the right boundary of the section are written to Section Memory. Then the next section is processed. Values are read from the Section Memory in the same order that they had been written, and they are passed through the multiplexers of Figure 12 to the left pipeline of Figure 12. So far as the rendering algorithm is concerned, it is as if they had been generated on the previous cycle and passed directly. That is, the same values are calculated in either case, but in different orders.

6 VolumePro PCI Card

The VolumePro PCI card is a single board with a 32-bit 66 MHz PCI interface. The board contains a single vg500 rendering ASIC, twenty 64 Mbit SDRAMs with 16-bit datapaths, a serial EEPROM, and a clock generation circuit. Figure 14 shows a block diagram of the components on the board and the busses connecting them.

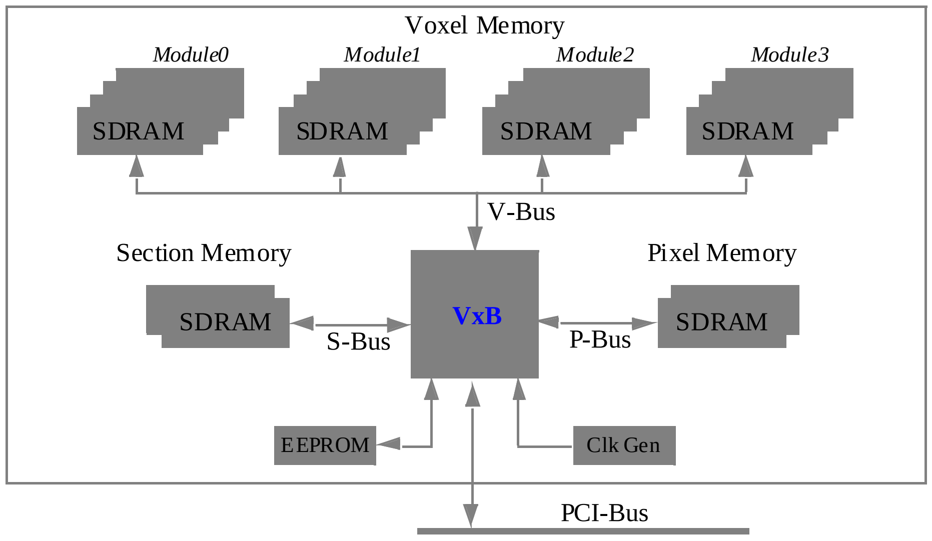


Figure 14: *VolumePro PCI board diagram.*

The vg500 ASIC interfaces directly to the system PCI-Bus. Access to the vg500's internal registers and to the off-chip memories is accomplished through the 32-bit 66 MHz PCI bus interface. The peak burst data rate of this interface is 264 MB/sec. Either Target R/W accesses to registers and off-chip memories or Direct Memory Access (DMA) to off-chip memories can be issued and may be interspersed.

Most registers are write-only and are memory-mapped via their own PCI base address register. Volume, pixel, and section memory are read/write-able and are directly memory-mapped into the PCI address space. The vg500 chip status is checked either by interrupts or by polling a status register that is on-chip. Alternatively, a copy of the status register gets DMA'ed to host memory when it changes. VolumePro supports many standard 16-bit and 32-bit pixel formats with on-the-fly conversion between formats during reads or writes.

The standard size of voxel memory is 128 MBytes, organized as four groups with four SDRAMs each. VolumePro also supports 32, 64, and 96 MByte configurations. Two 64 Mbit SDRAMs make up Section Memory and two 64 Mbit SDRAMs of Pixel Memory contain the rendered base plane image. They can hold up to sixteen base planes, each with up to 512×512 32-bit pixels. This allows double-buffering of several base planes on the PCI card and pipelined retrieval and warping of images. The base plane pixels are transferred over the PCI bus to a companion 3D graphics card for the final image warp and display.

The VolumePro system is inherently scalable. For example, several PCI cards can be connected to a high-speed interconnect network. Alternatively, several vg500 ASICs and their Voxel Memories can be integrated onto a single multi-processing rendering board. Volume data can be rendered in blocks, similar to supervolumes, on different vg500 ASICs using coarse-grain parallelism. However, at the present time we have no plans to implement such a multi-processing system.

7 VLI - The VoxelBlaster Interface

Figure 15 shows the software infrastructure of the VolumePro system. The VLI API is a set of C++ classes

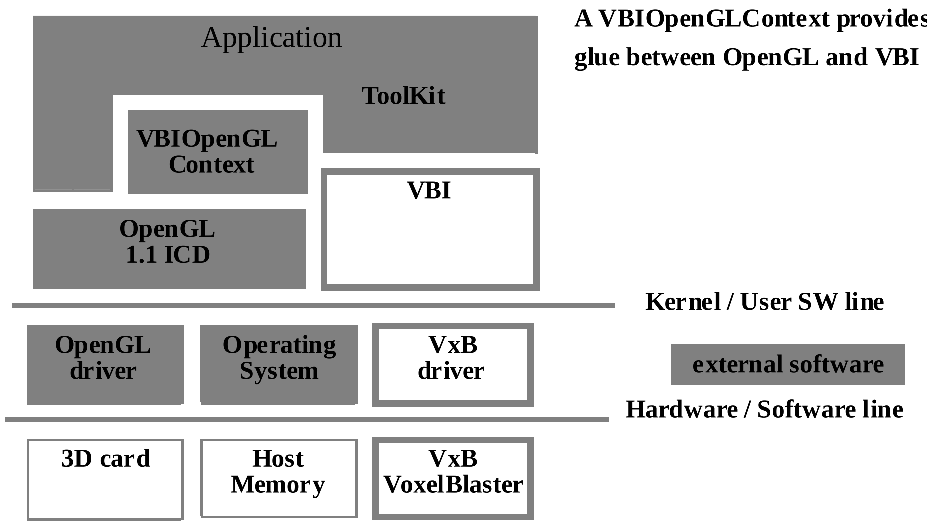


Figure 15: *Software infrastructure of the VolumePro system.*

that provide full access the vg500 chip features. VLI does not replace an existing graphics API. Rather, VLI works cooperatively with a 3D graphics library, such as OpenGL, to manage the rendering of volumes and displaying the results in a 3D scene. We envision higher level toolkits and scene graphs being the primary interface layer to applications on top of VLI.

The VLI classes can be grouped as follows:

- Volume data handling. The VLI class VLIVolume manages voxel data storage, voxel data format, and transformations to the volume data such as shearing, scaling, and positioning in world coordinate space.

- Rendering element creation. The following VLI classes are responsible for creating and specifying various rendering elements that control the final rendered image.

  VLILookupTable. Defines the lookup tables that contain the colors and opacity values to be mapped to the volume.

  VLICamera. Defines the viewing parameters of one or more cameras, which are stand-ins for the user in viewing the object being rendered.

  VLICrop. Defines the cropping characteristics, if used, for the rendering process.

  VLICutPlane. Defines the characteristics of the cut plane, if it is used in the rendering.

  VLILight. Creates and defines one or more lights that illuminate the scene.

  VLICursor. Creates and defines the 3D cursor.

  Several lookup tables, lights, and cameras can be created in each scene.

- Rendering context. The VLI class VLIContext is a container object for all attributes needed to render the volume. It is used to specify which volume data set, lookup table, light(s), and crop should be used for the current frame. It also specifies other rendering parameters, such as the gradient modulation parameters, material properties, specular highlight, supersampling, and blend mode.

The VLI automatically computes reflectance maps based on light placement, sets up α-correction based on viewing angle and sample spacing, supports anisotropic and gantry-tilted datasets by correcting the viewing and image warp matrices, and manages supervolumes, supersampling, and partial updates of volume data. In addition, there are VLI functions that provide initialization, configuration, and termination for the VolumePro hardware.

8 Conclusions

VolumePro is the world's first single chip real-time volume rendering system. This paper describes the algorithm, architecture, and advanced features of the VolumePro system and the vg500 rendering ASIC. The rendering capabilities of VolumePro – 500 million trilinear, Phong illuminated, composited samples per second – are two orders of magnitude higher than those of any existing system for PC class computers. Its core features, such as on-the-fly gradient estimation, per-sample Phong illumination with arbitrary number of light sources, 4K RGBA classification lookup table, α-blending with 12-bit precision, and gradient magnitude modulation of opacity and illumination, put it ahead of any existing hardware solution for volume rendering. Additional features, such as supersampling along rays, supervolumes, partial volumes updates, cropping and cut planes, and 3D line and plane cursors, enable the development of feature-rich high-performance volume visualization applications.

We believe that VolumePro will change the way volume rendering is used. For the first time, users on PC-class systems will be able to go beyond the volume rendering bottleneck. Application developers will focus on new and innovative interaction techniques with volume data, such as interactive experimentation with different rendering parameters. This may lead to new solutions for difficult problems, such as data segmentation and transfer function design. VolumePro allows visualizing dynamic volume datasets, such as data from interactive tissue cutting during surgical simulation, or continuous data input from 3D ultrasound. Future versions of VolumePro will include support for perspective projections and intermixing polygons and volume data while continually increasing the performance and reducing the cost of the system.

References

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

[2] I. Ernst, D. Jackel, H. Rüsseler, and O. Wittig. Hardware supported bump mapping: A step towards higher quality real-time rendering. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, pages 63–70, Maastricht, The Netherlands, August 1995.

[3] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, 2nd edition, 1990.

[4] B. M. Hemminger, T. J. Cullip, and M. J. North. Interactive visualization of 3D medical image data. Technical report, University of North Carolina at Chapel Hill, Department of Radiology and Radiation Oncology, 1994. TR 94-027.

[5] K. H. Höhne and R. Bernstein. Shading 3D-images from CT using gray-level gradients. *IEEE Transactions on Medical Imaging*, MI-5(1):45–47, March 1986.

[6] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. In *Computer Graphics*, Proceedings of SIGGRAPH 94, pages 451–457, July 1994.

[7] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(5):29–37, May 1988.

[8] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[9] R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibson, W. Hiatt, and T. Ohkami. EM-Cube: An architecture for low-cost real-time volume rendering. In *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*, pages 131–138, Los Angeles, CA, August 1997.

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

[11] F. Sheid. *Schaum's Outline of Theory and Problems of Numerical Analysis*. McGraw Hill, NY, 1968.

[12] J. van Scheltinga, J. Smit, and M. Bosma. Design of an on-chip reflectance map. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, pages 51–55, Maastricht, The Netherlands, August 1995.

[13] D. Voorhies and J. Foran. Reflection vector shading hardware. In *Computer Graphics*, Proceedings of SIGGRAPH 94, pages 163–166, Orlando, FL, July 1994.

[14] K. Z. Zuiderveld, A. H. J. Koning, and M. A. Viergever. Acceleration of ray casting using 3D distance transform. In R. A. Robb, editor, *Proceedings of Visualization in Biomedical Computing*, pages 324–335, Chapel Hill, NC, October 1992. SPIE, Vol. 1808.