A Ray-Slice-Sweep Volume Rendering Engine

Ingmar Bitter * and *Arie Kaufman* †

Center for Visual Computing (CVC)
State University of New York at Stony Brook ‡

Abstract

Ray-slice-sweeping is a plane sweep algorithm for volume rendering. The compositing buffer sweeps through the volume and combines the accumulated image with the new slice of just-projected voxels. The image combination is guided by sight rays from the view point through every voxel of the new slice. *Cube-4L* is a volume rendering architecture which employs a ray-slice-sweeping algorithm. It improves the *Cube-4* architecture in three ways. First, during perspective projection all voxels of the dataset contribute to the rendering. Second, it computes gradients at the voxel positions which improves accuracy and allows a more compact implementation. Third, *Cube-4L* has less control overhead than *Cube-4*.

Keywords: Volume Visualization, Volume Rendering Architecture, Hardware Design, Gradient Estimation, Compositing, Perspective Projection

1 Introduction

Volume visualization is a method of extracting information from volumetric datasets through the use of interactive graphics and imaging, and is concerned with the representation, manipulation and rendering of these datasets [5]. Most commonly, the data represents a continuous 3D function sampled on a regular, rectilinear 3D grid of volume elements called *voxels*. Depending on this 3D function (mathematical, MRI, CT, ultrasound, etc.) voxels accordingly represent function value, density, absorption, etc. There are many different approaches for rendering images from these volume datasets. ***Volume rendering*** includes only those techniques in which the image generation does not use any intermediate surface representations of the sampled data. These techniques fall into four groups: ***object-order, image-order, and hybrid algorithms*** as well as ***domain methods.***

*ingmar@cs.sunysb.edu, http://www.cs.sunysb.edu/~ingmar

†ari@cs.sunysb.edu, http://www.cs.sunysb.edu/~ari

‡Center for Visual Computing (http://www.cvc.sunysb.edu), Department of Computer Science (http://www.cs.sunysb.edu), State University of New York at Stony Brook, Stony Brook, NY 11794-4400

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee

1997 SIGGRAPH/Eurographics Workshop
Copyright 1997 ACM 0-89791-961-0/97/8..$3.50

Object-order algorithms compute each voxel contribution to all affected image pixels in object-storage order (e.g., splatting [18]). Image-order algorithms compute each pixel color in image scan line order. For each pixel, a ray is cast into the volume, and all the voxels in the neighborhood of the ray are processed and composited to generate a color value for the pixel (e.g., ray-casting [7], volumetric ray-tracing [15]). These algorithms usually deliver higher image quality than object-order algorithms, but they also require more computation and tend to access each voxel multiple times.

Hybrid algorithms combine object storage-order voxel access with image scan line processing order (e.g., slice transformation [1], cell-by-cell processing [16], template-based volume viewing [19], shear warp [6]). Good hybrid algorithms combine the high image quality of image-order algorithms with the efficient data access of object-order algorithms.

Domain methods transform the data set from the spatial into another domain, such as frequency, wavelet, compression, or light-field domain. Images are then rendered directly from the transformed domain data [2, 8, 9, 10, 11, 12, 3, 20].

Section 2 introduces ray-slice-sweeping — our new volume rendering algorithm. Section 3 describes the Cube-4L architecture, a hardware design based on the ray-slice-sweeping algorithm, followed by a short analysis of the energy distribution of a voxel in Section 4 and results from our software simulator of the Cube-4L architecture in Section 5.

2 Ray-Slice-Sweeping

Cube-4L is based on a new hybrid algorithm, called *ray-slice-sweeping*. Each volume slice projects all its voxels towards the viewpoint, but only one inter-slice unit in distance. In a sweep plane fashion, the compositing buffer sweeps through the volume from front-to-back and combines the images of the volume previously swept with the new slice of just-projected voxels. At the end of the sweep, the content of the compositing buffer — the *base plane image* — has to be warped onto the image plane. The sweep is done in object storage order yet it is image-pixel driven, while the warp is performed in scan line order; thus, ray-slice-sweeping belongs to the hybrid algorithm group. The algorithm is designed specifically for perspective projections. Parallel projections can be performed with the view point at infinity.

While sweeping through the volume in front-to-back processing order, each voxel of the current slice has to be colored, illuminated and classified resulting in RGBA intensities for that voxel. The algorithm sweeps the dataset always in positive z direction, independent of the view point. With the view point in front of the dataset (Fig. 1) front-to-back compositing (Fig. 3) is used to combine the RGBA values of the current slice with those of the compositing buffer. If the view point is behind the dataset (Fig. 2) the back-to-front

compositing equations are applied (Fig. 3). The main difference between ray-slice-sweeping and other ray casting based volume rendering algorithms is how the accumulated RGBA values are determined.

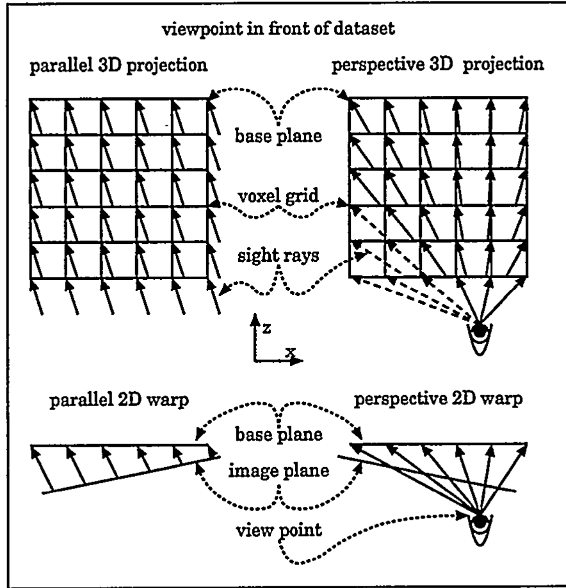

Figure 1: *Ray Slice Sweep (front-to-back compositing).*

Figure 1 depicts the rays cast by the algorithm within a horizontal cut through the dataset. Horizontal lines in the figure represent the data slices which the algorithm processes in positive z direction. Dashed sight rays exceed the 45° view angle limit and do not influence the final image. With the view point in front of the dataset the sight rays point towards the voxel grid positions and start in between voxels of the previous slice. For each voxel of a given slice the sight ray is traced back to its starting point on the previous slice. The compositing buffer contains accumulated colors of the whole volume up to the previous slice. Computing a bilinear interpolation of the four compositing buffer colors surrounding the rays' starting point yields the colors needed to perform the compositing calculation for the current voxel (Fig. 3). Thus, all values written to the compositing buffer are aligned with the voxel grid.

Figure 2 illustrates the sight rays for a viewpoint behind the dataset. Here, the sight rays start at the voxel positions and point towards the previously processed slice. The accumulated colors used in the back-to-front compositing equation are determined by bilinear interpolation of the four compositing buffer colors surrounding the rays' end point (Fig. 3).

All operations carried out during ray-slice-sweeping read and write data only within a local neighborhood of the currently processed voxel. All those small modifications add up from slice to slice, such that at the end of the sweep all voxels have been shifted to the image position required by perspective projection. Note that there are neither regions between rays in which voxel data is skipped, nor is any voxel used more than once. The algorithm automatically maintains a well balanced workload.

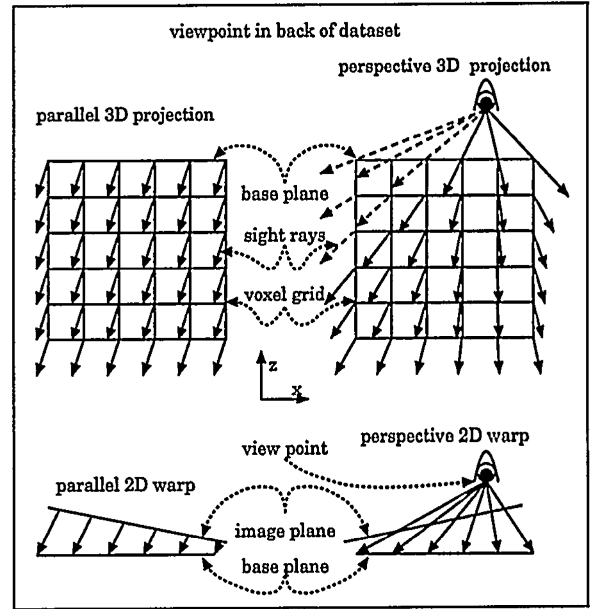

Figure 2: *Ray Slice Sweep (back-to-front compositing).*

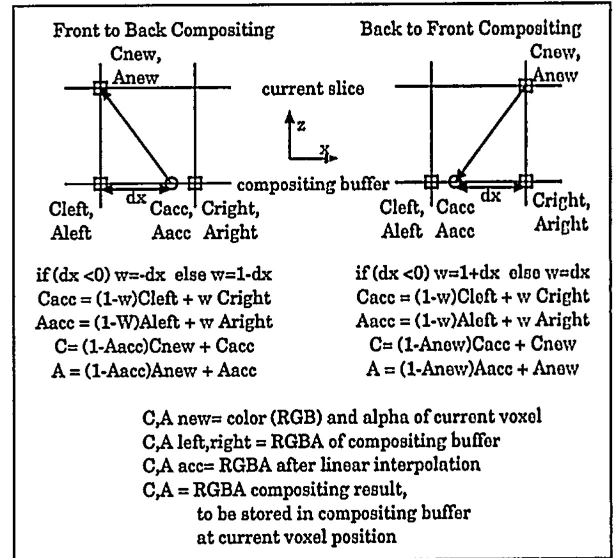

Figure 3: *Compositing.*

3 The Cube-4L Architecture

Cube-4 is a pipelined scalable volume rendering architecture based on ray casting [4, 13, 14]. Cube-4L is a modification of Cube-4 simplifying it by using an implementation of ray-slice-sweeping. Cube-4L uses sample points directly on the voxel grid. Therefore, the trilinear interpolation stage of Cube-4 is not needed and the gradient computation is simplified. Furthermore, in perspective projection all voxels of the dataset contribute to the rendering. As there is also less control information for rays, the implementation can be more compact — hence the name: Cube-4 Light.

Figure 4 gives an overview of the Cube-4L architecture. The voxel memory is distributed over several memory modules. Equally many rendering pipelines — each on a separate Cube-4L chip — are working on the 3D projection simulta-

neously. They only need nearest neighbor connections for horizontal communication until the final base plane pixels are computed. These pixels are sent over a global pixel bus to the host or graphics card memory to be assembled into one image and warped onto the final image plane.

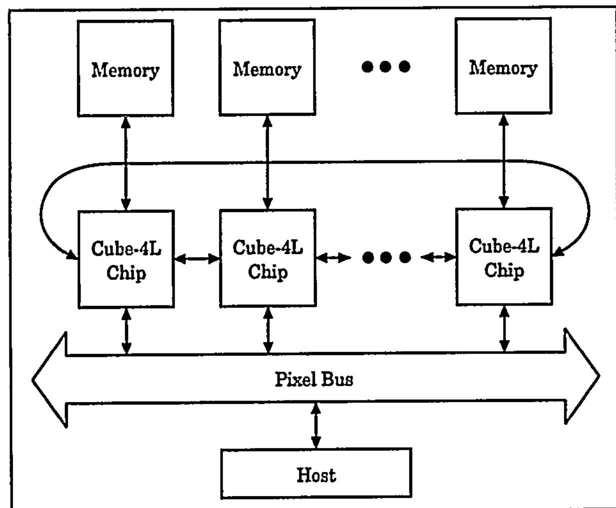

Figure 4: *The Cube-4L Architecture.*

The following detailed description of the Cube-4L rendering pipeline modules uses signal flow graphs (SFG) as described in [13]. Circles represent pipeline stages. Vertical arcs show the data flow within a pipeline, while diagonal arcs show data flow between pipelines. The weight on each arc represents for how many pipeline cycles the data has to be delayed before reaching the next pipeline stage. All units assume partial beam processing: beams are rows of voxels and breaking a beam into equal sized segments produces partial beams; the size of a partial beam is equal to the number of parallel pipelines implemented in hardware (Fig. 5). To handle the differences between partial-beam-end, beam-end and slice-end, each module has an extension unit (EX) at every pipeline stage with neighbor connections. Details about the Cube-4L architecture not mentioned in this paper are assumed to be handled the same way as in the Cube-4 architecture.

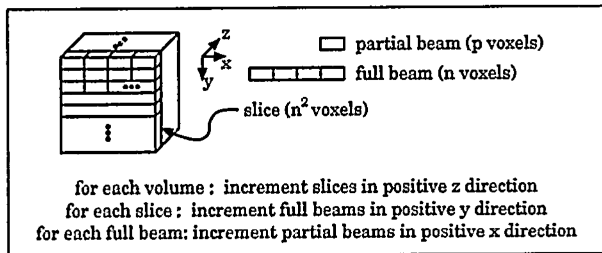

Figure 5: *The Partial Beam Processing Order.*

3.1 Cubic Frame Buffer

The Cubic Frame Buffer (CFB) contains memory modules for distributed skewed storage of voxels as well as an address generator and a control unit. It is the only stage in the Cube-4L rendering pipeline (Fig. 6) with global position information. It must make all global decisions for all pipeline

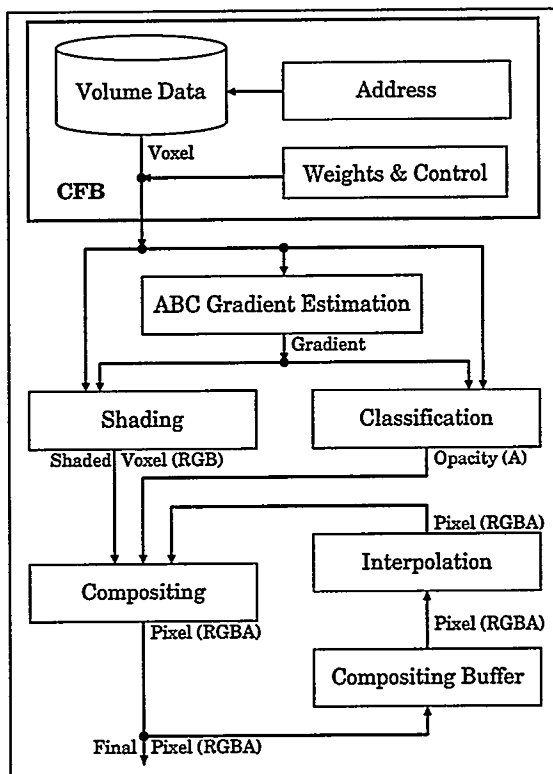

Figure 6: *The Cube-4L Rendering Pipeline.*

modules. The subsequent stages make their selections based on the control bits generated in the CFB.

These are a few abbreviations we employ:

(i, k) **Memory Space coordinates:** k = memory module number, i = address in module.

(u_d, v_d, w_d) **Dataset Space coordinates:** right handed, view independent and static.

(x_p, y_p, z_p) **Pipeline Space coordinates:** right handed, view dependent and changes if major view direction changes.

n: number of voxels along one axis of a cubic dataset.

p: number of parallel pipelines.

k: current pipeline number.

b: number of partial beams ($p \cdot b = n$).

N_s: current slice number.

N_B: current beam number.

N_b: current partial beam number.

SR: control bit for start of ray

ER: control bit for end of ray

First, the CFB has to calculate the position of the voxel starting to flow down the pipeline in the current clock cycle. An efficient incremental algorithm for this needs only a few local registers to store counters for the current partial beam number N_b, the beam number N_B, and the slice number

N_s. The result is the 3D voxel position $P = (x, y, z)$ in the pipeline coordinate system. The view point V is also defined in the pipeline coordinate system.

The *sight ray* S of a voxel is the vector from the viewpoint to the voxel position. Thus, the sight ray is the vector $S = P - V$ (see Fig. 7). Normalizing S yields $dx = S_x/S_z$ and $dy = S_y/S_z$. These normalized x and y components determine the bilinear interpolation weights (Fig. 6) and are forwarded to the compositing unit. They are also range checked. If $|dx| > 1$ or $|dy| > 1$, then the viewing angle exceeds 45° and the *invalid* bit is set to true; to make those voxels valid, the architecture would need more than nearest neighbor connections. The shading stage assigns complete transparency ($\alpha = 0$) to those shaded voxels which have the invalid bit set. As in Cube-4 for any arbitrary viewing direction one of the faces of the data cube has a normal within ±45° of the sight ray. The slices processed by the algorithm are parallel to that slice. Therefore, the restriction to using only nearest neighbors is feasible. For parallel viewing all sight rays are the same; sweeping the volume once delivers the complete image. In perspective projection mode the sight rays differ for each voxel, as might the faces for which the normal is within ±45°. This happens especially for viewpoints close to or inside the dataset. To acquire a full image with these settings, multiple sweeps through the data volume in different processing directions are necessary. The final image is assembled from the regions of valid pixels in the different base plane images.

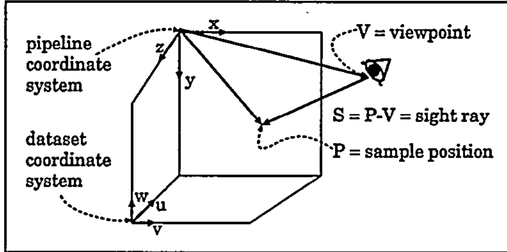

Figure 7: *Sight ray and different coordinate systems.*

All Cube architectures share the same memory layout. A voxel at position $P' = (u, v, w)$ is stored in memory module $k = (u + v + w) \bmod p$. This is called skewed memory. It distributes the voxels across memory modules such that all voxels of a partial beam parallel to the x,y or z axis reside in different memory modules. Thus, they can be accessed conflict free. All computation in the pipeline assumes working on slices perpendicular to the z axis and uses pipeline coordinates. To compute the voxel memory address, the vector $P = (x, y, z)$ has to be transformed into the dataset coordinate system. The transformed vector $P' = (u, v, w)$ is used to compute the address: $i = u \operatorname{div} p + v\, b + w\, b\, n_y$. While storing the data one has to ensure that a voxel at position (u, v, w) is written to memory module $k = (u+v+w) \bmod p$. However, in the rendering mode, this computation is implicit — each pipeline has only one dedicated memory module.

To be certain that voxels are only shaded if they have valid gradients, the CFB also has to set the invalid bit for all those voxels that have a direct neighbor on the other end of the data set. For those voxels the CFB might also have to set the *SR* or *ER* bits (i.e., *ER* should be true if the current voxel is on the left most slice of the dataset and $dx < 0$). The latter two bits are forwarded to the compositing unit, where they enable the compositing buffer reset (*SR*) and the pixel output to the final image (*ER*).

The CFB also has to set the start of beam (*BS*) bit whenever the current partial beam is the first on a full beam. This bit is needed in the extension units.

Finally, to output the final pixels to the unskewed base plane position, the CFB must compute the pixel address for each compositing buffer element. The coordinates are

$$x = (N_b \cdot p + k - N_s - (N_B - 1) + 2n) \bmod n + N_s \cdot \begin{cases} 1 & \text{if } dx > 0 \\ 0 & \text{if } dx = 0 \\ -1 & \text{if } dx < 0 \end{cases}$$

and

$$y = (N_B - 1) + N_s \cdot \begin{cases} 1 & \text{if } dy > 0 \\ 0 & \text{if } dy = 0 \\ -1 & \text{if } dy < 0 \end{cases}$$

3.2 ABC Gradient Estimation

The gradients used for shading in Cube-4L are central difference gradients (see Fig. 8). As a consequence, we need the six axis-aligned neighbors ($a-f$) of a voxel m to compute the gradient. In Cube-4 the gradients are computed using neighboring sample points. Fortunately, in Cube-4L all necessary neighbors are on the voxel grid. Thus, no interpolation and no gradient corrections are necessary.

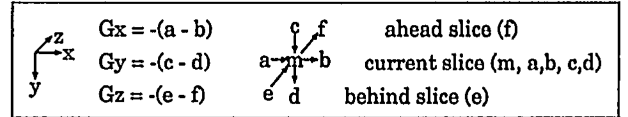

Figure 8: *Gradient components and the unskewed spatial relationship of the six computation parameters (voxels a-f).*

The voxels needed for the computation reside in three planes — the ahead, behind and current (ABC) slices — reflecting the processing order of the slices. They are stored in the ABC slice FIFO buffers. In fact, only the B and C slices are stored in a FIFO; the ahead slice comes directly from the CFB.

Figure 9 shows the spatial and temporal relationships between the gradient computation parameters considering the skewing. Voxel e is read first. After $b(n-1) = (s - b)$ cycles voxel c is available in the same pipeline. b cycles later, voxel a is read in the same pipeline and voxel b two pipelines to the right. That pipeline also reads voxels d and f after b and $b(n-1)$ more cycles. The following figures show the SFGs delaying and moveing those voxels so that they can be used to compute the three gradient components. Starting with the easiest possible approach, each new figure adds another inherent consideration. At the bottom of each graph new logical symbols are explained.

Figure 10 shows an SFG computing only the y-component of the gradient. It achieves delaying voxel c and moving voxel d such that they end up at the same clock cycle in the same pipeline. Hence, enabling the computation of the vertical gradient component $G_y = -(c - d)$.

Figure 11 modifies that approach such that only nearest neighbor connections are used. This makes one more pipeline stage necessary which receives voxel d from the pipeline to the right and sends it to the pipeline to the left. Figure 12 then changes the forwarding directions so that the y-gradient is computed in the pipeline which also holds the corresponding center voxel m.

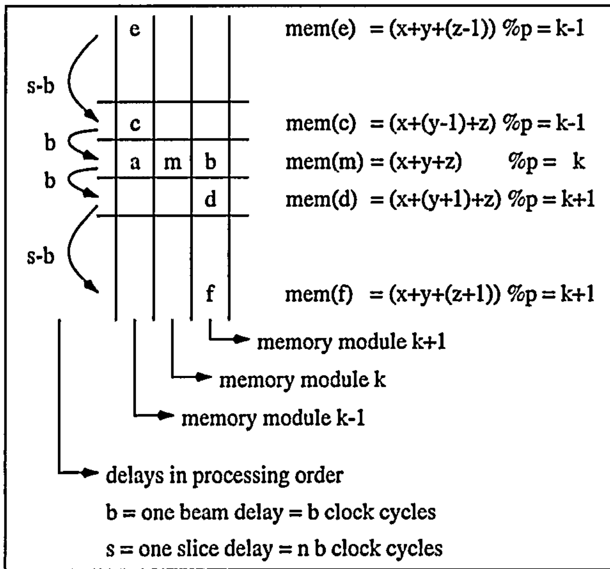

Figure 9: *Skewed positions of the gradient computation parameters (voxels a-f).*

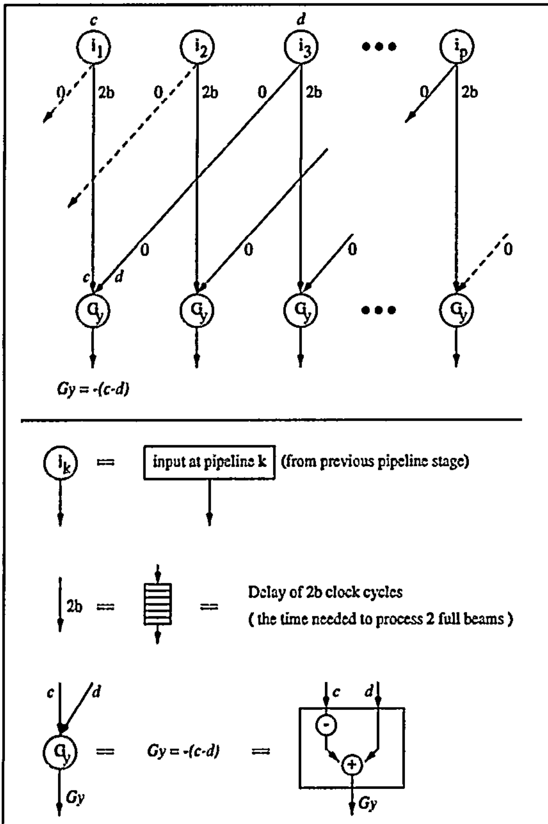

Figure 10: *Gradient y-component SFG, de-skewing and proper time lineup.*

Partial beams are tiled across each full beam. This is illustrated in Figure 13 for two full beams in skewed space each having five partial beams. The shaded circles represent voxels with different intensities. In this example each partial beam has four voxels.

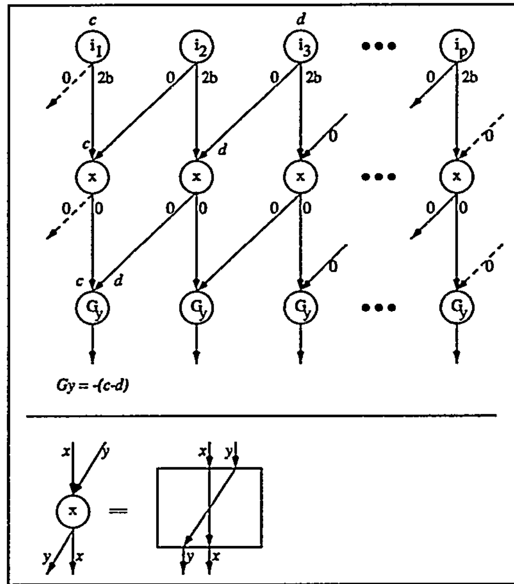

Figure 11: *Gradient y-component SFG, with only nearest neighbor connections.*

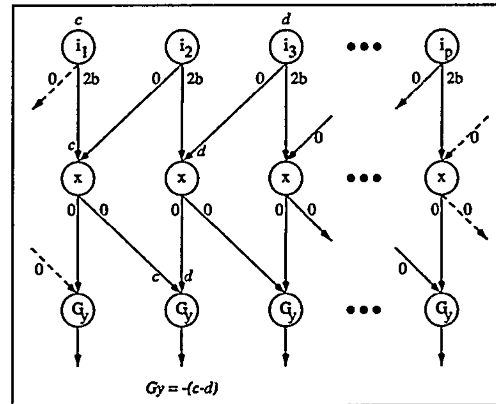

Figure 12: *Gradient y-component SFG, in which the result is computed in the pipeline of the corresponding center voxel m.*

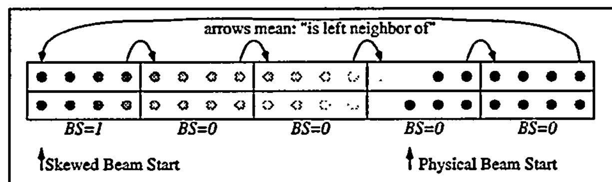

Figure 13: *Two consecutive full beams in skewed space.*

The voxel in the rightmost pipeline is usually the left neighbor of the voxel in the leftmost pipeline one clock cycle later. Consequently forwarding to the right requires buffering the rightmost voxel for one clock cycle. This buffer logically extends the next partial beam. Due to the skewing, the physical start of a beam changes from beam to beam. Thus the leftmost and rightmost voxels of a beam are also neighbors. Therefore, forwarding to the right requires the rightmost pipeline to send its data to the leftmost one. Unfortunately, it arrives there a full beam processing time (*b*

cycles) too late. The Cube-4 architecture handled this by reading the beam as soon as possible, but delaying the processing of the data by b cycles which requires buffering of the whole beam [4, 13]. Figure 14 suggests a better approach by shifting the beginning of a beam by one partial beam to the right. The figure shows the partial beams being read in the left column and the partial beams being processed in the right column. The first partial beam read is buffered in an extension unit for $b-1$ cycles. The following partial beams immediately process their data. After the last partial beam is finished its rightmost voxel and the voxels from the extension buffer are used to perform the computations on the first partial beam. This way, the extension buffer for each stage that forwards data to the right shrinks from full beam size to partial beam size. This increases the scalability of the architecture. Figure 15 shows the SFG using the just described ideas for the wrap around connections between the ends of partial beams.

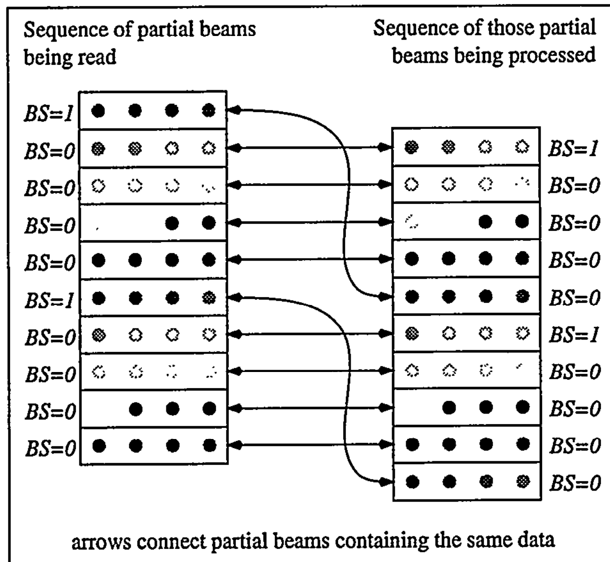

Figure 14: *Two consecutive full beams in time.*

Finally we show two alternatives for the complete x,y, and x gradient computation in one SFG. Fig. 16 minimizes pin count, while Fig. 17 minimizes on-chip buffer size. In Fig. 16 only two crossings between pipelines are needed. Therefore, considering the connections to the left and to the right pipeline, as well as 16 bits per voxel, this module needs 2x2x16=64 I/O-pins. The drawback is that four full slice buffers are necessary. In Fig. 17 only two full slice buffers are needed. However, each pipeline requires six connections to one neighboring pipeline. Thus, this version needs 6x2x16=192 IO-pins. For both alternatives, the pin count can be reduced by 75%, if only the four most significant bits are sent across pipelines. This is feasible if the shader uses only the four most significant bits of the computed gradient components.

		few pins	*small buffers*
buffer size		4 slices	2 slices
IO-pins	*16 bit*	64 pins	192 pins
	4 bit	16 pins	48 pins

Table 1: *Gradient hardware requirements*

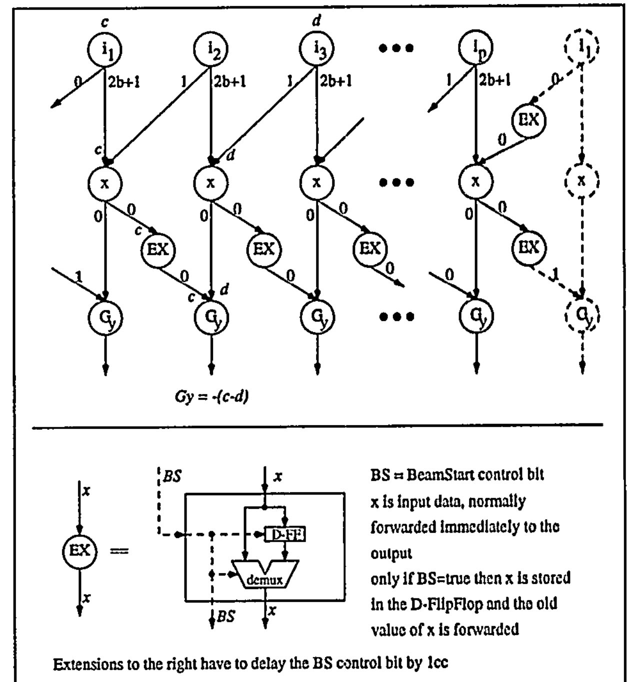

Figure 15: *Gradient y-component SFG, in which the wrap around connections can handle differences between full-beam-end and just partial-beam-end. An extension from the right stores the data of the leftmost pipeline in the first partial beam until the rest of the beam is processed. The extensions from the left store the complete first partial beam until the rest of the beam is processed.*

3.3 Shading

Once the voxel intensity and the corresponding gradient are available, the shader uses the intensity as an index into RGB color tables, and the gradient as an index into a reflectance map. The color tables map different intensities to different colors. This color transfer function can be used to segment the data on the fly. The reflectance map is a quantized solution table for the illumination equation for rays along the main viewing direction and any surface orientation [17]. The final color of a shaded voxel is then composed from the values returned from these tables. The most compact implementation just multiplies the RGB values from the color table with the intensity taken from the reflectance map.

3.4 Classification

Usually pixels are represented by RGBA where A represents the opacity α. The opacity for each voxel is determined by a classification function which depends on the voxel intensity and the gradient magnitude. This function is stored in another look-up table. Filling the table using different algorithms allows very flexible viewing modes such as x-ray, surface, fog, etc. Shading and classification are the same in Cube-4 and Cube-4L.

3.5 Compositing

Cube-4L uses either back-to-front or front-to-back compositing. The choice depends on the major viewing direction. If in the dataset coordinate system the viewpoint is left, above,

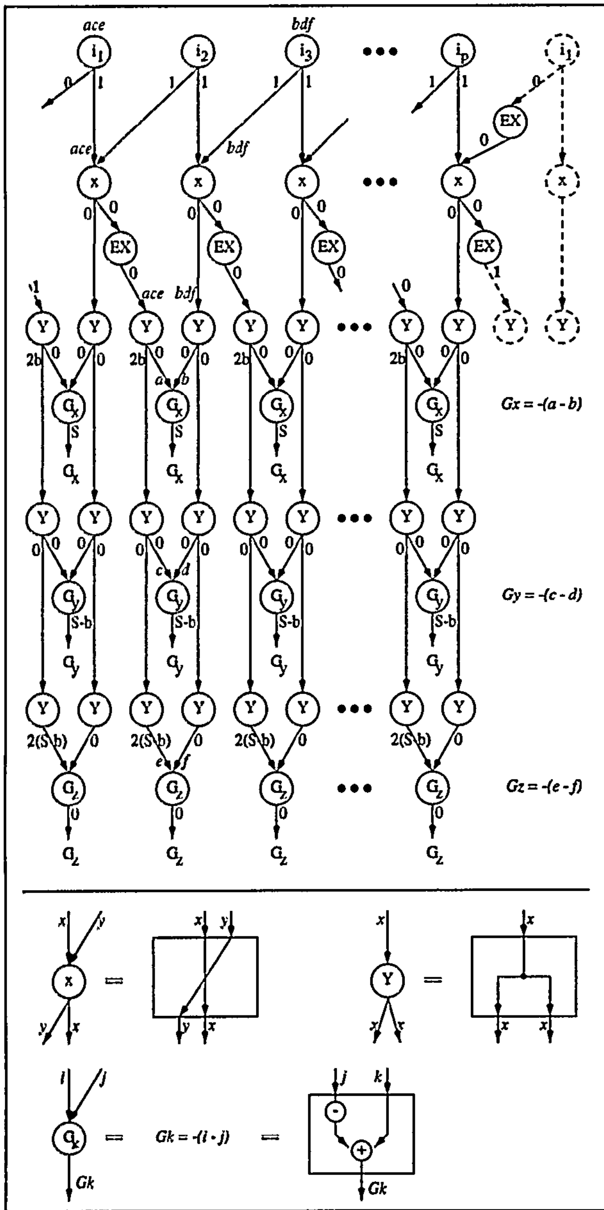

Figure 16: *SFG for all gradient components — minimizing pin count.*

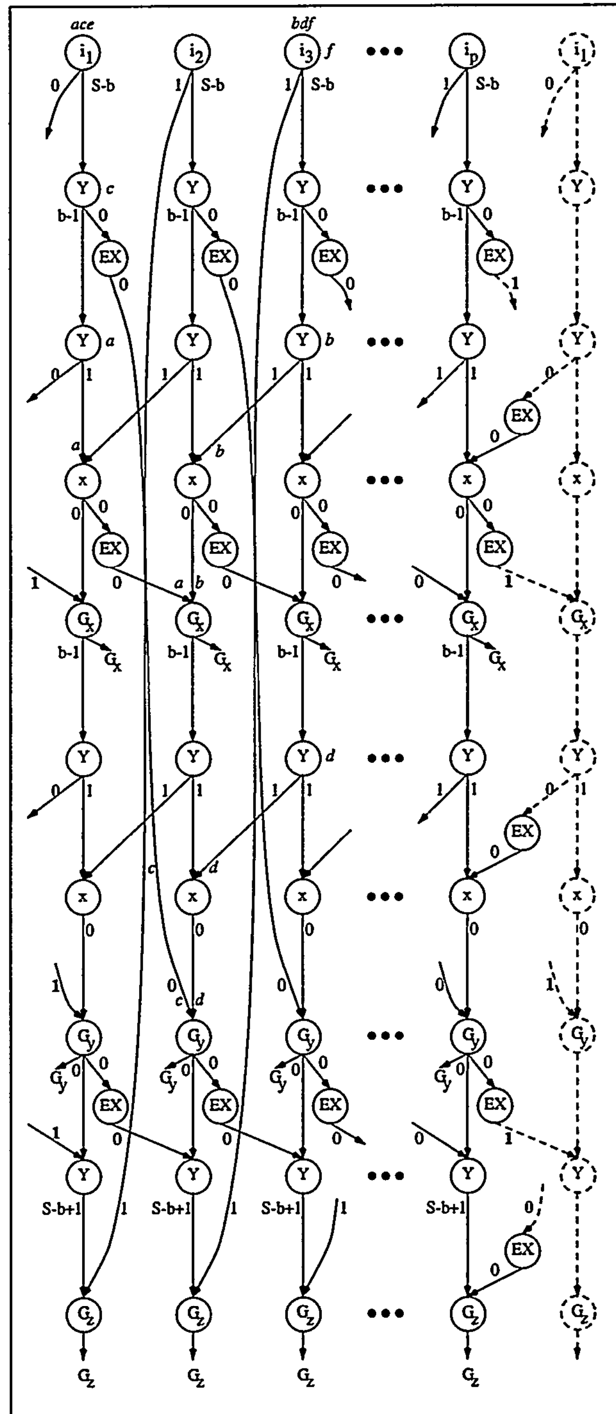

Figure 17: *SFG for all gradient components — minimizing internal buffer size.*

or in front of the dataset, front-to-back compositing is used — otherwise back-to-front is employed. The compositing equations read as follows:

Back-to-front compositing
$$C = (1 - A_{new}) \cdot C_{acc} + C_{new},$$
$$A = (1 - A_{new}) \cdot A_{acc} + A_{new}.$$

Front-to-back compositing
$$C = (1 - A_{acc}) \cdot C_{new} + C_{acc},$$
$$A = (1 - A_{acc}) \cdot A_{new} + A_{acc}.$$

Here we use C_{new} for any of the shaded voxel color components (RGB), and A_{new} for the classified voxel opacity, while using the subscript $_{acc}$ for the corresponding accumulated values taken from the compositing buffer. These equations are evaluated in the C_k circles of Fig. 18.

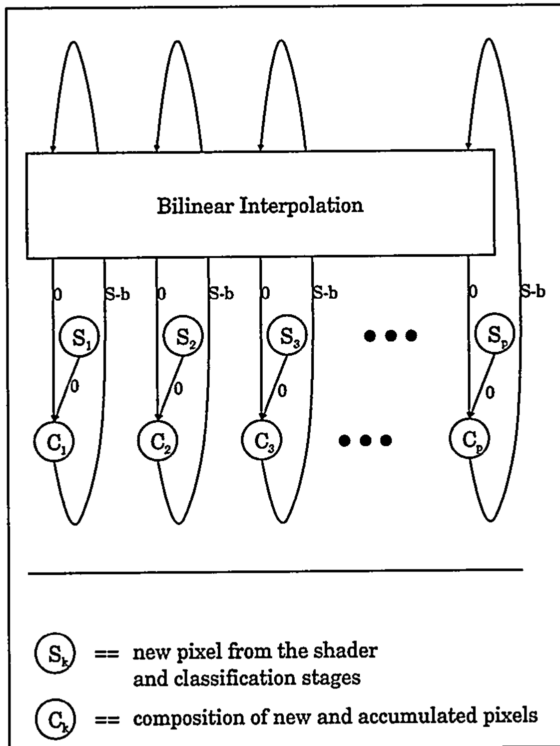

Figure 18: *Compositing with compositing buffer and bilinear interpolation.*

3.6 Compositing Buffer

The compositing buffer is a FIFO with a capacity of a full slice of RGBA values. The compositing unit writes new values to the bottom of the slice FIFO while the bilinear interpolation reads from the top.

	RGBA 48bit	*RGBA 32bit*
1 slice buffer	384 KBytes	256 KBytes
IO-pins	384 pins	256 pins

Table 2: *Compositing hardware requirements*

3.7 Bilinear Interpolation

The aims of the bilinear interpolation unit are to determine the color at the intersection between the compositing buffer plane and the sight ray through the already shaded voxel. Recall that because of the viewing angle being limited to ±45° the sight ray increments dx and dy are bounded. The intersection can therefore only occur within a 3x3 region around the current voxel. Figure 19 shows the skewed relative positions of the voxels in that region. In order to transfer the data from that region to the pipeline in which voxel m resides, a minimum of four pipeline stages is necessary — three that forward their data to their right neighbors and one that forwards to the left. This optimal configuration is used in the SFG of Fig. 21.

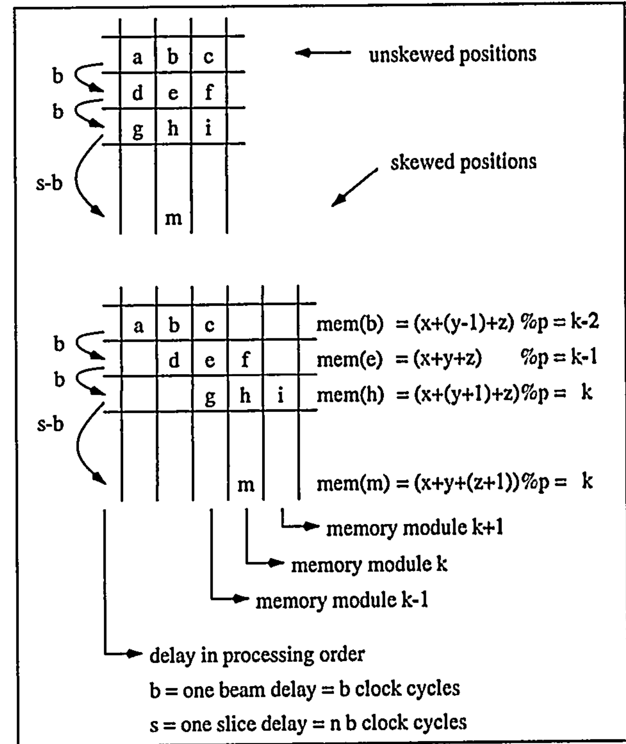

Figure 19: *Skewed relative positions of current voxel m, possibly influencing voxels a–i in the previous slice.*

The possible region can be narrowed to a 2x2 region just by evaluation of the sign of dx and dy (see the first two stages in Figs. 20 and 21). The second two stages in Figs. 20 and 21 then compute the actual bilinear interpolation from those four compositing buffer pixels. The dependency of the interpolation weights from the sight ray increments is given in Figs. 3 and 20.

4 Energy Distribution

In traditional ray casting each voxel can only contribute to a maximum of four rays. The voxel energy distribution on the image plane has very sharp edges. In our *ray-slice-sweeping* algorithm, the energy of a voxel is concentrated along the ray direction, but with each slice it spreads to the neighboring voxels. This spreading is caused by the bilinear interpolation. Thus, on average, a quarter of each voxel contribution flows towards the nearest four voxels of the next slice. In

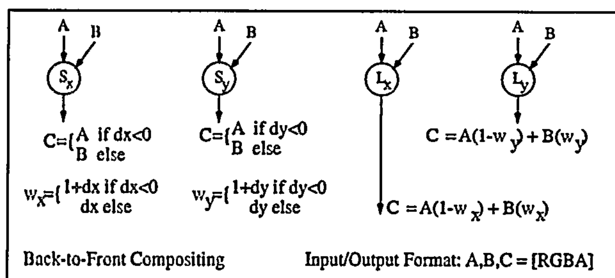

Figure 20: *Functionality of the pipeline stages in the bilinear interpolation SFG (Fig. 21) assuming back-to-front compositing. For front-to-back compositing dx and dy have the opposite sign slightly changing the computations.*

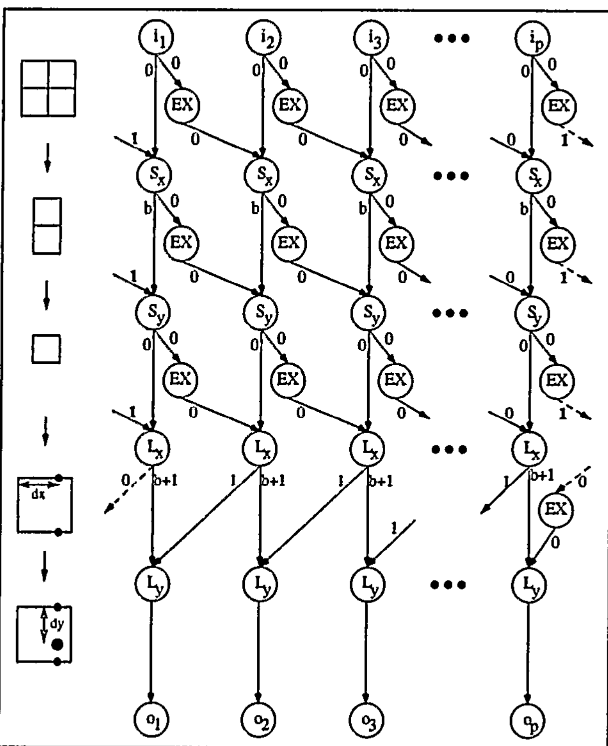

Figure 21: *Bilinear Interpolation SFG.*

the subsequent slice they spread over nine pixels, but also recombine in the center. As a result, the energy distribution function is a discrete approximation of the rapid decaying function $f(x) = 1/x^2$ rotated around the y-axis (see Fig. 22). This shows great similarity to the filter kernels used in splatting — and automatically provides anti-aliasing.

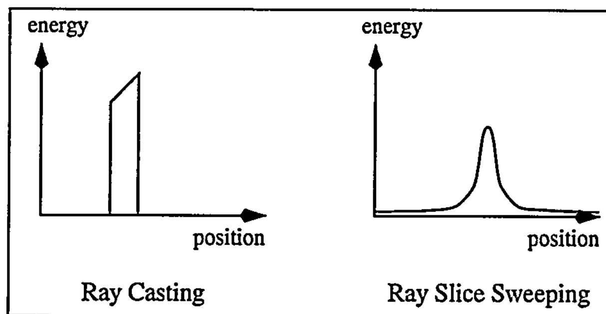

Figure 22: *Voxel Energy Distributions.*

5 Results

A bit accurate C++ simulation delivered the images in Figs. 23 and 24. The compositing buffer width was restricted to 8 bits per RGBA channel and all fixpoint operations used 8 fractional bits. Fig. 23 shows a 256^3 MRI head in parallel projection. Fig. 24 is an image of a 320x320x34 CT dataset of a lobster rendered in perspective mode.

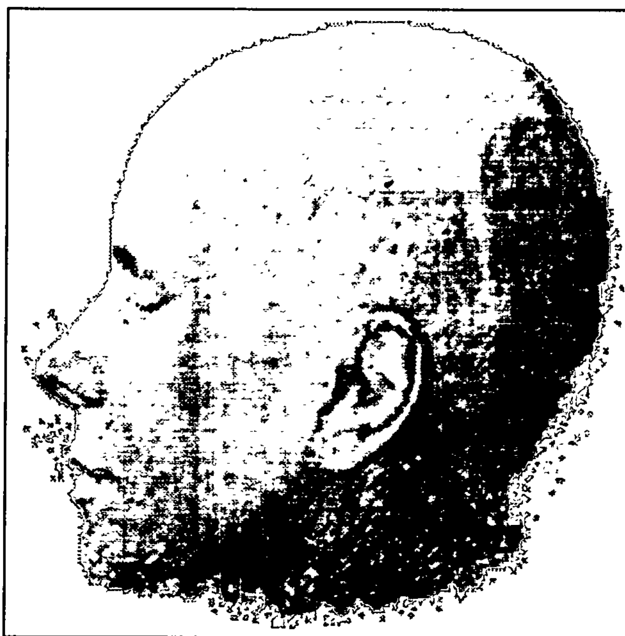

Figure 23: *MRI head, parallel projection.*

6 Conclusions

We have presented a volume rendering architecture which allows the rendering of parallel and perspective projections with low control overhead — especially in comparison to the Cube-4 architecture. In Cube-4L there are only nearest

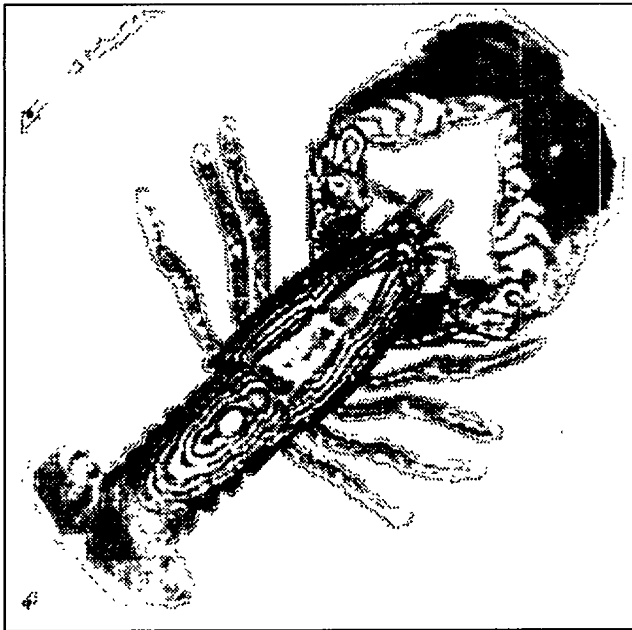

Figure 24: *CT lobster, perspective projection.*

neighbor connections required between rendering pipelines. An implementation could be realized on one chip with two parallel rendering pipelines. A higher number of pipelines would also be possible.

We are currently investigating modifications to the algorithm which will allow us to control the filter kernel size. Other anticipated improvements include carring out the compositing steps along trees rooted at each baseplane pixel, including merging/splitting branches of the trees using purely local information or globally predefined merges.

7 Acknowledgements

Many thanks to Hanspeter Pfister, Kevin Kreeger, Frank Dachille, and Baoquan Chen for numerous productive discussions. This work has been supported by NSF grant MIP-9527694, Japan Radio Corporation, Mitsubishi Electric Research Laboratory, and Hewlett Packard.

References

[1] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. In *Computer Graphics, SIGGRAPH 88*, volume 22(4), pages 65–74. ACM, August 1988.

[2] S. Dunne, S. Napel, and B. Rutt. Fast Reprojection of Volume Data. In *Proceedings of the 1st Conference on Visualization in Biomedical Computing*, pages 11–18, Boston, MA, 1990.

[3] J. Fowler and R. Yagel. Lossless Compression of Volume Data. In *Proceedings of 1994 Symposium on Volume Visualization*, pages 43–50, Washington, DC, October 1994.

[4] U. Kanus, M. Meissner, W. Strasser, H. Pfister, A. Kaufman, R. Amerson, R.J. Carter, B. Culbertson, P. Kuekes, and G. Snider. Implementations of Cube-4 on the Teramac Custom Computing Machine. *Computers and Graphics*, 21(2), 1997.

[5] A.E. Kaufman, editor. *Volume Visualization*. IEEE Computer Society Press, Los Alamitos, CA, 1991.

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

[7] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(5):29–37, May 1988.

[8] M. Levoy. Volume Rendering using the Fourier Projection-slice Theorem. In *Proceedings of Graphics Interface '92*, pages 61–69. Canadian Information Processing Society, 1992.

[9] M. Levoy and P. Hanrahan. Light Field Rendering. In *Computer Graphics, SIGGRAPH 96*, Anual Conference Series, pages 31–42, New Orleans, LA, August 1996. ACM.

[10] T. Malzbender. Fourier Volume Rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[11] S. Muraki. Volume Data and Wavelet Transform. *IEEE Computer Graphics & Applications*, 13(4):50–56, July 1993.

[12] P. Ning and L. Hesselink. Fast Volume Rendering of Compressed Data. In *Proceedings of Visualization '93*, pages 11–18, October 1993.

[13] H. Pfister. *Architectures for Real-Time Volume Rendering*. PhD thesis, State University of New York at Stony Brook, Computer Science Department, Stony Brook, NY 11794-4400, January 1997.

[14] H. Pfister and A. Kaufman. Cube-4: A Scalable Architecture for Real-Time Volume Rendering. In *Proceedings of 1996 Symposium on Volume Visualization*, pages 47–54, San Francisco, CA, October 1996.

[15] L. Sobierajski and A. Kaufman. Volumetric Ray Tracing. *Volume Visualization Symposium Proceedings*, pages 11–19, October 1994.

[16] C. Upson and M. Keeler. V-BUFFER: Visible Volume Rendering. In *Computer Graphics, SIGGRAPH 88*, volume 22(4), pages 59–64. ACM, August 1988.

[17] 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.

[18] L. Westover. Footprint Evaluation for Volume Rendering. In *Computer Graphics, SIGGRAPH '90*, volume 24, pages 367–376. ACM, August 1990.

[19] R. Yagel and A. Kaufman. Template-based Volume Viewing. In *Proceedings Eurographics*, volume 11(3), pages 153–167. Eurographics Association, September 1992.

[20] B.-L. Yeo and B. Liu. Volume Rendering of Dct-Based Compressed 3d Scalar Data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):29–43, March 1995.