Interactive Rendering of Volumetric Data Sets

Scott Juskiw and Nelson G. Durdle

Department of Electrical Engineering
University of Alberta, Edmonton, Alberta, Canada, T6G-2G7

V. James Raso and Doug L. Hill

Glenrose Rehabilitation Hospital, Edmonton, Alberta, Canada, T5G-0B7

ABSTRACT

The *bela* architecture for interactive rendering of regularly structured volumetric data sets is presented. The proposed architecture is scalable and uses custom processors to achieve high-speed shading, projection, and composition of voxel primitives. A general purpose image composition network supports the accumulation of both volumetric and geometric elements into the final rendered scene. Data access contentions between processors are eliminated via the use of an enhanced dual object space and image space partitioning scheme that does not require replication or redistribution of rendered data. The *bela* architecture is intended for rendering large data sets and meets the performance requirements of a full frame interactive image generation system.

KEYWORDS

volume rendering, image composition, parallel processing, computer architecture, scientific visualization, medical imaging.

I. INTRODUCTION

Numerous graphics applications including geometric modeling, scientific visualization, medical imaging, and virtual reality require the rapid processing and display of a dynamic computer generated environment. Instantaneous feedback to operator actions maintains a sensation of immediacy by permitting real-time observation, manipulation, and analysis. The massive computational requirements necessary to achieve high resolution images with low latency and frame rates of thirty or more updates per second for data sets comprising millions of elements demands a distributed network of dedicated processors [4,13,17]. Parallel accumulation, or *composition*, of multiple individually imaged picture elements into a final scene is a viable approach to distributed rendering [18]. View-independent parallel processing of sampled volumetric or abstract geometric primitives avoids the data redistribution bottlenecks of conventional rendering architectures leading to potentially higher performance systems [16].

The complexity of the rendering algorithm dictates the maximum number of primitives per processor to maintain interactive image generation rates. As data set density increases, the number of processors must rise to accommodate the load resulting in wider bandwidth requirements for the image generating system. Volume rendering applications, in particular, tax the capacity of existing hardware to shade, resample, and composite discrete three dimensional data sets of moderate complexity [2,12]. In addition, the evolution of raster display technology to higher resolutions, faster refresh rates, and deeper bit depths will place additional demands on graphics systems. An effective means of generating and composing rendered images from a distributed network of processors is critical to maintaining real-time visualization performance.

This paper presents a scalable architecture for high-speed volumetric rendering of discrete three dimensional data sets. The system employs custom processors for rendering volumetric primitives and a high-performance accumulation network that supports the merging of both volumetric and geometric elements into a composite image. Data access contentions are avoided through an enhanced dual partitioning scheme that enables parallel processing in both object space and image space without the redistribution or replication of primitives. The proposed architecture is intended for rendering data sets comprising 10^7 to 10^9 elements and meets the performance requirements of a full frame interactive image generation system.

An overview of the hybrid rendering architecture with dual object and image space partitions is presented in section II. *bela*, an implementation of the hybrid architecture for volumetric rendering of regularly structured three dimensional data sets, is introduced in section III. A discussion follows of the four main sub-systems in the *bela* architecture, the shading processor, the projection processor, the image assembler, and the hierarchical composition tree. Performance estimates, including latency and storage requirements, to implement a *bela* system are given in section IV. Suggestions for future enhancements are presented in section V along with a summary of the *bela_1* prototype currently under development. The structure and benefits of the hybrid rendering architecture are recapitulated in the final section.

II. OVERVIEW OF HYBRID RENDERING ARCHITECTURE

The rendering task can be partitioned into parallel object space or image space processes depending on whether concurrent jobs are operating on the object space primitives or

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

the image space viewing plane. The success of either scheme depends on a particular implementation's prowess to minimize object/image access contentions between the multiple processors. Object space partitioning distributes the primitives among multiple processors. Each processor renders its assigned primitives that contribute to the final image. Memory contentions arise in image space as multiple processors attempt to access the image plane simultaneously. Image space partitioning distributes the image plane among multiple rendering processors. Each processor renders the primitives which contributes to its assigned portion of the image plane. Memory contentions arise in object space as multiple processors attempt to access the object primitives simultaneously.

Either approach suffers load imbalance from idle processors due to strict front-to-back (FTB) or back-to-front (BTF) primitive rendering sequence for maintaining spatial coherence. Dynamic allocation of the object primitives can mitigate these disadvantages by aligning processors to non-occluding tiles of the image plane. However, this "reallocation of resources" requires either replication of the object primitives for each processor or a redistribution of primitives prior to rendering. Neither approach is acceptable for scenes comprising millions of primitives as the former has extensive memory requirements while the latter increases latency and image generation time.

A rendering engine for visualizing volumetric or geometric primitives can be constructed using a dual partitioning system—a hybrid partition. Both object space and image space processing are combined into a network of rendering and compositing processors as outlined in Figure 1. This hybrid architecture uses "slice-based" image composition to achieve integration of volumetric and geometric objects. The supposition is that the generation of an image can be accomplished through the accumulation of numerous individually rendered point (or *atomic*) primitives. Point primitives produce point images; point images are then accumulated into slice images; slice images are accumulated into slab images; and slab images are accumulated into the final image. Object space partitioning gathers spatially connected primitives into sub-cubes and distributes these groupings among multiple object space processors. Each processor renders the data in its assigned sub-cubes at screen resolution. Primitives within a sub-cube are processed in orthogonal planes "most parallel" to the viewing plane. This yields a series of sub-cube images representing an intermediate view of object space from image space. Image space partitioning gathers adjacent viewing plane pixels into sub-screens and distributes these groupings among multiple image space processors. Each processor composites a set of parallel sub-screen images representing slices of the primitive data set "most parallel" to the viewing plane. The parallel images are produced by combining relevant portions of the sub-cube images generated by the object space processors in a process termed *image assembly*. The use of intermediate slice images eliminates memory contentions between object to image space data transfer. By keeping the point images smaller than the slice images, no contention arises in the image assembler interface.

A simplified 2D example of the hybrid partition in object space is given in Figure 2a. Thirty six primitives are grouped into four sub-cubes, c_0 to c_3. Processing order is determined by taking the dot product of the image plane axis x_i with the object space axes x_o and y_o. In this example, x_i is "more parallel" to x_o than y_o, hence slices s_0 to s_5 are defined parallel to the x_o axis. Image space processing for this example is given in Figure 2b. The viewing plane is divided into four sub-screens, each of which receives sub-screen images from two parallel image planes p_0 and p_1. The parallel image planes receive sub-cube images from object space such that, for the given orientation, c_0 and c_1 supply p_0 , while c_2 and c_3 supply p_1.

The object space processors produce fully rendered images in RGBA format [18] representing a slice of the data. Since slice planes are parallel amongst the object space processors, each slice image contains an inherent "depth" value. The rendered slice images are passed to the composition network which accumulates the slices in the correct FTB or BTF order for each pixel to produce the correct image. The composition network is unaware of the source of the slice images, thus it accumulates both volumetric and geometric primitives equally. Data reallocation or replication is unnecessary since the correct ordering of rendered primitives is removed from the object space partition to the image assembly and composition partitions. In addition, load imbalance in object space is eliminated by providing multiple parallel slice planes for intermediate images and load imbalance in image space by providing dedicated parallel compositing planes for each image space processor. From software simulations, slice-based image composition does not produce errors beyond those encountered with traditional rendering techniques, if done correctly.

The scalability of the proposed architecture is linear in both object and image space. Arbitrarily large data sets are accommodated by increasing the number of object space processors to achieve the desired frame rate. The frame rate also governs the number of image space processors required to achieve a given image size. Thus the architecture is largely "technology-driven" in that limitations in object and image size will be determined by data access times and the available system bandwidth between processors. In the following sections a design example for real-time volume rendering using the hybrid architecture is presented.

III. Volume Rendering of Regularly Structured Grids

The *bela* architecture is an implementation of the slice-based hybrid partitioning scheme presented in the previous section. *bela* is intended for the rapid image generation of regularly structured three dimensional data sets. The architecture is optimized for rendering data sets comprising 10^7 to 10^9 elements to frame buffers of 1024^2 pixels and larger. A *bela* system comprises three major components (Figure 3): object space shading and projection processors for slice image generation, an image composition network with integrated image assembler to accumulate the slice images, and a host computer. The generation of slice images is handled differently depending on the source of the primitive data. Slice image generation of geometric data or *discretized geometry* [9] is beyond the scope of this paper; the focus is on volumetric rendering of discrete three dimensional data sets.

Numerous techniques for generating images from voxel data are applicable in the hybrid partition architecture including: ray casting [12,20,24], energy projection (*splatting*) [25], polyhedral decomposition [21,26], and view transformation [2,10]. *bela* implements voxel rendering via energy projection since, for regular grids, it can be largely table driven leading to significantly fewer calculations. In addition, no data interpolation (or resampling) is required; only the original voxel data is rendered—and all of it—leading to fewer artifacts. As well, rendering time with energy projection is constant for a given data set regardless of orientation and image size with fine retention of detail in magnified views.

The object space processors generate images at screen resolution. This entails two operations: shading and projection. Although the voxel images could be generated independently of the screen resolution to a neutral plane, this would introduce an additional resampling/quantizing stage in the image assembler potentially leading to more artifacts. The sub-cube is logically partitioned into several *sub-cube image planes*, each of which is dynamically allocated according to the viewing orientation. The voxel data is scanned in a front-to-back sequence within each sub-cube image plane and alternately between planes. Voxels are passed to the shading processor for conversion from raw discrete data to an RGBA quadruple. These shaded voxels are then projected to local image planes at screen resolution via a look-up-table mechanism. Object space processors generate parallel planes of voxel images, pre-sorted into sub-cube image planes and correctly ordered from front-to-back.

The image space partition consists of image assemblers and a hierarchical tree structure of compositors. After completion of an entire slice rendering, the image assemblers accumulate the voxel images, and any rendered geometric primitives, into slice images. The slice images—though not explicitly generated—are directly composited onto parallel *slab buffers* which effectively accumulate all primitives rendered between slab planes. Successful rendering of all primitives in the scene signals the hierarchical composition tree to accumulate the slab images onto the output image buffer.

The *bela* architecture requires a host computer to perform a number of non time-critical functions as well as to provide a user interface to the rendering process. The host also serves as the destination for the 24-bit colour image produced by the image space processors. The host has read/write access to the voxel primitives for maintaining the contents of the data set and configuring the voxel *tags*. Tags are necessary to differentiate voxels via their spatial location in the data set (a local classification), rather than on their raw voxel value (a global classification). The host will typically perform a 3D edge detection [6] or region-growing operation [1] on the voxel data set to define structures not adequately segmented via cutting planes or global classification. Tags are used by the shading processor to apply different rendering parameters to the segmented structures. The host sends info-packets to the object space processors and image assembler indicating changes to the current viewing parameters as set by the operator. Packet size is small, less than 128 bytes for changes to the viewing orientation, and less than 1024 bytes to reload a look-up-table. The host also loads the rendering code into the shading processor. With the compute-intensive rendering task transferred to the hybrid partition, the host need not be too powerful and thus inexpensive.

A. Shading Processor

The shading processor is a high performance device optimized for volumetric rendering algorithms (Figure 4). The processor's core is programmable with reconfigurable data paths and an array of multi-purpose computational cells (*CCs*). A very long instruction word (VLIW) format specifies the operation performed by each CC and governs the flow of intermediate data between CCs. Shader programs are typically very small (less than 10 instructions) and have minimal branching and comparison operations. Both super-scalar and super-pipelined methodologies are exploited to maintain shading at the maximum rate of data retrieval from the voxel volume. The host administered look-up-tables designate unshaded RGBA values according to tag and voxel value. Light source vectors, lighting coefficients, and other shading parameters are also accessible by CCs via the reconfigurable data paths. A small register file with programmable levels of delay latches are provided to store intermediate CC results.

Shading algorithms typically comprise the following operations: gradient calculation, dot product, scaling, absolute value, maximum/minimum, power functions, and linear mixing. The basic implementation of these functions are decomposed into multiply and accumulate operations that lend themselves to a regular macrocell layout strategy which simplifies the design of a large CC shading sub-system. RAM access times are typically the performance limiting factor in volume rendering architectures [5,11]. The shading processor's on-chip execution rate is much higher than the external data rate to minimize the number of CCs required to maintain complex shading algorithms at the maximum voxel retrieval rate. Such super-pipelined implementations maximize CC performance with only a modest increase in circuit complexity [15]. External to internal access time ratios of 10:1 are easily achievable with readily available technologies [22] while even greater performance is possible with low voltage sub-micron technologies [23].

Memory skewing increases performance through parallelism [8,10,11]. The voxel sub-cube is stored in a skewed memory format to achieve parallel access to arbitrary planar or cubic groupings of connected voxels, termed *sub-cells*. Sub-cell access is key to preserving voxel data flow through the shading processor when local pseudo-surface information is required by the rendering algorithm. The skewed memory format permits parallel access of a voxel and its immediate neighbours for gradient generation. A dedicated off-chip processor comprising voxel data substitution tables and a normalized vector and magnitude generator provides uninterrupted pseudo-surface generation in tandem with voxel shading. This skewed memory format requires numerous small capacity devices that evidently leads to higher performance, although much wider data paths. The data flow through a shading processor configured for surface enhancement via local gradient magnitude [12] with ambient, diffuse, and specular lighting models [3] is given in Figure 5.

Numerous high speed data paths reside in the shader to transfer intermediate results between CCs, registers, and look-up-tables. These data paths are effected as narrow fixed-point vectors for design simplicity. Narrow data paths do not impair the quality of the generated images since the output of the volume rendering engine is a reconstructed image, an approximation to reality, using rendering parameters that are determined arbitrarily by the user. Pixel RGB errors less than 2% are difficult to detect with the human eye. The narrow data paths also enable a greater number of denser CCs to be constructed on a given die area which increases shader performance while reducing the number of chips needed to produce a shading processor. The specific bit-precision supported by a shading processor to eliminate visible errors is dependent on the resolution of the voxel data, the minimum usable level of transparency, and the number of composites performed per pixel in the image composition network. Therefore bit-precision must be evaluated and defined on an application specific basis prior to shader design.

B. Projection Processor

The projection processor maps the shaded RGBA voxel to image space and generates an energy footprint, or texture map, indicating the density of energy distributed over a range of pixels. The contents of the energy footprint depend on the function used for the convolution kernel, the distribution width of the kernel, the voxel's *centre–of–projection*, and the ratio of inter-voxel spacing to pixel size. The distribution width determines the size of the volume encompassed by the convolution kernel over which the energy is spread in object space. Generating an energy footprint requires projecting a voxel's energy distribution onto the viewing plane and integrating the projection across the bounds of each pixel that falls under it. This involves a triple integration of the convolution kernel that is approximated by quantizing the voxel's centre–of–projection and indexing into a precomputed look-up table. The basic procedure is outlined in Figure 6.

Aliasing artifacts from the sampling process are largely eliminated by super-sampling the energy projection. Subdividing each pixel into four sub-pixels results in four possible quantized locations for a voxel's centre–of–projection. In Figure 6, the voxel's quantized centre–of–projection resides in sub-pixel #1 which retrieves the corresponding energy footprint from look-up table #1. A 3x3 pixel region, termed the *extent width*, is needed to cover all four possible locations for the centre–of–projection. By limiting extent widths to odd values, a voxel's centre–of–projection is guaranteed to lie in the centre of the energy footprint which simplifies the look-up process. Note that although there are four look-up tables each containing nine entries, there can be only six unique values due to symmetry. This greatly reduces the size of the look-up-tables.

The host creates the energy footprint from the current viewing parameters assuming isometric spacing of the voxel volume elements. This simplifies the generation of energy footprints since equi-spaced elements imply a rotationally invariant system. All distributions are spherical in object space and always project to a circle in image space, regardless of the viewing position. To support data sets with unequal sample spacing, hardware support for spherical to elliptical projections is provided via affine transformations to the projected kernel [7]. By providing transformation and projection on a voxel-by-voxel basis for later assembly, projected energy overlap errors are avoided [17].

Prior to projection, the shaded RGBA voxel is "normalized" for image composition by pre-multiplying the RGB components by the opacity term A [18]. The energy footprint is then retrieved from the look-up-tables and the voxel image is generated to separate RGB planes in parallel. Due to the interleaving of voxels within sub-cube image planes, voxel image generation can occur over several voxel accesses, if required by the projection hardware.

C. Image Assembler

The image assembler is essentially a "smart" compositor whose fundamental operation is to align the voxel images generated by the projection processor and composite them in the correct front-to-back order onto a slab buffer. Image assembly can be performed in either image space scan-line order or object space primitive order. Scan-line order is inefficient in that a large number of image space pixels are typically unaffected by the rendering of individual slices through an object. Primitive order, as implemented in *bela*, focuses on the pixels that may, potentially, be affected by the current slice rendering and is accommodating to the parallel projection technique presented in the previous section.

An array of image assemblers are dynamically distributed across the viewing plane as shown in Figure 7. Each assembler is responsible for the correct accumulation of primitives in its assigned *sub-screen*. Since a full slice of rendered primitives can be mapped to a small grouping of sub-screens, image assemblers must be designed to handle this worst case scenario otherwise the rendering pipeline will stall. The use of multiple sub-cube image planes mitigates this performance bottleneck by routing rendered slices to alternating image assembly planes. As well, sub-screens are assigned to alternate assemblers within image assembly planes to equalize the load. Within each sub-screen, voxel images are retrieved from the appropriate sub-cube partition and composited in parallel. Sub-screen pixels are interleaved in a planar configuration similar to the voxel volume elements to enable parallel access. Since voxel image access is directed from image space processors, no image space data contention is introduced. Object space contention is avoided by the virtual extension of "null" voxel images across the viewing plane. Voxel images completely internal to a sub-screen need only be accessed once. Voxel images that cross sub-screen boundaries cannot occupy the same relative spatial location across sub-screens. Since front-to-back processing is spatially ordered, voxel images are thus independent across sub-screens and will not normally be accessed simultaneously from neighbouring sub-screens.

Image assembly is computationally intensive requiring $[\text{extent width}]^2$ compositions per voxel. To maintain synchronization with object space processing, each assembler consists of an array of dedicated compositing cells with on-chip storage. This results in large bandwidth requirements for the image assembler but is not uncommon for image composition systems [19]. The on-chip compositor storage functions as the slab image accumulator. Upon completion of each slab, the contents are available for readout on a separate "pixel addressable" port by the hierarchical composition tree.

D. Hierarchical Composition Tree

The final stage in the *bela* architecture is a hierarchical composition tree to combine the slab images generated by the image assemblers into the final image. Both parallel plane and tree configurations of compositors require N-1 compositors to accumulate N image planes. Tree structures are preferred over parallel planes as each image plane encounters $\log_2 N$ compositions to reach the final image which can lead to fewer round-off errors in narrow fixed-point implementations. The slab image accumulator and composition tree support image space partitioning similar to that used in the image assembler. In most cases, such partitioning is not required since composition of full frame images (1024^2 pixels) at 30 Hz is readily attainable with custom hardware (see section V).

The tree depth is determined by the total number of sub-cube image planes supported. RGBA pixels are read from the slice buffers—in scan-line order—into the pipelined composition tree to yield one RGBA output pixel per cycle. The final image is then passed through two additional compositors to provide an overlay and underlay. The overlay is typically a transparent surface while the underlay is usually an opaque black background. The most significant eight bits of each RGB channel are stored in a conventional frame buffer for manipulation and display by the host.

IV. Performance & Resource Estimates

The separation of object image and image space in the *bela* architecture enable processors at each level to function at optimum speed. In most circumstances, the maximum rate is determined by memory access times and the available system bandwidth. Once the critical timing constraint is identified in each stage, the various processors are then synchronized to maintain a constant frame rate. Throughout this discussion, parallel data transfer is assumed unless otherwise noted. While this assumption often requires excessive data lines between processors, it nevertheless serves to indicate an absolute upper performance limit.

The maximum time intervals to shade a voxel (T_S), project a pixel (T_P), assemble a projection element (T_A), and composite a pixel (T_C) are given by:

$$T_S = \frac{n^3}{N^3 f} \qquad T_P = \frac{n^3 p v_p^2}{N^3 f w^2}$$

$$T_A = \frac{i^2 p v_A^2 n}{N^3 f w^2} \qquad T_C = \frac{m^2}{M^2 f}$$

where

$w =$ extent width $= z[\frac{2kM}{N}]+1$

$p =$ sub cube image planes

$v_p =$ voxel projection parallelism, $M =$ image size

$v_A =$ voxel assembler parallelism, $N =$ volume size

$n =$ object space partition, $z =$ magnification

$i =$ image assembler partition, $f =$ frame rate

$m =$ image space partition, $k =$ distribution width

For simplicity, the voxel volume is assumed to be cubic, although not a requirement. In the ideal case, the voxel projection parallelism is equal to the extent width, *i.e.*. the entire projection is processed in parallel. Similarly, if the voxel assembler parallelism is equal to the extent width, the entire voxel image is composited in parallel. A magnification level of 1.0 sets the inter-voxel to pixel size ratio such that the rendered volume completely fills the image plane. Simulations indicate that distribution widths of 1.0 to 1.4 are suitable for most convolution kernels.

Equally important to frame rate is system latency; the time interval between when the operator first makes a change to the viewing parameters and when the image is finally updated to reflect that change. Assuming a system designed such that all processes are synchronized without stalling the rendering pipeline, minimum latency (L_{MIN}) in *bela* is dependent on: the reaction time of the host to inform the system of an update (T_U), the time to shade a sub-cube, the time to project a voxel, the time to assemble a slice image, and the time to composite the slab buffers.

$$L_{MIN} = T_U + T_S\frac{N^3}{n^3} + T_P + T_A\frac{N^2 w^2}{i^2 v_c^2} + T_C[\frac{M^2}{m^2} + \log_2 pn]$$

The dominant terms are volume shading and slab buffer composition. At best, minimum latency is slightly more than one frame if fast compositors with large m are used, and two frames at worst.

Volumetric rendering via image composition demands extensive memory resources for storage of the voxel volume and the subsequent voxel images. Ignoring shader and projector look-up-tables (considered diminutive by comparison), memory requirements, in bits, for the volume storage (M_S), voxel images (M_P), slab buffers (M_A), and frame buffer (M_C) are estimated by:

$$M_S = N^3 b_{VT} \qquad M_P = dpnN^2 w^2 b_{RGBA}$$

$$M_A = dpnM^2 b_{RGBA} \qquad M_C = dM^2 b_{FB}$$

where

$b_{VT} =$ bits per voxel and tag

$b_{RGBA} =$ bits per RGBA component

$b_{FB} =$ bits per frame buffer pixel

$d = 2$ if double buffered, otherwise 1

Single buffering voxel images and slab buffers almost halves the storage requirements but significantly reduces the frame rate while processors idle.

From the above discussion and formulae, the performance of *bela* systems can be estimated. Given a voxel data set with N=256, a desired image size of M=512, and using 40 ns RAM, a frame rate of 12 Hz is achieved with eight shading processors, eight projection processors, eight image assemblers, and one hierarchical compositor. System latency is estimated at 95 ms, or 1.13 frames. In terms of storage, with double buffered 16 bit precision, 24 MBytes are needed for the voxel volume (four bit tags), 50 MBytes for the voxel images (w=5), 8 MBytes for the slab buffers, and 1.5 MBytes for the frame buffer. A data set with N=1024 and a desired image size of M=1024 rendered at the same frame rate with 40 ns RAM requires considerably more resources. Since object space has increased sixty four times, the number of shading and projection processors increases accordingly to 512. Similarly, the number of image assemblers increases to 512 despite image space only quadrupling in size. Seven compositors are required in the hierarchical composition tree to accumulate eight slab buffers; no image space partitioning is required. System latency is estimated at 126 ms or 1.5 frames. Storage increases to 1536 MBytes for the voxel volume, 1152 MBytes for the voxel images (w=3), 128 MBytes for slab buffers, and 6 MBytes for the frame buffer. The assembly of a system with these processing and storage requirements represents the limit of what can reasonably be achieved with the *bela* architecture.

V. THE FUTURE

The *bela* architecture is continually evolving to increase functionality and frame rate. Enhancements currently under investigation include: object space support for irregular grids, sparse grids, perspective projection, and hardware for slice image generation of geometric primitives. The implementation of a "smarter" scan-line ordered image assembler that avoids compositing transparent pixels onto the slab buffer and skips over slab buffer pixels that reach an opacity threshold [14] is also under consideration.

A prototype system, *bela_1*, based on the N=256 example presented in section IV is currently under construction. Extensive bit level software simulations have been completed to verify the correct operation of the hybrid architecture and to determine processor bit precision to eliminate visible artifacts from fixed-point round-off error. A general purpose 16-bit compositing cell, *Compose16*, has been designed in 1.2μ CMOS and is undergoing fabrication. Compose16 achieves a non-pipelined composition in 8.5 ns, on-chip, and will be used in the hierarchical composition tree. The shading processor, *Shade16*, is being designed in a 0.8μ BiCMOS technology to be fabricated in the fall of 1994. An on-chip execution rate of

200 MHz provides 25 million shaded voxels per second, according to the algorithm in Figure 5. With simpler algorithms, up to 50 million voxels can be shaded per second. Design and fabrication of the image assembler and projection processor will follow completion of the shading processor. Expected completion date of the prototype is spring 1995.

VI. Conclusions

The scalable *bela* architecture for high-speed volumetric rendering of discrete three dimensional data sets was presented. *bela* comprises custom processors for rendering volumetric primitives and a high-performance composition network for merging volumetric and geometric elements. Data access contentions are avoided through an enhanced dual partitioning scheme that enables parallel processing in both object space and image space without the redistribution or replication of primitives. Object space parallelism and versatility is achieved via allocation of primitives to programmable shading processors. Primitive projections are spatially ordered for efficient slice-based accumulation via the image assembler and hierarchical composition tree.

Image composition networks permit a high degree of parallel processing to achieve interactive volume rendering generation at the expense of memory resources and system bandwidth. The proposed architecture attempts to allay these limiting criteria through hierarchical accumulation of rendered primitives. *bela* is suitable for rendering data sets comprising 10^7 to 10^9 elements at interactive rates.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council (NSERC), the Alberta Microelectronic Centre (AMC), and the Canadian Microelectronics Corporation (CMC).

References

[1] M.F. Cohen, J. Painter, M. Mehta, and K.L. Ma, "Volume Seedlings," in *Computer Graphics Special Issue on 1992 Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, 139-145, 1992.

[2] R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics*, vol. 22, no. 4, 65-74, August 1988.

[3] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics, Principles and Practice*. Addison-Wesley, 1990.

[4] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, vol. 23, no. 3, 79-88, July 1989.

[5] S.M. Goldwasser and R.A. Reynolds, "Real-Time Display and Manipulation of 3-D Medical Objects: The Voxel Processor Architecture," *Computer Vision, Graphics, and Image Processing*, vol. 39, 1-27, 1987.

[6] K.H. Höhne, M. Bomans, A. Pommert, M. Riemer, C. Shiers, U. Tiede, and G. Wiebecke, "3D Visualization of Tomographic Volume Data Using the Generalized Voxel Model," *The Visual Computer*, vol. 6, no. 2, 28-36, Feb 1990.

[7] P. Hanrahan, "Three-Pass Affine Transforms for Volume Rendering," *Computer Graphics*, vol. 24, no. 5, 71-78, November 1990.

[8] D. Jackel, "Reconstructing Solids from Tomographic Scans, The PARCUM II System," in *Advances in Computer Graphics Hardware II*, 101-109, 1988.

[9] A. Kaufman and E. Shimony, "3D Scan-Conversion Algorithms for Voxel-Based Graphics," in *ACM Workshop on Interactive 3D Graphics*, 45-76, 1986.

[10] A. Kaufman and R. Bakalash, "Memory and Processing Architecture for 3D Voxel-Based Imagery," *IEEE Computer Graphics and Applications*, vol. 8, 10-23, November 1988.

[11] G. Knittel, "Verve: Voxel Engine for Real-time Visualization and Examination," *Computer Graphics Forum*, vol. 12, no. 3, C37-C48, 1993, Proceedings of Eurographics '93.

[12] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, 29-37, May 1988.

[13] M. Levoy, "Design for a Real-Time High-Quality Volume Rendering Workstation," in *Chapel Hill Workshop on Volume Visualization*, 85-92, 1989.

[14] M. Levoy, "Efficient Ray Tracing of Volume Data," *ACM Transactions on Graphics*, vol. 9, no. 3, 245-261, July 1990.

[15] F. Lu and H. Samueli, "A 200-MHz CMOS Pipelined Multiplier-Accumulator Using a Quasi-Domino Dynamic Full-Adder Cell Design," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 2, 123-132, February 1993.

[16] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics*, vol. 26, no. 2, 231-240, July 1992.

[17] U. Neumann, "Interactive Volume Rendering on a Multicomputer," in *Computer Graphics Special Issue on 1992 Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, 87-93, 1992.

[18] T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics*, vol. 18, no. 3, 253-260, July 1984.

[19] J. Poulton, J. Eyles, S. Molnar, and H. Fuchs, "Breaking the Frame-Buffer Bottleneck with Logic-Enhanced Memories," *IEEE Computer Graphics and Applications*, vol. 12, no. 6, 65-74, November 1992.

[20] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Computer Graphics*, vol. 22, no. 4, 51-58, August 1988.

[21] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics*, vol. 24, no. 5, 60-70, November 1990.

[22] D. Somasekhar and V. Visvanathan, "A 230-MHz Half-Bit Level Pipelined Multiplier Using True Single-Phase Clocking," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 4, 415-422, December 1993.

[23] M. Suzuki, N. Ohkubo, T. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome, "A 1.5-ns 32-b CMOS ALU in Double Pass-Transistor Logic," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 11, 1145-1151, November 1993.

[24] C. Upson and M. Keeler, "V-BUFFER: Visible Volume Rendering," *Computer Graphics*, vol. 22, no. 4, 59-64, August 1988.

[25] L. Westover, "Footprint Evaluation for Volume Rendering," *Computer Graphics*, vol. 24, no. 4, 367-376, August 1990.

[26] J. Wilhelms and A. van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics*, vol. 25, no. 4, 275-284, July 1991.

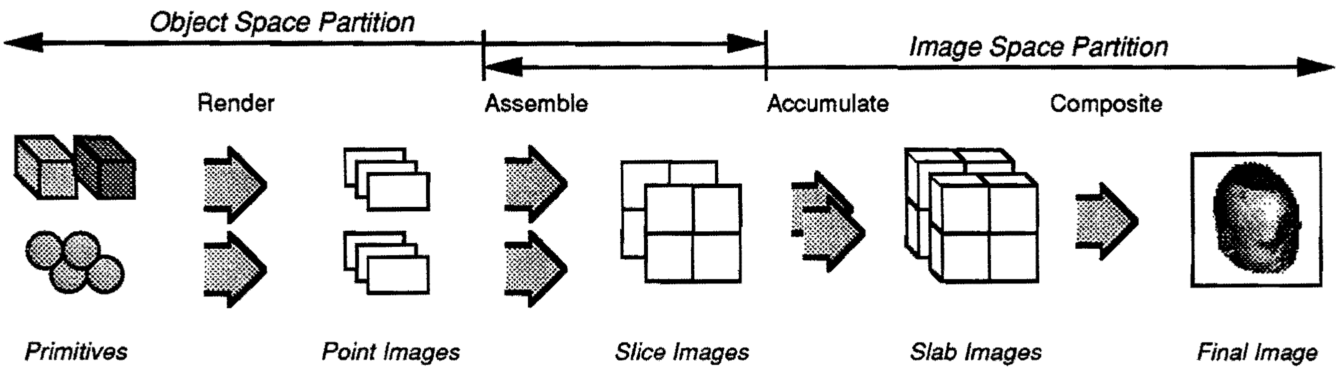

Figure 1. Hybrid Partition for Slice-Based Rendering

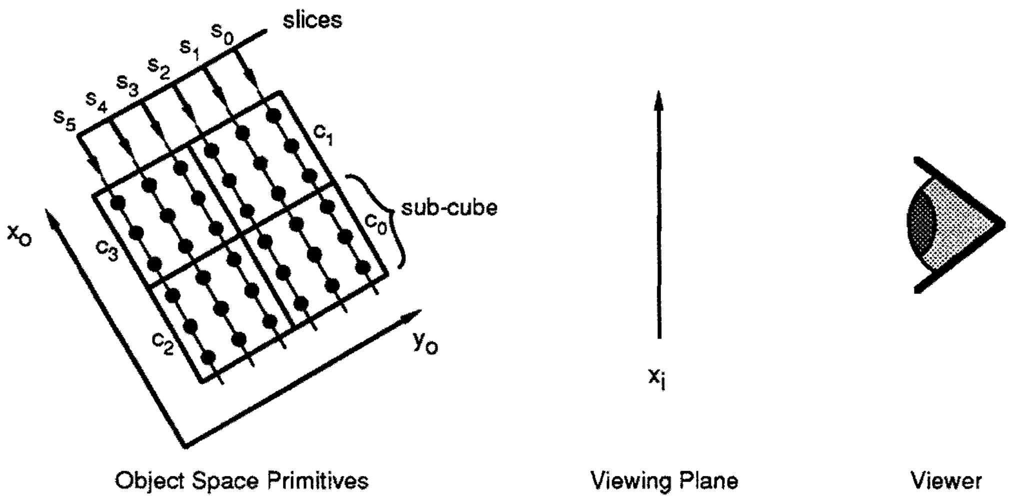

Figure 2a. Simplified 2D Example of the Hybrid Partition in Object Space

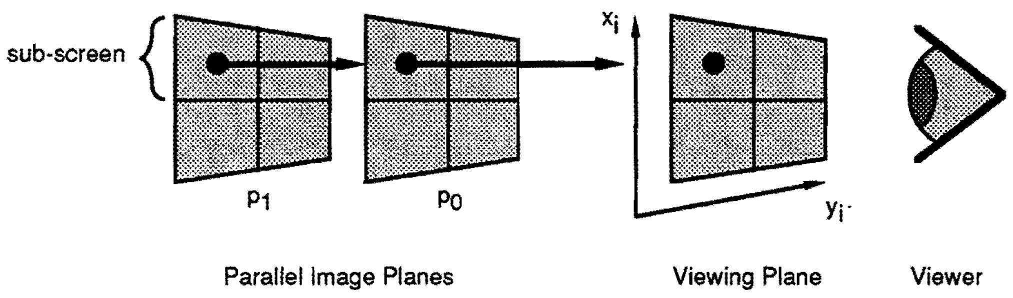

Figure 2b. Example of the Hybrid Partition in Image Space

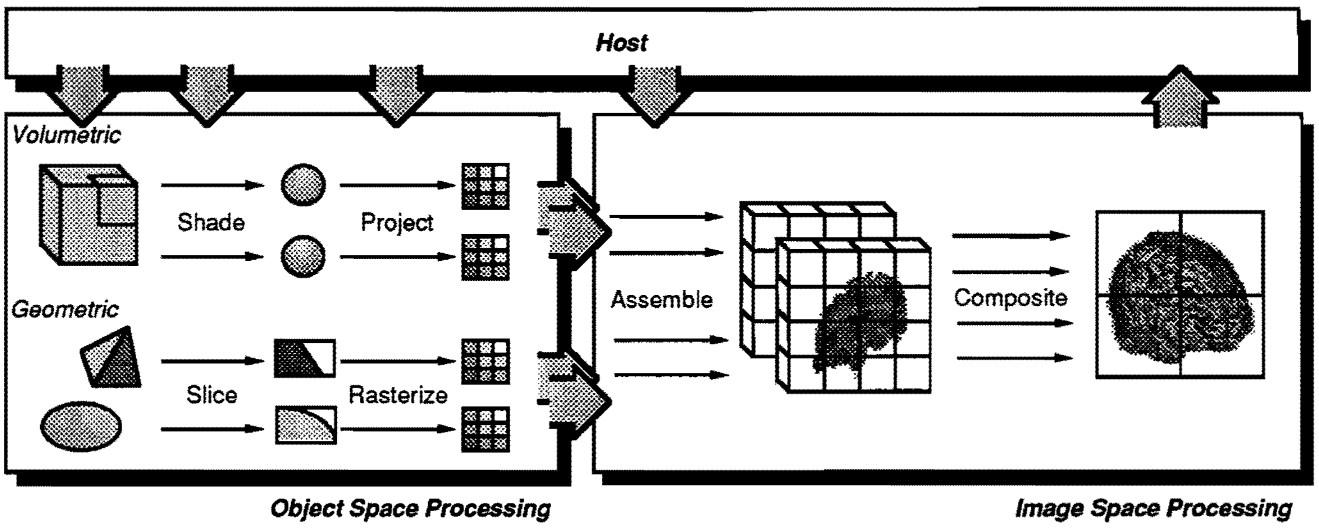

Figure 3. *bela* Architectural Overview

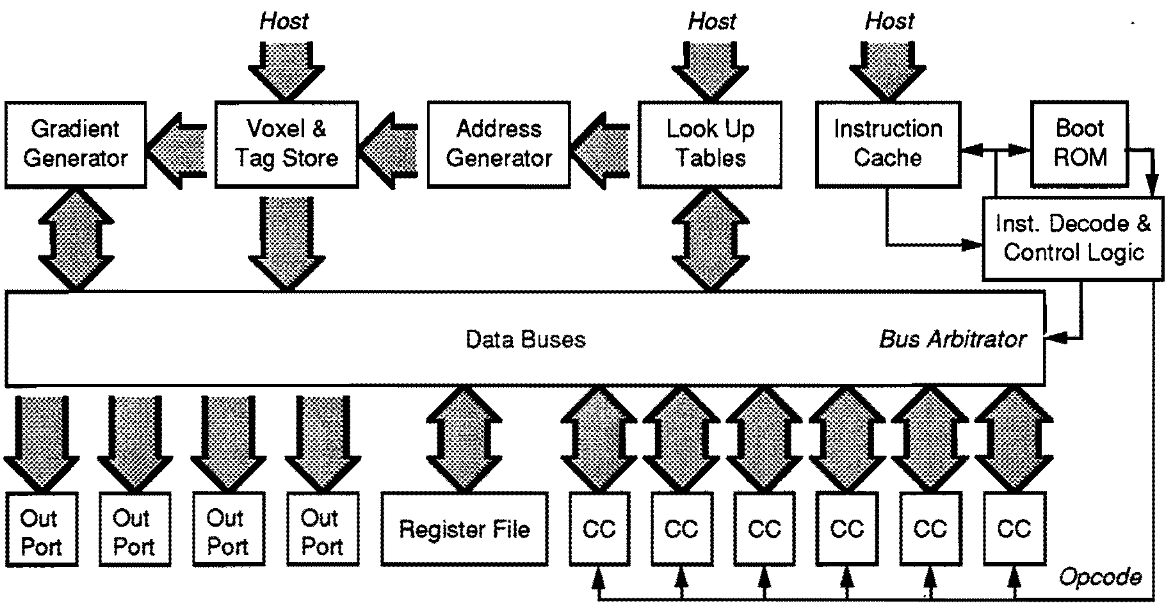

Figure 4. Shading Processor

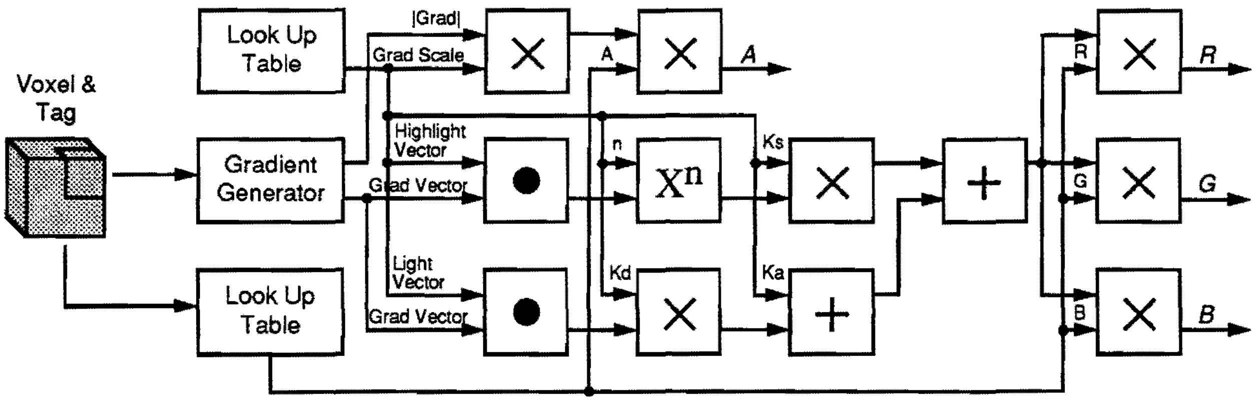

Figure 5. Shader Data Flow Example

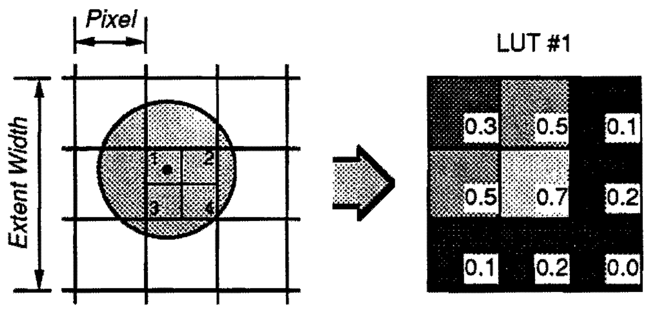

Figure 6. Quantized Centre-of-Projection

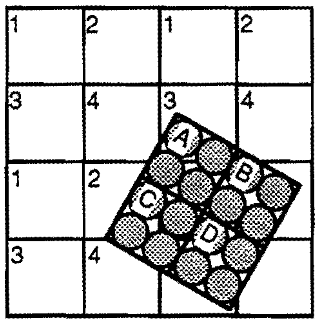

Figure 7. Image Assembler
(Sub-Screens = 1234, voxel FTB order = ABCD)