A PCI-based Volume Rendering Accelerator

Günter Knittel
WSI / GRIS †
University of Tübingen, Germany

Abstract

We discuss the underlying algorithms, design principles and implementation issues of an extremely compact and cost-efficient volume rendering accelerator for PCI-based systems. It operates on classified and shaded data sets which have been coded and compressed using *Redundant Block Compression* (RBC), a technique originating from 2D-imaging and extended to 3D. This specific encoding scheme reduces drastically the required data traffic between the volume memory and the processing units. Thus, the volume data set can be placed into the main memory of the host, eliminating the need of a separate volume memory. Furthermore, the tri-linear interpolation needed for perspective raycasting is very much simplified for RBC-transformed data sets.

All in all, these techniques allow a volume rendering accelerator to be implemented as a single-chip coprocessor, or as an FPGA-based prototype for monochrome data sets as presented in this work. Although using a lossy compression scheme, image quality is still high, and expected frame rates are between 2 and 5Hz for typical data sets of 256^3 voxels.

Keywords: graphics hardware, volume rendering, ray casting, data compression

1 Motivation

Real-time volume rendering requires such a high computing power that it can only be achieved on supercomputers, high-end workstations, workstation networks or special-purpose hardware. The high computing requirements stem from the algorithmic complexity of volume visualization and the sheer amount of data to be processed. The visualization of volume data sets typically involves the segmentation, or classification of structures of interest, and their meaningful display on the screen. In a raycasting pipeline, this can easily turn into 100 operations per raypoint, giving 16GOPS for 256^3 data sets to be rendered at 10Hz. Depending on the chosen algorithm and the machine architecture, the needed memory bandwidth can also reach the GByte/s-range.

However, most potential users are not in reach of a supercomputer or a high-end workstation. Workstation networks most often do not deliver the expected speed due to a limited interconnection bandwidth or heavy loads by other users. Current academic and commercial designs tend to be large, massively parallel architectures (see [9], [12], [14] and various contributions in these proceedings) and thus, tend to be very expensive as well.

To make the benefits of volume rendering available to a wide audience (e.g., in medical education) we must take every effort to reduce the costs of a volume visualization system.

Although highly desirable in general, interactive classification is not needed in a large number of applications, e.g., when the materials of the data set are well-known, or when automatic segmentation is not reliable enough and must therefore be done manually by human specialists (for example, in medical diagnosis).

For the moment, we consider the classification to be a preprocessing step, performed once before a large number of views are produced. For a comprehensive understanding of the data set, however, a number of requirements should be satisfied:

- the users should be placed into a virtual environment, where they can step right through the data set to gain the maximum insight.
- The system must preserve the three-dimensional nature of the data, provide perspective views and retain the depth information.
- If the data set can be classified into different materials, it should be possible to display either one of them or any combination translucently without performing a re-classification.
- Finally, the system must provide high rendering speed and high image quality.

† Universität Tübingen
Wilhelm-Schickard-Institut für Informatik -
Graphisch-Interaktive Systeme (WSI / GRIS)
Auf der Morgenstelle 10, C9
D-72076 Tübingen, Germany
Phone: ..49 7071 29 5463
FAX: ..49 7071 29 5466
email: knittel@gris.informatik.uni-tuebingen.de
www: http://greco.gris.informatik.uni-tuebingen.de/

However, for a broad acceptance on the market, a volume rendering accelerator should not increase the price of a graphics workstation or even a PC significantly.
Our approach to fulfil these contradictory requirements is based on a very simple data encoding scheme, which is discussed in detail in chapter 2. Chapter 3 explains the rendering method. The underlying technologies (the PCI-bus and FPGAs) are introduced in short in chapter 4 and 5, respectively. Implementation issues are discussed in chapter 6. Image quality and expected rendering performance are illustrated in chapter 7.

2 Algorithm

Processing starts with the classification of the data set. The voxels are grouped and tagged according to the material they belong to (e.g., bone and tissue, see [6]). Each material is shaded separately according to whether only its surface should be displayed or its entire region [13]. Thus, we use one opacity transfer function for each material having an upper bound equal to 1. Then the data is encoded and compressed as explained in the following section.
The data set is visualized using the raycasting algorithm. Our method offers arbitrary perspective projections and even walk-throughs. Consequently, the raypoints do not coincide with the grid points and the data set is therefore tri-linearly interpolated at the resample locations. The visual appearance is further improved by performing depth-cueing. The material tags in conjunction with user-supplied transparency parameters allow us to display a given structure exclusively or to blend different materials during rendering.

2.1 Compression and redundant Coding

The compression algorithms are borrowed from 2D image processing [4],[5],[10] and are briefly reviewed for completeness.
Given an 8-bit grey scale picture, the image is divided into 4×4 pixel blocks in which the pixels are grouped according to whether their greyvalue is above (or equal to) or below the average greyvalue η of the entire block. The result of this operation is a 16 bit decision vector D for each block. For each group of "lower" and "upper" pixels new greyvalues a and b are computed such that the block mean η and the variance σ^2 are preserved. Given q and p as the number of pixels above and below the block mean, respectively, a and b are computed by:

$$a = \eta - \sigma \cdot \sqrt{q/p} \quad \text{and} \quad b = \eta + \sigma \cdot \sqrt{p/q} \qquad (1)$$

The new values a and b are appended to the 16 bit decision vector to form the code element for one 4×4 pixel block. Using 8 bits for both a and b, this scheme achieves a reduction to 2 bits per pixel. The example demonstrated above is a variant of the *block truncation coding* (BTC) algorithm [5]. Note that the decompression is particularly inexpensive: for each pixel in the image to be created, examine the corresponding decision bit and write either a or b.
The reduction to 2 bits per pixel can be maintained for 24-bit RGB pictures using the *color cell compression* (CCC) technique [4]. The decision criterion now is the mean luminance $\bar{Y}$, where

$$Y = 0.3 \cdot R + 0.59 \cdot G + 0.11 \cdot B \qquad (2)$$

The mean values R_u, G_u and B_u, computed from the color components of the upper pixels, are assigned to the upper group. The same is done for the lower pixels. At this time, a 64 bit word $DR_lG_lB_lR_uG_uB_u$ represents a 16 pixel block. In the next step, a look-up table of 256 colors is constructed which best represents the set of colors present in the code elements, for example, by using the median cut algorithm [11]. Each *RGB*-triple is then replaced by a pointer P into that look-up table, so that finally a 4×4 pixel block is compressed into a 32 bit code element DP_lP_u. Optionally, two separate color look-up tables could be used for the upper and lower colors.
The decompression expenses for CCC are slightly increased, since two table look-ups must be performed additionally for each pixel block.
Note that these compression schemes do not achieve the optimal quantization in terms of mean square error or mean absolute error [5], but offer an easy implementation and a high speed.
The application of these compression techniques to volume data sets is straightforward. Since most workstations can handle 32 bit words conveniently, we chose to pack 12 voxels into one 32 bit code element, as shown in Figure 1.

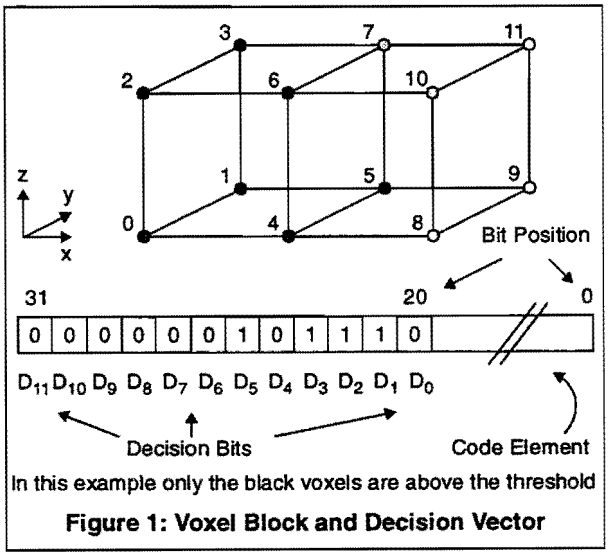


Figure 1: Voxel Block and Decision Vector

The decision vector occupies 12 bits in any given code element. Thus we can spend 10 bits to characterize the upper and the lower group. The actual contents of these bit fields depend on the kind of data set and the visualization method used. For example:

- Blood vessels are commonly visualized from MR data sets using a maximum projection. The resolution of the scanning devices is typically 12 bits. Since only the maximum value along a ray is displayed, one can discard all voxels below a certain threshold during compression without losing relevant information. The remaining voxel values are then quantized in 10 bits, e.g., by a histogram equalization [8].
- For grey-level gradient shading, the upper and lower bit fields hold the emitted light intensity in up to 10 bit resolution.
- For colored gradient shading, pointers into 1K entry color look-up tables can be used. $P_u = 0$ denotes an empty code element, i.e., all voxel values are zero.
- If only 8-bit greyvalues or pointers are used, the remaining bits can be used to group the voxels into different materials. For each code element, the bits are set according to the material tag which occurs the most often.

Consequently, each volume rendering method requires its own coding scheme.

Considering just a single code element, we can see the first major advantage of this method: after performing a single memory access, all voxels needed for tri-linear interpolation are available. To make this true for the entire data set, we have to compress and code all voxels redundantly, as shown in Figure 2. As a consequence, all voxels at positions 0, 1, 2, 3, 8, 9, 10 and 11 in Figure 1 are represented 8 times, and the voxels at position 4, 5, 6 and 7 are stored 4 times in the coded data set. This is why we call this method *Redundant Block Compression* (RBC). Thus, for original data sets of 16 bit voxels, the coded data set has just the same size. To be precise, if the original data set dimensions were $xxyxz$, the coded data set occupies $2xxxyxz$ bytes.

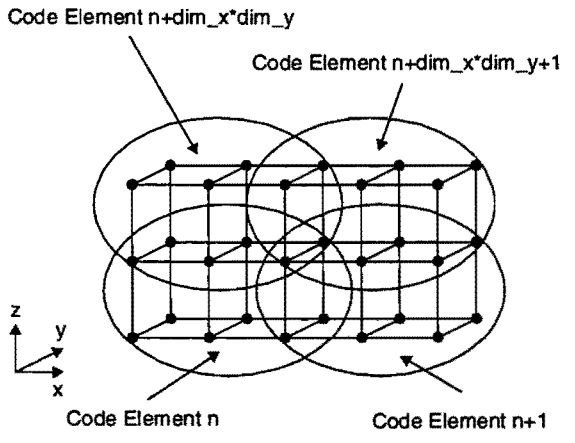


Figure 2: Redundant Block Compression

2.2 Simplified Tri-Linear Interpolation

The second major advantage of this coding scheme is that the tri-linear interpolation is simplified to the largest extent. Let's consider a volume cell with the eight greyvalues $C_0..C_7$ at the corners as shown in Figure 3.

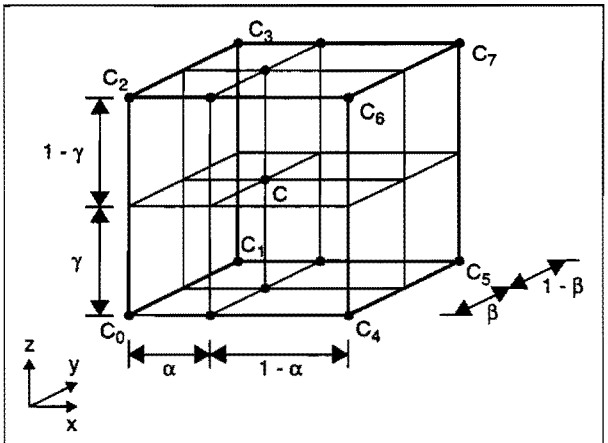


Figure 3: Tri-Linear Interpolation

The desired greyvalue C at the offset (α,β,γ) within the cell is given by:

$$\begin{aligned} C = {} & C_0 \cdot (1-\alpha) \cdot (1-\beta) \cdot (1-\gamma) \\ & + C_1 \cdot (1-\alpha) \cdot \beta \cdot (1-\gamma) \\ & + C_2 \cdot (1-\alpha) \cdot (1-\beta) \cdot \gamma \\ & \quad \ldots \\ & + C_7 \cdot \alpha \cdot \beta \cdot \gamma \end{aligned} \tag{3}$$

Written differently:

$$C = C_0 \cdot \omega_0 + C_1 \cdot \omega_1 + C_2 \cdot \omega_2 + \ldots + C_7 \cdot \omega_7 \tag{4}$$

Since there are only two different greyvalues C_u and C_l in any given volume cell, we can factor out (4) in 256 possible ways:

$$\begin{aligned} C = {} & C_u \cdot (\omega_a + \omega_b + \ldots + \omega_c) \\ & + C_l \cdot (\omega_d + \omega_e + \ldots + \omega_f) \end{aligned} \tag{5}$$

The weightfactors ω_n can be considered as the contents of the subvolumes shown in Figure 4.

Obviously, they sum up to 1. If ω_l is the compound weight for C_l, then

$$C = C_u \cdot (1-\omega_l) + C_l \cdot \omega_l = C_u - \omega_l \cdot (C_u - C_l) \tag{6}$$

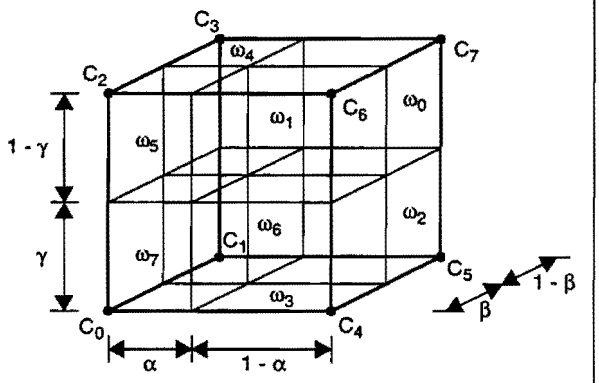


Figure 4: Weightfactors

If all α, β and γ have 4 bit precision, which is sufficient in most practical cases, then any given distribution of upper and lower values in (5) can give 4096 possible values for ω_l. All in all, ω_l depends on an 8 bit decision vector and three 4 bit offsets, giving a total of $IM = 2^{20}$ different configurations. Thus we can easily precompute the weightfactors for each possible configuration and store them in a table. Furthermore, as implied by (6), we do not store C_u and C_l in the code elements, but instead C_u and $(C_u\text{-}C_l)$. Then, a complete tri-linear interpolation is performed by

- assembling the weightfactor address from the decision vector and the offsets,
- one table look-up and
- one multiplication and one subtraction.

For maximum performance, the weightfactor table should be stored in a local memory (e.g., a PROM) placed on the accelerator sub-system.

For colored data sets, the accelerator additionally needs one or optionally two color look-up tables (CLUT). Accesses to the weightfactor table and the CLUT(s) can be done in parallel. The rendering speed is maintained if the accelerator contains separate processing units for each of the color components.

The size of the tables is not critical: if truncated to 8 bits, we need 1MByte for the weights, (available in a single EPROM device) and at most 1.5KBytes for the CLUTs (possibly integrated on the accelerator chip), if 8-bit indices are used.

2.3 Distance Coding

Classified and shaded data sets contain a high percentage of empty voxel blocks, for which $P_u;C_u = 0$. In this case, the remaining bits of the code element are redefined. We define 8 overlapping neighborhood octants (corresponding to the 8 possible orientations of a ray in terms of the sign of its components) and determine for each octant the largest integer radius (in grid units) which can be

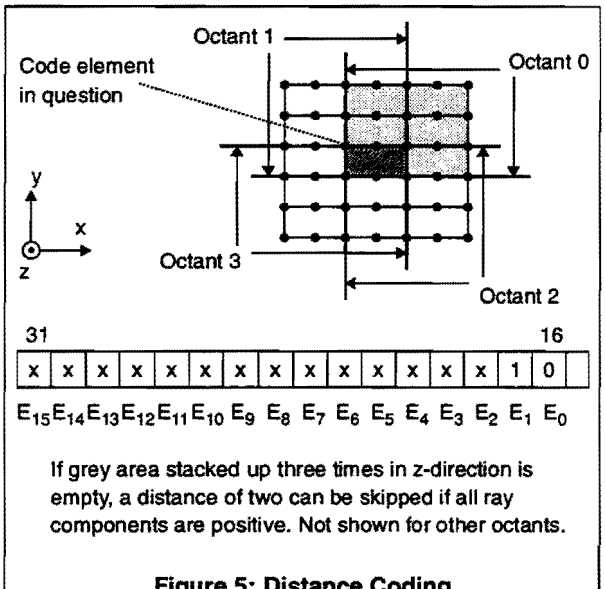


Figure 5: Distance Coding

skipped safely due to an empty neighborhood. See Figure 5 for an explanation in 2D. The largest step can be 8 grid units if C_u or P_u have 8 bits, or 4 units if 10 bits are used for these quantities.

The advantage of this technique is that no separate acceleration data structure (e.g., an octree) is needed, and thus, skipping empty space requires neither additional memory capacity nor bandwidth. The spatial arrangement of objects in the neighborhood of an empty block is described more precisely as in previous work (see [15]), so that an unnecessary reduction of the stepsize in the vicinity of objects that will not be hit occurs less often.

The maximum stepsize of 8 (in grid units) is usually satisfactory except for the regions outside the bounding volume of the objects. Thus, techniques like PARC [3], which use a large set of polygons to better describe the bounding volume of an object can very well complement our method, and will be implemented in a later version.

3 Rendering

In the remainder of this paper we will only consider monochrome data sets, since the accelerator prototype only works on greyvalues.

The coded data set is placed into the main memory of the PCI-host and rendered by the accelerator using perspective raycasting. Computation of camera parameters according to user inputs and ray generation is performed by the host-CPU. After having obtained all ray-parameters from the CPU, the accelerator processes all points of that ray autonomously and returns the results to the CPU or writes the pixel color directly into the frame buffer of the system. The ray-parameters include:

- the physical address of the data set, set up once per session,

- a threshold value, a "landing run," a set of material properties (see section 3.1) and the coordinates of the Manhattan Distance Reference Point (see section 6.3), set up once per frame,
- the coordinates $X.\alpha$, $Y.\beta$, $Z.\gamma$ of the first resample point, the total number of raypoints, the initial attenuation factor f_i and the increments of these quantities, set up once per ray.

Linear depth-cueing [7] is performed according to the nearest and the farthest point of the volume, as depicted in Figure 6:

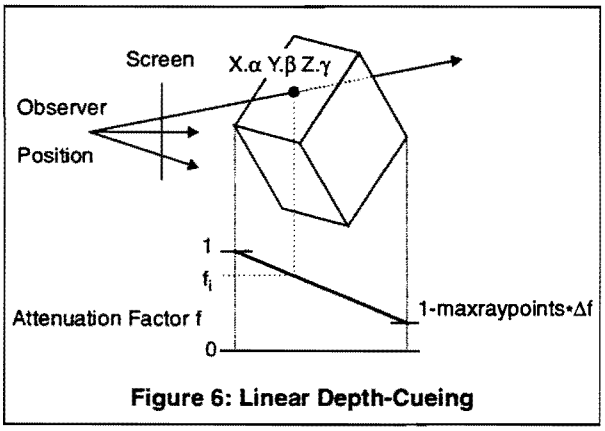


Figure 6: Linear Depth-Cueing

In the following section we will describe the rendering procedure in more detail.

3.1 Rendering Details

After being set up with the ray-parameters, the accelerator computes the physical address of the first raypoint, arbitrates for PCI-ownership and fetches the code element. If the code element is empty, the skip distance within the code element is selected by the value in the ray orientation register. The raypoint counter is decremented and the address of the next raypoint and its depth-cueing factor are calculated according to the skip distance (which is set to zero for non-empty cells). The next memory access is initiated immediately thereafter. Parallel to that, if the volume cell is non-empty, the address of the weightfactor is assembled, and the local look-up-table is accessed. Besides that, the material tag bits within the code element address the material properties RAM, which holds a user-definable property for each specific material. The possible properties are:

- Invalid: The material is currently of no interest, and the raypoint is discarded.
- Opaque: The material is subject to the threshold operator.
- Translucent: The light intensities of all raypoints of that material are accumulated.

As soon as the weightfactor is available, the simplified tri-linear interpolation is performed if the raypoint is valid. The raypoint can still be discarded if the interpolated intensity is below the threshold. In the other case, the light intensity is attenuated according to the associated depth cueing factor.

In case of translucent material, the final value is added to all values of previous points of the same material.

In case of opaque material, a special threshold operator applies. For the first point in opaque material exceeding the threshold, the raypoint counter is set to a user-programmable landing run (or to the actual value, whichever is less), and only the largest value found within that distance will be returned to the host.

In the prototype implementation currently under development, four different materials can be defined, and thus, the accelerator has four accumulators in case all materials are set to translucent. The grey value of an opaque material is stored in a separate register.

Thus, separate frames for each material are produced, from which the final screen is composed by the host-CPU. Note that this is an inexpensive 2D operation, which is performed very quickly. Thus, if the user wishes to blend the structures differently, but the camera parameters haven't changed, no volume rendering operation must be done and the system responds immediately.

A maximum projection is performed by setting all material to opaque and the landing run to infinity.

4 The PCI-Bus

In the PC-market, the PCI-Bus (Peripheral Component Interconnect [1]) has almost completely replaced all existing bus standards within a few months. Besides that, it is more and more becoming a standard in the workstation area as well. The PCI-bus constitutes an acceptable trade-off between low costs and high system speed.

A typical PCI-system is shown in Figure 7. The PCI-bus is a multi-master bus, allowing a device on an expansion card to initiate a transfer while the CPU can potentially continue its work out of the cache.

The PCI-bus is designed for burst transfers, i.e., a large number of data transfers following a single address transfer as shown in Figure 8. For this reason, and to reduce system costs, it has a multiplexed address/data-bus of 32 bits. Both the initiator and the target control a handshake signal (IRDY and TRDY in Figure 8) to insert waitstates if necessary.

All transactions are synchronous to a common clock. Current specifications allow clock rates of up to 33MHz and thus, a 32-bit PCI-system has a theoretical peak transfer rate of 132MByte/s.

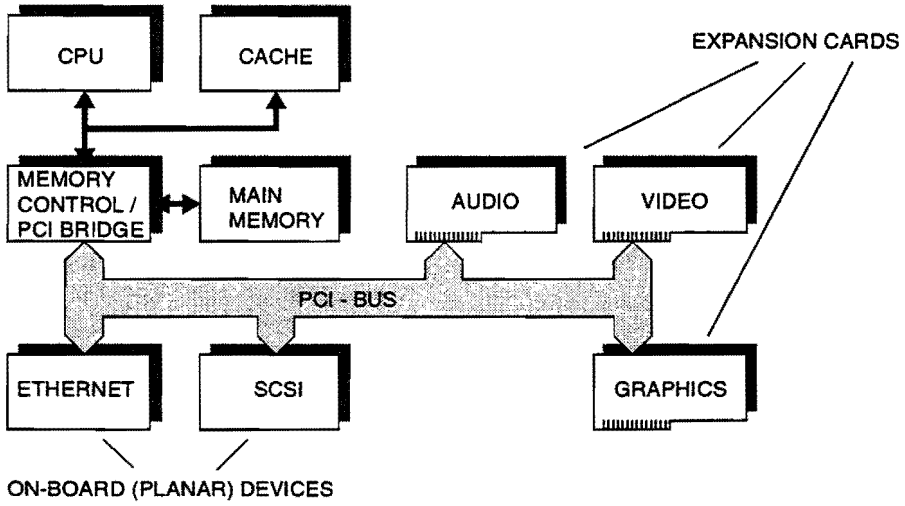


Figure 7: Typical PCI-System

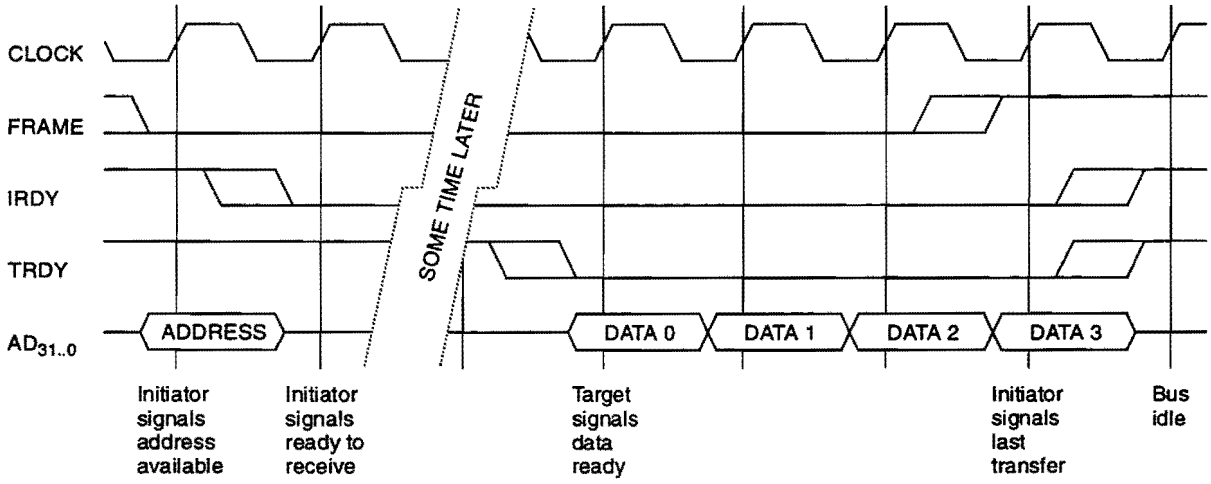


Figure 8: PCI-Burst-Transaction (Example)

5 XILINX FPGA Architecture

The two different parts used in this design provide a high number of I/O-blocks (176 for the XC3195A and 160 for the XC4013), each of them offering a bi-directional buffer and register pair [2]. All arithmetic and logical units are implemented in so-called Configurable Logic Blocks (CLBs), which basically consist of two flipflops and a set of function generators. A function generator is a look-up table (SRAM) which is addressed by the input variables. The number of input variables varies between four and five. Implementing a boolean function is done by loading the function generator RAM with the appropriate data. The number of CLBs is 484 for the XC3195A and 576 for the XC4013. For the XC4013, each CLB can be configured as a 16x2 bit SRAM, so that this device offers up to 18.432 bits storage capacity.

6 Implementation

A simplified block diagram of the accelerator is given in Figure 9. The on-board bus structure is given for clarity only and can be adapted to other requirements; essentially there is one 68 bit local bus and one 25 bit bus connecting the 4013 devices. The different units are described in the following sections.

6.1 PCI-Interface Unit

The PCI-Interface Unit manages the dataflow through several buses and controls the operation of all other units via point-to-point control lines. On the PCI-side, it offers a combined Master/Target-interface.

Data from memory is fetched in packets of four code elements, which form one cache block, via a read-burst transfer. Since there is no principal ray direction, we place a subcube of 2x2x2 volume

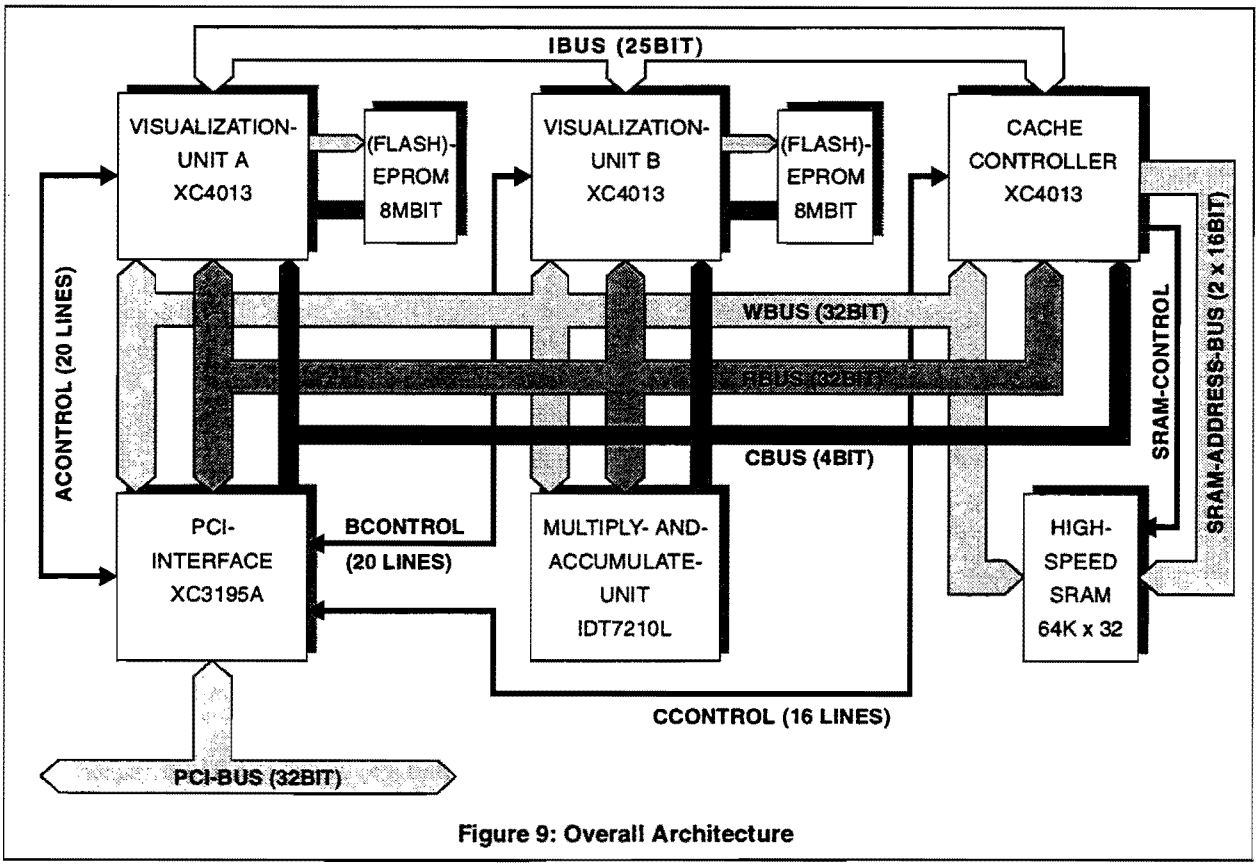


Figure 9: Overall Architecture

cells into one cache block, as shown in Figure 10. The linear offset Ω of a code element in memory is derived from its logical coordinates by

$$\Omega = Z_{7\ldots1}Y_{7\ldots1}X_{7\ldots1}Z_0Y_000 \quad (7)$$

A cache block is stored in four consecutive locations in the high-speed SRAM. The grid point with the smallest X-, Y- and Z-coordinates within a cache block is called the Reference Point.

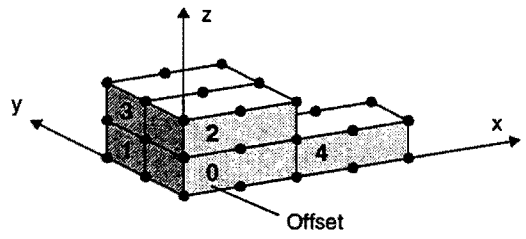


Figure 10: Cache Block Organization

6.2 Visualization Unit

Each of the two Visualization Units implements the complete rendering algorithm. The reason for the presence of two such units is as follows: Since the address of the next resample location depends on the actual volume cell (by means of the distance coding), there is a certain delay before a Visualization Unit can hand out the next address after having obtained a code element. To avoid idle times in the system, we use two identical Visualization Units which follow two different rays in parallel.

The Visualization Unit basically consists of three sub-units: the address sequencer, the PROM interface and the greyvalue processor as shown in Figure 11.

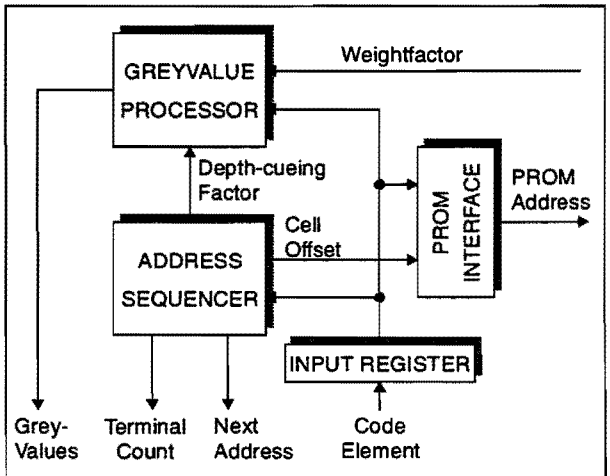


Figure 11: Visualization Unit

The address sequencer is a collection of incrementers, one for each the x-, y- and z-component of the raypoints, the depth-cueing factor and the raypoint number. Increments are added according to the skip distance. An inexpensive implementation uses an 8-entry on-chip SRAM, which holds

the pre-computed increments, and which is addressed by the skip distance as shown in Figure 12.

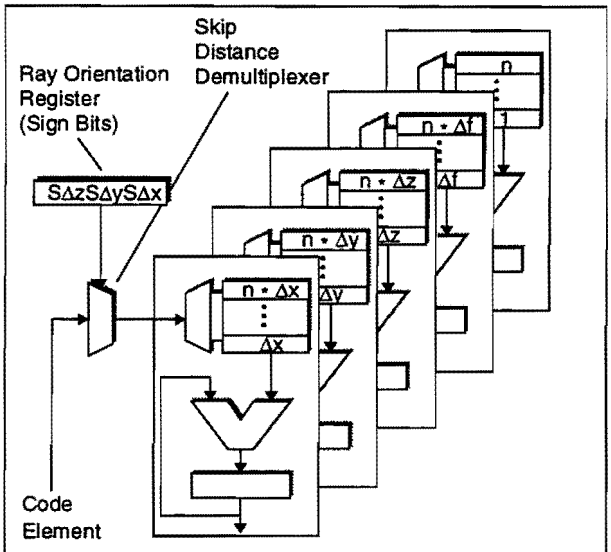


Figure 12: SRAM-based Incrementers

The weightfactor address is assembled as

$D_{7..0}\gamma_{-1..-4}\beta_{-1..-4}\alpha_{-1..-4}$ if $X_0=0$ or

$D_{11..4}\gamma_{-1..-4}\beta_{-1..-4}\alpha_{-1..-4}$ if $X_0=1$.

The greyvalue processor computes $C_u - \omega_l*(C_u-C_l)$ and, if the result is greater than the threshold, multiplies the greyvalue with the depth-cueing factor in a second pass. It has an 8×8→8 bits multiplier with a delay time of 50ns. As mentioned above, the greyvalue processor has four accumulators and one register for opaque material, from where the host-CPU can read the results.

6.3 Cache

The cache system consists of a XILINX 4013 device and two 64K×16 high-speed SRAM devices. The cache organization is direct mapped, however, *relative to the one-dimensional coordinate system of the actual ray.* We use the Manhattan Distance of the Reference Point of the actual resample location to the Reference Point which has the smallest distance to the eye point (which is called the Manhattan Distance Reference Point) as cache index. Therefore, the cache tag RAM needs 384 entries. Each tag RAM entry holds the logical coordinates of the Reference Point of the cache block and a valid flag and thus stores 22 bits. The tag RAM has a total capacity of 8448 bits, and thus fits into the XC4013 device. A simplified block diagram of the tag RAM is shown in Figure 13.

The operation is as follows: whenever the PCI-Interface Unit reads a code element address from either one of the Visualization Units, the Manhattan Distance D and the logical coordinates

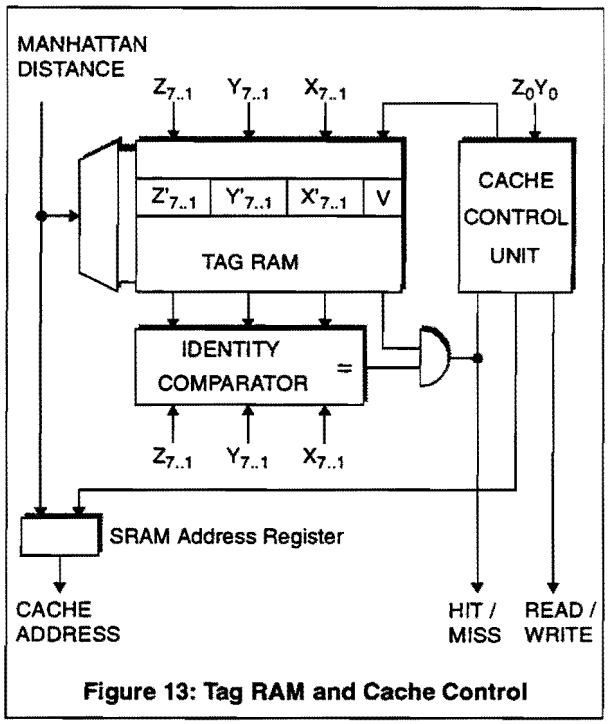


Figure 13: Tag RAM and Cache Control

$Z_{7..0}Y_{7..0}X_{7..1}$ of the code element are transferred to the Cache Controller. The tag RAM is addressed by the Manhattan Distance, and the stored logical cache block coordinates $Z'_{7..1}Y'_{7..1}X'_{7..1}$ are compared with the newly generated ones. In case of equality and a set valid flag, a cache hit is signalled to the PCI-Interface Unit, which does not start a PCI-transfer. The cache RAM is read at (DZ_0Y_0), and the code element is passed to the requesting Visualization Unit.

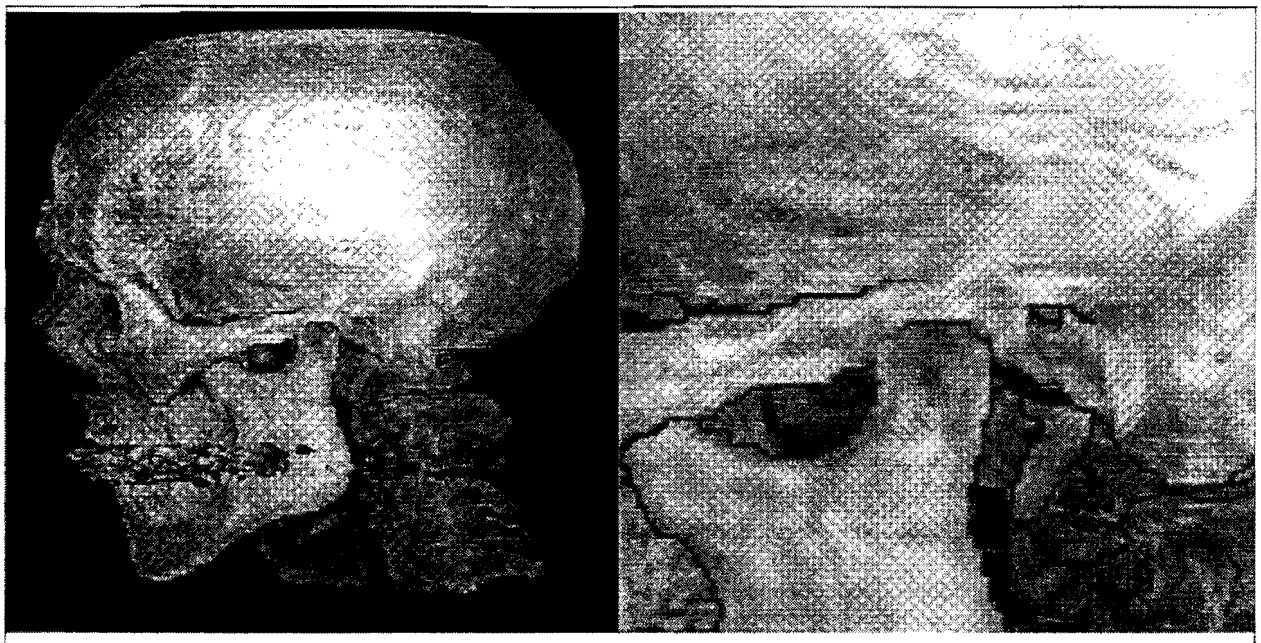
In case of a cache miss, the new logical cache block coordinates are written into the tag RAM together with a set valid flag. The Cache Controller then waits for the four code elements forming a cache block and writes them sequentially at addresses $(D00)$, $(D01)$, $(D10)$ and $(D11)$ into the cache RAM.

Note that this is a read-only cache, i.e., no modified entries must ever be written back to main memory, and that the valid flags need to be reset only once after a data set was loaded.

According to the cache block organization, rays are generated in the order of square screen blocks instead of scanlines to maximize the cache hit ratio.

7 Performance and Image Quality

For verification and measurement purposes, a software implementation of the algorithm was made. The example given here is a CT data set from a human skull. Data set dimensions are 256x256x216. All material except bone was removed during the pre-processing step. The bone surface was shaded using six light sources at infinity.



a) b)

Figure 14: Human Skull

The picture in Figure 14a was created by sending 256x256 rays through the data set, and bi-linearly interpolating the frame to the final 512x512 screen resolution. The bone surface was detected by using the threshold operator as explained in section 3.1. Figure 14b shows a magnified portion of the same data set.

For a round-trip or a walk-through, between 1 and 2 million raypoints must be processed for each frame. The cache hit ratio varies between 35% for large viewing distances and close to 98% for walk-throughs.

A software simulator for a single Visualization Unit was written based on the following (pessimistic) assumptions:

- The processing of a raypoint takes 240ns (empty cell or material discarded), 360ns (raypoint intensity is smaller than threshold) or 480ns (complete processing).
- A read-burst of four code elements (a cache line fill) takes 450ns.
- A cache hit takes 60ns.

Let's consider Figure 14a: the frame required exactly 1.058.672 raypoints, of which 580.276 caused a cache miss. 801152 raypoints were in empty space. Execution time is 568ms, giving 1.76Hz. Since no simulator for parallel operation of two Visualization Units is available, we can only estimate that the frame rate will be in the range of 2.5 to 3.5Hz.

The magnified view in Figure 14b required 775.312 raypoints, of which 140.417 caused a miss. 411.214 visited volume cells were empty. Execution time is 332ms, giving a frame rate of about 5Hz. A summary is given in Table 1. For these performance measurements, the distance of two raypoints was set to 0.95, and the maximum skip radius was 3.

8 Future Work

So far we have only talked about the *display* of RBC-data. In the current implementation, the entire pre-processing stage including classification, shading, encoding and distance coding is done in software. This can take up to 30 minutes, which is far from being interactive. For this task, however, we can take advantage of the in-system-

Ref.	# of Raypoints	Misses	Hits	Hit Ratio [%]	Empty Cells	Execution Time [s]	Estimated Frame Rate [Hz]
Figure 14a	1.058.672	580.276	478.396	45.2	801.152	0.568	2.5 - 3.5
Figure 14b	775.312	140.417	634.895	81.9	411.214	0.332	4 - 5

Table 1: Performance Summary

programmability of the FPGA devices, and employ what is called *reconfigurable computing*. Thus, for each step of the pre-processing algorithm, the accelerator is configured differently to provide maximum speed-up. Hardware resources are plenty (finally we got the reason for the presence of the multiply-and-accumulate unit), so that we're optimistic to bring the pre-processing time into the range of seconds.

9 Conclusions

We presented a compact and cost-efficient volume rendering accelerator which can bring volume visualization even into the PC market. Using a standard Gate Array technology, production costs can very well meet the absolute low-cost requirements of that market segment. Nevertheless, image quality and rendering speed are still high. Application areas which will benefit from this development are education in medicine, biology, chemistry and more, but also medical diagnosis and non-destructive testing.

10 Acknowledgments

This work was done for the research project SFB 328, funded by the German Science Foundation DFG, and was supervised by Prof. Straßer. Special thanks to Andreas Schilling, to whom I owe the idea of using the Manhattan distance as cache index, and many others which were born during exciting discussions.

11 References

1 **Anonymous,** "*PCI Local Bus Specification, Rev. 2.0*", PCI Special Interest Group, PO Box 14070, Portland, OR 97214, April 1993

2 **Anonymous,** "*The Programmable Logic Data Book*", XILINX Inc., San Jose, CA, 1994

3 **R. S. Avila, L. M. Sobierajski and A. E. Kaufman,** "*Towards a Comprehensive Volume Visualization System*", Proceedings of the IEEE Visualization '92 Conference, Boston, MA, October 19-23, 1992, pages 13-20

4 **G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, L. A. Leske, J. A. Lindberg and D. J. Sandin,** "*Two Bit/Pixel Full Color Encoding*", SIGGRAPH '86 Conference Proceedings, Computer Graphics, Vol. 20, No. 4, August 1986, pages 215-223

5 **E. J. Delp and O. R. Mitchell,** "*Image Compression Using Block Truncation Coding*", IEEE Transactions on Communications, Vol. COM-27, No. 9, Sept. 1979, pages 1335-1342

6 **R. A. Drebin, L. Carpenter, P. Hanrahan,** "*Volume Rendering*", Computer Graphics, Vol. 22, No. 4, August 1988, pages 65-74

7 **J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes,** "*Computer Graphics: Principles and Practice*", Addison-Wesley, Reading, MA, 1990, pages 727-728

8 **R. C. Gonzalez and P. Wintz,** "*Digital Image Processing*", Addison-Wesley, Reading, MA, 1987

9 **T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, H.-P. Meinzer, H.-J. Baur,** "*VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine*", Proceedings of the 9. Eurographics Hardware Workshop, Oslo, September 12-13, 1994

10 **D. R. Halverson,** "*On the Implementation of a Block Truncation Coding Algorithm*", IEEE Transactions on Communications, Vol. COM-30, No. 11, Nov. 1982, pages 2482-2484

11 **P. Heckbert,** "*Color Image Quantization for Frame Buffer Display*", SIGGRAPH '82 Proceedings, Computer Graphics, Vol. 16, No. 4, August 1982, pages 297-307

12 **A. Krikelis,** "*A Modular Massively Parallel Processor for Volumetric Visualisation Processing*", Proceedings of the Workshop on High Performance Computing for Computer Graphics and Visualisation, Swansea, UK, July 3-4, 1995

13 **M. Levoy,** "*Display of Surfaces from Volume Data*", IEEE Computer Graphics & Applications, Vol. 8, No. 5, May 1988, pages 29-37

14 **H. Pfister and A. Kaufman,** "*Real-Time Architecture for High-Resolution Volume Visualization*", Proceedings of the 8. Eurographics Hardware Workshop, Barcelona, September 6-7, 1993, pages 72-80

15 **K. J. Zuiderveld, A. H. J. Koning and M. A. Viergever,** "*Acceleration of Ray-Casting Using 3D Distance Transforms*", Proceedings of Visualization in Biomedical Computing, Chapel Hill, NC, October 13-16, 1992, pages 324-335