Design of a Fast Voxel Processor for Parallel Volume Visualization

Jan Lichtermann

University of Kaiserslautern
Germany

Abstract

The basics of a parallel real-time volume visualization architecture are introduced. Volume data is divided into subcubes that are distributed among multiple image processors and stored in their private voxel memories. Rays fall into ray segments at the subcube borders. Each image processor is responsible for the ray segments within its assigned subcubes. Results of the ray segments are passed to the image processor where the ray continues. The enumeration of resampling points on the ray segments and the interpolation at resampling points is accelerated by the voxel processor. The voxel processor can additionally compute a normalized gradient vector at a resampling point used as a surface normal estimation for shading calculations. In the paper the focus is on operation and hardware implementation of this pipeline processor and the organization of voxel memory. The instruction set of the voxel processor is explained. A performance of 20 images per second for a 256^3 voxel volume and 16 image processors can be achieved.

CR Descriptors: C.1.2 **[Processor Architectures]**: Multiple Data Stream Architectures- *Parallel Processors, Pipeline Processors*; I.3.1 **[Computer Graphics]**: Hardware Architecture - *Graphics processors*; I.3.2 **[Computer Graphics]**: Graphics Systems - *Distributed/network graphics*; I.3.7 **[Computer Graphics]**:Three-Dimensional Graphics and Realism Animation - *Raytracing*; I.4.10 **[Image Processing]**: Image Representation - *Volumetric*.

1. Introduction

In medicine, modern imaging techniques such as computer tomography, magnetic resonance imaging, positron emission tomography, and others produce enormous amounts of two-dimensional data. By stacking the 2D cross sections and interpolating between them, volumetric data sets can be obtained. A volumetric data set can be imagined as a large rectilinear 3D grid of voxels. Voxels are unit volume elements or cells, each voxel carrying either a scalar, vector, or tensor volume. Rendering volume data helps in understanding complex structures, i.e. anatomy in medical applications. Different rendering techniques can be applied to produce images from the data sets [4].

We use a pixel order approach for non-binary voxels similar to the one published by Levoy [8]. It is a raycasting approach enumerating each pixel in 2D image space and using a ray to determine the influence of each voxel projected on the examined pixel.

The visualization process is supported by 3D segmentation [11] of the volume data. As a result of the segmentation step each voxel carries a segment number besides its scalar value. The segment number in combination with a segment lookup table enables the assignment of a color, opacity, and specular highlight coefficient for shading calculations to a voxel. Opacity is a value between 1.0 meaning fully opaque and 0 meaning fully transparent material. Therefore several semi-transparent voxels on a ray may contribute to a single pixel resulting in a very expensive computation.

Three ways to accelerate image computation are reported in the literature.

The first approach is the use of software techniques, like adaptive refinement [9]. It is also possible to decompose the general viewing transformation matrix into matrices describing simple shear and scale operations and to exploit object and image coherence [7]. But with today's workstation technology it seems to be impossible to reach real-time rendering rates.

The second approach makes use of commercially available parallel computers. The visualization process is partitioned and the computation is distributed among the nodes of the computer. Because volume data sets are very large, simple data replication of the volume data set on all parallel nodes must be avoided. Two partitioning schemes have evolved in literature.

Image space partitioning assigns regions of the image to a node. A node has to get access to the voxels of the volumetric data set that influence the pixels in the region assigned to it.

In object space partitioning, parts of the volumetric data set are assigned to each node and each node has to compute a subimage of the voxels it can access. A final composing step of the subimages is necessary at the end.

Both partitioning schemes cause communication between computational nodes. A comparison of the communication costs can be found in [12]. It is the authors's opinion that use of commercially available parallel machines is inadequate for widespread use of volume visualization, e.g. in medicine, because of the high costs for this kind of computers.

The third approach makes use of dedicated architectures for volume rendering.

The Silicon Graphics Reality Engine is a commercial machine that

University of Kaiserslautern, Department of Computer Science, Postfach 3049, D-67653 Kaiserslautern, Germany,
email lichterm@informatik.uni-kl.de.

can be used for that purpose. Its 3D texture hardware can accelerate volume visualization. A performance of 10 frames per second for a 128x128x64 voxel dataset and 2.5 frames per second for a 256x256x64 voxel dataset is reported in [13]. A new generation of high end graphics machines with improved texture mapping hardware may lead to better frame rates for large data volumes.
During the last year, three dedicated volume visualization architectures were published. A good overview about more recent approaches can be found in [3].
The Cube-3 Architecture is designed as a real-time architecture for high-resolution volume visualization. A performance of about 30 frames per second for a 512^3 16 bit voxel volume is estimated in [14], but some technical challenges are left to the hardware implementation.
VIRIM is another approach to real-time volume rendering using the Heidelberg Raytracing Model. A performance estimate of 10 frames per second for a 256x256x128 voxel dataset is given in [2].
A Voxel Engine for Real-time Visualization and Examination (VERVE) is proposed in [5]. A single voxel engine is designed for 2.5 frames per second for a 256^3 voxel dataset. Eight voxel engines can work in parallel and result in a performance of 20 frames per second.
We think that the third approach may lead to rendering accelerators used in conjunction with an off-the-shelf workstation at moderate costs.
Our goal is to perform the image computation with a parallel architecture at a rate of 20 images per second for a volume of 256^3 voxels. This allows a human observer to choose the observation point interactively or even to rotate the whole volumetric data set in real-time.

2. The Distributed Volume Visualization Architecture

2.1 Volume rendering with raycasting

As stated in the introduction a raycasting method is used. The volume data consists of sampling points on an isotropic grid. Each sampling point carries a scalar value and a segment number. The segment number can be transformed into voxel color, opacity, and highlight coefficient for shading. Raycasting is a pixel order approach.
So each pixel in the image plane is enumerated and at least one ray through each pixel is initiated. On the rays resampling points are enumerated. All resampling points are equidistant and it is possible to advance from one resampling point to the next by simply adding an incremental vector. Normally the resampling points on the rays do not match the voxel lattice. Therefore the values at the resampling points are computed from the surrounding voxel values by trilinear interpolation. Figure 2.1 shows trilinear interpolation at the resampling point P.

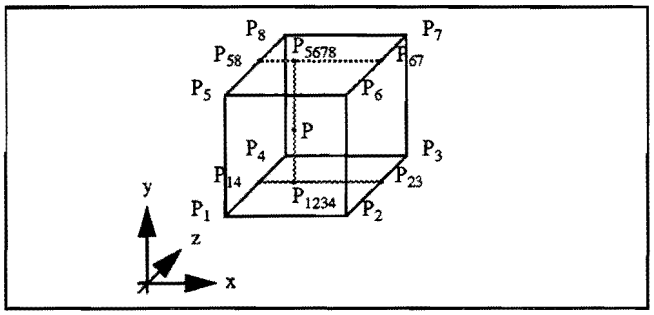

Figure 2.1 Trilinear Interpolation.

At the resampling points light emission computations are performed. The computations consider energy emission due to lumious gas or particles and energy emission due to reflection on surfaces. Both energy terms are mixed by a term called surface probability which is a value between 0 and 1.0. For shading calculations a surface normal is estimated by gradient computation. Emitted light on a resampling point is attenuated by the material between the resampling point and the pixel.
If we find fully opaque material between a resampling point and a pixel or accumulated opacity reaches a certain level the resampling point is invisible. This observation may be exploited to speed up computation [9]. We stop computation for rays with such a point (early ray termination) because everything behind this point is obscured.
Most datasets contain almost empty regions surrounding the object that was sampled. The voxel values in these regions are smaller than a typical threshold value representing air plus a certain noise level of the sampling device. So initially all resampling points with values below the threshold can be skipped until a resampling point with a value above the threshold is reached. We call this speed-up technique previewing. Previewing dramatically reduces the number of points where light emission computations have to be performed. To our experience up to 80% of a volume dataset (e.g. the well known UNC head) is traced in previewing mode. For resampling points in these regions we only have to perform trilinear interpolation.
The gain of both speed-up techniques, early ray termination and previewing, is of course depending on the data set. We can exploit both techniques in the DIV^2-Architecture.
Two design goals are pursued in the design of the DIV^2-Architecture:
The first design goal is to achieve good load balancing between the processors. This implies that equal portions of the image computation task are given to each processor.
The second design goal is to keep communication as low as possible and as local as possible. Keeping communication low saves communication time and bandwidth on the communication network. Keeping communication local simplifies the communication network and facilitates the use of hardware links between neighboring processors. The total communication traffic distributes among the links in the architecture.

2.2 Parallel rendering on DIV^2A

To speed up computation the task of computing one image is subdivided into smaller tasks and these tasks are scheduled on multiple processors. Therefore we partition our data cube into small subcubes of equal size. To achieve good load balancing we assign a set of subcubes to each processor and store the subcubes data within the processors local memory. Each processor gets equal numbers of subcubes from the inner and outer parts of the data volume. A ray through the data cube typically intersects multiple subcubes. We call the part of a ray through one subcube a ray segment. Each processor is responsible for the computation of the ray segments within its subcubes. A ray message for a processor consists of the first resampling point on a ray segment, the incremental vector between two resampling points, the intermediate result of light emission computation up to but excluding the first resampling point on the ray segment, and the pixel address within the image the ray is belonging to. When a processor receives a ray message it is its task to construct all further resampling points on that ray segment and perform light emission calculation until calculation stops due to opacity or the ray segment is expired. In the case the ray leaves the data cube and hits background computation of that ray stops, too. If ray computation stops, the final value is sent together with the pixel address to a frame buffer unit where the computed

image is assembled.
In the third case the ray continues in a neighboring subcube. So the processor has to send a ray message to continue raytracing with the actual data to the responsible processor of the neighboring subcube. So far we have a object space partitioning scheme. In contrast to [1] we do not need a final composing step of intermediate images because computation on a ray proceeds strongly from front to back and the composing step is implicitly performed when a message with the intermediate result is passed from one processor to its successor. This is the reason why we can exploit early ray termination due to opacity.
To start raytracing the rays of one image must be initiated. Rays are originated in subcubes that compose the faces of the large data cube. So all processors responsible for these subcubes test visibility of these subcubes and initiate rays. This computation is performed independently by all processors. So we have a image space partitioning scheme, too. The assignment of processors to pixels for ray initiation is not static but depends on the visualization parameters, like the position of the image plane, the observers position, and the kind of projection to be performed.

2.3 Architectural Overview

The Distributed Volume Visualization Architecture is shown in figure 2.2. A Unix workstation is used to store volume data on disk and hosts the user interface. DIV^{2}A-hardware is connected to the workstation via an interface card. The hardware consists of the central processor, multiple image processors, and a communication network.

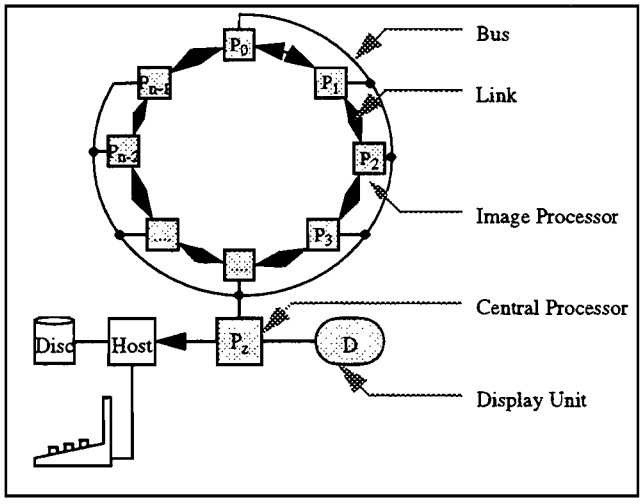

Figure 2.2 DIV^2-Architecture.

The central processor connects to all image processors via a bus. This bus is used to transfer volume data from the central processor to the image processors. The data of terminated rays is sent from the image processors to the central processor via the bus. A specialized unit in the central processor assembles these ray messages to the final image which is stored for display in a frame buffer.
The image processor (figure 2.3) consists of three subprocessors, which are the raytracing processor, the I/O processor, and the voxel processor.

The raytracing processor initiates rays for image computation, conducts raytracing, and performs shading operations. It consists of a Texas Instruments floating-point DSP TMS320C40.
Image processors send ray messages to other image processors. For this kind of communication the image processors are arranged

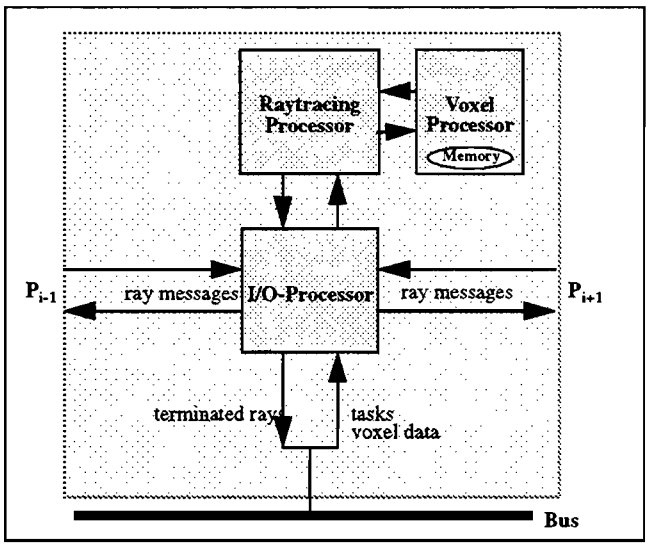

Figure 2.3 Image processor of DIV^{2}A.

in a ring. Two neighbors in a ring are connected by a bidirectional link allowing message transfer in two directions simultaneously. Network hardware supports message routing across image processors. Communication is not restricted to neighboring processors. The links do not build a ring network in the classical sense because all processors can exchange messages with their neighbors at the same time. Communication is handled by the I/O processor.
Voxel data is stored in the voxel processors memory. The instruction set of the voxel processor enables the user to retrieve a voxel value together with segment information and an estimated surface normal at given non lattice coordinates. Additionally the voxel processor is able to generate the resampling points on ray segments from the first resampling point on a ray segment and the incremental vector between two resampling points. The voxel processor will be discussed in detail in section 3.
The number of image processors must be a power of two to simplify address calculations.

2.4 Assignment of image processors to subcubes

Voxels are set up in a three-dimensional orthonormal x,y,z-coordinate system. Coordinates range from 0 to 255 for a 256^3 voxel data volume. We denote the length of the data cube with *clen* which is 256 in our example. Each resampling point on a ray within the data volume can be described with global coordinates $(x_g, y_g, z_g), x_g, y_g, z_g \in [0, 255]$. We denote the number of processors in the DIV^2-Architecture with p. An image processor in the ring has number $p_i, i \in \{0,..., p-1\}$. We can determine its left and right neighbor by

$$\text{left\_neigbor}(p_i) = \begin{cases} p_{(i-1+p)} & \text{if } i-1<0 \\ p_{i-1} & else \end{cases}$$

$$\text{right\_neigbor}(p_i) = p_{(i+1) \bmod p}$$

The directed distance *ddist* between two processors p_i and p_j is the shortest distance between them along the ring. It indicates whether to travel clockwise or counterclockwise along the ring to get with minimal steps from p_i to p_j.

$$\text{ddist}(p_i, p_j) = \begin{cases} p_j - p_i \text{ if } |p_j - p_i| \le \frac{p}{2} \\ p_j - p_i + p \text{ if } p_j - p_i < -\frac{p}{2} \\ p_j - p_i - p \text{ if } p_j - p_i > \frac{p}{2} \end{cases}$$

We divide the data cube into subcubes such that the number of subcubes along a random axis is p multiplied by a power of two. We call the power of two the interleaving factor *ilv* because it describes how many subcubes are assigned to the same image processor along one axis. Thus the edge length of a subcube is

$\text{sclen} = \text{clen} / (p \cdot \text{ilv})$.

To describe subcubes we define subcube coordinates x_{sc}, y_{sc}, z_{sc} with $x_{sc}, y_{sc}, z_{sc} \in \{0, \ldots, p \cdot \text{ilv} - 1\}$. Given a point ($x_g$, y_g, z_g) we can find the subcube containing this point by

$(x_{sc}, y_{sc}, z_{sc}) = (x_g/\text{sclen}, y_g/\text{sclen}, z_g/\text{sclen})$.

We define a processor function *proc* which assigns a processor to a subcube:

$\text{proc}(x_{sc}, y_{sc}, z_{sc}) = (x_{sc} + y_{sc} + z_{sc}) \bmod p$.

Each subcube (x_{sc}, y_{sc}, z_{sc}) except those at the border of the large data cube is surrounded by 26 other subcubes. Six of them with coordinates ($x_{sc}\pm1$, y_{sc}, z_{sc}), (x_{sc}, $y_{sc}\pm1$, z_{sc}), and (x_{sc}, y_{sc}, $z_{sc}\pm1$) share a common face with the subcube. We call them face neighbors. Twelve with coordinates ($x_{sc}\pm1$, $y_{sc}\pm1$, z_{sc}), ($x_{sc}\pm1$, y_{sc}, $z_{sc}\pm1$), and (x_{sc}, $y_{sc}\pm1$, $z_{sc}\pm1$) share a common edge with the subcube. We call them edge neighbors. Eight with coordinates ($x_{sc}\pm1$, $y_{sc}\pm1$, $z_{sc}\pm1$) share a common vertex with the subcube. We call them vertex neighbors. We will refer to this again in section 3.4.

In table 2.1 we show the directed distances between the processors of (x_{sc}, y_{sc}, z_{sc}) and the processors assigned to the neighbors

	Σ	-3	-2	-1	0	+1	+2	+3
face nbrs	6	0	0	3	0	3	0	0
edge nbrs	12	0	3	0	6	0	3	0
vertex nbrs	8	1	0	3	0	3	0	1

Table 2.1: directed communication distance for face, edge, and vertex neighbors

Each of the 26 neighbors is a potential candidate for a ray to proceed if a ray segment in a subcube is expired. The table shows that in no cases distance is larger than 3. Because in most cases a ray continues in a face neighbor a communication distance of 1 is most probable. Our processor assignment restricts ray message communication to local communication (design goal) and permits the use of point-to-point communication on the links between neighbors on the ring. Figure 2.4 shows the subcube distribution for a 16 processor architecture with ilv=1. Processor numbers are coded by different gray levels.

3. The Voxel Processor

The voxel processor (figure 3.1) supports raytracing. The instruction set of the voxel processor enables the user to retrieve a voxel value together with segment information and an estimated surface normal at given non lattice coordinates. Additionally the voxel

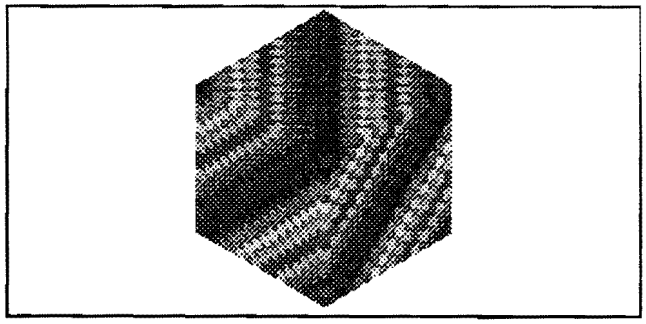

Figure 2.4 Example of a 16 processor architecture.

processor is able to construct the resampling points on ray segments from the first resampling point of a ray segment and the incremental vector between two resampling points. Operations to be performed are additions to compute the coordinates of resampling points, trilinear interpolations to retrieve voxel values at non lattice points, and normalized gradient computation for surface normal estimation. Because trilinear interpolation and normalized gradient computation are complex operations, they are subdivided into smaller subcomputations and they are pipelined. Pipelining substantially increases throughput.

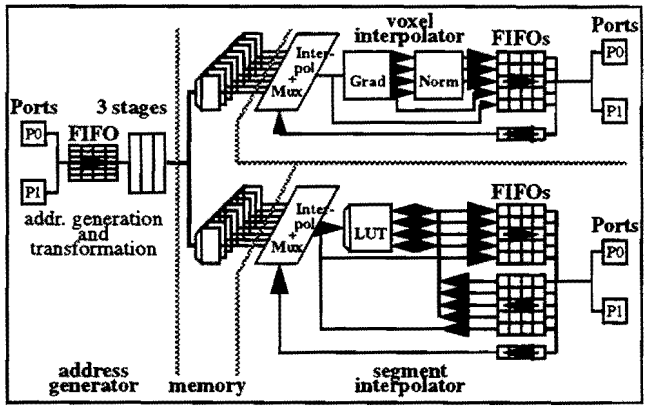

Figure 3-1 DIV²A voxel processor.

The voxel processor is a large pipeline. It consists of the address generator, voxel memory, the voxel interpolator, and the segment interpolator. The pipeline is fed with operations and operands from the raytracing processor through the ports P0 and P1. To decouple the voxel processor from the raytracing processor, operations and operands are stored in FIFOs. A scheduler contained in the voxel processor puts operations on the pipeline if resources become available. Results of pipeline operations are stored in FIFOs in the voxel and segment interpolator. From there they can be read by the raytracing processor. Decoupling the voxel processor from the raytracing processor enables them to work in parallel.

3.1 Processor interface

The voxel processor is memory mapped into the address space of the raytracing processor. Two interfaces P0 and P1 are implemented to increase bandwidth. This enables us to profit from modern DSPs with two independent busses in the raytracing processor. Operands for pipeline operations are successively written into special register locations. If all operands for an operation are written, these values are copied together with the instruction code into the FIFO. The FIFO realizes an instruction queue where the instructions wait for being scheduled. Results of a pipeline instruction are stored in FIFOs. A result of a pipeline instruction may consists of several values, e.g. the three components of the gradient vector

together with the value at the resampling point. Values belonging to one resampling point are stored in parallel in the FIFOs. If an instruction generates multiple resampling points, the results at these resampling points are stored in sequence in the FIFOs. The first entry of the FIFO can be read by the raytracing processor through register locations. If a result is no longer needed, the raytracing processor can flush the first entry out of the FIFO. Afterwards the next result in the sequence of resampling points can be read. The processor interface together with the FIFOs allows the voxel processor to work at another clock speed than the raytracing processor. Eight instructions can be stored in the instruction FIFO at the beginning of the pipeline. Thirty results (resampling points) can be stored in the FIFOs at the end of the pipeline.

3.2 Pipeline design

As we mentioned above, the voxel processor is a large multifunction pipeline. A short estimate shows that the pipeline does not fit into a single chip: The processor interface consists of 32 data lines plus 8 address and control lines per port. For two ports this sums to 80 lines. To exploit full parallelism we divide voxel memory into 8 memory banks. To address a volume of 256^3 voxels distributed on 16 processors with 8 memory banks each, we need 17 address lines plus a CS and WE line per bank. Voxel values have 12 bit resolution and segment information is represented with 8 bit resolution. So we have 20 data lines. This results in a total of 2x80+8x39=472 pins without power supply pins. For 20 I/O-pins we need one pair of power supply pins which increases the number of pins to 520. Chip size grows quadratically with the number of pins. Assuming one pad every 170µ plus some margin for corner cells we come to a chip size of approximately $500mm^2$.
Because we only can fabricate chips with a size of up to $190mm^2$ (1.0µ or 0.7µ CMOS standard cell) in the Eurochip project, we have decided to partition our design into 3 chips. A natural way is to partition the design into an address generator, a voxel interpolator, and a segment interpolator. The functions of these units are discussed in the following sections. Voxel memory is implemented with discrete memory chips. It is one stage of the pipeline. As a consequence of the partitioning the processor interface is distributed to the three chips.
The multifunction pipeline needs a scheduler. The scheduling strategy is to schedule instructions strongly in the sequence of arrival. An instruction is scheduled as early as possible without causing collisions (structural hazards) with other instructions already streaming through the pipeline. The scheduler implementation is hardwired.
We have decided to implement a dynamically configured multifunction pipeline. This means that the type of function performed in the pipeline may change with every instruction entering the pipeline. A statically configured pipeline would not have been sufficient because we cannot guarantee that long sequences of the same function are input to the voxel processor. Search and trace instructions are arbitrarily mixed during raytracing.
Each stage of the pipeline needs control. Control signals determine the function of a pipeline stage depending on the operation to be performed and may indicate the presence or absence of data (null cycle) and the routing of data within the pipeline. Two extreme control strategies are possible, time-stationary and data-stationary control [6]. In time-stationary control a central controller is the global source of the control and route signals going to each stage. In data-stationary control the control signals accompany the data. The control signals can be the opcode indicating the function to be performed on that data. In this case decoders are necessary at the pipeline stages generating the control and route signals for that particular stage. It is also possible to send a wide vector of control and route signals through the pipeline.
We have decided to implement data-stationary control. This facilitates the design of the multifunction pipeline. After a operation is initiated in the pipeline we do not have to care about the control of that operation in the central controller. The control vector accompanying the data takes 27 bit in our design.
The necessity to partition the pipeline into multiple chips leads to new problems. First, if a single controller is used in one chip, the control vector has to leave the chip and has to enter two other chips. This results in an additional number of pins that have to be added to the chips.
Second, the first pipeline stage, which realizes the readout of data from the FIFOs into the pipeline, is partitioned to three chips. This leads to timing problems because the control signals for this purpose are generated within one cycle and have to cross chip boundaries within the same clock cycle.
We have replicated the controller in all three chips to overcome both problems (figure 3.2). The three controllers are synchronized with the scheduler contained in the address generator. This ensures that all three controllers have the same global view of the pipeline state. The controller is described with VHDL. After synthesis parts of the controller and the shift registers for the control vector that are not needed in a particular chip are automatically eliminated by logic minimization. This helps to save chip area.

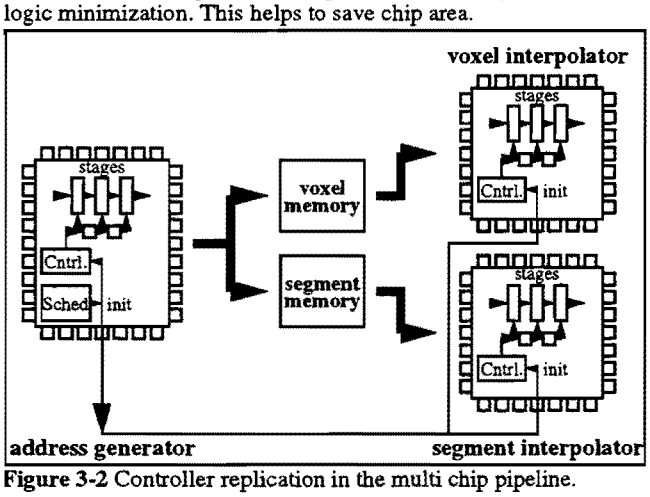

Figure 3-2 Controller replication in the multi chip pipeline.

3.3 Voxel Memory

Voxel data is stored in voxel memory. For each voxel a 12 bit sample value and a 8 bit segment number is stored. For resampling trilinear interpolation is used. To exploit full parallelism in the interpolation hardware voxel memory is divided into eight memory banks. We can access the eight voxel values for one trilinear interpolation in one access cycle. Memory access can be considered one stage of the pipeline. Therefore memory access time must be less than the clock cycle of the pipeline (25 ns). Voxel memory consists of static memory with 20 ns access time.
The partitioning of voxel space into subcubes and the assignment of subcubes to processors arises a new problem. A resampling point on a ray may fall between two voxels stored in neighboring subcubes. Figure 3.3 gives an example for the planar case. To perform the trilinear (example: bilinear) interpolation the voxel processor processing the resampling point has to access voxel values stored in subcubes not assigned to it. The same problem arises for gradient computation near the border of a subcube. For shading calculations a surface normal is estimated by the gradient vector (G(x+1,y,z) - G(x-1,y,z), G(x,y+1,z) - G(x,y-1,z), G(x,y,z+1) - G(x,y,z-)). (x,y,z) is the resampling point and G(x,y,z) denotes the interpolated voxel value at point (x,y,z). For gradient computations

the problem is worse (figure 3.4).

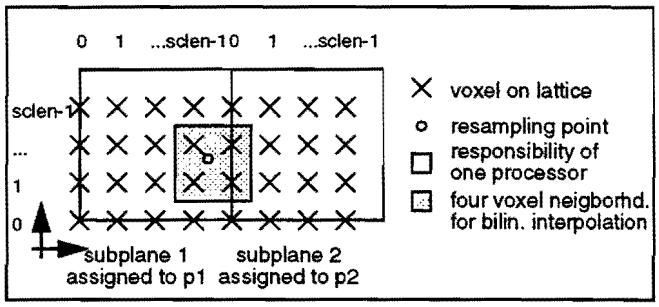

Figure 3-3 Bilinear resampling near the border of two subplanes.

Three design alternatives exist to overcome the problem: data replication, communication request for voxel values to neighboring processors and shared (multiport) voxel memories.
First we estimate the frequency of the access problem. We denote the subcube length with sclen. Along one axis the voxels within a subcube can be enumerated from 0 to sclen - 1. A voxel processor has to cope with resampling points falling onto voxel 0 up to resampling points falling an indefinite small distance left to voxel 0 of the neighbor. Thus during resampling there are no problems with resampling points at the left border of a subcube but with all resampling points between voxel sclen - 1 and voxel 0 of the neighbor. For the subcube we can estimate the percentage of these points:

$$rp = \frac{sclen^3 - (sclen-1)^3}{sclen^3} 100\%.$$

For sclen = 16 we get rp = 17.6%.
For gradient computation all resampling points between voxel 0 and voxel 1 and all resampling points between voxel sclen - 2 and voxel 0 of the neighbor are concerned (see figure 3.4 for the planar case). The percentage of these points can be estimated:

$$rg = \frac{sclen^3 - (sclen-1-2)^3}{sclen^3} 100\%.$$

For sclen = 16 we get rg = 46.4%.
The estimate shows that a good solution must be chosen because for sclen = 16 and gradient computation the access problem arises for nearly half of all resampling points in a subcube.

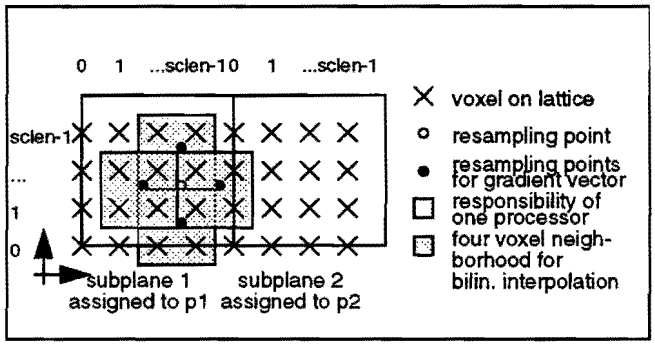

Figure 3.4 Gradient computation near the subplane boundaries.

Communication requests to neighboring voxel processors is not a good solution. First, the above estimate shows that a lot of communication must occur. Communication takes time and leads to hold-ups in the voxel pipeline. Additional hardware is necessary in the voxel processor to service voxel request from other processors. A pipeline working at maximal speed needs 8 voxel values every clock cycle of 25 ns. Because of the hold-ups in the pipeline due to communication delays throughput dramatically drops.
Shared or multiport memory is too costly. As we showed in section 2.4 a subcube has 26 neighbors. The neighboring subcubes are assigned to processors within a communication distance of -3, ... , +3. Thus a voxel processor has to access the voxel memory of 6 neighboring voxel processors. This leads to a very high connectivity because of the high number of address and data lines that are necessary. Voxel memory can either be realized as a seven-port memory or as a single port memory together with an arbitration logic. Seven-port memories are no standard components and are not available. For a single port memory access contentions have to be resolved by an arbitration logic which leads to access delays. The effect of access delays is again pipeline hold-up with a decrease of throughput.
Replication of voxel data avoids both, communication and memory contention. Pipeline hold-ups are avoided because memory accesses are local. As a disadvantage of data replication voxel memory has to be enlarged which results in higher memory costs. These higher memory costs can be justified because a high performance should be achieved and the complexity for the hardware implementation is low compared to the before mentioned alternatives.
Memory overhead is the ratio of memory needed to store additional voxels from other voxel processors and the memory needed to store only the subcube data. The additional number of voxels to be stored can be determined from the discussion about resampling and gradient computation near subcube boundaries. Near the 3 subcube borders with low voxel coordinates a region of thickness 1 has to be stored. Near the 3 subcube borders with high voxel coordinates a region of thickness 2 has to be stored. Thus the overhead is given by

$$mo = \frac{(1+sclen+2)^3}{sclen^3} 100\%.$$

For sclen = 16 the equation evaluates to mo = 67.4%.
Addressing is simplified if a region of 2 is stored around each subcube. This increases the overhead to

$$mo' = \frac{(2+sclen+2)^3}{sclen^3} 100\%$$ and evaluates to mo' = 95.3% for sclen = 16.

We have chosen the last solution because we can simply use chips with double address space to consider the memory overhead. So no additional chips are necessary and board space can be saved.
For a 256^3 voxel volume and 16 processors each voxel memory stores $256^3/16$=1M voxels. Assuming 8 memory banks 1M/8 = 128K voxels are stored in each memory bank. With data replication address space is doubled and chips with a 256K organization can be used for the implementation of the voxel memory.

3.4 Voxel Addressing

In this section we describe the addressing scheme to map voxel lattice coordinates on memory addresses. Voxel memory is divided into two parts. Voxels which belong to subcubes that are assigned to the processor are stored in the first half. Voxels belonging to replicated data are stored in the second half.
First, we describe voxel addressing within subcubes that are assigned to the processor. In section 2.4 we have introduced subcube coordinates (x_{sc},y_{sc},z_{sc}), clen, sclen, and p. By means of subcube coordinates all subcubes of one processor can be enumerated:

$$s_{cl} = (x_{sc} + y_{sc}\,(clen \text{ div } sclen) + z_{sc}\,(clen \text{ div } sclen)^2) \text{ div } p$$

scl is the first part of a voxel address determining the number of the subcube the voxel belongs to. For trilinear interpolation a set of eight voxel values must be accessed in one cycle. We call a set of voxels with addresses (x,y,z), (x,y+1,z), (x,y,z+1), (x,y+1,z+1), (x+1,y,z), (x+1,y+1,z), (x+1,y,z+1), (x+1,y+1,z+1) a voxel octet. The coordinates of a voxel with minimal x,y,z coordinates in a voxel octet are called the base coordinates of the voxel octet. A voxel octet which has base coordinates with even x,y,z coordinates is called a supervoxel. Each voxel belongs to exactly one super-

voxel. For each voxel with coordinates (x,y,z) local supervoxel coordinates within a subcube can be computed:

$$(x_{ls}, y_{ls}, z_{ls}) = \begin{matrix}((x \bmod sclen) \text{ div } 2,\\ (y \bmod sclen) \text{ div } 2,\\ (z \bmod sclen) \text{ div } 2\ )\end{matrix}$$

The concatenation $z_{ls}.y_{ls}.x_{ls}$ enumerates all supervoxels within one subcube.

The eight voxels of a supervoxel are assigned to the eight memory banks by generating the 3 bit number $b_z b_y b_x$ with $b_z = z \bmod 2$, $b_y = y \bmod 2$, $b_x = x \bmod 2$. Thus for addressing within a memory bank the concatenation $s_{cl}.z_{ls}.y_{ls}.x_{ls}$ can be used to map coordinates (x,y,z) on a memory address. It is evident that a random voxel octet can be fetched in parallel from the eight memory banks when using the mapping shown above. If the voxel octet is a supervoxel then the same addresses are applied to all memory banks. Otherwise the voxel values are stored in different supervoxels depending on the base coordinates. If x of the base coordinates is even, then x_{ls} has to be applied to the memory addresses of all voxels in the octet. If x of the base coordinates is odd, then x_{ls} has to be applied to the memory addresses of all voxels with coordinate x and $x_{ls}+1$ has to be applied to the memory addresses of all voxels with coordinate x+1. The coordinates y and z are treated in the same way.

Now the addressing scheme for replicated data is described. As already mentioned in section 2.4 a subcube has 26 neighbors. These neighbors are classified as face, edge, or vertex neighbors. So the region of thickness 2 we want to store around each subcube can be decomposed into face, edge, and vertex subvolumes (figure 3.5). There are 6 face subvolumes of size $sclen^2*2$, 12 edge subvolumes of size $sclen*2^2$, and 8 vertex subvolumes of size 2^3. The subvolumes of one type can be enumerated.

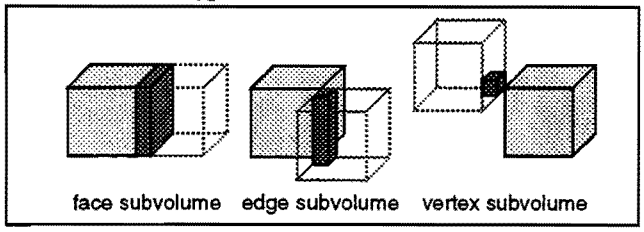

Figure 3-5 Three subvolume types for the storage of replicated data.

If a voxel is read from a subvolume the type of the subvolume together with the number of the subcube which is extended by the subvolume is computed. So the first part of the memory address is again the subcube number. Then the subcube type together with the enumeration number of the subvolume is used as the second part.The last part consists of the local supervoxel address within the subvolume. Supervoxel addresses are defined separately for each type of subvolume. The assignment of voxels to one of the eight memory banks is the same which is used for voxels contained in assigned subcubes.

The mapping of voxel coordinates on a subcube number and a subcube type is not definite. A voxel may belong to up to three different subvolumes. We have made the mapping table definite. This eliminates the need to store a voxel value in three different subvolumes when the voxel memory is loaded. Otherwise the 'store voxel' operation would have consumed up to three pipeline cycles depending on the voxel coordinates instead of just one cycle.

3.5 Voxel Interpolator

The voxel interpolator (upper part of figure 3.1) is designed to perform trilinear interpolation of voxel values. A single interpolation takes 25 ns. The eight voxel values for one interpolation are fetched from memory in one cycle. The eight values are input to the interpolation tree (Figure 3.6) through a multiplexor stage. The multiplexor stage rearranges the eight voxel values coming from memory. This is necessary because the eight voxel values of an octet may arrive from memory in different patterns. The pattern depends on the base coordinates of the octet. In the interpolation tree four interpolations along the x-axis, then 2 interpolations along the y-axis and finally one linear interpolation along the z-axis is performed. Linear interpolation is performed with 12 bit voxel resolution and 12 bit interpolation weights x_f, y_f, z_f. The multiplexor together with the interpolation tree is implemented in 8 pipeline stages.

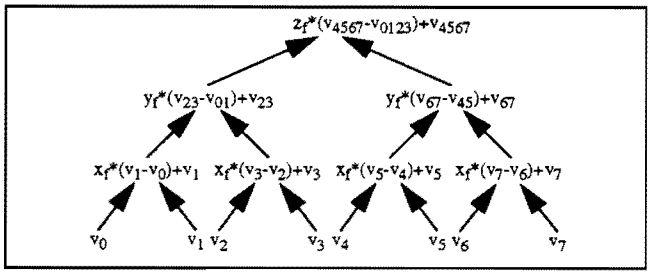

Figure 3-6 Trilinear interpolation tree.

To perform gradient computations the address generator enumerates the coordinates of the six necessary points. For each of the six points trilinear interpolation is performed. The gradient unit of the pipeline collects the three pairs of values and computes the x,y,z components G(x+1,y,z) - G(x-1,y,z), G(x,y+1,z) - G(x,y-1,z), G(x,y,z+1) - G(x,y,z-1) of the gradient vector. After that the vector length is computed to normalize the vector. Thus the vector components are squared and summed. The cordic algorithm is used to compute the square root. To perform these computations 16 clock cycles of 25 ns are necessary. The vector length is a 16 bit integer value. The implementation of the square root computation takes into account that one gradient can be computed every 7 clock cycles of 25 ns. Therefore the stages in this part of the pipeline are clocked with a lower frequency.

The output of the gradient unit is fed into the normalization unit. There the three gradient vector components are normalized by division with the vector length. This takes 9 clock cycles of 25 ns. The normalized gradient vector components are computed with 15 bit accuracy plus a sign bit. The design of the normalization unit takes into account that one gradient normalization is performed at most every 7 clock cycles of 25 ns.

The voxel interpolator contains additional hardware to support the generation of resampling points along a ray segment. This is necessary for the search and trace instructions described in section 3.8. To generate the resampling points along a ray segment the first resampling point and the difference vector between two resampling points is given. The (x,y,z) coordinates of a random resampling point can be split into an integer part (x_i,y_i,z_i) and a fractional part (x_f,y_f,z_f) with $0 \le x_f,y_f,z_f < 1$. The fractional part is stored in the voxel interpolator and presents the weights for linear interpolation in each direction. To generate the next resampling point from the current resampling point the difference vector has to be added. The addition of the fractional parts is performed in the voxel interpolator. The addition of the integer parts and the carries from the fractional parts is done in the address generator. The generation of a new resampling point and thus the new interpolation weights is possible within one clock cycle.

During previewing, points along a ray segment are generated until a threshold is exceeded. To implement this instruction a threshold register together with a comparator is contained in the voxel interpolator. The voxel value at the resampling point is compared with

the threshold. A signal is sent to the address generator to terminate the further generation of resampling points if the threshold is exceeded.
A counter is implemented to count the number of resampling points from the beginning of a ray segment until the threshold is exceeded. The number of resampling points of a ray segment is counted if threshold is not exceeded. It is also possible to count the number of resampling points of a ray segment for the trace instructions.
The voxel interpolator is implemented in a chip with 190 pins. The chip area is 150 mm^2 using 1.0µ CMOS technology. FIFOs are implemented as dual port RAMs using a megacell compiler. The logic of the design is implemented with standard cells.

3.6 Segment Interpolator

The segment interpolator (lower part of figure 3.1) is able to determine the nearest voxel to a given resampling point on a ray. It selects the segment number of that voxel. Selection is implemented in two pipeline stages. It then transforms the segment number into voxel color (RGB), voxel opacity, and a highlight coefficient for shading. Transformation is done by means of an on-chip translation lookup table within one clock cycle. The translation lookup table can be loaded from the raytracing processor.
To determine the nearest neighbor to a given resampling point the segment interpolator uses the fractional part (x_f,y_f,z_f) of the coordinates. The segment interpolator contains the same adder circuitry that is implemented in the voxel interpolator to support generation of resampling points along a ray segment. This is necessary for the trace instructions. A new resampling point can be generated within one clock cycle.
The segment interpolator is implemented in a chip with 148 pins. The chip area is 80 mm^2 using 1.0µ CMOS technology. FIFOs are implemented as dual port RAMs using a megacell compiler. The logic of the design is implemented with standard cells.

3.7 Address Generator

The address generator (left part of figure 3.1) contains the scheduler. As described in section 3.2 operations are scheduled as soon as pipeline resources become available.
The address generator performs several tasks.
First, it selects one of the eight memory banks if a single voxel with integer coordinates is written into voxel memory or is read from memory. A mapping of the voxel coordinates to a memory address is performed as described in section 3.4. An error signal is generated if the voxel that should be read is not stored in the voxel memory.
Second, the address generator supports resampling. It generates the voxel addresses of all eight voxels of the octet that is used to perform the trilinear interpolation for a given resampling point. The eight voxel addresses are mapped to memory addresses. The memory addresses determine the voxels in the corresponding memory banks. The address mapping for the eight voxels is done in parallel.
Third, the address generator automates gradient computation. It generates all six resampling points that are necessary to compute the gradient vector. For each of the six resampling points a resampling operation is performed like described above.
Fourth, the address generator generates resampling points along a ray. It uses the first resampling point of a ray segment together with the incremental vector between two resampling points to iteratively generate the next resampling point. Only the integer coordinate parts of the resampling point and the incremental vector together with the carries from the fractional sums are added in the address generator. Coordinates of resampling points are checked. If a resampling point is outside the assigned subcubes of the processor, an error signal is generated. This mechanism is used to determine wether a newly generated resampling point still belongs to the ray segment or the ray leaves the processors responsibility.

The address generator consists of three pipeline stages. In the first stage the voxel addresses for the eight voxels of an octet are generated. During gradient computation the coordinates of the six resampling points are generated in this stage. The coordinates of the successor of a resampling point are generated in the first stage, too. Some work of the coordinate to address mapping concerning supervoxels is also done in the first stage.
The second stage performs the next part of the coordinate to address mapping. The type of the subcube is determined if an access to replicated data takes place. Read and write signals for the memories are generated. An error bit is generated if a voxel address is outside the processors responsibility.
In the third stage coordinate to address mapping completes. Chip select signals for the memory banks are generated. An error signal for gradient computation is composed from the sequence of error signals of stage 2. This helps to mark a gradient invalid if one of the six resampling points for the gradient computation is outside the processors responsibility.
After the third stage signals leave the chip and are connected to the address and control lines of the memory banks. Memory is the fourth stage of the voxel processor pipeline.
To bring pipeline clock cycle close to memory access time, we have not implemented the registers after the third stage within the address generator chip. Instead, we have used external registers, as shown in figure 3.8.
In figure 3.7 the clock period is determined by the following delays: At the beginning of a clock cycle signals of stage 3 are valid at the output of register R3 after a register delay T_{R3}. Then signals have to leave the address generator which causes a pin delay $T_{pin,out}$. After memory access time $T_{acc,mem}$ signals are valid at the memory output. Then the signal has to enter the voxel or segment interpolator which causes an additional pin delay $T_{pin,in}$. Before the next clock cycle the setup time of register R4 $t_{setup,R4}$ has to be met.
Thus: $T_{min} = T_{R3} + T_{pin,out} + T_{acc,mem} + T_{pin,in} + t_{setup,R4}$

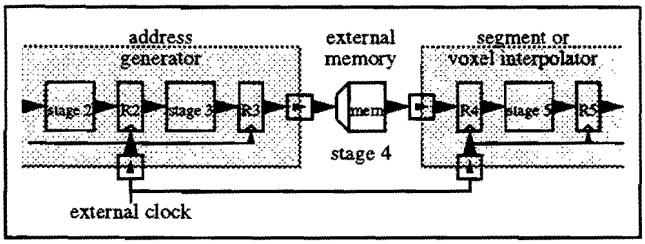

Figure 3-7 Pipeline with register R3 inside the address generator.

The solution shown in figure 3.8 performs better as the comparison shows:
At the beginning of the first clock cycle the signals of stage 3 have already left the address generator and have propagated to the inputs of external register R3. There is a clock skew between the clock at external register R3 and the clock signals within the chips at internal registers R2 in the address generator and R4 in the voxel or segment interpolator. The clock signal at R3 is earlier because there is a pin delay for the external clock to enter the chips and an additional in-chip delay caused by the clock distribution networks inside the chips. So, after the first clock cycle, we have a register delay caused by external register R3. After that, the memory access time plus the pin propagation delay to enter the voxel or segment interpolator must be considered. Before the next clock

cycle the setup time of register R4 has to be met.
Thus $T_{min} = T_{R3} - T_{skew} + T_{acc,mem} + T_{pin,in} + t_{setup,R4}$

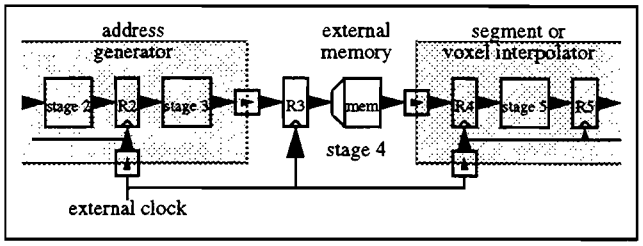

Figure 3-8 Pipeline with external register R3.

Because the external register is implemented with Advanced BICMOS technology (Texas Instruments) propagation delay is very small (clock to output 1.5 ns min., 4.5 ns max.).
The design of figure 3.8 allows a 25 ns pipeline clock period with 20 ns access time memories. With the solution of figure 3.7 we would have been 10% slower for the typical case and 29% slower for the worst case. The solution of figure 3.8 has the additional advantage that the adress lines of the memory chips are externaly buffered. This avoids high peak currents for the address generator chips caused by charge effects of the memories input capacitances.
The address generator is implemented in a chip with 244 pins. The chip area is 135 mm^2 using 1.0µ CMOS technology. FIFOs are implemented as dual port RAMs using a megacell compiler. The logic of the design is implemented with standard cells.

3.8 Instruction set and performance

We must distinguish between the number of clock cycles (25 ns for DIV^2A) it takes from the beginning of an instruction until we have a valid result and the number of clock cycles the pipeline is blocked and no other instruction can be performed in meantime. Throughput of the pipeline is determined by the last value if the pipeline can be kept filled all the time. The FIFOs at the beginning and at the end of the pipeline decoupling the voxel processor from the raytracing processor help to keep the pipeline filled.
The first group of instructions stores voxel values, segment numbers, or both in voxel memory. If the data does not belong to one of the assigned subcubes or replicated data of the voxel processor, the store instruction behaves like a no-operation. This simplifies the loading of the voxel memories because all raytracing processors present the whole data volume to their voxel processor. The voxel processors store what they need.
There are instructions to retrieve voxel values and segment numbers from voxel memory. The instructions assume that the coordinates are on the voxel grid. All instructions of the first group block the pipeline for one clock cycle.
The second group of instructions performs resampling operations. We can get the voxel value and segment information at a resampling point. This blocks the pipeline for one clock cycle. Additionally we can get the voxel value, segment information, and gradient at a resampling point. This blocks the pipeline for seven clock cycles.
The third group of instructions supports the construction of ray segments. Let n be the number of resampling points of a ray segment.
The search instruction gets the first resampling point of a ray segment and generates all further resampling points until a subcube is left or a threshold at a resampling point is exceeded. The search instruction is used in previewing mode.
The instruction blocks the pipeline for 2 + n clock cycles if threshold is not exceeded. To perform the resampling at the n resampling points takes n clock cycles. Two additional clock cycles are necessary to detect that a resampling point is outside the subcubes assigned to the processor. The error signal is complete after the third pipeline stage. Therefore two invalid resampling points have been generated in the meantime. If threshold is exceeded at point m in the ray segment the pipeline is blocked for min(13 + m,2 + n) clock cycles. In this case m resampling operations have to be performed which takes m clock cycles. Because exceeding of the threshold is detected in stage 13 of the pipeline (voxel interpolator) up to 13 superfluous resampling points have been generated and are streaming through the pipeline. If additionally a resampling point outside the processors responsibility was generated in the address generator, it has already stopped generating resampling points for that ray segment. Then 2+n can be less than 13+m. In every case the number of clock cycles is at most the number of clock cycles we had needed if threshold were not exceeded.
The trace instructions get the first resampling point of a ray segment and generate all further resampling points in a subcube. Trace1 computes the voxel value, segment information, and gradient at each resampling point. It blocks the pipeline 2 + 7*n clock cycles. Trace2 does not compute the gradients at the resampling points and takes only 2 + n clock cycles. To perform the resampling at n resampling points takes n clock cycles for the trace2 instruction because of gradient computation and 7*n clock cycles for the trace1 instruction. Two additional clock cycles are necessary to detect that a resampling point is outside the subcubes assigned to the processor. The error signal is complete after the third pipeline stage. Therefore two invalid resampling points have been generated in the meantime.
The fourth group deals with loading the segment lookup table in the segment interpolator. Reading or writing a segment entry takes one clock cycle.
There are some miscellaneous instructions that can be used to reset the pipeline or to clock the FIFOs when the results of an instruction are read. Clocking the FIFOs does not consume pipeline clock cycles.

4. Conclusion

We have introduced a parallel architecture designed for real-time volume visualization. Volume data is divided into small cubes and distributed among multiple image processors. Raytracing is supported by a powerful pipelined voxel processor. A dynamically configured multifunction pipeline with data-stationary control is used. Because of the high pin count the pipeline is implemented in three chips. The pipeline controllers are replicated in each chip to reduce the number of pins and to avoid timing constraints. The voxel processor uses trilinear interpolation for resampling. It is able to perform gradient computation and to generate resampling points on ray segments. At the subcube boundaries additional voxel data is needed to perform resampling and gradient computation. Data replication is used to overcome the problem and to guarantee high pipeline throughput. Voxel memory consists of 8 independent memory banks. A coordinate to memory address mapping is presented that allows to fetch a set of eight voxel values for trilinear interpolation from memory in one cycle.
The voxel processor can perform a trilinear interpolation and the computation of the next resampling point in one 25 ns pipeline clock cycle. A 16 processor architecture is able to visualize a 256^3 voxel volume at a rate of 20 images per second. Each image consists of 256^2 pixels and uses a byte for each RGB color component.

We are currently building a prototype consisting of 4 image processors interfaced to a Sun workstation. This prototype is not expected to perform in real-time. The voxel processor is designed for a 40 MHz clock. We use a subcube size of 16 for a 256^3 voxel volume. The voxel processor consists of 3 custom chips designed with 1.0 µ CMOS standard cells of ES2. Chips are fabricated by

ES2 within the Eurochip Project.
The next prototype will have 16 image processors and is expected to perform in real-time. We plan to implement 3D segmentation on DIV^2A and two expand the architecture for radiation treatment planning in medicine which is very time consuming. Because radiation has similarities to the behavior of light, raytracing could be used, too.
This research is supported by the German Science Foundation.

References

[1] Camahort, Emilio and Indranil Chakravarty, 'Integrating Volume Data Analysis and Rendering on Distributed Memory Architecture,' *1993 Parallel Rendering Symposium Proceedings*, (San Jose, California, October, 1993), 89-96

[2] Guenther, T. and C. Poliwoda and C. Reinhart and J. Hesser and R. Maenner and Hans-Peter Meinzer and H.-J. Baur, 'VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine,' *Proc. of the 9th Eurographics Hardware Workshop*, (Oslo, Norway, September, 1994), 103-108

[3] Kaufman, Arie et al., 'A Survey of Architectures for Volume Rendering,' *IEEE Engineering in Medicine and Biology*, (December, 1990), 18-23

[4] Kaufman, Arie. '3D Volume Visualization,' *Advances in Computer Graphics VI*, Series Eurographics Seminars, Springer, (1991), 175-203

[5] Knittel, Guenter and Wolfgang Strasser, 'A Compact Volume Rendering Accelerator,' *ACM/IEEE Symposium on Volume Visualization*, (Washington, DC, October, 1994), 67-74

[6] Kogge, P. M., 'The Microprogramming of Pipelined Processors', *Proceedings 4th Annual Conference Computer Architecture*, IEEE No. 77CH 1182-5C, March, 1977, 63-69

[7] Lacroute Philippe and Marc Levoy, 'Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation,' *SIGGRAPH 94 Computer Graphics Proceedings*, (Orlando, Florida, July, 1994), 451-458

[8] Levoy, Marc. 'Display of Surfaces from Volume Data,' *IEEE Computer Graphics and Applications*, Vol. 8, No. 3, (May, 1988), 29-37

[9] Levoy, Marc. 'Volume Rendering by Adaptive Refinement', *The Visual Computer* (1990) 6, (1990), 2-7

[10] Meagher, Donald J. 'Applying Solids Processing Methods to Medical Planning,' *Proceedings NCGS'85*, (Dallas, TX, April, 1985), 101-109

[11] Mittelhäußer, Gangolf and Frithjof Kruggel, 'Fast Segmentation of Brain Magnetic Resonance Tomograms,' *Computer Vison, Virtual Reality and Robotics in Medicine*, First International Conference, CVRMed'95, (Nice, France, April, 1995), 237-241

[12] Neumann, Ulrich. 'Communication Costs for Parallel Volume-Rendering Algorithms,' *IEEE Computer Graphics and Applications*, (July, 1994), 49-58

[13] Neumann, Ulrich and Timothy J. Cullip, 'Accelerating Volume Reconstruction With 3D Texture Hardware,' Radiation Oncology Department, Department of Computer Science, University of North Carolina at Chapel Hill, *TR93-027*, (May, 1994)

[14] Pfister, Hanspeter and Arie Kaufman and Tzi-cker Chiueh, 'Cube-3: A Real-Time Architecture for High-Resolution Volume Visualization,' *ACM/IEEE Symposium on Volume Visualization*, (Washington, DC, October, 1994), 75-83