An array based design for Real-Time Volume Rendering

Michael Doggett*
School of Computer Science and Engineering
The University of New South Wales †‡

Abstract

This paper describes a new algorithm and hardware design for the generation of two dimensional images from volume data using the ray casting technique. The algorithm is part of an image generation system that is broken down into three subsystems. The first subsystem stores the input data in a buffered memory using a rearrangement of the original address value. The second subsystem reads data points from the buffered memory and shifts the data to computational elements in order to complete the viewing calculations for the image synthesis process. The final stage takes the results of the viewing calculations combined with the original input data to complete the surface rendering and pixel compositing to create the final image.

This paper focusses on the second subsystem which consists of two, two dimensional arrays of processing elements. The first array performs a limited angle, single dimension rotation by shifting the data. The second array performs a two dimensional ray casting operation where viewing rays are assigned to each processing element. The first stage is outlined in this paper and the final rendering stages are the subject of previous work. The hardware design associated with these algorithms is described and tested. It is estimated that this architecture is capable of producing 384×384 pixel images at speeds of 15 frames per second for 256^3 data sets. Real time generation of images of volume data is important in scientific applications of volume visualization and computer graphics applications which use volume graphics.

Additional Key Words and Phrases: volume visualization, graphics hardware, image generation , volume graphics

1 Introduction

New custom architectures for graphics systems are required to process the large amounts of data associated with volume data at video rates [10]. Custom hardware architectures have been proposed that are capable of generating real-time images from volume data [8, 16, 5, 7, 11, 15]. Volume data is a large data space made up of discrete points which typically lie on a three dimensional grid where each discrete point holds a data value commonly called a voxel. The synthesis of video rate images from large volume data is used in various applications of visualization such as medical imaging, and video rate image generation is essential in computer interface technologies such as virtual reality.

The generation of two dimensional images from volume data using transparency techniques, such as volume rendering, is essential for Volume Visualization. Examples of volume data which are sampled include, magnetic resonance imaging (MRI) and computed tomography (CT) data. Volume data can also be synthesized from traditional computer graphics primitives such as triangles. The generation of images from synthetic data is referred to as Volume Graphics. As hardware processing power increases Volume Graphics has the potential to replace polygon based graphics and place real-time 3D computer graphics on every computer [9]. This can be achieved by replacing two dimensional frame buffers with three dimensional frame buffers that can work with voxelized polygons stored as volume data.

2 Previous Work

Previous work on the system [2, 1] has concentrated on shading which occurs in the final stages of image generation. This paper details the design of part of the front end, being two arrays that shift input data to perform a partial rotation and processing elements that calculate the path of viewing rays through the volume data. The memory subsystem that enters the data into the first array is briefly outlined in this paper and is the subject of continuing investigation.

The design presented in this paper aims to describe a scalable and adaptable architecture for real-time volume visualization systems. To achieve this, a rotation and ray casting algorithm which uses arrays of processing elements to create a parallel pipelined subsystem was designed. The complete system uses an initial subsystem for volume data storage and a final subsystem for rendering. The array based algorithm was implemented and tested in software. A hardware description language was used to describe, simulate and verify the hardware functionality and calculate performance estimates for the array processing elements. The hardware description was then translated into the LSI gate array design environment in order to estimate the characteristics of a gate array implementation. Using the gate array results, the per-

*Email: miked@vast.unsw.edu.au
†Sydney 2052, AUSTRALIA
‡WWW: http://www.vast.unsw.edu.au/~miked/index.html

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

formance is estimated to achieve a video rate of 15 frames per second (f/s) for images of volume data.

3 Related Work

The Cube-3 architecture [15] is estimated to be capable of processing a 512^3 data set at 30 f/s. The Cube architecture uses a bus between input volume data and a set of buffers where data is stored before the image generation process begins. The transfer of data using this bus involves a template of values being calculated which determine the correct storage addresses for the data held in the buffers. The buffers store the data for the next stage of processing. The purpose of the bus transfer operation is to transfer data representing one row of one slice of a volume data cube in one transfer operation. The bus transfer operation is the projection calculation for parallel or perspective viewing of the data set.

The array based design reported in this paper differs from the Cube-3 architecture by having a simpler parallel transfer method between memory and the second subsystem for processing. The simplified addressing then requires the introduction of the two array designs to calculate arbitrary screen projections.

The Knittel system [11] is able to render images of 256^3 data sets at 2.5 f/s, and uses a VLSI pipeline for the calculation of surface normals and Phong shading. The performance of this architecture is improved by using a distributed volume data memory and sending packets between processing elements to represent the traversal of rays through the data space. This parallel implementation is capable of 20 f/s for 512^3 data sets using a 64 processing element array.

The architecture presented here eliminates the need for a network of processing elements and packets containing ray traversal information. The ray casting operation is performed within one array that aligns the volume data with the correct ray for image generation.

Results from algorithms capable of speeds of a few frames per second for data sets of 128^3 to 256^3 using supercomputers [17] and standard workstations [12] have also been reported. These real time results are dependent on either large supercomputer systems or sparse data sets. For higher frame rates and larger data sets, increased processing power is required.

4 Image Generation from Volume Data

4.1 Introduction

The synthesis of images from volume data is referred to as volume rendering [13, 3]. There are several approaches to this rendering process, one being ray casting [13]. Ray casting is similar to ray tracing, except that in ray casting rays travel through the data set once without creating reflection rays. The ray casting used in this system is based on discrete increments along each ray similar to discrete ray tracing [18]. As the rays travel through the data, sample points are extracted and used to calculate local gradients surrounding the sample point which is used in opacity calculations to combine the sample points to create the final pixel value. The shading of the surface is calculated using either the diffuse shading or Phong shading equations [4].

Ray casting and ray tracing are examples of image synthesis that is based on a pixel by pixel calculation. The standard ray casting algorithm allows rays to travel from any viewpoint through the data space. This requires that all data points in the memory are available to each processor that calculates the traversal of a ray. This creates a bottleneck at the input memory for a single processor calculating all ray traversals. A solution for parallel systems is to pass data packets containing ray traversal information from one parallel processing element to another. This passing of ray packets requires a network and adds overheads to the ray casting algorithm. The objective of the ray casting algorithm presented in this paper is to create a parallel algorithm that passes data in the correct order to a set of processing elements without the memory or network bottlenecks. To achieve this the algorithm presented uses a limited range of possible viewpoints for the ray casting algorithm. Arbitrary viewing is accomplished by adding two preprocessing stages. The first stage is called coordinate swapping, and the second, X axis rotation. The final stage requires a modified ray casting algorithm in the X, Z plane.

The data that enters the first stage are voxel values, V_a, which are represented using a right handed coordinate system called the world coordinate system. In the same coordinate system a viewing direction is specified using a cylindrical coordinate system represented by the values θ and ϕ. The coordinate value for one of the eight corners of the volume data set is equal to the origin of the world coordinate system and the data set increases in size along the positive x, y and z axes.

4.2 Coordinate Swapping

The coordinate swapping stage of the image generation process involves the rearrangement of coordinate values for each voxel which reorders the input data in terms of the new coordinate system. The world coordinate system used for input voxel data is changed to the new coordinate system and called the limited view coordinate system and changes all possible viewing angles in the range of $(0 < \theta < 360, 0 < \phi < 180)$ to $(225 < \theta < 315, 90 < \phi < 180)$. This mapping is accomplished by swapping and/or inverting the x, y, z coordinates of the input voxels depending on the viewpoint represented by the cylindrical coordinate values, θ and ϕ. The new coordinates are used in the X axis rotation. The required coordinate swapping for each view angle are shown in Table 1.

	Viewing Angles		*Axes Change*
Region	θ	ϕ	X+,Y+,Z+
1	$\frac{\pi}{4}$ - $\frac{3\pi}{4}$	0 - $\frac{\pi}{2}$	$Z-, Y-, X-$
2	$\frac{3\pi}{4}$ - $\frac{5\pi}{4}$	0 - $\frac{\pi}{2}$	$Y+, Z-, X-$
3	$\frac{5\pi}{4}$ - $\frac{7\pi}{4}$	0 - $\frac{\pi}{2}$	$Z+, Y+, X-$
4	$\frac{7\pi}{4}$ - $\frac{\pi}{4}$	0 - $\frac{\pi}{2}$	$Y-, Z+, X-$
5	$\frac{\pi}{4}$ - $\frac{3\pi}{4}$	$\frac{\pi}{2}$ - π	$X+, Y-, Z-$
6	$\frac{3\pi}{4}$ - $\frac{5\pi}{4}$	$\frac{\pi}{2}$ - π	$X+, Z-, Y+$
7	$\frac{5\pi}{4}$ - $\frac{7\pi}{4}$	$\frac{\pi}{2}$ - π	$X+, Y+, Z+$
8	$\frac{7\pi}{4}$ - $\frac{\pi}{4}$	$\frac{\pi}{2}$ - π	$X+, Z+, Y-$

Table 1: Input data rearrangements required for ranges of viewing angles.

4.3 X axis rotation

The second stage of the image generation process involves a two dimensional array which effectively performs a small rotation of the voxel data values. The rotation is about the X axis of the limited view coordinate system. The complete rotation calculation involves a translation to the origin, a three dimensional rotation, and a translation back to the original position. Using a homogeneous coordinate system [4], the translation and rotation operations result in the following equations :

$$R_x = x$$

$$R_y = y\cos(\theta_x) - z\sin(\theta_x) - \frac{n}{2}\cos(\theta_x) + \frac{n}{2}\sin(\theta_x) + \frac{n}{2}$$

$$R_z = y\sin(\theta_x) + z\cos(\theta_x) - \frac{n}{2}\sin(\theta_x) - \frac{n}{2}\cos(\theta_x) + \frac{n}{2}$$

where

x, y, z = world coordinate values
R_x, R_y, R_z = rotated coordinate values in the limited view coordinate system
θ_x = angle of rotation about the X axis calculated from the θ view point angle
n = size of the data set

To reduce the calculation complexity involved in performing this calculation for every coordinate an incremental calculation is used. The coordinates of a data point in the first plane and the last plane of the volume data set are rotated using the above equations. The rotated coordinates of the first plane point are used as the starting point and incremental values are added for each new point. The incremental value is calculated using the following equations :

$$y_{inc} = \frac{y_{z=n} - y_{z=0}}{n}$$

$$z_{inc} = \frac{z_{z=n} - z_{z=0}}{n}$$

where

y_{inc}, z_{inc} = incremental values for y, z coordinates respectively
$y_{z=n}, z_{z=n}$ = rotated y,z coordinates for points on the $z = n$ plane
$y_{z=0}, z_{z=0}$ = rotated y,z coordinates for points on the $z = 0$ plane
n = size of the data set

Once all initial and incremental values for coordinate rotation are calculated they are used in an algorithm for the X axis rotation. The algorithm is based on a two dimensional array data structure $W(x, y)$. Each element of the array stores the voxel data V_a, the voxel y value V_y and three boolean values *eq*, *gt* and *lt*. A comparison is made between the V_y value and the y value stored in each array element and the results stored in the three boolean values. The voxel data is also shifted along the x axis of the array. As data is shifted along the x axis the boolean values indicate whether the data moves to the adjacent array element or a row above or below the adjacent element. The x dimension of the array is n and the y dimension is $\frac{3}{2}n$ in order to handle up to 45 degree rotations. The input into the array is from the volume data and proceeds in a plane-by-plane, column-by-column processing order. For each column of each plane the data is input and rotated using the following algorithm :

```
for ( y = 1 to 3/2 n )
{
    W(n,y).V_y = y_{z=0} + (y × y_inc)
    W(n,y).V_a = V_{xc,y,zc}
    for ( x = 1 to n ) {
    {
        W(x,y).eq := (W(x,y).V_y equal to W(x,y).Y_pe)
        W(x,y).gt := (W(x,y).V_y greater than
            W(x,y).Y_pe)
        W(x,y).lt := (W(x,y).V_y less than W(x,y).Y_pe)
        if W(x,y).eq then W(x - 1, y) = W(x, y)
        if W(x,y).gt then W(x - 1, y) = W(x, y - 1)
        if W(x,y).lt then W(x - 1, y) = W(x, y + 1)
    }
}
```

where

eq = boolean value for equal y coordinates
gt = boolean value for voxel y coordinate greater than array y coordinate
lt = boolean value for voxel y coordinate less than array y coordinate
xc = current column
zc = current plane

4.4 Ray Casting

The final stage of the new image generation process is a modified ray casting algorithm that uses an array of rays that tra-

verse through the volume data in the X, Z plane. A two dimensional array, $R(x, y)$, stores voxel values, voxel coordinates, and ray coordinates in each array element. The ray coordinates, R_x and R_z represent the current location of the ray in the x, z plane. The dimensions of the $R(x, y)$ array are $\frac{3}{2}n$ in both x and y dimensions.

The ray at each array location moves through the data in an incremental fashion detecting intersections with data values as it progresses. The starting point and incremental values for each ray need to be calculated for each image. The viewing ray is determined by taking an initial value at the screen plane and a terminating value on the opposite side of the volume data. A line is created between the two values and the initial point of intersection with the volume data set is found. The incremental values are found using these initial and final values for each ray.

The view plane is the x, y plane at $z = -\frac{n}{4}$ in the view coordinate system. This view plane has to be translated to the same coordinate system that the X axis rotation uses so that coordinate intersections can be found between rays and the voxel values stored in the $W(x, y)$ array. The transformation of the ray coordinate values to the correct coordinate system requires a translation to the origin, then rotation, and then another translation back to volume data coordinates. The equations for the calculation of x, z values in the X axis rotation coordinate space are :

$$R_x = x\cos(\theta_y) + z\sin(\theta_y) + \frac{n}{2} - \frac{n}{2}\cos(\theta_y) - \frac{n}{2}\sin(\theta_y)$$

$$R_z = z\cos(\theta_y) - x\sin(\theta_y) + \frac{n}{2} - \frac{n}{2}\cos(\theta_y) + \frac{n}{2}\sin(\theta_y)$$

where

x, z	= original coordinate values
R_x, R_z	= rotated coordinate values
θ_y	= angle of rotation about the Y axis calculated from the ϕ view point angle
n	= size of the data set

The points on the viewing plane at $z = -\frac{n}{4}$ are transformed to the correct coordinate space with the viewing angle ϕ. Another x, y plane at $z = \frac{5n}{4}$, which represents the plane where rays terminate, is also transformed. Using the two planes as initial and final values for each ray the intersection point between the ray and the voxel data space is calculated. This intersection is found by finding the intersection between the line connecting the corresponding point on the two rotated planes and the faces of the cube which defines the voxel data space. This intersection point is then the initial value for each ray. The increment values for each ray are calculated using the corresponding points of the two planes and the following formula:

$$x_{inc} = \frac{R_{x=\frac{5n}{4}} - R_{x=\frac{-n}{4}}}{n}$$

$$z_{inc} = \frac{R_{z=\frac{5n}{4}} - R_{z=\frac{-n}{4}}}{n}$$

where

x_{inc}, z_{inc}	= incremental values for x, z ray coordinates respectively
$x_{z=\frac{5n}{4}}, z_{z=\frac{5n}{4}}$	= rotated x, z coordinates for points on the $z = \frac{5n}{4}$ plane
$x_{z=\frac{-n}{4}}, z_{z=\frac{-n}{4}}$	= rotated x, z coordinates for points on the $z = \frac{-n}{4}$ plane
n	= size of the data set

A two dimensional array, $R(x, y)$, stores at each array element three voxel values, the coordinates for the current plane voxel value, and the coordinates of the ray associated with the array element. The array element also stores a series of boolean values. The $x = 1$ column of the $W(x, y)$ array contains the voxel values and associated coordinates which are assigned to the $n = \frac{3}{2}n$ column of the $R(x, y)$ array. The columns of data effectively move across the $W(x, y)$ array and into the $R(x, y)$ array. The $R(x, y)$ array is driven by the same data processing as the $W(x, y)$ array and uses the following algorithm to complete the ray casting operation :

for (y = *1 to* $\frac{3}{2}n$)
{
    $R(\frac{3}{2}n, y).V_x = xc$
    $R(\frac{3}{2}n, y).V_z = zIR + (y \times z_{inc})$
    $R(\frac{3}{2}n, y).V_a = W(1, y).V_a$
    $R(\frac{3}{2}n, y).V_b = R(1, y).V_a$
    $R(\frac{3}{2}n, y).V_c = R(1, y).V_b$
    for (x = *1 to* $\frac{3}{2}n$)
    {
        $R(x, y).eq = (R(x, y).R_x$ *equal to* $R(x, y).V_x)$
            and $(R(x, y).R_z$ *equal to* $R(x, y).V_z)$
        $R(x, y).V_a = R(x + 1, y).V_a$
        $R(x, y).V_b = R(x + 1, y).V_b$
        $R(x, y).V_c = R(x + 1, y).V_c$
        $R(x, y).V_x = R(x + 1, y).V_x$
        $R(x, y).V_z = R(x + 1, y).V_z$
        $R(x, y).v1 = R(x + 1, y).v1$
        if $R(x, y).eq$ *then*
            if $(R(x, y).v1$ *used) then*
                $R(x, y).pv2 = R(x, y).R_{xf}, R(x, y).R_{zf}$
                $R(x, y).R_x = R(x, y).R_x$
                    $+R(x, y).x_{inc}$
                $R(x, y).R_z = R(x, y).R_z$
                    $+R(x, y).z_{inc}$
            else
                $R(x, y).pv1 = R(x, y).R_x, R(x, y).R_z$
                $R(x, y).v1$ = *true*
    }
}

eq	= boolean for ray intersection with data value
$v1$	= boolean for when first passed value is used
$pv1$	= first set of passed ray intersection values
$pv2$	= second set of passed ray intersection values
V_b	= previously processed plane of voxel values
V_c	= plane of voxels processed before the V_b plane

The output from the $R(x, y)$ array is the intersection points and associated voxel values which are used in surface shading and image compositing. When an intersection is found the fractional parts of the current ray location are passed through the array to the the $n = 1$ column where they are used in the shading calculations. If more than one intersection is found at a particular voxel location a second set of intersection values are passed for shading calculation. When a ray intersects a voxel value, the coordinates of the ray are incremented and the ray isn't incremented again until the next intersection. The distance between the sample points that a ray takes is equal to the distance between voxel values in the data set to ensure that no more than two samples are ever taken at one voxel location.

4.5 Surface shading and image compositing

The voxels values in the column $R(1, y)$ are used to calculate values for the pixels in the final image. The y value of the voxel in the $R(x, y)$ array is the screen y value for the pixel. The x pixel value is stored with the voxel in the $R(x, y)$ array. A window of 3^3 voxel values is taken from the $R(1, y)$ column one 3×3 plane for each iteration of the complete algorithm. The gradient at the centre of the window of values is calculated and used as a surface gradient for the centre point. The surface gradient is used in a diffuse lighting equation to calculate the light intensity and hence pixel value at the location of the centre voxel in the window. The complete description for surface shading and image compositing used in this system is described in [2, 1]. Alternative algorithms for surface shading and image compositing, that are implemented in real-time systems, are outlined in [11, 15].

5 Hardware Design

The architecture of the system into which the new image generation algorithm fits is shown in Figure 1. The system is broken into several major components including two custom hardware arrays of processing elements based on the algorithms described in the previous section outlining the new image generation algorithm. The system is capable of accepting input from a real-time data acquisition device and storing it in the memory system. The specialised memory represents the first stage and is a double buffered memory with each buffer holding one copy of the volume data. As data is entered into the double buffer the coordinate swapping operation is performed. The output from the memory subsystem is connected to the warp array, which performs

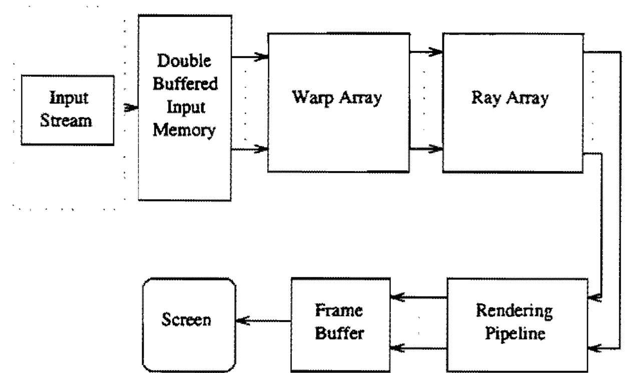


Figure 1: System level organisation

the X axis rotation described in Section 4.3. The warp array function is similar to a shear warp operation, but is a rotation and not a shear. The output from the warp array is connected to the input of the ray array, which performs the modified ray casting algorithm described in Section 4.4. The last stage performs the surface shading and image compositing and is broken into several rendering pipelines and a screen buffer.

5.1 Double buffered memory

The first stage of the system is a double buffered memory which performs the coordinate swapping operation described in Section 4.2. A double buffered memory is used so that data can enter the system at the same time as data is read into the warp array without conflicting accesses. The address of voxel data is altered as it is written into the double buffered memory.

An incrementor starting at the origin point of the voxel values and incrementing to the last row and column of the final plane of the voxel data is used to calculate the address of input data. The incrementor value represents the X, Y and Z coordinates with 8 bits for each coordinate to accommodate a 256^3 data set. The coordinate values are then recalculated according to the view point with coordinate swapping described in Section 4.2. Three multiplexors and inverters are used to create the new address where the data is stored in the active buffer for input data. Once the incrementor reaches the maximum value the buffer switch is changed and the process repeats. The buffer switch is used to select between the input data buffer and the buffer where values are read into the warp array. The design of the double buffer is shown in Figure 2.

The input data value addresses are calculated sequentially. To improve the performance of the address calculation process multiple address calculations units can run in parallel. Data set sizes of 512^3 and 1024^3 would require parallel address calculation units.

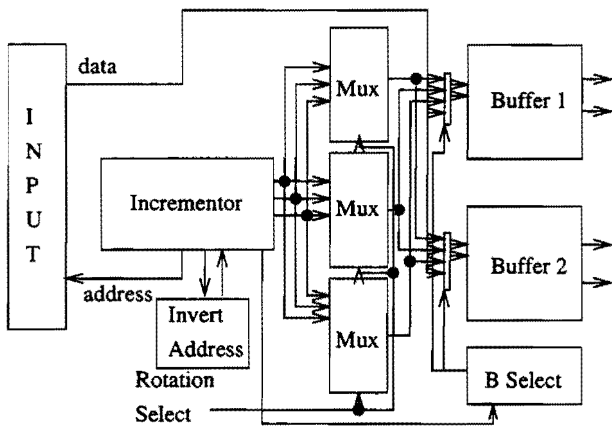


Figure 2: Input data double buffer for data rearrangement

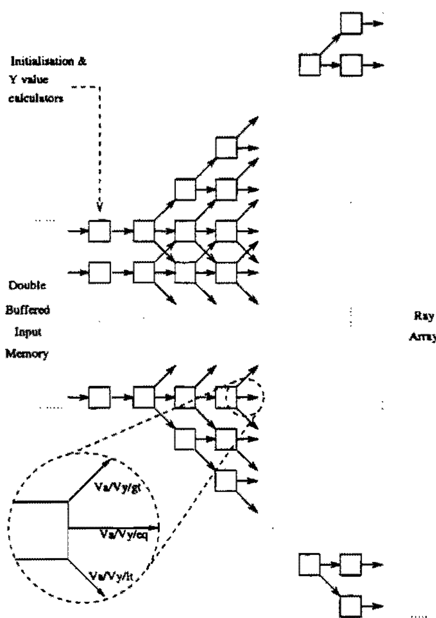


Figure 3: The Warp Array organization

5.2 The Warp Array

The warp array is an array of processing elements which perform the viewing rotation described in Section 4.3. The organization of the warp array is shown in Figure 3. Initialisation calculations are required for the coordinate data used in the warp array. The incremental calculations require one addition operation for each voxel value. The projection initialisation operations are more complex and require a dedicated digital signal processor (DSP) to calculate the initial values once per frame. The initial value for each row and an incremental value are loaded into two registers which, combined with an adder, can calculate the Y value input for each row of the warp array.

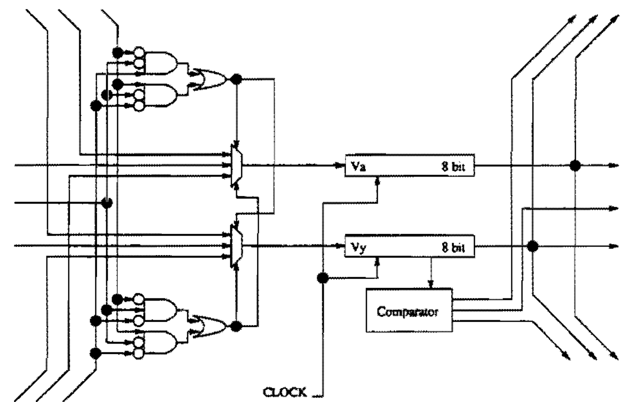


Figure 4: Warp Array processing element

V_a	Original voxel data value
V_x, V_y, V_z	X, Y and Z coordinates of data value after X axis rotation
Y_{pe}	Y-position of processing element from top of warp array
R_{xi}, R_{xf}	Integer and fractional components of the X-coordinate of the ray
R_{zi}, R_{zf}	Integer and fractional components of the Z-coordinate of the ray

Table 2: Notation for Warp and Ray Array processing elements.

5.3 The Warp Array processing element

The warp array processing element takes one set of inputs from three possible sets and sends the input to one of three possible outputs depending on the data's associated Y value. The entire array acts as a large shift register where data is stored and shifted towards its correct position within the array. Each processing element has a Y_{pe} register which holds the y value of the array element. The Y_{pe} value and the V_y value are compared and the voxel data and y value are passed to a processing element in the next column which is either above, below or adjacent to the current element.

The processing element uses two flip flops as registers to store the voxel value and its Y coordinate. The input to the registers is selected from the outputs of the adjacent processing elements. There are two select lines to the multiplexers which are driven by the comparator output of the adjacent processing elements. The output of the register values are passed to the adjacent processing elements on the output side. A comparator is used for the Y coordinate comparison. The layout for the warp processing element can be seen in Figure 4 and the values are described in Table 2. The output of the final column of the warp array is connected to the input of the first column of the ray array.

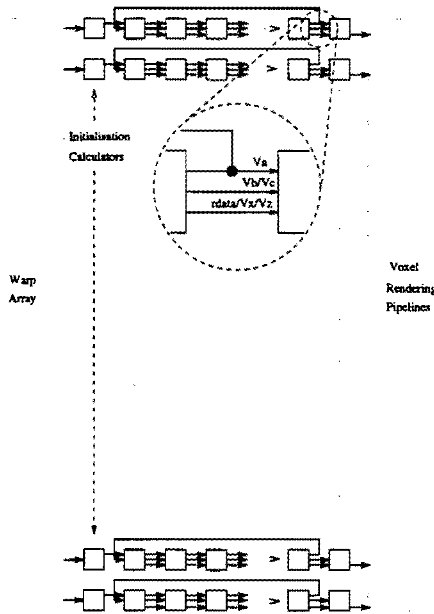


Figure 5: Ray Array organization

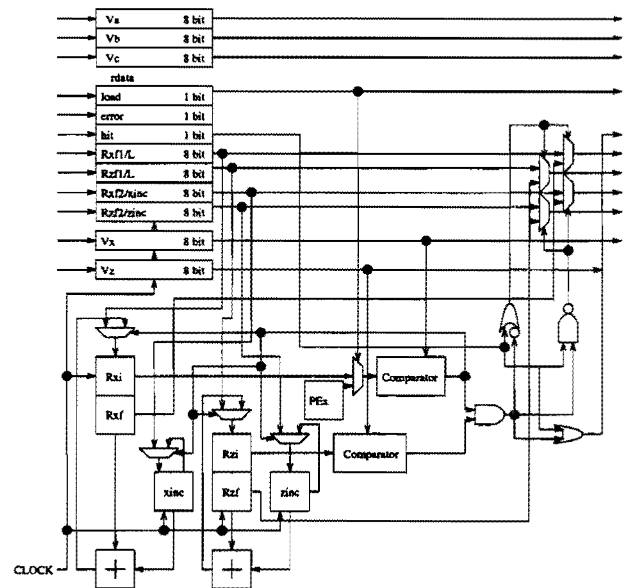


Figure 6: Ray Array processing element

5.4 The Ray Array

The ray array is a two dimensional array of processing elements which perform the operation of ray casting as described in Section 4.4. The initial values for the ray coordinates are calculated by a dedicated digital signal processor (DSP) chip and passed through the ray array to the correct processing element. The DSP chip calculates the initialisations described in Section 4.4, which are performed once per frame. The coordinates of the ray are stored in each processing element using a register with both an 8-bit integer and 8-bit fractional component. The integer component of the ray and the coordinates of the voxel data are compared to determine intersections with the data and when an intersection occurs the fractional component of the intersection is passed through the remainder of the array. Each ray array processing element accepts a set of values that contain intersection information from previous processing elements. After the intersection calculation is performed the result is combined with the input intersection data and sent to the following processing element. The output data is used by the shading operation in the surface rendering and image compositing stage. The organization of the ray array is shown in Figure 5.

The voxel data value V_a from the warp array is placed directly into the adjacent processing elements in the ray array. The voxel y coordinate, V_y, is not passed on to the ray array as it is no longer needed for computation. Each ray array element stores a voxel value from the current plane, V_a and the two previous planes, V_b and V_c.

5.5 The Ray Array processing element

Initialisation values are loaded into each ray array processing element using the a load bit and the registers that pass the intersection data. The load bit is used to switch multiplexors that input the initialisation data into the appropriate registers in the processing element. Each ray array processing element stores data using flip flops as data registers. The ray coordinate is incremented through the array using two 16-bit carry look ahead adders. An intersection is detected using a comparator to detect when the coordinates of a voxel and a ray are equal. At each intersection point, the fractional components of the ray location , R_{xf} and R_{zf} are output to the next processing element.

Once an intersection is detected by the comparator the intersection data in the input register is checked to see if a value already exists, if so the intersection data from the current processing element is placed in the second location. In Figure 6 the layout for a ray processing element is shown with the values explained in Table 2.

6 Results

The viewing and shading operations of the system were implemented in software and results were obtained. The processing elements in the warp and ray array were implemented in a hardware description language (HDL) and tested for functionality and performance. The viewing and shading operations were tested on both artificial and real data sets. Only the real data set results are discussed in this paper. The artificial data sets showed that all arbitrary viewing angles worked correctly.

6.1 Software Simulation

The results from the software simulation for a 40^3 sized data set containing several spheres is shown in Figure 7(a). In Figure 7(a) the viewpoint is set to $\theta = 45$ degrees and $\phi = 45$ degrees showing the performance of both the ray and warp arrays.

To test the two arrays on real sampled data an MRI of a human heart was used. Figure 7(b) shows the data set with no rotation.

6.2 Hardware Simulation

To test hardware functionality and to estimate the processing speeds of the warp and ray array, the processing elements were defined in an HDL and simulated using a switch level simulator [6]. The simulator takes account of gate delays and fan-in and fan-out conditions. Both processing elements functioned as described, with the warp array processing element operating at $38MHz$, and the ray array processing element at $25MHz$.

6.3 System Performance

To estimate the performance of the complete system the results from the hardware simulations were used. A conservative clock rate of $20MHz$ will produce a pixel result every $100ns$. To estimate the requirements of a real time system, an example data set of size 256^3 was used to generate 384^2 sized images. This was found to require $32MB$ of 8-bit data in the double buffered memory. Assuming $10ns$ memory is used then 3 parallel read and write lines are required to perform at $15f/s$. The warp array requires an array of size 256×384 and the ray array requires an array of 384×384. The technology used for simulating the gate array layout is LSI logic's 0.6-micron LCA300K [14]. The maximum number of gates on one chip is 300,000. The hardware description was translated to the description language used by the LSI toolset and a gate array schematic generated to determine the number of gates that each processing element required.

Using these details the warp array requires a set of 25 chips configured in a 5×5 array. The larger ray array is implementable using a similar set of 5×5 chips, but only contains a 384×40 ray array. This reduced size requires that the ray array be reused approximately 10 times per frame. The following calculation estimates the time required to generate one frame in this system :

$100ns \times 256 \times 256 \times 10 = 65ms$

The frame time translates to a performance of 15 f/s.

7 Conclusion

This paper has presented a new algorithm and hardware design for the visualization of volume data. The warp array and ray array store the data as it is processed and perform the viewing and ray casting operations required for volume visualization. The reduction of the three dimensional ray casting algorithm to two dimensional is a direct result of the coordinate swapping process and the X axis rotation. The separation of ray casting from data storage and rendering allows these aspects to be customized for particular applications. The system is designed to allow flexibility in the sizing of both the warp and ray arrays to cost and performance considerations. The system utilises a high level of both pipelining and parallelism to provide real time frame rates for volume data sets. The system design is scalable and therefore allows it to be used to process larger data sets at higher frame rates.

8 Acknowledgements

The author would like to thank Professor Graham Hellestrand for his supervision and essential feedback, Dr Jayasooriah, Mr Günter Knittel and Mr Stephen Avery for their helpful discussions with regard to this work, and Dr Jürgen Hesser for the MRI data set of a human heart.

References

[1] DOGGETT, M., AND HELLESTRAND, G. A hardware architecture for video rate shading of volume data. In *International Symposium of Circuits and Systems* (May 1995), IEEE.

[2] DOGGETT, M. C., AND HELLESTRAND, G. R. A hardware architecture for video rate smooth shading of volume data. In *EuroGraphics Workshop on Graphics Hardware* (September 1994), EuroGraphics, pp. 95–102.

[3] DREBIN, R., CARPENTER, L., AND HANRAHAN, P. Volume rendering. *Computer Graphics 22*, 4 (August 1988), 51–58.

[4] FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. *Computer Graphics: Principles and Practice*. Addison Wesley, 1989.

[5] GÜNTHER, T., POLIWODA, C., REINHART, C., HESSER, J., MÄNNER, R., MEINZER, H.-P., AND BAUR, H.-J. Virim: A massively parallel processor for real-time volume visualization in medicine. In *Eurographics workshop on Graphics Hardware* (September 1994), pp. 103–108.

[6] HELLESTRAND, G. R. Modal: A system for digital hardware description and simulation. *Journal of Digital Systems 4*, 3 (1980), 241–303.

[7] JUSKIW, S., AND DURDLE, N. G. Interactive rendering of voumetric data sets. In *Eurographics workshop on Graphics Hardware* (September 1994), pp. 86–94.

[8] KAUFMAN, A., AND BAKALASH, R. Memory and processing architecture for 3d voxel-based imagery. *IEEE Computer Graphics and Applications 8*, 11 (November 1988), 10–23.

[9] KAUFMAN, A., COHEN, D., AND YAGEL, R. Volume graphics. *IEEE Computer 26*, 7 (July 1993), 51–64.

[10] KAUFMAN, A., HÖHNE, K. H., KRUGER, W., ROSENBLUM, L., AND SCHROEDER, P. Research issues in volume visualization. *IEEE Computer Graphics and Applications 14*, 2 (March 1994), 63–67.

[11] KNITTEL, G. A scalable architecture for volume rendering. In *Eurographics Workshop on Graphics Hardware* (September 1994), pp. 58–69.

[12] LACROUTE, P., AND LEVOY, M. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Computer Graphics* (July 1994), ACM SIGGRAPH, pp. 451–458.

[13] LEVOY, M. Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 5 (May 1988), 29–37.

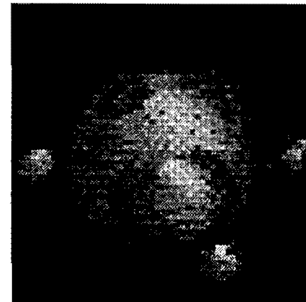
[14] LSI LOGIC CORPORATION. *LCA300K Gate Array 5 Volt Series Products Databook*, October 1993.

[15] PFISTER, H., KAUFMAN, A., AND CHIUEH, T.-C. Cube 3: A real-time architecture for high resolution volume vizualization. In *ACM/IEEE Symposium on Volume Visualization* (October 1994).

[16] STYTZ, M. R., AND FRIEDER, O. Volume-primitive based three-dimensional medical image rendering: Customized architectural approaches. *Computers and Graphics 16*, 1 (1992), 85–100.

[17] VÉZINA, G., FLETCHER, P. A., AND ROBERTSON, P. K. Volume rendering on the maspar mp-1. In *ACM Workshop on Volume Visualization* (October 1992), pp. 3–8.

[18] YAGEL, R., COHEN, D., AND KAUFMAN, A. Discrete ray tracing. *IEEE Computer Graphics and Applications 12*, 5 (September 1992), 19–28.



(a)

(b)

Figure 7: (a) A data set comprised of spheres at an arbitrary view point. (b) MRI heart Scan with no rotation