Hardware Architecture for Voxelization-based Volume Rendering of Unstructured Grids

C. E. Prakash and S. Manohar
Supercomputer Education and Research Centre
Indian Institute of Science
INDIA *

Abstract

Interactive volume visualization of unstructured grid data is a much sought after, but as yet elusive, goal in many scientific visualization applications. We present an architecture that can possibly bring this goal within reach. In this architecture we combine the recently identified method of using texture mapping for volume rendering[5, 2] with anti-aliased voxelization. We show how the proposed architecture can be implemented with simple extensions to existing high-end graphics systems, using the SGI RealityEngine as an example. The architecture has the advantage of providing both direct volume rendering and polygon based rendering at high performance levels on the same hardware platform. We present some simulation results that demonstrate the validity of our architecture.

1 Introduction

Volume visualization has become indispensable to the computational sciences in recent years. The sources for volume data, both computed and acquired, have become diverse: medical, molecular, geological, astronomical, to name a few areas. However, there has not been commensurate growth in the hardware support for volume rendering. The current high-end graphics systems are all tuned to provide support for polygon-based rendering. With one exception, volume rendering is implemented using one of the following methods:

- Extract a polygonal representation from the volume data and use the polygon-rendering hardware to render it.
- Use direct volume rendering methods in software, making use of any hardware support for image compositing, if available.

*Bangalore - 560 012, INDIA. email:manohar@csa.iisc.ernet.in

The former approach loses the benefits of direct volume rendering, like transparency, presentation of internal structure etc, while the latter suffers from long rendering times forcing the application to become non-interactive, or at best provide very low interaction rates. In this paper we focus on architectures for direct volume rendering.

Several research efforts are underway to provide hardware support for volume rendering. A survey of these architectures is presented in [19]. The CUBE architecture [6] and related work described in [19] are based on the supposition that it is time for volume graphics [13] to supersede surface graphics, just as surface graphics replaced vector graphics in the seventies. The long-term prospects for such an approach are quite promising. However, in the near-term such an approach will not be competitive with the volume rendering methods, listed above, using polygon-rendering hardware. The only hardware support for volume rendering that is available on current commercial workstations is the texture mapping hardware of the RealityEngine (RE) system of Silicon Graphics [1]. The use of this texture mapping hardware is to provide rapid volume rendering as described in [11]. However, for this method, it is assumed that the data is available as a regular 3D scalar grid. Most applications in the computational sciences generate unstructured grids of data.

Volume rendering of unstructured grids has recently received much attention. The two most common methods are *Cell Projection* (see [15, 24] and the references therein.) and *Ray Casting* (for example [10, 24, 3]). Both methods have been found to be extremely costly. There are several research efforts currently underway to provide rendering of unstructured grids at interactive rates [9, 25]. These methods are faster but compromise on the quality and, more important, on the accuracy of the rendered images.

None of the above methods have been implemented in hardware. In this paper we propose an architecture for the interactive visualization of unstructured grids. This architecture is based on a voxelization-based approach to the rendering of unstructured grids proposed in [18]: *Voxelization*, the process of converting a geometrically represented 3D object into a voxel model, is the 3D extension of the conventional scan-conversion process. Graphics hardware exists for scan-conversion, but we know of no commercial hardware that provides voxelization in hardware. We show that this architecture provides a seamlessly integrated hardware solution for voxelization and volume rendering that makes interactive volume visualization of unstructured grids feasible with current hardware. We indicate how this architecture can be implemented as a simple extension of an existing high-end system so that the resulting solution is highly competitive. We have chosen the Silicon Graphics RealityEngine(RE) architecture as an example, since it is the most popular high-end system and also because details of the architecture have been readily available. However, we note that the proposed architecture could be implemented by extending other high performance triangle rendering architectures.

The major features of the proposed architecture are:

- Supports both polygon-based rendering and voxel-based rendering in the same platform
- Requires simple extensions of the existing RE architecture
- Supports anti-aliased voxelization
- Enables interactive volume visualization of unstructured grids.

We consider volume visualization of unstructured grids as a two-step process: anti-aliased voxelization of the unstructured grid to generate a regular scalar grid, followed by direct volume rendering of the resulting scalar grid. We note that in several applications, like CFD, stress analysis, etc., The scalar values of the grid, or the geometry of the grid or both are time variant and hence voxelization has to be in the inner rendering loop. We enhance the voxelization step by the use of an anti-aliasing method called the 3D accumulation volume buffer (AVB) method.

In the next section we describe the voxelization algorithm that we use, followed by the anti-aliased voxelization algorithm in Section 3. A brief outline of using texture mapping for rendering regular volumes is presented in Section 4. The hardware architecture for implementing the above steps is described in Section 5. This section details the extension needed to the RealityEngine architecture to implement the proposed architecture. Results of simulations that demonstrate the validity of the proposed architecture are given in Section 6. We conclude by indicating possible extensions of this work.

2 Voxelization

In this section we tackle the issue of voxelization of unstructured grids. The most well known algorithms for voxelization are due to [14] that voxelize a polyhedron using a scan-plane method. The scan-plane is used to obtain the polygon of intersection and the polygon is filled to get the voxels which lie on the scan-plane. A set of scan-planes from top to bottom gives all the voxels in the polyhedron. In the following we present a recent algorithm ([18]) which will be used as the basis for the hardware implementation. This voxelization algorithm assumes the unstructured grid to consists of arbitrary convex polyhedra. For the hardware implementation however, we restrict the input cells to be tetrahedral. The need for this restriction are dealt with in Section 2.2.2.

2.1 Two-Buffers approach

We use a two-buffer approach to voxelization. The use of two buffers for graphics algorithms is not new. Two buffers have been used to integrate the scalar value within a cell [16, 23]. Our work extends the two-buffer idea for voxelization and volume rendering of unstructured grids.

Since the input cells are tetrahedra, the faces are all triangular. A given cell has four triangular faces of which some are facing the viewer 'known as the front faces' and the other faces are back faces which face away from the viewer. If the front faces are scan converted into one buffer and the back faces into a separate buffer, then for each pixel, the values at the corresponding location in the two buffers are used to interpolate the scalar values of all voxels which lie between the front and back faces of the cell. The depth buffer stores the corresponding one-dimensional buffers one for the left and another for the right extent of the polygon. The X, Z and $Scalar$ values are stored in the left and right buffers. These values are used in the scan-conversion and depth-conversion of the polygon into front and back framebuffers/z-buffers. The different buffers used to voxelize a tetrahedron are shown in Fig. 1. The overall voxelization procedure is described in the next section.

2.2 Voxelization

This section describes the coherent voxelization technique. The voxel is a unit cube and each voxel has a scalar value associated with it. The voxel volume consists of closely stacked $< N_x, N_y, N_z >$ voxels in the

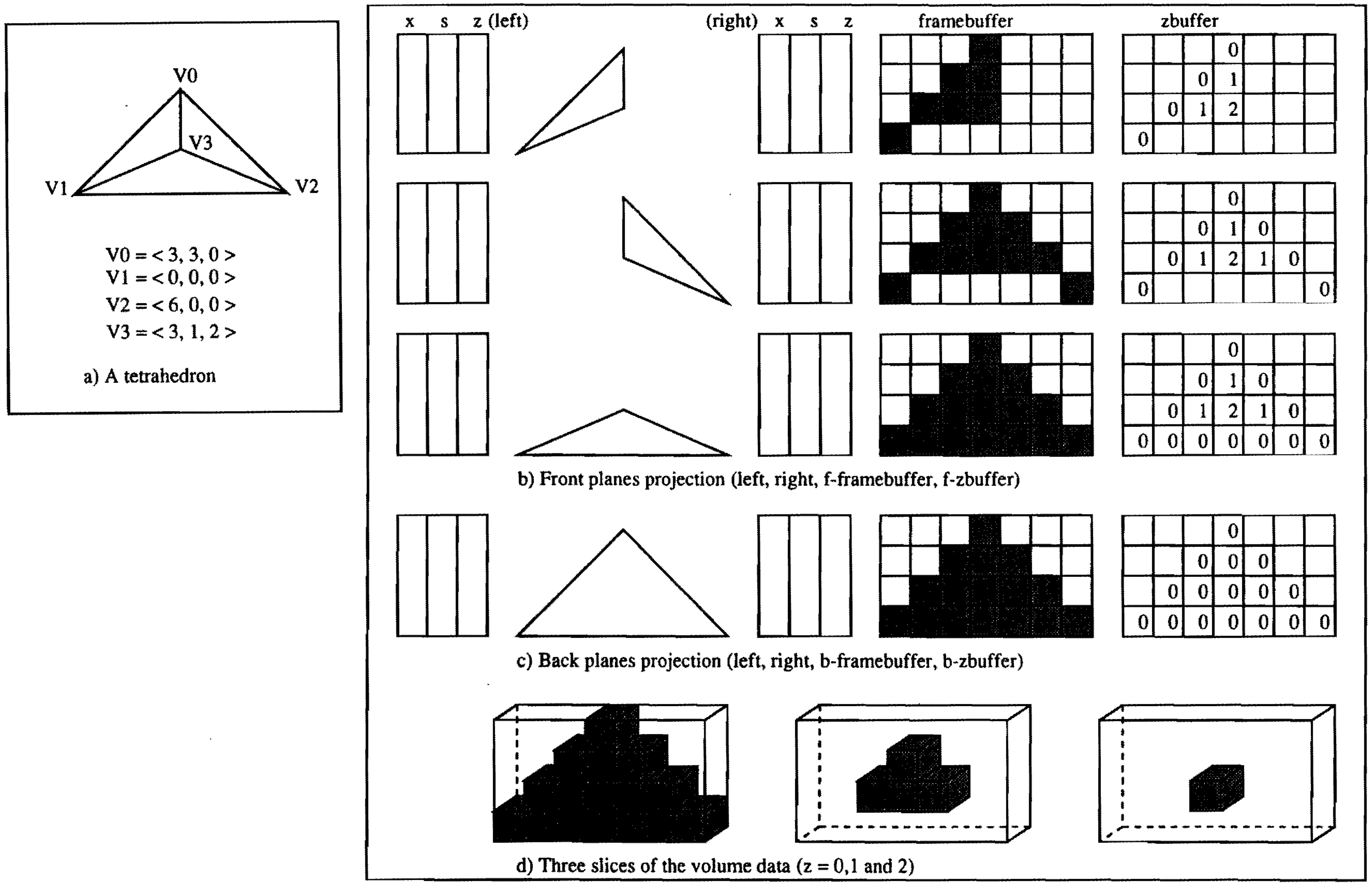
V0
V3
V1
V2
V0 = < 3, 3, 0 >
V1 = < 0, 0, 0 >
V2 = < 6, 0, 0 >
V3 = < 3, 1, 2 >
a) A tetrahedron
x s z (left)
(right) x s z
framebuffer
zbuffer
b) Front planes projection (left, right, f-framebuffer, f-zbuffer)
c) Back planes projection (left, right, b-framebuffer, b-zbuffer)
d) Three slices of the volume data (z = 0,1 and 2)

Fig. 1. Voxelization of a tetrahedron

$< x, y, z >$ direction respectively. The voxelization is done in cell-order. For each cell in the unstructured grid we need to compute the voxels which lie within the cell. Each cell is organized as a collection of polygonal faces. The front faces are projected into the front framebuffer and the back faces into the back framebuffer. The depth of the front and back polygonal faces are stored in the front and back z-buffers respectively. Since we maintain two buffers the order in which the faces arrive is immaterial. The next step is to compute the scalar values for voxels which lie between the front and back faces of the cell. The z-buffers give the extents of voxels within the cell along the projected pixel. The framebuffers contain the scalar value of the front and back voxel of the cell at that pixel. The summary of the algorithm is given below:

```
// VOXELIZATION of unstructured grid
voxelize_all_cells()
{
for ( All Tetrahedral Cells in the Volume) {
   for ( Each Triangular Face of the Cell){
     form_buffers(current_triangle)
     if(Front Face){
       //scan convert into front buffer
       scan_convert(current_triangle)
     }
     else if(Back Face){
       //scan convert into back buffer
       scan_convert(current_triangle)
     }
   }
   voxelize_between_two_buffers()
}
}
```

The *form_buffers*() procedure incrementally computes the extents of a triangle for each scan-line. The extents of each scan-line are used in the scan-conversion process. The *scan_convert*() procedure computes the scalar values and depth for all pixels in the projection of the triangle. The *voxelize_between_ two_buffers*() procedure interpolates the scalar values for all voxels within the cell.

2.2.1 Incremental voxelization between Two-Buffers

This section describes the incremental estimation of scalar values for all voxels which lie between the front and back voxels. The scalar values for the front and back voxels are obtained from the respective framebuffers at the corresponding pixel locations.

Since the projections of the cells are known, it is enough to compute scalar values for all voxels between the front and back projections of the cell. The voxels which belong to a cell lie between the depth values stored in the front and back z-buffers. An incremental value is computed from the scalar values available in the front and back framebuffers for each projected pixel. This value is added to the voxels incrementally along that projected pixel to get the scalar voxels within the cell. The following procedure shows the incrementation of voxel values from the framebuffer values.

```
// voxelize_between  two buffers
\\to get scalar value of VOXELS
voxelize_between_two_buffers()
{
for(Cell Ymin to Ymax){
   for(Cell Xmin to Xmax){
      if(The cell projects on this pixel){
         Get scalar value from front Fbuf
         //from front and back Fbuf
         Compute Incr_scalar
         // between front and back Zbuf
         for(All voxels ){
            V[x][y][z] = transfer_fn(scalar);
            scalar += Incr_scalar;
         }
      }
   }
}
}
```

By adding a 0.5 to the first voxel[14] a close rounding to the voxels is obtained. The scalar value for the front voxel gets the value from the front framebuffer. A more accurate method would be to interpolate from the neighboring pixels to obtain the scalar value of the start voxel. The scalar value is scaled to fit the voxel size (8, 16 or 32 bits). However, the transfer function allows non-linear or piecewise linear functions within a cell with trilinearly interpolated scalar values.

The main advantages of this algorithm over other algorithms for converting an unstructured grid composed of tetrahedral cells into a regular voxel grid are:

1. Uses a cell-order approach for voxelization (Completes all the voxels within a tetrahedral cell before marching to the next cell).
2. Each cell is accessed only once, each triangle in a cell only once and each edge of the triangle only once.
3. Handles one cell at a time. Information about other cells on the grid is not required. Hence there is no need to store the entire grid and no extra connectivity information is required. Therefore it is suitable for hardware voxel-engines with small buffers.
4. Easy to implement in hardware: No complicated data structures.

5. This is an accurate method for rendering unstructured grids (since depth information is properly accounted for) and helps to interpret the data in an unambiguous way.

6. Voxelization of 3D grids is decomposed into an one-dimensional problem. Linear interpolation is done along the edge, along the scan-line, and along the depth.

7. The algorithm is incremental where all the computations are reduced to incrementations (additions).

2.2.2 Tetrahedral cells

Even though our method works for any convex cell, we restrict the implementation to handle tetrahedral cells due to the following reasons:

- Tetrahedron has triangle faces only and the number of inputs are constant, viz, x, y, z, color for three vertices
- For polygons with more than 3 vertices it is difficult to ensure that all points lie on the same plane whereas triangles are planar.
- Interpolation on a polygon is dependent on the direction of scan conversion. In a triangle interpolation is not direction dependent.

3 Anti-aliased voxelization

Just as aliasing is a problem for the scan conversion of 2D primitives, the voxelization of tetrahedra is a discretization process and hence suffers from aliasing effects. This has been the focus of recent work especially in the context of volume graphics (see [21, 22, 17]). We have developed a new anti-aliasing algorithm for voxelization, called the 3D accumulation buffer method[17], that is highly suitable for hardware implementation.

This method is an extension of the accumulation buffer method [12] used for anti-aliasing 2D polygons. The Accumulation buffer accumulates/ integrates multiple point samples taken in the region of each pixel. The Point Sampling hardware is used to render multiple images, each with the sample point jittered by a specific amount. These images are then integrated to form the final, anti-aliased result. The method was first proposed by [8]. It uses accumulation to create anti-aliased images by rendering the scene repeatedly with subpixel offsets.

The 3D accumulation buffer is a straightforward extension of the above method: accumulate the voxel values sampled at multiple sub-voxel sample points located within a voxel.

3.1 Accumulation Volume Buffer (AVB)

The Accumulation Volume Buffer (AVB) method is independent of the voxelization algorithm used. What is required is to voxelize each object multiple times, each time providing an offset to the object, and then accumulating the partial voxel values generated by the voxelization step. For example, to voxelize a tetrahedron with anti-aliasing, the tetrahedron is transformed by sub-pixel offsets and voxelized for each offset position. The average scalar value is accumulated to the existing voxel value. To voxelize unstructured grids, all the cells in the grid are offset by sub-voxel positions, and the net effect is same as mentioned above for a tetrahedral cell. The standard way to generate these offsets is to have $2 \times 2 \times 2$ offset points within a voxel, where each offset is the center of the octant within the voxel. When the voxel has eight sub-samples, positioned at 8 sub-voxel locations, then the effective volume buffer resolution is eight times the original resolution. The next higher version in our method uses $3 \times 3 \times 3$ offset points(ie 27 sample points within the voxel). We have also generated antialised voxelizations with $4 \times 4 \times 4$ and $5 \times 5 \times 5$ sample points (ie 64 and 125 samples within a voxel). The pseudocode for the AVB method is as follows:

```
voxel_accumulation_buffer()
{
  for(each offset point within the voxel)
  {
    // tet is the tetrahedron
    translate(tet, offset, new_tet)
    voxelize(new_tet)
  }
}
```

From a programming standpoint, all that is required is multiple passes through the grid data base, each preceded by a small translation of the voxel offset. No change is necessary to the data base or to the process of its traversal. Even though we have used a $n \times n \times n$ super-sampling in our work, there is no reason to be confined to regular $n \times n \times n$ sample points. A possible method would be to have N random sample points within the voxel. The random function can be a Gaussian as used by [12]. Several experiments were done and we have found that our antialising method improves the quality of voxelization as measured by some new error measures that have also been proposed in [17].

The same accumulation technique may be extended to other voxelization methods: In the case of the scan plane method[14], the scan plane is translated to sub-voxel positions and intersected with the grid. The values of each pixel in the scan-plane are then accumulated to obtain the anti-aliased scan-plane. An alternative method is to translate the entire grid to sub-voxel positions and then do the scan-plane intersection. Multiple passes for the entire grid is done for the required number of sub-voxel sample points. The same technique can be used for the octree based voxelization [20] to support anti-aliasing.

4 Volume rendering with texture mapping

A conceptually simple way of rendering a voxel volume is to treat each plane of the volume as a 2D texture and then to apply the texture to a rectangle. For example to render a 128x128x128 volume, we can consider the volume to be an array of 128 textures of size 128x128. This volume can be rendered from any view point simply by defining 128 rectangles and mapping successive elements of the array of textures on to the rectangles. If the texture mapping and rendering is done using composition of the overlapping textured rectangles then the resulting image is very close to a direct volume rendered image. Alternately, we can consider the volume as a 3D texture array and apply the texture during the rendering stage onto a stack of polygons which intersect the volume in an orientation perpendicular to the view-position. Such an approach has been proposed by Akeley [1] [7], who uses the texture mapping hardware of the Silicon Graphics RealityEngine to render volume data. A similar approach using the Kubota Pacific Denali hardware is used in [11].

5 Architecture for unstructured grid volume visualization

Our objective is to develop a hardware architecture for the two-step volume visualization of unstructured grids described above. The two steps are, voxelization with anti-aliasing, and texture-mapped volume rendering. Clearly, the texture mapping step has already been well implemented in hardware ([1] and [11]). Thus we need not re-invent the wheel, but simply use the texture mapping hardware of any RE-like hardware. What is not obvious is that most of the computations required for the voxelization and anti-aliasing, are already being performed in an RE-like architecture. Thus it will be seen that with simple modifications to an RE-like architecture, the two steps can be seamlessly integrated in hardware. We take the RE architecture as the base architecture since more information is available on this architecture than any other comparable system. We believe that the extension similar to the one we propose below can be made to any high end graphics system that supports texture mapping in hardware.

5.1 RealityEngine architecture

The board-level block diagram of the RE architecture[1] is shown in Fig. 2. Details of the Fragment GeneratorFG) and Image Engine(IE) (from [1]) are given below:

> Each Fragment Generator is responsible for the rasterization of 1/5, 1/10, or 1/20 of the pixels in the framebuffer, with the pixel assignments finely interleaved to ensure that even small triangles are partially rasterized by each of the Fragment Generators. Each Fragment Generator computes the intersection of the set of pixels that are fully or partially covered by the triangle and the set of pixels in the framebuffer that it is responsible for, generating a fragment for each of these pixels. Color, depth, and texture coordinates are assigned to each fragment based on the initial and slope values computed by the Geometry Engine. A subsample mask is assigned to the fragment based on the portion of each pixel that is covered by the triangle. The local copy of the texture memory is indexed by the texture coordinates, and the 8 resulting samples are reduced by linear interpolation to a single color value, which then modulates the fragment's color.
>
> The resulting fragments, each comprising a pixel coordinate, a color, a depth, and a coverage mask, are then distributed to the Image Engines. Like the Fragment Generators, the Image Engines, are each assigned a fixed subset of the pixels in the framebuffer. These subsets are themselves subsets of the Fragment Generator allocations, so that each Fragment Generator communicates only with the 16 Image Engines assigned to it. Each Image Engine manages its own dynamic RAM that implements its subset of the framebuffer. When a fragment is received by an Image Engine, its depth and color sample data are merged with the data already stored at that pixel, and a new aggregate pixel color is immediately computed. Thus the image is complete as soon as the last primitive has been rendered; there is no need for a final framebuffer operation to resolve the multiple color samples at each pixel location to a single displayable color.

5.2 Hardware voxelization - the computational requirements

The voxelization algorithm presented in Section 2.2 consist of the following steps:

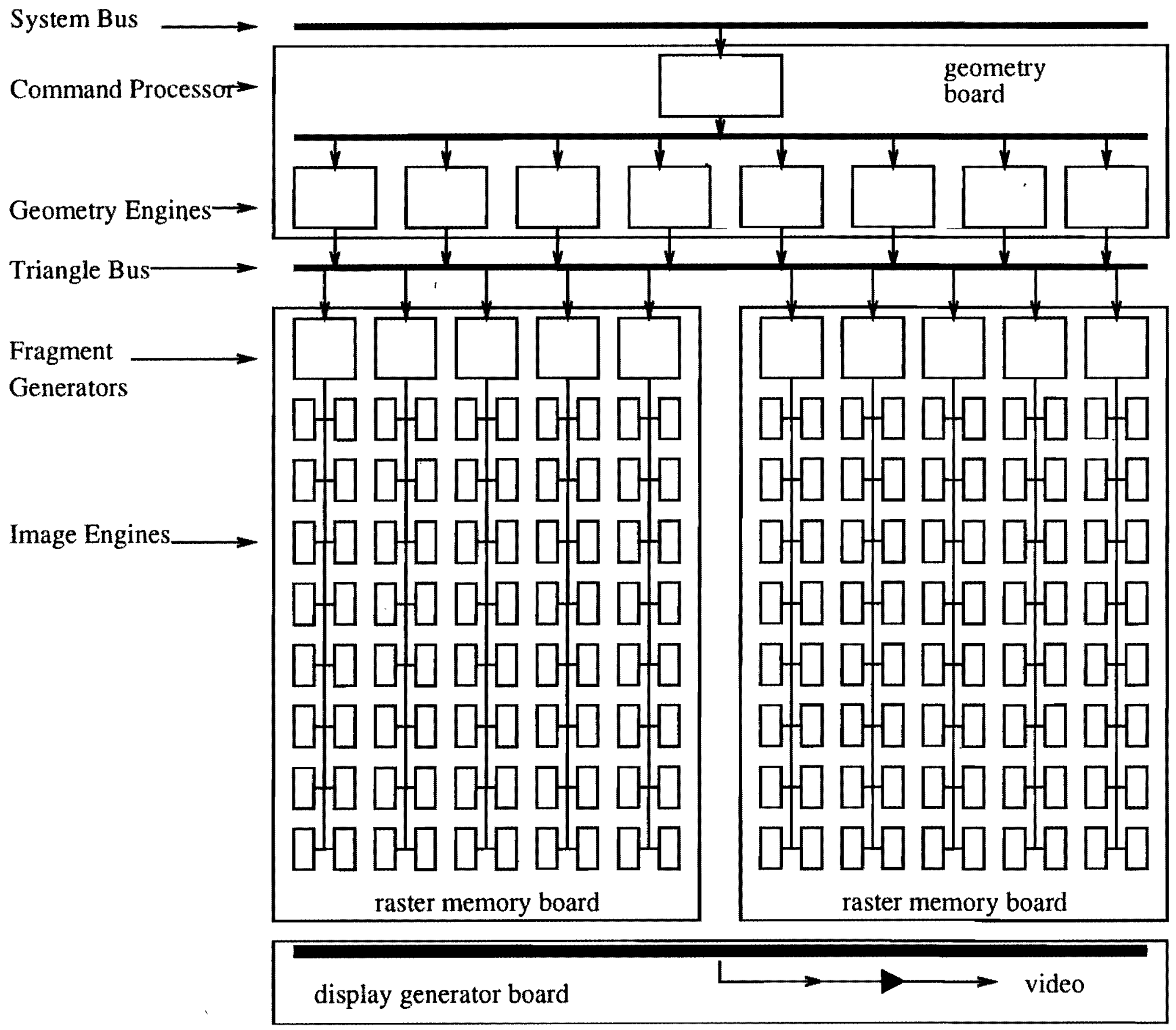

Figure 2. Block diagram of an intermediate configuration of RealityEngine
(8 GEs on the geometry board, 2 raster memory boards and a display generator board).

- Determine the front and the back faces of the tetrahedron.
- Scan convert the front triangles to the front framebuffer and the front Z-buffer.
- Scan-convert the back triangles to the back framebuffer and back Z-buffer.
- Interpolate all the voxel values between corresponding pixels in the front and the back framebuffers and store the results in the volume buffer.

Clearly, the first three steps are the same as what the RE does for rendering triangles except that we need two pairs of frame- and z-buffers. However, the RE can already configure the bitplanes into two frame buffers, the front and the back. Thus the only additional requirement is the back z-buffer. In the RE architecture, the Image Engines directly manipulate the bit-planes. Each IE can configure up to 1024 bits for each pixel. If the IE is enabled to split the available bitplanes into two pairs of buffers, the above can be realized without the need for additional bitplanes.

The last step in the voxelization process, namely the interpolation of the voxel values and writing to the volume buffer is not handled by the present RE architecture.

The computation necessary for the interpolation step involves *increment scalar estimation* and *scalar incrementation*. Along the depth (volume) of each cell, the scalar value increment is given by

$$s_{incr\_volume} = (s_{front} - s_{back})/(z_{front} - z_{back})$$

where s_{front}, s_{back} are the scalar values in the front and back framebuffer for the projected pixel. The z_{front} and z_{back} are the front and back z values available in the zbuffers for the projected pixel. This computation is performed once for each projected pixel of the cell.

Given the value s_{i-1}, the scalar incrementation of the adjacent voxel along the depth s_i is computed as follows:

$$s_i = s_{i-1} + s_{incr\_edge}$$

A total of $(z_{front} - z_{back})$ scalar incrementations are done for each projected pixel. The above computations are simple linear interpolation computations.

Since we will be using the texture mapping hardware of the RE to perform the volume rendering, the obvious destination for the output of the interpolation step should be the texture memory.

5.3 The proposed architecture

The extended RE architecture is to perform the following steps in order to generate a volume rendered image of an unstructured grid:

1. The CPU outputs tetrahedral data to the Geometry Engine.
2. For all the tetrahedral cells in the unstructured grid the voxelization step below is performed.
    - Triangle faces of tetrahedron classified as front and back faces(with respect to observer) - done by Geometry Engine
    - Transformation to volume buffer coordinates - done by Geometry Engine
    - Generate fragments for the back faces- done by Fragment Generator
    - Merge fragments into back frame buffer and back z-buffer - done by the Image Engines
    - Communication of a pixels depth and color from both front and back buffers

      by Image Engines to the corresponding Fragment Generators
    - Interpolation - by Fragment Generator for each projected pixel and update of texture memory with generated voxel values (Note that a FG updates only all the interleaved locations in texture memory corresponding to the IEs associated with the FG).
3. After the voxelization step is over it is necessary to merge texture memory from different FGs to generate a consistent copy of the texture for all the FGs. This can be done using one of the following techniques:
    - Broadcast of all the affected texels to the other FGs.
    - Read textures from FGs into CPU/GE and accumulate to a new texture.
4. Texture mapping onto polygons in back-to-front order - GL hardware

The necessary modifications to the RE are:

1. FGs receive pixel information from the IEs for all projected pixels.
2. FGs perform interpolation and incrementations for each projected pixel.
3. FGs write voxel values into texture memory for the run of voxels between front and back z-buffers.
4. FGs Broadcast texture memory updated by the FG after voxelization of all the cells.
5. The IE needs to perform the following actions: After the scan conversion of front and back triangles is complete, each IE needs to examine all the pixels managed by it to determine which ones have

been affected by the projection of the triangles. The front and back frame- and z-buffer values of all such pixels need to be transferred to the FG. The time involved in this sequential search can be minimized by making use of the fact that we are dealing only with convex polygons (triangles in this case)

We note that the interleaved organisation of pixels on an IE helps load balancing and that with the addition of more RMs each IE and FG needs to handle fewer pixels. We point out that the above design is based on incomplete knowledge of the RE-architecture. Various alternate designs could be possible. For example, the IEs can voxelize directly into their frame buffer and transfer the result into the texture memory.

5.4 Anti-aliasing in hardware

Next we examine the hardware architecture for the accumulation buffer anti-aliasing algorithm. The only additional facility needed to implement the above is the ability to update a texture memory location with a new value which is a linear combination of the present value and an incoming value. With this facility, anti-aliased voxelization is implemented by repeated voxelization of a given cell with sample offsets.

The hardware support for 2D sub-pixel positioning, accumulation buffer and area sub-sampling helps to generate an anti-aliased polygon projection. When these options are turned on, the quality of projection/scan conversion is much better than the normal scan-conversion. The impact of this on the voxelization is not known currently. However the error measures proposed in [17] helps to study the difference between two voxel volume buffers.

5.5 Extension to OGL

We propose the following new routines to be added to the Open GL to facilitate the use of the extended hardware capabilities:

- *vox_prefsize*()
- *vox_ortho*()
- *vox_bgntetra*() and *vox_endtetra*()

With these routines, the OGL programmer can visualize unstructured grids in real-time, in addition to rendering of polygonal data.

The *vox_prefsize*() function specifies the dimensions for the resulting voxel volume. This is analogous to the GL function *prefsize*() used to specify screen pixels. The *vox_ortho*() function specifies the user dimensions for the input volume. This is analogous to the GL function *ortho*() used to specify a 3D user volume. The two functions *vox_prefsize*() and *vox_ortho*(), helps to compute the scaling required to transform the user space to the volume buffer. In our implementation we transform the user space into the uniformly scaled voxel volume which lies totally inside the volume buffer specified by *vox_prefsize*(). For example the user space is $< 100, 20, 60 >$ and the volume buffer specified is $< 100, 100, 100 >$, then the actual volume buffer is $< 100, 20, 60 >$. This helps to maintain a uniformly scaled object in the voxel volume and also reduces the data handled by eliminating empty voxels. The pseudocode for the hardware based voxelized volume rendering is given in Appendix I.

6 Simulation studies

To validate the above algorithms and to demonstrate the feasibility of the proposed architecture we resorted to some hardware-assisted simulation. This has been necessary since we do not have access either to the details of the RE implementation (other than the cited references) nor to any simulator of the RE internals. We implemented our algorithm on an SGI ONYX VTX system with hardware support for texture mapping. The VTX Graphics board performs all the operations of a RE with 1 Raster Manager. The system has 128 Mb of CPU memory and 2 R4400 150MHz CPUs.

6.1 Simulation set-up

The implementation has the following steps:

1. Hexahedron as input - done in the CPU
2. Polygon faces of hexahedron classified as front and back faces(with respect to observer) - done by CPU
3. Scan convert back faces into frame buffer and zbuffer - VTX hardware
4. Use the GL *lrectread*() function to read both framebuffer and zbuffer for back faces - VTX hardware
5. Scan convert front faces into frame buffer and zbuffer - VTX hardware
6. Use the GL *lrectread*() function to read both framebuffer and zbuffer for front faces - VTX hardware
7. Interpolation - done by CPU
8. Load volume as texture - VTX hardware
9. Texture mapping onto polygons in back-to-front order - VTX hardware

6.2 Experiments

Two data sets were studied: the staircase data set containing 23 cells and the Bluntfin data set with 37479 cells. The staircase data set is a regular grid and the Bluntfin model is a curvilinear grid.

We have used tetrahedron in our algorithm description. This is because in a given hexahedral description the order of the vertices and connectivity information are additional input required. In addition, even if the above information are available and accurate, interpolation of voxel values at the interior using the vertex scalar values is dependent on the direction of voxelization. In the two cases considered, the accurate hexahedral information is available. In addition for the staircase data all the vertices have the same scalar value and hence the interpolation problem does not arise. For the blunt-fin data, the original computational grid is a curvilinear grid with hexahedral cells. A unique tetrahedralization of the grid is not possible and hence we have used the hexahedral grid itself. Hence there could be variation in the interior voxel scalar values. However as described in [17] this is a problem general to all tri-linear interpolation based approaches.

6.3 Results

Our simulation setup concentrated on three aspects of the hardware implementation namely, voxelization, anti-aliased voxelization and hardware texture mapped volume Rendering. The results obtained are described in the next few paragraphs.

6.3.1 Voxelization

Fig. 3a. shows the stair case voxelized using software and BOB for rendering[4]. Fig. 4a. shows the stair case voxelized using the VTX hardware and BOB for rendering.

6.3.2 Anti-aliased Voxelization

Fig. 3b. shows the antialiased voxelization of stair case using software and BOB for rendering. Fig. 4b. shows the stair case using the VTX hardware for antialised voxelization and BOB for rendering.

6.3.3 Hardware Texture Mapped Rendering

Fig. 5a. shows the stair case voxelized using the VTX hardware and volume rendered using hardware texture mapping. Fig. 5b. shows the stair case using the VTX hardware for antialised voxelization and rendered using hardware texture mapping.

6.3.4 Voxelized curvilinear grids

The algorithm has been tested on the Bluntfin curvilinear grid. This curvilinear grid has hexahedral cells. The density of gas flow over a blunt fin is visualized as a scalar value. Fig. 6a. shows the voxelization of bluntfin using software and BOB for rendering. Fig. 6b. shows the bluntfin voxelized using the VTX hardware and BOB for rendering. Fig. 6c. shows the bluntfin voxelized using the VTX hardware and rendered using hardware texture mapping.

7 Summary

We have presented an architecture that combines anti-aliased voxelization with volume rendering using texture mapping to provide a high performance visualization of unstructured data. The attractive aspect of the architecture is that it can be implemented on a high-end polygon rendering architecture with simple extensions. The other advantage is that rendering time does not depend on the displayed image size. Since we use the hardware for scan-conversion, the full frame-buffer can be used for voxelization.

There are however some drawbacks to this approach. The largest voxel volume that can be rendered is tied to the size of the texture memory. The texture memory currently supported on commercial systems is 4 to 16 Mb. But to support 1024 × 1024 × 1024 voxels with 8 bits per voxel we need 1 GB of texture memory. The quality of the rendered images is not as good as those from raycasting based approaches (for example using Volvis[6]). However for rapid interaction with the volume data these images are adequate. The other limitation to our approach is that our algorithm works only on convex cells. But in reality this is not a serious limitation because most of the successful graphics hardware works only on lines and triangles. And efficient algorithms for tetrahedralization of arbitrary polyhedra are available [25]. Another problem is the uniform voxelization of all the cells of the unstructured grid. This will result either in the loss of detail, if the resolution is low, or in data explosion, if the resolution is high. Hierarchical voxelization methods and their hardware voxelization are the topics of ongoing research.

We have also not dwelt on the expected performance of the voxelization step. We can only surmise that the excellent polygon rendering rate of the RE will be available to the voxelization step and this in combination with the interleaved interpolation performance by the FGs will provide rapid voxelization. Since we do not have access to the internal details of the RE implementation, any attempts to provide more specific numbers will be speculation on our part. Further refinements to this architecture can only be done by system designers

GL Crectread( )

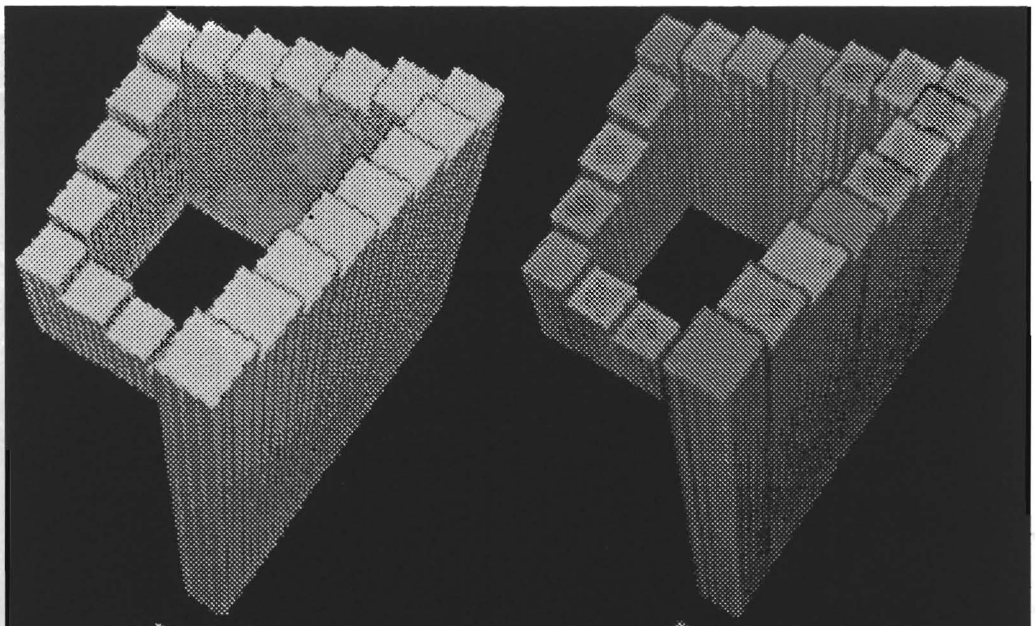
Fig. 3.

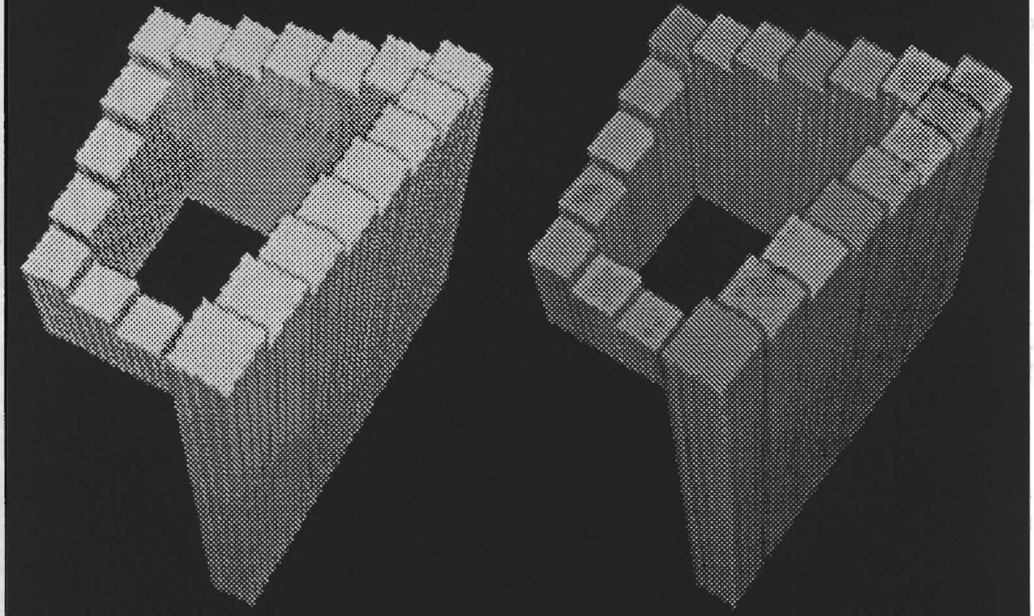
Fig. 4.

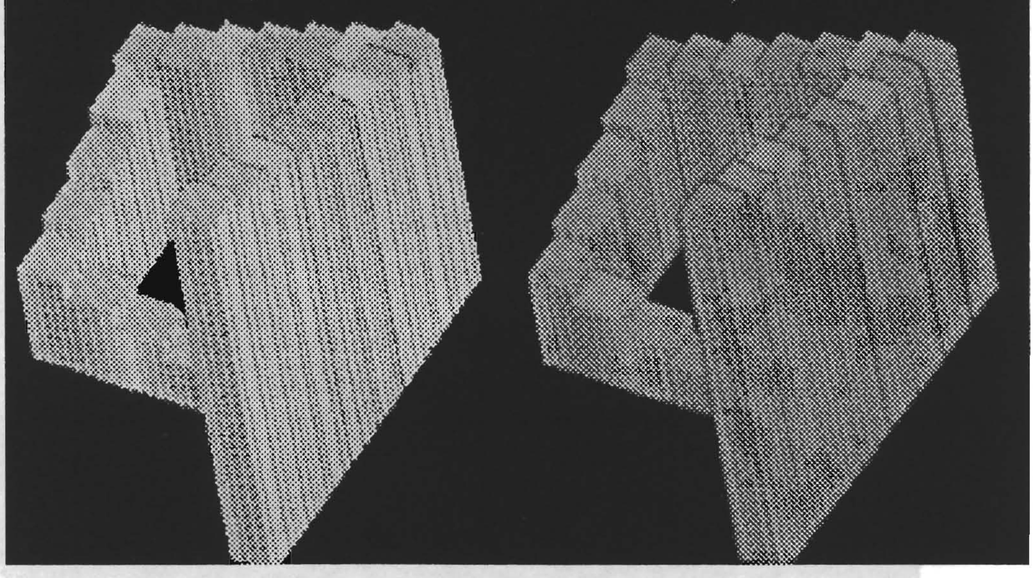
Fig. 5.

(a) (b)

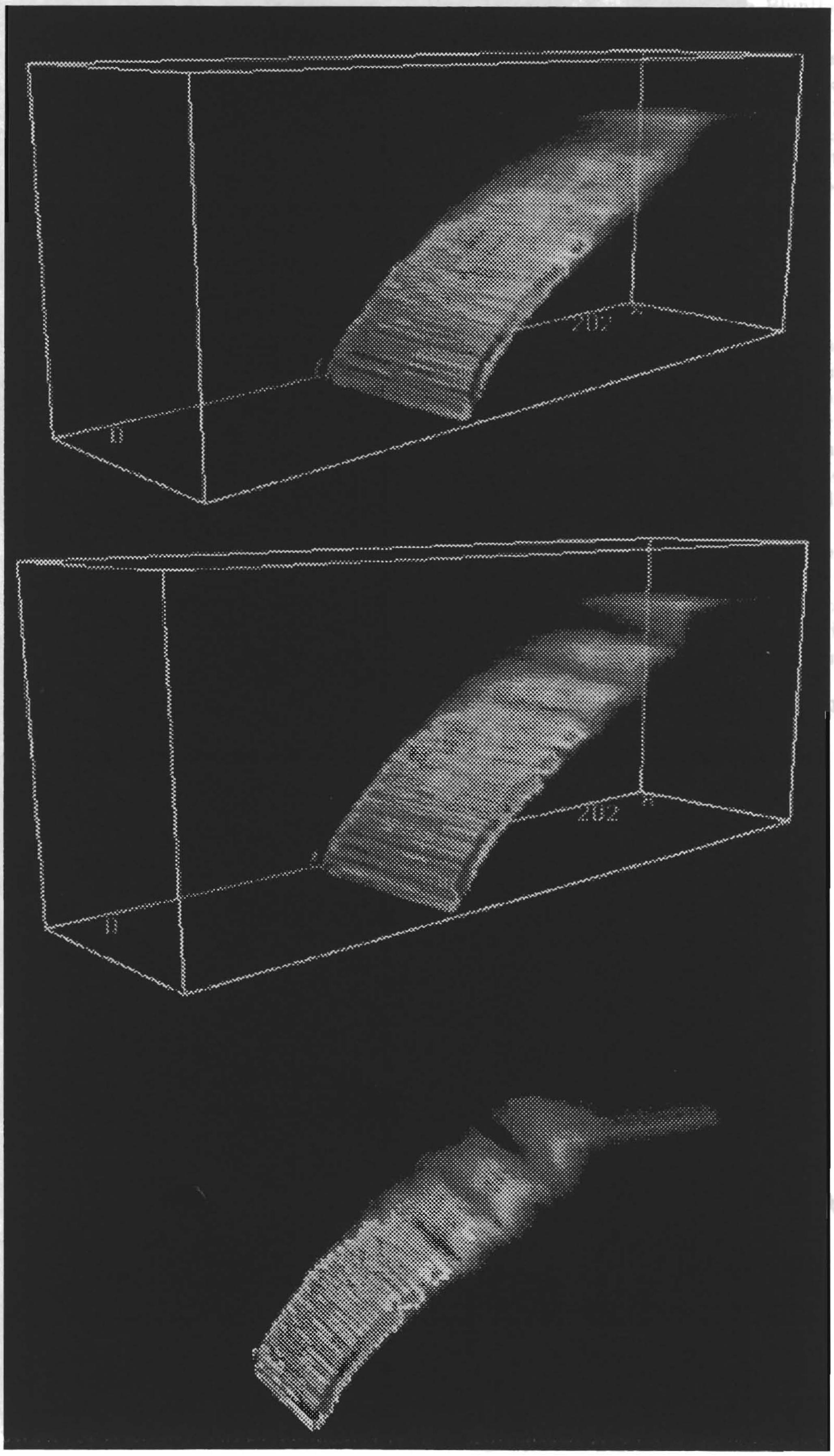

Fig. 6. Volume rendering of Bluntfin data

who have access to the implementation details of the RE.

References

[1] AKELEY, K. Realityengine graphics. *Computer Graphics (SIGGRAPH 93* (Aug 1993), 109-116.

[2] CABRAL, CAM, AND FORAN. Texture mapped volume rendering. *Proc. of IEEE Visualization 1994* (October 1994).

[3] CHALLINGER, J. Scalable parallel volume raycasting for nonrectilinear computational grids. *Proc. of Parallel Rendering Symposium 1993* (Oct 1993), 81–88.

[4] CHIN-PURCELL, K. BOB - Brick Of Bytes. *Minnesota Supercomputer Center, Inc* (1993).

[5] CRAWFIS, R. A., AND MAX, N. Texture splats for 3D scalar and vector field visualization. *Proc. of Visualization 93, San Jose* (Oct 1993), 261–266.

[6] FRASER, R. Interactive volumme rendering using advanced graphics architectures. *SGI Developer News* (Dec 1994), 5–9.

[7] FUCHS, H. Fast spheres, shadows, texture, transparencies and image enhancements in pixel-planes. *Computer Graphics, (SIGGRAPH '85 Proc.) 19* (July 1985), 111-120.

[8] GELDER, A. V., AND WILHELMS, J. Rapid exploration of curvilinear grids using direct volume rendering. *Proc. of IEEE Visualization 93* (1993), 70–77.

[9] GIERTSEN, C., AND PETERSEN, J. Parallel volume rendering on a network of workstations. *IEEE CG&A* (Nov 1993), 16–23.

[10] GUAN, S.-Y., BLEIWERS, A., AND LIPES, R. Parallel implementation of volume rendering on Denali graphics systems. *Proceedings of the International Parallel Processing Symposium* (1995).

[11] HAEBERLI, P., AND AKELEY, K. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics 25*, 4 (Aug 1990), 309–318.

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

[13] KAUFMAN, A., AND SHIMONY, E. 3D scan-conversion algorithms for voxel-based graphics. *Proc. of ACM Workshop on Interactive 3D Graphics, Computer Graphics, Chapel Hill, NC* (October 1986), 45–75.

[14] KAUFMAN, A. E. Towards a comprehensive volume visualization system. *Proc. Visualization 92* (Oct 1992), 37–44.

[15] MAX, N., BECKER, B., AND CRAWFIS, R. Flow volumes for interactive vector field visualization. *Proc. of Visualization 93, San Jose* (Oct 1993), 19–24.

[16] MAX, N., HANRAHAN, P., AND CRAWFIS, R. Area and volume coherence for efficient visualization of 3D scalar functions. *Proc. of San Diego Workshop on Volume Visualization, Computer Graphics 24*, 5 (Nov 1990), 27–33.

[17] PRAKASH, C. E., AND MANOHAR, S. Error measures and 3D anti-aliasing for voxel data. *Pacific Graphics 95* (1995).

[18] PRAKASH, C. E., AND MANOHAR, S. Voxelization of unstructured grids. *to appear in Computers and Graphics also available as Technical Report IISc-CSA-94-04, Department of Computer Science and Automation, IISc, Bangalore-560 012, INDIA* (1995).

[19] STYTZ, M. R., FRIEDER, G., AND FRIEDER, O. Volume rendering and visualization for scientific data. *ACM Computing Surveys 23*, 4 (Dec 1991), 421–499.

[20] TAUBIN, G. Rasterizing algebraic curves and surfaces. *IEEE CG & A* (March 1994), 14–23.

[21] WANG, S. W., AND KAUFMAN, A. Volume sampled voxelization of geometric primitives. *Proc. of IEEE Visualization 93* (Oct 1993), 78–84.

[22] WANG, S. W., AND KAUFMAN, A. Volume sampled 3D modeling. *IEEE CG & A 14*, 5 (Sep 1994), 26–32.

[23] WILHELMS, J. A coherent projection approach to direct volume rendering. *Computer Graphics(SIGGRAPH '91 Proceedings) 25*, 4 (July 1991), 275–284.

[24] WILHELMS, J. Pursuing interactive visualization of irregular grids. *The Visual Computer 9* (1993), 450–458.

[25] WILLIAMS, P. L. Interactive direct volume rendering of curvilinear and unstructured data. *PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign* (1992).