Efficiently Using Graphics Hardware in Volume Rendering Applications

Rüdiger Westermann, Thomas Ertl

Computer Graphics Group
Universität Erlangen-Nürnberg, Germany[*]

Abstract

OpenGL and its extensions provide access to advanced per-pixel operations available in the rasterization stage and in the frame buffer hardware of modern graphics workstations. With these mechanisms, completely new rendering algorithms can be designed and implemented in a very particular way. In this paper we extend the idea of extensively using graphics hardware for the rendering of volumetric data sets in various ways. First, we introduce the concept of clipping geometries by means of stencil buffer operations, and we exploit pixel textures for the mapping of volume data to spherical domains. We show ways to use 3D textures for the rendering of lighted and shaded iso-surfaces in real-time without extracting any polygonal representation. Second, we demonstrate that even for volume data on unstructured grids, where only software solutions exist up to now, both methods, iso-surface extraction and direct volume rendering, can be accelerated to new rates of interactivity by simple polygon drawing and frame buffer operations.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—-*Graphics Hardware, 3D Textures, Volume Rendering, Unstructured Grids*

1 Introduction

Over the past few years workstations with hardware support for the interactive rendering of complex 3D polygonal scenes consisting of directly lit and shaded triangles have become widely available. The last two generations of high-end graphics workstations [1, 17], however, besides providing impressive rates of geometry processing, also introduced new functionality in the rasterization and frame buffer hardware, like texture and environment mapping, fragment tests and manipulation as well as auxiliary buffers. The ability to exploit these features through OpenGL and its extensions allows completely new classes of rendering algorithms to be developed. Anticipating similar trends for the more advanced imaging functionality of todays high-end machines we are actively investigating possibilities to accelerate expensive visualization algorithms by using these extensions.

In this paper we are dealing with the efficient generation of a visual representation of the information present in volumetric data sets. For scalar-valued volume data two standard techniques, the rendering of iso-surfaces, and the direct volume rendering, have been developed to a high degree of sophistication. However, due to the huge number of volume cells which have to be processed and to the variety of different cell types only a few approaches allow parameter modifications and navigation at interactive rates for realistically sized data sets. To overcome these limitations we provide a basis for hardware accelerated interactive visualization of both iso-surfaces and direct volume rendering on arbitrary topologies.

Direct volume rendering tries to convey a visual impression of the complete 3D data set by taking into account the emission and absorption effects as seen by an outside viewer. The underlying theory of the physics of light transport is simplified to the well known volume rendering integral when scattering and frequency effects are neglected [9, 10, 15, 29]. A few standard algorithms exist for computing the intensity contribution along a ray of sight, enhanced by a wide variety of optimization strategies [13, 15, 12, 4, 11]. But only recently, since hardware supported 3D texture mapping is available, has direct volume rendering become interactively feasible on graphics workstations [2, 3, 30]. We extend this approach with respect to flexible editing options and advanced mapping and rendering techniques.

Our goal is the visualization and manipulation of volumetric data sets of arbitrary data type and grid topology at interactive rates within one application on standard graphics workstations. In this paper we focus on scalar-valued volumes and show how to accelerate the rendering process by exploiting features of advanced graphics hardware implementations through standard APIs like OpenGL. Our approach is pixel oriented, taking advantage of rasterization functionality such as color interpolation, texture mapping, color manipulation in the pixel transfer path, various fragment and stencil tests, and blending operations. In this way we

- **extend volume rendering via 3D textures** with respect to arbitrary clipping geometries and exploit pixel textures for volume rendering in spherical domains
- **render shaded iso-surfaces** at interactive rates combining 3D textures and fragment operations thus avoiding any polygonal representation
- **accelerate volume visualization of tetrahedral grids** employing polygon rendering of cell faces and fragment operations for both shaded iso-surfaces and direct volume rendering.

In the remainder of this paper we first introduce the basic concept of direct volume rendering via 3D textures. We then describe our extension for arbitrary clipping geometries, and we introduce the pixel texture mechanism for spherical domains be reused later on. Basic algorithms for rendering shaded iso-surfaces from regular voxel grids will be described. Finally, we propose a general framework for the visualization of unstructured grids. Some of the earlier ideas can be used here again, but 3D textures have to be abandoned in favor of polygon rendering of the cell faces. We conclude our paper with detailed results and additional ideas for future work.

[*]Lehrstuhl für Graphische Datenverarbeitung (IMMD9), Universität Erlangen-Nürnberg, Am Weichselgarten 9, 91054 Erlangen, Germany, Email: `wester@informatik.uni-erlangen.de`

2 Volume rendering via 3D textures

When 3D textures became available on graphics workstations their potential benefit in volume rendering applications was soon recognized [3, 2]. The basic idea is to interpret the voxel array as a 3D texture defined over $[0,1]^3$ and to understand 3D texture mapping as the trilinear interpolation of the volume data set at an arbitrary point within this domain. At the core of the algorithm multiple planes parallel to the image plane are clipped against the parametric texture domain (see Figure 1) and sent to the geometry processing unit. The hardware is then exploited for interpolating 3D texture coordinates issued at the polygon vertices and for reconstructing the texture samples by trilinearly interpolating within the volume. Finally, pixel values are blended appropriately into the frame buffer in order to approximate the continuous volume rendering integral.

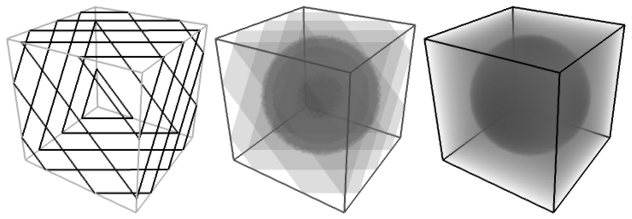


Figure 1: Volume rendering by 3D texture slicing.

Nevertheless, besides interactive frame rates, in many practical applications editing the data in a free and easy way is of particular interest. Although texture lookup tables might be modified in order to enhance or suppress portions of the data, the relevant structures can often be separated in a much more convenient and intuitive way by using additional clipping geometries. Planar clipping planes available as core OpenGL mechanisms may be utilized, but from the user's point of view more complex geometries are necessary.

2.1 Arbitrary clipping geometries

A straightforward approach which is implemented quite often is the use of multiple clipping planes to construct more complex geometries. However, notice that the simple task of clipping an arbitrarily scaled box cannot be realized in this way.

Even more flexibility and ease of manipulation can be achieved by taking advantage of the per-pixel operations provided in the rasterization stage. As we will outline, as long as the object against which the volume is to be clipped is a closed surface represented by a list of triangles it can be efficiently used as the clipping geometry.

The basic idea is to determine all pixels which are covered by the cross-section between the object and the actual slicing plane (see Figure 2). These pixels, then, are locked, thus preventing the textured polygon from getting drawn to these locations.

The locking mechanism is implemented exploiting the OpenGL stencil buffer test. It allows pixel updates to be accepted or rejected based on the outcome of a comparison between a user defined reference value and the value of the corresponding entry in the stencil buffer. Before the textured polygon gets rendered the stencil buffer has to be initialized in such a way that all color values written to pixels inside the cross-section will be rejected.

In order to determine for a certain plane whether a pixel is covered by a cross-section or not we render the clipping object in polygon mode. However, since we are only interested in setting the stencil buffer we do not alter any of the frame buffer values. At first, an additional clipping plane is enabled which has the same orientation and position as the slicing plane. All back faces with respect to the actual viewing direction are drawn, and everything in front of

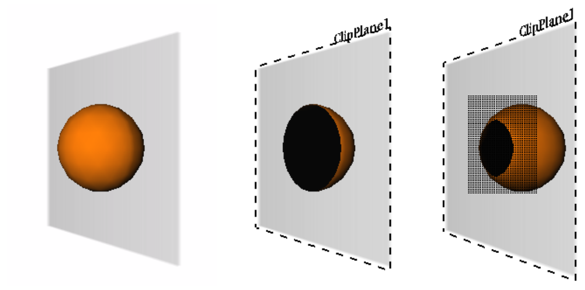


Figure 2: The use of arbitrary clipping geometries is demonstrated for the case of a sphere. In regions where the object intersects the actual slice the stencil buffer is locked. The intuitive approach of rendering only the back faces might result in the patterned erroneous region.

the plane is clipped. Wherever a pixel would have been drawn the stencil buffer is set. Finally, by changing the stencil test appropriately, rendering the textured polygon, now, only affects those pixels where the stencil buffer is unchanged.

In general, however, depending on the clipping geometry this procedure fails in determining the cross-section exactly (see rightmost image in Figure 2). Therefore, before the textured polygon is rendered all stencil buffer entries which are set improperly have to be updated. Notice that in front of a back face which was written erroneously there is always a front face due to the topology of the clipping object. The front faces are thus rendered into those pixels where the stencil buffer is set and the stencil buffer is cleared where a pixel also passes the depth test. Now the stencil buffer is correctly initialized and all further drawing operations are restricted to those pixels where it is set or vice versa. Of course, the stencil buffer has to be cleared in order to process the next slice.

Since this approach is independent of the used geometry it allows arbitrary shapes to be specified. In particular, it turns out that adaptive manipulations of individual vertices can be handled quite easily thus providing a flexible tool for carving portions out of the data in an intuitive way.

2.2 Spherical domains

Traditionally, 3D texture space is parameterized over a Cartesian domain. Because the texture is mapped to a cube, all cross-sections between a slice parallel to the image plane and the volume still remain planar in parametric texture space. As a consequence it suffices to assign texture coordinates on a per-vertex basis and to bilinearly interpolate between them during rasterization.

However, in many applications the texture has to be mapped to domains which are not Cartesian. For instance, observe that in geoscience atmospheric data is often parameterized in spherical coordinates and mapped to the corresponding domain.

Now 3D texture mapping becomes difficult because planar slicing planes are mapped to non-planar surfaces in texture space. No longer can bilinear interpolation on a per-vertex basis be used. One way to deal with this limitation is to assign texture coordinates for each pixel separately rather than to interpolate the values across polygons. With the **pixel texgen** OpenGL extension available on SGI Impact architectures this becomes possible in a quite efficient way by giving the user direct control of texture coordinates on a per-pixel basis.

Pixel textures are specified in the same way as standard 3D textures. Once a pixel texture has been activated all pixel values which are drawn from main memory into the frame buffer are interpreted as texture coordinates into this texture. At first, each RGB color triple is mapped to the texture. Then, instead of the color values the interpolated texture values are drawn.

Let us consider a simple example to demonstrate the relevance of pixel textures in volume rendering. A spherical object is rendered having smooth color interpolation across polygons. Red and green color components are set according to the normalized spherical coordinates. The blue component is initialized by a constant value (see left of Figure 3). By reading the frame buffer and drawing it back with enabled pixel texture each of the RGB values is treated as a texture coordinate into the 3D map.

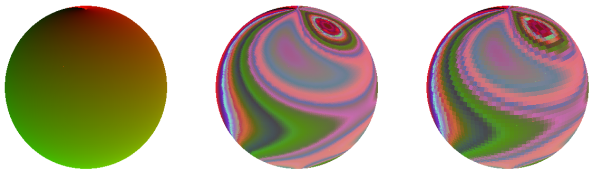


Figure 3: Using pixel textures to perform texture mapping and distortion on spherical domains. The right image was created by using constant color across faces.

Since texture coordinates can be modified in main memory on the per-pixel basis it is easy to apply arbitrary transformations without manipulating and re-loading the texture. For example, in Figure 3 the original texture was just a simple striped pattern, but texture coordinates were distorted with respect to a vector field before they got mapped (see middle of Figure 3).

Arbitrary homogeneous transformations can be applied using the OpenGL pixel transfer operations. Before pixel data gets written to the frame buffer it is multiplied with the **color matrix** and finally scaled and biased by vector-valued factors. More precisely, if **CM** is the 4x4 color matrix and **b** and **s** are the four component bias and scale vectors providing separate factors for each color channel, then each RGBα quadruple, **p**, becomes $\mathbf{b} + \mathbf{s} \cdot \mathbf{CM} \cdot \mathbf{p}$.

Even more efficiently pixel textures can be exploited in volume rendering applications. For example, image c) in Figure 10 was produced by rendering the spherical object multiple times, each time which a slightly increased radius coded into the blue color channel. The pixel data was then re-used to map to the 3D texture before it was blended into the frame buffer thus simulating volumetric effects.

3 Rendering shaded iso-surfaces via 3D textures

So far, with our extensions to texture mapped volume rendering we introduced concepts for adaptive exploration of volumetric data sets as well as for the application to rather unusual domains.

In practice, however, the display of shaded iso-surfaces has been shown as one of the most dominant visualization options, which is particularly useful to enhance the spatial relationship between structures. Moreover, this kind of representation often meets the physical characteristics of the real object in a more natural way.

Different algorithms have been proposed for efficiently reconstructing polygonal representations of iso-surfaces from scalar volume data [14, 16, 19, 26], but unfortunately none of these approaches can effectively be used in interactive applications. This is due to the effort that has to be spent to fit the surface and also to the enormous amount of triangles produced.

For example, the iso-surface shown in Figure 4 was reconstructed from a 512^2x128 abdomen data set. It took about half a minute to generate 1.4 million triangles. Rendering the triangle list on a high-end graphics workstation takes several seconds. Thus, interactively manipulating the iso-value is quite impossible, and also rendering the surface at acceptable frame rates can hardly be achieved. In contrast, with the method we propose the surface can be rendered in approximately one second, including arbitrary updates of the iso-value.

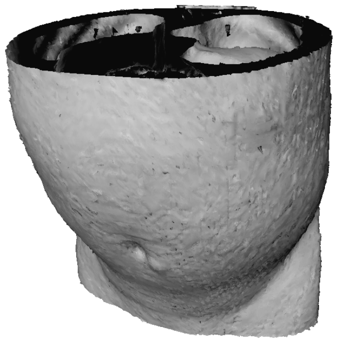


Figure 4: Iso-surface reconstructed with the MC algorithm.

In order to design an algorithm that completely avoids any polygonal representation we combine 3D texture mapping and advanced pixel transfer operations in a way that allows the iso-surface to be rendered on a per-pixel basis. Since the maximum possible feature size is a single pixel we expect our method to be able to capture even the smallest features in the data.

Recently, first approaches for combining hardware accelerated volume rendering via 3D texture maps with lighting and shading were presented. In [24] the sum of pre-computed ambient and reflected light components is stored in the texture volume and standard 3D texture composition is performed. On the contrary, in [8] the orientation of voxel gradients is stored together with the volume density as the 3D texture map. Lighting is achieved by indexing into an appropriately replicated color table. The inherent drawbacks to these techniques is the need for reloading the texture memory each time any of the lighting parameters change (including changes in the orientation of the object) [24], and the difficulty to achieve smoothly shaded surfaces due to the limited quantization of the normal orientation and the intrinsic hardware interpolation problems [8].

Basically, our approach is similar to the one used in traditional volume ray-casting for the display of shaded iso-surfaces. Let us consider that the surface is hit if the material values along the ray do exceed the iso-value for the first time. At this location the material gradient is computed which is then used in the lighting calculations.

By recognizing that we do already exploit texture interpolation to re-sample the data, all that needs to be evaluated is how to capture those texture samples above the iso-value which are nearest to the image plane. Therefore we have employed the OpenGL **alpha test**, which is used to reject pixels based on the outcome of a comparison between their alpha component and a reference value.

Each element of the 3D texture gets assigned the material value as it's alpha component. Then, texture mapped volume rendering is performed as usual, but pixel values are only drawn if they pass the z-buffer test and if the alpha value is larger than or equal to the selected iso-value. In any of the affected pixels in the frame buffer, now, the color present at the first surface point is being displayed.

In order to obtain the shaded iso-surface from the pixel values already drawn into the frame buffer we propose two different approaches:

- **Gradient shading**: A four component 3D texture is stored which holds in each element the material gradient as well as the material value. Shading is performed in image space by means of matrix multiplication using an appropriately initialized color matrix.

- **Gradientless shading**: Shading is simulated by simple frame buffer arithmetic computing forward differences with respect to the light source direction. Pixel texturing is exploited to encompass multiple rendering passes.

Both approaches account for diffuse shading with respect to a parallel light source positioned at infinity. Then the diffuse term reduces to the scalar product between the surface normal, **N**, and the direction of the light source, **L**, scaled by the material diffuse reflectivity, k_d.

The texture elements in gradient shading each consist of an RGBα quadruple which holds the gradient components in the color channels and the material value in the alpha channel. Before the texture is stored and internally clamped to the range [0,1] the gradient components are being scaled and translated by a factor of 0.5.

By slicing the texture thereby exploiting the alpha test as described the transformed gradients at the surface points are finally displayed in the RGB frame buffer components (see left image in Figure 5). For the surface shading to proceed properly, pixel values have to be scaled and translated back to the range [-1,1]. We also have to account for changes in the orientation of the object. Thus, the normal vectors have to be transformed by the model rotation matrix. Finally, the diffuse shading term is calculated by computing the scalar product between the light source direction and the transformed normals.

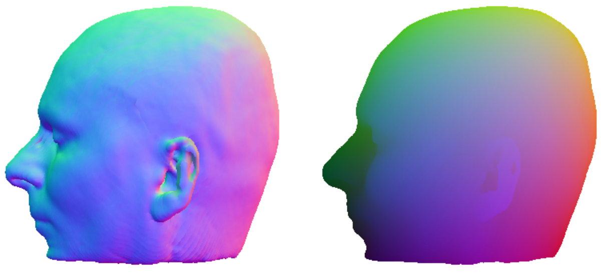


Figure 5: On the left, for an iso-surface the gradient components are displayed in the RGB pixel values. On the right, for the same iso-surface the coordinates in texture space are displayed in the RGB components.

All three transformations can be applied simultaneously using one 4x4 matrix. It is stored in the currently selected color matrix which post-multiplies each of the four-component pixel values if pixel data is copied within the active frame buffer.

For the color matrix to accomplish the transformations it has to be initialized as follows:

$$CM = \begin{pmatrix} L_x & L_y & L_z & 0 \\ L_x & L_y & L_z & 0 \\ L_x & L_y & L_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{M}_{rot} \begin{pmatrix} 2 & 0 & 0 & -1 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

By just copying the frame buffer contents onto itself each pixel gets multiplied by the color matrix. In addition, it is scaled and biased in order to account for the material diffuse reflectivity and the ambient term. The resulting pixel values are

$$\begin{bmatrix} I_a \\ I_a \\ I_a \\ 0 \end{bmatrix} + \begin{bmatrix} k_d \\ k_d \\ k_d \\ 1 \end{bmatrix} \mathbf{CM} \begin{bmatrix} R \\ G \\ B \\ \alpha \end{bmatrix} = \begin{bmatrix} k_d \langle L, N_{rot} \rangle + I_a \\ k_d \langle L, N_{rot} \rangle + I_a \\ k_d \langle L, N_{rot} \rangle + I_a \\ \alpha \end{bmatrix}$$

where obviously different ambient terms and reflectivities can be specified for each color component.

To circumvent the additional amount of memory that is needed to store the gradient texture we propose a second technique which applies concepts borrowed from [18] but in an essentially different scenario. The diffuse shading term can be simulated by simple frame buffer arithmetic if the surface is locally orthogonal to the surface normal and the normal as well as the light source direction are orthonormal vectors.

Notice that the diffuse shading term is then proportional to the directional derivative towards the light source. Thus, it can be simulated by taking forward differences toward the light source with respect to the material values:

$$\frac{\partial \mathbf{X}}{\partial L} \approx \mathbf{X}(\vec{p}_0) - \mathbf{X}(\vec{p}_0 + \triangle \cdot \vec{L})$$

By rendering the material values twice, once at the original surface points and then shifted towards the light source, OpenGL blending operations can be exploited to compute the differences.

In order to obtain the coordinates of the surface points we take advantage of the alpha test as proposed and we also apply pixel textures to re-sample the material values. Therefore it is important to know that each vertex comes with a texture coordinate as well as a color value. Usually the color values provide a base color and opacity in order to modulate the interpolated texture samples.

Let us consider that to each vertex the computed texture coordinate (u, v, w) is assigned as RGB color value. Since texture coordinates are computed in parametric texture space they are within the range [0,1]. Moreover, the color values interpolated during rasterization correspond to the texture space coordinates of points on the slicing plane. As a consequence we now have the position of surface points available in the frame buffer rather than the material gradients.

In order to display the correct color values they must not be modulated by the texture samples. However, remember that in gradientless shading we use the same texture format as in traditional texture slicing. Each element comprises a single-valued color entry which is mapped via a RGBα lookup table. This allows us to temporarily set all RGB values in the lookup table to one thus avoiding any modulation of color values.

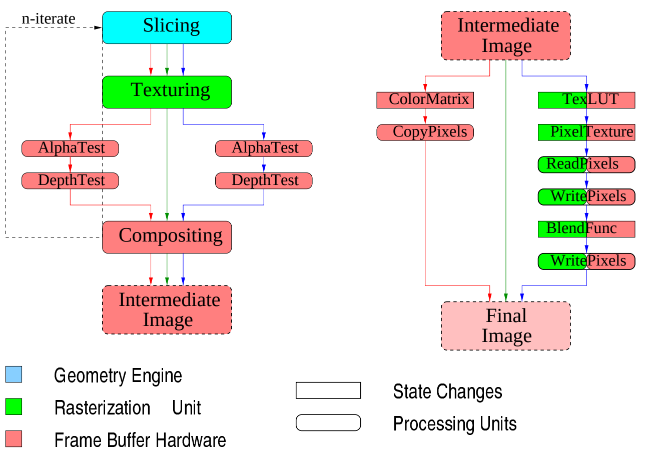


Figure 6: Process flow for texture based volume rendering techniques. Green: Standard; Red: Gradient Shading; Blue: Gradientless Shading

At this point, the real strength of pixel textures can be exploited. The RGB entries of the texture lookup table are reset in order to produce the original scalar values. Then, the pixel data is read into main memory and it is drawn twice into the frame buffer with enabled pixel texture. In the second pass pixel values are shifted towards the light source by means of the OpenGL pixel bias. Changing the blending equation appropriately let values get subtracted from those already in the frame buffer. Figure 6 summarizes the basic stages and instructions involved in the outlined methods.

So far, we have accepted that only the first material value exceeding the iso-value will be captured. In practice, however, it is often useful to visualize the exterior and the interior of an iso-surface, the latter one often characterized by material values falling below the iso-value. For example, the left image in Figure 7 was generated using our approach. Since close to the volume boundaries data larger than the iso-value is present the first pixel drawn locks the frame buffer.

This problem is solved by a multi-pass rendering approach. At first, the depth buffer is initialized with a value right behind the first intersection with the texture bounding box. All structures close to the boundary which are drawn, therefore, fail the depth test. At these locations the stencil buffer is set. Now the volume is rendered with the alpha test performed as usual into pixels where the stencil buffer is unchanged. Then, the alpha test is reversed and the volume is rendered into pixels where the stencil buffer has already been set. The result is shown in Figure 7.

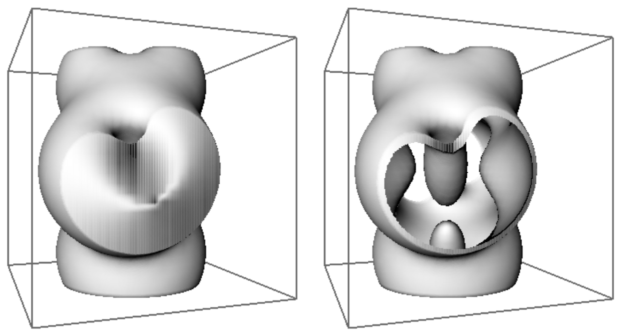


Figure 7: Multi-pass rendering to determine the exterior and the interior of iso-surfaces.

3.1 Bricking

All presented approaches work equally well if the volume data has to be split into smaller blocks which are small enough to fit into texture memory. Each brick is rendered separately from front to back starting with the one nearest to the image plane. Since the stencil buffer is immediately locked whenever a hit with the surface occurs, once a value has been set correctly it will never be changed.

Even when pixel textures are used the same algorithm proceeds without essential modifications. Since the rendered color values are always specified with respect to the parametric domain of the actual block the pixel texture will always be accessed correctly.

4 Volume rendering of unstructured grids

Now we turn our attention to tetrahedral grids most familiar in CFD simulation, which have also recently shown their importance in adaptive refinement strategies. Since most grid types which provide the data at unevenly spaced sample points can be quite easily converted into this representation, our technique, in general, is potentially attractive to a wide area of different applications.

Caused by the irregular topology of the grids to be processed the intrinsic problem showing up in direct volume rendering is to find the correct visibility ordering of the involved primitives. Different ways have been proposed to attack this problem, e.g. by improving sorting algorithms [23, 28, 5], by using space partitioning strategies [27], by taking advantage of hardware assisted polygon rendering [20, 23, 31, 25] and by exploiting the coherence within cutting planes in object space [6, 21]. Similar to the marching cubes algorithm, the marching tetrahedra approach suffers from the same limitations, i.e., the large amount of generated triangles.

In each tetrahedron (hereafter termed the volume primitive or cell) we have a linear range in the material distribution and therefore a constant gradient. The affine interpolation function $f(x,y,z) = a + bx + cy + dz$ which defines the material distribution within one cell is computed by solving the system of equations

$$\begin{pmatrix} 1 & x_0 & y_0 & z_0 \\ 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \end{pmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

for the unknowns a, b, c and d. f_i are the function values given at locations (x_i, y_i, z_i).

Now the partial derivatives, b, c and d, provide the gradient components of each cell. Gradients at the vertices are computed by simply averaging all contributions from different cells. These are stored in addition to the vertex coordinates and the scalar material values, the latter ones given as one-component color indices into a RGBα lookup table.

4.1 Shaded iso-surfaces

In contrast to regular volume data we are no longer able to perform the rendering process by means of 3D textures. However, geometry processing and advanced per-pixel operations will be exploited in a highly effective way which again allows us to avoid any polygonal representation.

At first, let us consider a ray of sight passing through one tetrahedron in order to re-sample the material values. Since along the ray the material distribution is linear it suffices to evaluate the data within the appropriate front and back face and to linearly interpolate in between. This, again, can be solved quite efficiently using the graphics hardware. Therefore the material values are issued as the color of each vertex before the smoothly shaded cell faces are rendered. The correctly interpolated samples are then being displayed and can be grabbed from the frame buffer.

Obviously, the same procedure can be applied by choosing an appropriate shading model and by issuing the material gradient as the vertex normal. Then the rendered triangles will be illuminated with respect to the gradients of the volume material.

Nevertheless, since we are interested in rendering a specific iso-surface we have to find those elements the surface is passing through. But even more difficult, the exact location of the surface within these cells has to be determined in order to compute appropriate interpolation weights (see Figure 8).

The key idea lies in a multi-pass approach:

a: **Faces are rendered having smooth color interpolation in order to compute the interpolation weights.**

b: **Faces are rendered having smooth shading in order to compute illuminated pixels.**

c: **The interpolation weights are used to modulate the results properly.**

In order to compute the interpolation weights we duplicate the material values given at the vertices into RGBα-quadruples. These are used as the current color values. Next, all the back faces are rendered, but only those pixels nearest to the image plane with an alpha value larger than the threshold are maintained by exploiting the alpha test and the z-buffer test. The stencil buffer is set whenever a pixel passes both tests.

We proceed by inverting the alpha test and by drawing the front faces. Although z-buffer comparison is in effect, z-values are never going to be changed since they are needed later. Pixel values may

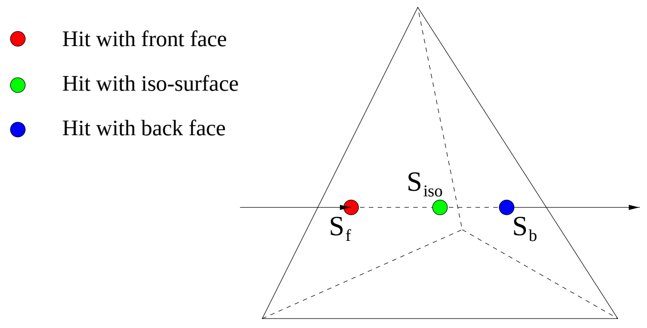


Figure 8: Interpolation weights are computed as $q = \frac{(S_b - S_f)}{S_{iso} - S_f}$ and $1 - q$ in order to obtain the iso-surface.

be affected where the stencil buffer is set, but, in fact, not all components will be altered in order to retain the previously written results.

Instead of processing all front faces at once we alternatively render each element's back faces again. By setting their alpha values to zero it is guaranteed that they always pass the alpha test. Notice that if pixels were accepted in the first pass the corresponding z-values are still present since we did not alter the z-buffer. Choosing an appropriate stencil function allows the stencil buffer to be locked whenever a pixel is written with a z-value equal to the stored one. At these locations the frame buffer is locked in order to prevent the correctly drawn pixels from being destroyed. Finally, the pixel data is read into main memory and the interpolation weights are computed and stored into two distinct pixel images I_f and I_b, respectively.

Once again, the entire procedure is repeated, but now the hardware is exploited to render illuminated faces instead of colored ones. The results of the first rendering pass are blended with the pixel image I_f and the modulated pixel data is transfered to the accumulation buffer [7]. Pixel data produced in the second pass is blended with the pixel image I_b and added to the data already stored in the accumulation buffer. During both passes blending ensures that the shaded faces are interpolated correctly in order to produce the surface shading. Finally, entire image is drawn back into the frame buffer.

However, even without computing the interpolation weights, just by equally blending the lighted front and back faces, this method produces sufficient results (see Figure 9). It is quite evident that sometimes the geometric structure of the cells shows up, but this seems to be tolerable during interactive sessions.

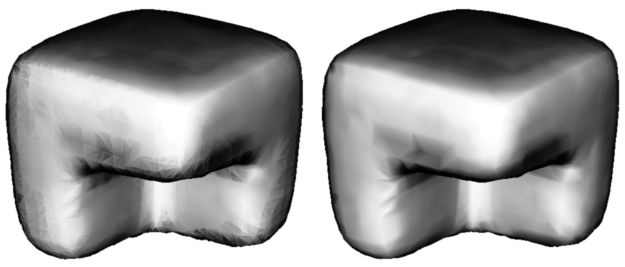


Figure 9: Iso-surface reconstructed from a tetrahedral grid. Color values of the image were equalized to enhance the effects. The left image was generated without using the interpolation weights necessary to achieve smooth results.

4.2 Directly slicing unstructured grids

It is now easy to derive an algorithm which allows us to reconstruct arbitrary slices out of the data. For each vertex we also store it's distance to the image plane and use it temporarily as the scalar material value. But then, a slice just corresponds to a planar iso-surface defined by an iso-value that is equal to the distance of that slice to the image plane. As a consequence our method for reconstructing shaded iso-surfaces can be applied directly. Even more efficiently, since we are only interested in the scalar material values faces are always rendered having smooth color interpolation.

For the method to proceed properly, we store the scalar values in the RG color components and the distance values in the Bα components issued at each vertex. Again we start rendering the back faces. For a slice at the distance d from the image plane only pixel values with an alpha value larger than or equal to d are accepted. We only allow RB pixel values to be written to the frame buffer. As usual, the stencil buffer is set where a pixel passes the depth test and the alpha test. Now the front faces are rendered but only the Gα components are going to be altered. Locking the stencil buffer is done in the same way as described.

Finally, all values necessary to correctly interpolate the scalar values within the actual slice are available in the pixel data. We read the data and calculate

$$S = \frac{R-d}{R-B} \cdot G + \frac{d-B}{R-B} \cdot \alpha,$$

for each RGBα pixel value. Before the resulting scalar value are written back into the frame buffer they get mapped via a lookup table provided by the graphics hardware. In order to approximate the volume rendering integral the grid is sliced multiple times and the generated images are blended properly.

The remarkable fact is that we never need to explicitly use the geometric description of primitives or the topology of the underlying grid. Sorting is implicitly done on the per-pixel basis in the rasterization unit. Polygon drawing is exploited for reconstructing scalar values on the cell faces. All values necessary to interpolate within one slice are accessed from the frame buffer. Moreover, adaptive slicing can easily be performed. For example, to preview the result at a very coarse level only a few slices have to be rendered.

4.3 Data Structures

In this section we briefly discuss the basic data structures used in the presented approaches. In particular the following two issues will be addressed:

- **Find all elements a particular iso-surface passes through**
- **Which elements contribute to a particular slice**

It is important that the decision whether a cell has to be rendered or not can be done in a predictive way. The naive approach to render every cell in each pass of the algorithm would cause the graphics hardware to collapse for sufficiently large data sets.

In iso-surface rendering we prefer a simple byte stream data structure which is easy to access and efficient to store. All scalar material values are scaled to the range [0,1] which is partitioned into an arbitrary number of equally spaced intervals.

Let us consider that we have N cells to process. For each interval a stream of $\lceil N/8 \rceil$ bytes is stored. A bit is set if not all of the values given at the vertices of the corresponding cell are completely less or larger than the bounds of the interval. For a surface to be rendered with respect to a particular iso-value it now suffices to find those entries in the relevant stream which are set. Since in general many adjacent entries within the same interval will be empty this kind of representation provides an effective data structure to be runlength encoded. In addition, this encoding scheme allows empty intervals to be skipped quite efficiently.

In our slicing approach to simulate direct volume rendering we mainly borrowed ideas from [31]. Each cell occurs twice in a set of lists, each of which is stored for exactly one slice. Since each element, in general, contribute to multiple slices, it is included into those lists which are stored for the first and the last slice that is covered by the element. In addition, an active cell list is utilized which captures all cells contributing to the slice that is actually being rendered. Obviously, other data structures [27] might be well suited for this kind of application, but we found the present one to be optimal in our test cases. Tree-like data structures, of course, avoid the incremental update from slice to slice and from viewpoint to viewpoint, but in order to keep the memory overhead moderate much more cells have to be rendered in general. In all our experiments the time necessary to update the cell lists was negligible compared to the final rendering times. We believe that the additional amount of memory introduced by this approach is fairly acceptable compared to the gains it offers.

5 Results

Compared to pure software solutions the accuracy of the results produced by our approaches strongly depends on the available frame buffer hardware. Whenever pixel values are read and processed further on the number of bits per color channel determines the precision that can be achieved. This turns out to be most critical in rendering shaded iso-surfaces with the pixel texture where we may access wrong locations in the texture domain. For example, let us consider a 1024^3 texture which should be processed using a 8 Bit frame buffer. Since pixel values are only precise within $\frac{1}{256}$ we may access voxel values which are about 4 cells aside.

Limited precision is also a problem in direct rendering of unstructured grids where the frame buffer is accessed multiple times to retrieve the scalar material values and the values necessary to perform the interpolation. By repeatedly blending the quantized data samples the final image might get degraded.

Throughout our experiments we used the 8 bit frame buffer on a SGI Maximum Impact where pixel textures have been exploited and in all other cases the 12 bit frame buffer of the RealityEngineII. The **SGIX_pbuffer**, an additional frame buffer invisible to the user, was used throughout the implementations since it can be locked exclusively in order to prevent other applications from drawing into pixel data to be read.

All the results were run on different data sets carefully chosen to outline the basic features of our approaches. The first row of Figure 10 shows a selection of images demonstrating the extensions to the traditional texture based volume rendering we developed. In the images a) and b) a simple box was used to clip the interior and the exterior of a MRI-scan. Image c) shows the benefits of pixel textures for the visualization of atmospheric data. The multi-pass algorithm as described was employed.

In the second row we compare results of the proposed rendering techniques for shaded iso-surfaces. The surface on the leftmost image was rendered in roughly 15 seconds using a software based ray-caster. 3D texture based gradient shading was run with 6 frames per second on the next image. The distance between successive slices was equal to the sampling intervals used in the software approach. The surface on the right appears somewhat brighter with a little less contrast, but basically there can hardly be seen any differences.

The next two images show the comparison between gradient shading and gradientless shading. Obviously, surfaces rendered by the latter one exhibit low contrast and even incorrect results are produced especially in regions where the variation of the gradient magnitude across the surface is high. Although the material distribution in the example data is almost iso-metric, at some points the differences can be easily recognized. At these surface points the step size used to compute the forward difference has to be increased, which, of course, can not be realized by our approach.

However, only one fourth of the memory needed in gradient shading is used in gradientless shading, and also the rendering times differ insignificantly. The only difference lies in the way the shading is finally computed. In gradient shading we copy the whole frame buffer once. In gradientless shading we have to read the pixel data and we have to write it twice with enabled pixel texturing. For a 512x512 viewport the difference was 0.08 seconds. Compared to the traditional rendering via 3D textures gradient shading ran about 0.04 seconds slower. On the other hand, since the overhead does not depend on the data resolution but on the size of the viewport we expect it's relative contribution to decrease rapidly with increasing data size.

Our final results illustrate the rendering of scalar volume data defined on tetrahedral grids (see last row in Figure 10). The first image shows an iso-surface from the NASA bluntfin which was converted into 225000 tetrahedra. To generate the 512x512 pixel image it took 0.2 seconds on a RE2 with one R10000 195 Mhz cpu.

Direct volume rendering of a finite-element data set is demonstrated by the second example. Notice the adaptive manipulation of the transfer function in order to indicate increasing temperature from blue to yellow. The glowing inner kernel can be clearly distinguished. In Table 1 we compare timings for various parts of the algorithm. These are mainly the elapsed times needed to read and write the frame buffer (**FbOps**) and to render the element's faces (**GrOps**), and the cpu time required to calculate the interpolation weights for all slices (**Cmp**).

Table 1: Processed primitives and timings (seconds) for the finite-element data set. (400 slices, 400x400 viewport)

	#Tetra	FbOps	GrOps	Cmp	Total
fedata0	60000	5.0	9.2	6.4	20.6
fedata1	110000	5.0	11.5	6.4	22.9
fedata2	150000	5.0	15.3	6.4	25.7
fedata3	200000	5.0	18.0	6.4	29.4

Observe that the time needed to compute the interpolation weights does not change since the number of pixels which have to be processed remains constant.

The experimental times are significantly faster than the times proposed in [31, 25, 22] without noticeable losses in the image quality. Compared to the lazy sweep algorithm [21], however, our method is slightly slower. The significant improvement is that the expected times do not depend on the grid topology. As a consequence we end up with constant frame rates for arbitrary topologies but equal number of primitives. This is a major difference to a variety of existing approaches which exploit the connectivity between cells. Once the first intersection with the boundaries has been computed the grid can be traversed very efficiently taking benefit of the pre-computed neighborhood information.

Our final image shows a particular state of a binary cellular automaton. A test case for which we expect other techniques taking advantage of the coherence between primitives to be significantly slower. Each primitive represents a living cell in the automaton. Originally, 8000 cells were generated. Each cell has been converted to 12 tetrahedra which have the center point of the original cell in common. In order to show a potential surface around the centers of the cells, there, the material values were set to one. At all other vertices they were set to zero. No connectivity information was used. The time needed to render the shown iso-surface was 1.8 seconds. Direct volume rendering with the same settings as described took 14.1 seconds.

Obviously, since the performance of our algorithms strongly depends on the throughput of the geometry engine as well as the rasterization unit, we expect them to be accelerated considerably if run on the currently available high-end systems like the InfiniteReality or future architectures.

6 Conclusion

In this paper we have presented many different ideas to exploit advanced features offered by modern high-end graphics workstations through standard APIs like OpenGL in volume rendering applications. In particular, real-time rendering of shaded iso-surfaces has been made possible for Cartesian and tetrahedral grids avoiding any polygonal representation. Furthermore, we have presented a direct volume rendering algorithm for unstructured grids with arbitrary topology by means of hardware supported geometry processing and color interpolation. Since any connectivity information has been abandoned we expect the frame rates to be independent of the grid topology.

Our results have shown that the presented methods are significantly faster than other methods previously proposed while only introducing slight image degradations. Since we take advantage of graphics hardware whenever possible, the number of operations to be performed in software is minimized making the algorithms relatively simple and easy to implement. However, there are still several areas to be explored in this research:

- **Hardware supported specular lighting effects and shadows** will help to improve the spatial perception of volumetric objects and may also be integrated in global illumination algorithms.
- **Image based pixel manipulations** are useful in a variety of applications, e.g. image based rendering, volume morphing or vector field visualization. In this context, the benefits of pixel textures have to be explored more carefully.
- **Multi-block data sets and arbitrary cell primitives** are a major challenge in scientific visualization. Efficient rendering algorithms for these kinds of representation still need to be developed.

7 Acknowledgments

Special thanks to Wolfgang Heidrich for valuable suggestions and advice concerning the OpenGL implementation, and Ove Sommer who did integrate some of the ideas into our OpenInventor based visualization tool.

References

[1] K. Akeley. Reality Engine Graphics. *ACM Computer Graphics, Proc. SIGGRAPH '93*, pages 109–116, July 1993.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *ACM Symposium on Volume Visualization '94*, pages 91–98, 1994.

[3] T.J. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.

[4] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *ACM Workshop on Volume Visualization '92*, pages 91–98, 1992.

[5] M. Garrity. Ray Tracing Irregular Volume Data. In *ACM Workshop on Volume Visualization '90*, pages 35–40, 1990.

[6] C. Giertsen. Volume Visualization of Sparse Irregular Meshes. *Computer Graphics and Applications*, 12(2):40–48, 1992.

[7] P. Haeberli and K. Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. *ACM Computer Graphics, Proc. SIGGRAPH '90*, pages 309–318, July 1990.

[8] M. Haubner, Ch. Krapichler, A. Lösch, K.-H. Englmeier, and van Eimeren W. Virtual Reality in Medicine - Computer Graphics and Interaction Techiques. *IEEE Transactions on Information Technology in Biomedicine*, 1996.

[9] J. T. Kajiya and B. P. Von Herzen. Ray Tracing Volume Densities. *ACM Computer Graphics, Proc. SIGGRAPH '84*, 18(3):165–174, July 1984.

[10] W. Krüger. The Application of Transport Theory to the Visualization of 3-D Scalar Data Fields. In *IEEE Visualization '90*, pages 273–280, 1990.

[11] P. Lacroute and M Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform. *Computer Graphics, Proc. SIGGRAPH '94*, 28(4):451–458, 1994.

[12] D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *ACM Computer Graphics, Proc. SIGGRAPH '93*, 25(4):285–288, July 1991.

[13] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[14] W.E. Lorensen and H.E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *ACM Computer Graphics, Proc. SIGGRAPH '87*, 21(4):163–169, 1987.

[15] N. Max, P. Hanrahan, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. In *ACM Workshop on Volume Visualization '91*, pages 27–33, 1991.

[16] C. Montani, R. Scateni, and R. Scopigno. Discretized Marching Cubes. In *IEEE Visualization'94*, pages 281–287, 1994.

[17] J. Montrym, D. Baum, D. Dignam, and C. Migdal. Infinite Reality: A Real-Time Graphics System. *Computer Graphics, Proc. SIGGRAPH '97*, pages 293–303, July 1997.

[18] M. Peercy, J. Airy, and B. Cabral. Efficient Bump Mapping Hardware. *Computer Graphics, Proc. SIGGRAPH '97*, pages 303–307, July 1997.

[19] H. Shen and C. Johnson. Sweeping Simplices: A Fast Iso-Surface Axtraction Algorithm for Unstructured Grids. In *IEEE Visualization '95*, pages 143–150, 1995.

[20] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *ACM Computer Graphics, Proc. SIGGRAPH '90*, 24(5):63–70, 1990.

[21] C. Silva and J. Mitchell. The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids. *Transactions on Visualization and Computer Graphics*, 4(2), June 1997.

[22] C. Silva, J. Mitchell, and A. Kaufman. Fast Rendering of Irregular Grids. In *ACM Symposium on Volume Visualization '96*, pages 15–23, 1996.

[23] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *ACM Symposium on Volume Visualization '94*, pages 83–90, 1994.

[24] A. Van Gelder and K. Kwansik. Direct Volume Rendering with Shading via Three-Dimensional Textures. In R. Crawfis and Ch. Hansen, editors, *ACM Symposium on Volume Visualization '96*, pages 23–30, 1996.

[25] R. Westermann and T. Ertl. The VSBUFFER: Visibility Ordering unstructured Volume Primitives by Polygon Drawing. In *IEEE Visualization '97*, pages 35–43, 1997.

[26] J. Wilhelms and A. Van Gelder. Octrees for faster Iso-Surface Generation. *ACM Transactions on Graphics*, 11(3):201–297, July 1992.

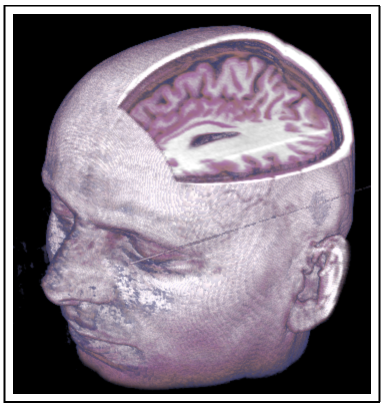
[27] J. Wilhelms, A. van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids. In *IEEE Visualization 1996*, pages 57–65, 1996.

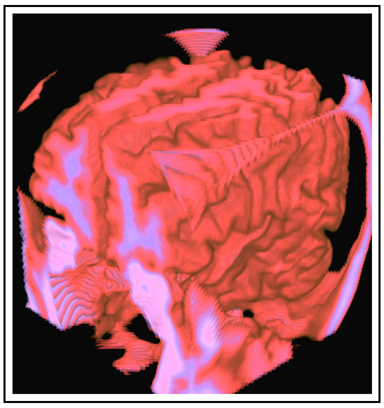
[28] P. Williams. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):102–126, 1992.

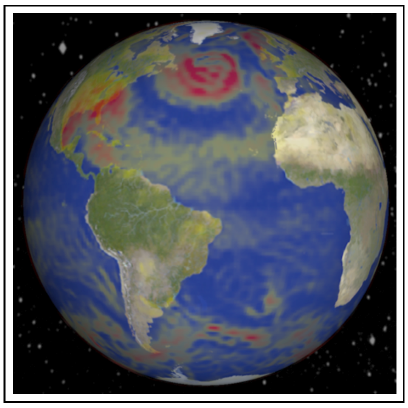
[29] P. Williams and N. Max. A Volume Density Optical Model. In *ACM Workshop on Volume Visualization '92*, pages 61–69, 1992.

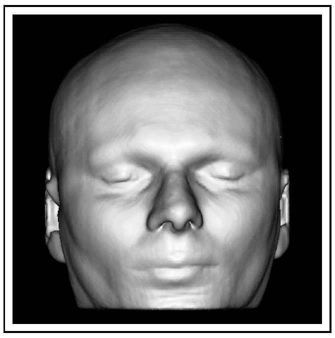
[30] O. Wilson, A. Van Gelder, and J. Wilhelms. Direct Volume Rendering via 3D Textures. Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, 1994.

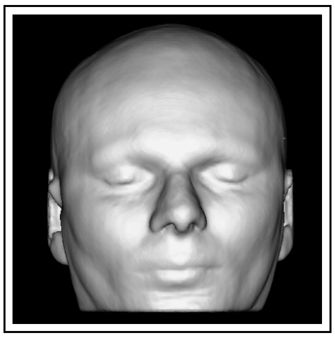
[31] R. Yagel, D. Reed, A. Law, P. Shih, and N. Shareef. Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. In *ACM Symposium on Volume Visualization '96*, pages 55–63, 1996.

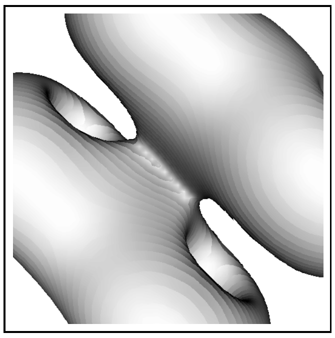

(a) Box clipping performed with the stencil buffer.

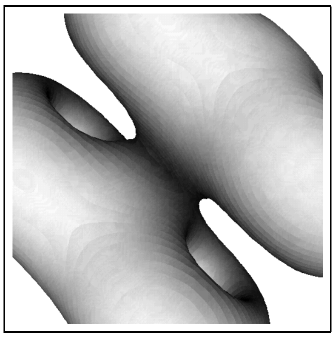

(b) Inverse box clipping of the brain.

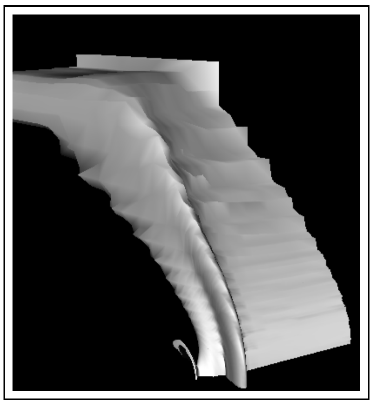

(c) Visualizing atmospheric volume data with the pixel texture.

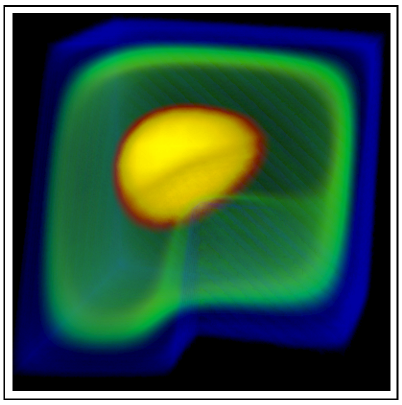

(d) Iso-surface rendering by software ray-casting.


(e) Iso-surface rendering performed with a 3D gradient texture.


(f) Iso-surface rendering performed with a 3D gradient texture.


(g) Iso-surface rendering by frame buffer arithmetic.


(h) Shaded iso-surface extracted from the NASA bluntfin data set.


(i) Direct volume rendering of a finite-element data set.

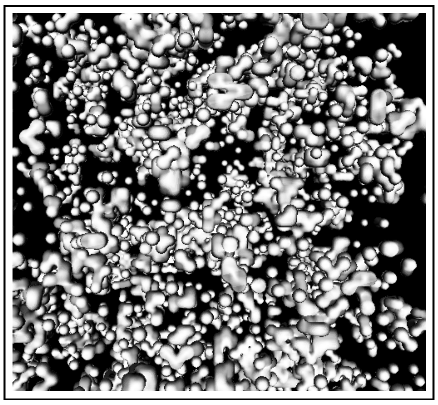

(j) Iso-surface displayed from a highly irregular topology.

Figure 10: Image plate showing the results of hardware supported volume visualization.