Volume Visualization and Volume Rendering Techniques

M. Meißner[†], H. Pfister[‡], R. Westermann[§], and C.M. Wittenbrink[¶]

Abstract

There is a wide range of devices and scientific simulation generating volumetric data. Visualizing such data, ranging from regular data sets to scattered data, is a challenging task.
This course will give an introduction to the volume rendering transport theory and the involved issues such as interpolation, illumination, classification and others. Different volume rendering techniques will be presented illustrating their fundamental features and differences as well as their limitations. Furthermore, acceleration techniques will be presented including pure software optimizations as well as utilizing special purpose hardware as VolumePro but also dedicated hardware such as polygon graphics subsystems.

1. Introduction

Volume rendering is a key technology with increasing importance for the visualization of 3D sampled, computed, or modeled datasets. The task is to display volumetric data as a meaningful two-dimensional image which reveals insights to the user. In contrast to conventional computer graphics where one has to deal with surfaces, volume visualization takes structured or unstructured 3D data which is the rendered into two-dimensional image. Depending on the structure and type of data, different rendering algorithms can be applied and a variety of optimization techniques are available. Within these algorithms, several rendering stages can be used to achieve a variety of different visualization results at differen cost. These stages might change their order from algorithm to algorithm or might even not be used by certain approaches.

In the following section, we will give a general introduction to volume rendering and the involved issues. Section 3 then presents a scheme to classify different approaches to volume rendering in categories. Acceleration techniques to speed up the rendering process in section 4. Section 3 and 4 are a modified version of tutorial notes from R. Yagel which we would like to thankfully acknowledge.
A side by side comparison of the four most common volume rendering algorithms is given in section 5. Special purpose hardware achieving interactive or real-time frame-rates is presented in section 6 while section 7 focuses on applications based on 3D texture mapping. Finally, we present rendering techniques and approaches for volume data not represented on rectilinear cartesian grids but on curvilinear and unstructured grids.

2. Volume rendering

Volume rendering differs from conventional computer graphics in many ways but also shares rendering techniques such as shading or blending. Within this section, we will give a short introduction into the types of data and where it originates from. Furthermore, we present the principle of volume rendering, the different rendering stages, and the issues involved when interpolating data or color.

2.1. Volume data acquisition

Volumetric data can be computed, sampled, or modeled and there are many different areas where volumetric data is available. Medical imaging is one area where volumetric data is frequently generated. Using different scanning techniques, internals of the human body can be acquired using MRI, CT, PET, or ultrasound. Volume rendering can

[†] WSI/GRIS, University of Tübingen, Auf der Morgenstelle 10/C9, Germany, e-Mail: meissner@gris.uni-tuebingen.de

[‡] MERL - A Mitsubishi Electric Research Laboratory, 201 Broadway, Cambridge, MA 02139, USA, e-Mail: pfister@merl.com

[§] Computer Graphics Group, University of Stuttgart, *something road 4711*, 74711 Stuttgart, Germany, e-Mail: Ruediger.Westermann@informatik.uni-stuttgart.de

[¶] Hewlett-Packard Laboratories, PaloAlto, CA 94304-1126, USA, e-Mail: craig_wittenbrink@hpl.hp.com

be applied to color the usually scalar data and visualize different structures transparent, semi-transparent, or opaque and hence, can give useful insights. Different applications evolved within this area such as cancer detection, visualization of aneurisms, surgical planning, and even real-time monitoring during surgery.

Nondestructive material testing and rapid prototyping is another example where frequently volumetric data is generated. Here, the structure of an object is of interest to either verify the quality or to reproduce the objects. Industrial CT scanners and ultrasound are mainly used for these applications.

The disadvantage of the above described acquisition devices is the missing color information which needs to be added during the visualization process since each acquisition techniques generates scalar values representing density (CT), oscillation (MRI), echoes (ultrasound), and others. For educational purposes where destructing the original object is acceptable, one can slice the material and take images of each layer. This reveals color information which so far cannot be captured by other acquisition devices. A well-known example is the visible human project where this technique has been applied to a male and a female cadaver.

Microscopic analysis is yet another application field of volume rendering. With confocal microscopes, it is possible to get high-resolution optical slices of a microscopic object without having to disturb the specimen.

Geoseismic data is probably one of the sources that generates the largest junk of data. Usually, at least 1024^3 voxels (1 GByte and more) are generated and need to be visualized. The most common application field is oil exploration where the costs can be tremendously reduced by finding the right location where to drill the whole.

Another large source of volumetric data is physical simulations where fluid dynamics are simulated. This is often done using particles or sample points which move around following physical laws resulting in unstructured points. These points can either be visualized directly or resampled into any grid structure possibly sacrificing quality.

Besides all the above mentioned areas, there are many others. For further reading we recommend [61].

2.2. Grid structures

Depending on the source where volumetric data comes from it might be given as a cartesian rectilinear grid, or as a curvilinear grid, or maybe even completely unstructured. While scanning devices mostly generate rectilinear grids (isotropic or anisotropic), physical simulations mostly generate unstructured data. Figure 1 illustrates these different grid types for the 2D case. For the different grid structures different algorithms can be used to visualize the volumetric data. Within the next sections, we will focus on rectilinear grids before presenting approaches for the other grid types in section 8.

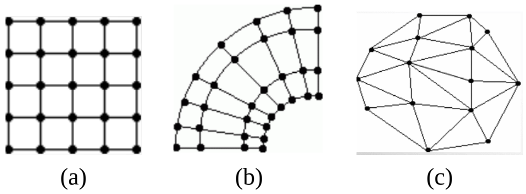


Figure 1: *Different grid structures: Rectilinear (a), curvilinear (b), and unstructured (c).*

2.3. Absorption and emission

In contrast to conventional computer graphics where objects are represented as surfaces with material properties, volume rendering does not directly deal with surfaces even though surfaces can be extracted from volumetric data in a preprocessing step.

Each element of the volumetric data (voxel) can emit light as well as absorb light. The emission of light can be quite different depending on the model used. I.e., one can implement models where voxels simply emit their own light or where they additionally realize single scattering or even multiple scattering. Depending on the model used, different visualization effects can be realized. Generally, scattering is much more costly to realize than a simple emission and absorption model, one of the reasons why they are hardly used in interactive or real-time applications. While the emission determines the color and intensity a voxel is emitting, the absorption can be expressed as opacity of a voxel. Only a certain amount of light will be passed through a voxel which can be expressed by $1 - opacity$ and is usually referred to as the transparency of a voxel.

The parameters of the emission (color and intensity) as well as the parameters of the absorption (opacity/transparency) can be specified on a per voxel-value base using classification. This is described in more detail in the following section. For different optical models for volume rendering refer to [66].

2.4. Classification

Classification enables the user to find structures within volume data without explicitly defining the shape and extent of that structure. It allows the user to see inside an object and explore its inside structure instead of only visualizing the surface of that structure as done in conventional computer graphics.

In the classification stage, certain properties are assigned to a sample such as color, opacity, and other material properties. Also shading parameters indicating how shiny a structure should appear can be assigned. The assignment of opacity to a sample can be a very complex operation and has a major impact on the final 2D image generated. In order to assign these material properties it is usually helpful to use histograms illustrating the distribution of voxel values across the dataset.

The actual assignment of color, opacity, and other properties can be based on the sample value only but other values can be as well taken as input parameters. Using the gradient magnitude as further input parameter, samples within homogeneous space can be interpreted differently than the ones with heterogeneous space. This is a powerful technique in geoseismic data where the scalar values only change noticeably in between different layers in the ground.

2.5. Segmentation

Empowering the user to see a certain structure using classification is not always possible. A structure can be some organ or tissue but is represented as a simple scalar value. When looking at volumetric data acquired with a CT scanner, different types of tissue will result in same density values due to the nature of CT. Therefore, no classification of density values can be found such that structures which similarly absorb X-rays could be separated. To separate such structures, they need to be labeled or segmented such that they can be differentiated from each other. Depending on the acquisition method and the scanned object, it can be relatively easily, hard, or even impossible to segment some of the structures automatically. Most algorithms are semi-automatic or optimized for segmenting a specific structure.
Once a volumetric dataset is segmented, for each segment a certain classification can be assigned and applied during rendering.

2.6. Shading

Shading or illumination refer to a well-know technique used in conventional computer graphics to greatly enhance the appearance of a geometric object that is being rendered. Shading tries to model effects like shadows, light scattering, and absorption in the real world when light falls on an object. Shading can be classified into global methods, direct methods, and local methods. While global illumination computes the light being exchanged between all objects, direct illumination only accounts for the light the directly falls onto an object. Unfortunately, both methods depend on the complexity of the objects to be rendered and are usually not interactive. Therefore, the local illumination method has been widely used. Figure 2 shows a skull rendered with local and with direct illumination. While direct illumination takes into account how much light is present at each sample (figure 2(b)), local illumination is much cheaper to compute but still achieves reasonable image quality (figure 2(a)). Local illumination consists of an ambient, a diffuse and a specular component. While ambient component is available everywhere, the diffuse component can be computed using the angle between the normal vector at the given location and the vector to the light. The specular component depends on the angle to the light and the angle to the eye position. All three components can be combined by weighting each of them differently using material properties. While tissue is less likely to have specular components, teeth might reflect more light. Figure 3 shows a skull without and with shading.

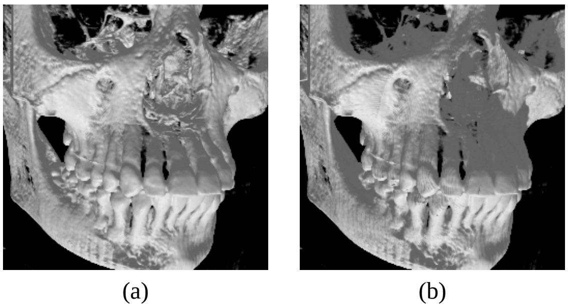


Figure 2: *Comparison of shading: Local illumination (a) and direct illumination (b).*

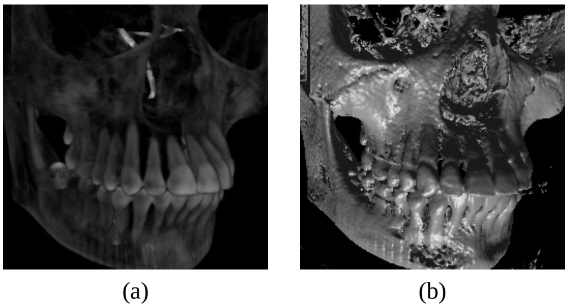


Figure 3: *Comparison of shading: No shading (a) and local shading (b).*

For further reading refer to [61, 66].

2.7. Gradient computation

As mentioned in the previous section, a normal is required to be able to integrate shading effects. However, volumetric data itself does not explicitly consist of surfaces with associated normals but of sampled data being available on grid positions. This grid of scalar values can be considered as a grey level volume and several techniques have been investigated in the past to compute grey-level gradients from volumetric data.

A frequently used gradient operator is the central difference operator. For each dimension of the volume, the central difference of the two neighbouring voxels is computed which gives an approximation of the local change of the gray value. It can be written as $Gradient_{x,y,z} = [-1 \quad 0 \quad 1]$. Generally, the central difference operator is not the necessarily the best one but very cheap to compute since it requires only six voxels and three subtractions. A disadvantage of the central difference operator is that it produces anisotropic gradients.

The intermediate difference operator is similar to the central difference operator but has a smaller kernel. It can be written as $Gradient_{x,y,z} = [-1 \quad 1]$. The advantage of this operator is that it detects high frequencies which can be lost when using the central difference operator. However, when flipping the orientation a different gradient is computed for the identical voxel position which can cause undesired effects.

A much better gradient operator is the Sobel operator which uses all 26 voxels that surround one voxel. This gradient operator was developed for 2D imaging but volume rendering borrows many techniques from image processing and the Sobel operator can easily be extended to 3D. A nice property of this operator is that it produces nearly isotropic gradients but it is fairly complex to compute [61].

2.8. Compositing

All samples taken during rendering need to be combined into a final image which means that for each pixel of the image we need to combine the color of the contributing samples. This can be done in random order if only opaque samples are involved but since we deal with semi-transparent data, the blending needs to be performed in sorted order which can be accomplished in two ways: Front to back or back to front. For front to back, the discrete ray casting integral can then be written as:

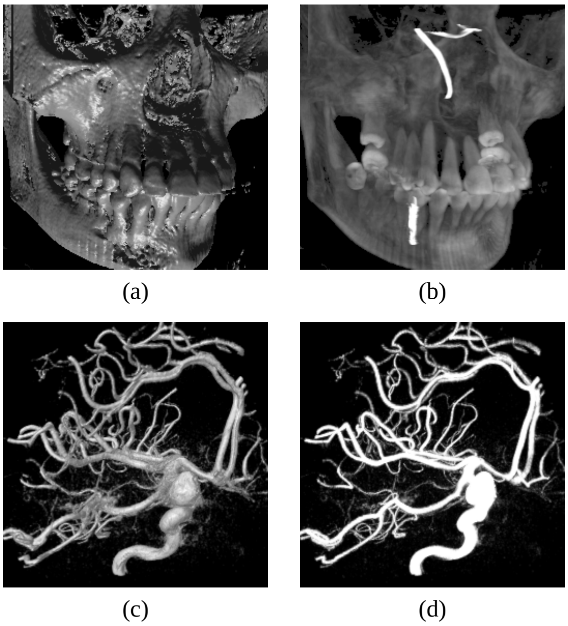
```
Trans = 1.0;  - full
Inten = I[0]; - initial value
for (i=0; i<n; i++) {
   Trans *= T[i-1];
   Inten += Trans * I[i];
}
```

The advantage is that the computation can be terminated once the transparency reaches a certain threshold where no further contribution will be noticeable, i.e. 0.01.

For back to front, compositing is much less work since we do not need to keep track of the remaining transparency. However, then all samples need to be processed and no early termination criteria can be exploited:

```
Inten = I[0]; - initial value
for (i=0; i<n; i++) {
   Inten = Inten + T[i] * I[i];
}
```

Instead of accumulating the color for each pixel over all samples using the above described blending operations, one can chose other operators. Another famous operator simply takes the maximum density value of all samples of a pixel, known as maximum intensity projection (MIP). This is mostly used in medical applications dealing with MRI data (magnetic resonance angiography) visualizing arteries that have been acquired using contrast agents.



(a) (b)

(c) (d)

Figure 4: *Compositing operators: Blending shaded samples of skull (a) and arteries (c) and maximum intensity projection of skull (b) and arteries (d).*

2.9. Filtering

Many volume rendering algorithms resample the volumetric data in a certain way using rays, planes, or random sample points. These sample points seldomly coincide with the actual grid positions and require the interpolation of a value based on the neighbouring values at grid position.

There are numerous different interpolation methods. Each of them is controlled by an interpolation kernel. The shape of the interpolation kernel provides the coefficients for the weighted interpolation sum. Interpolation kernels can be thought of as overlays. When a value needs to be interpolated, the kernel is placed onto of the neighbouring values. The kernel is centered at the interpolation point and everywhere the interpolation kernel intersects with the voxels, the values are multiplied. One dimensional interpolation kernels can be applied to interpolate in two, three, and even more dimensions if the kernel is separable. All of the following interpolation kernels are separable.

The nearest neighbour interpolation method is the simplest and crudest method. The value of the closest of all neighbouring voxel values is assigned to the sample. Hence, it is more a selection than a real implementation. Therefore, when using nearest neighbour interpolation, the image quality is fairly low and when using magnification, a blobby structure appears.

Trilinear interpolation assumes a linear relation between neighbouring voxels and it is separable. Therefore, it can be decomposed into seven linear interpolations. The achievable image quality is much higher than with nearest neighbour interpolation. However, when using large magnification factors, three dimensional diamond structures or crosses appear due to the nature of the trilinear kernel.

Better quality can be achieved using even higher order interpolation methods such as cubic convolution or B-spline interpolation. However, there is a trade-off between quality and computational cost as well as memory bandwidth. These filters require a neighbourhood of 64 voxels and a significant larger amount of computations than trilinear interpolation. It depends on the application and the requirements which interpolation scheme should be used.

2.10. Color filtering

The previously mentioned techniques can be performed in different order resulting in different image quality as well as being prone to certain artifacts. Interpolation of scalar values is usually prone to aliasing since depending on the classification used, high frequency details might be missed. On the other side, color interpolation by the means of classification and shading of available voxel values and interpolation of the resulting color values is prone to color bleeding when interpolating color and α-value independent from each other [119]. A simple example of this is bone surrounded by flesh where the bone is classified opaque white and the flesh is transparent but red. When sampling this color volume one needs to chose the appropriate interpolation scheme. Simply interpolating the neighbouring color and opacity values results in color bleeding as illustrated in figure 5. To obtain the correct color and opacity, one needs to mulitply each color with the corresponding opacity value before interpolating the color. While it can easily be noticed in figure 5 due to the chosen color scheme, it is less obvious in monochrome images.

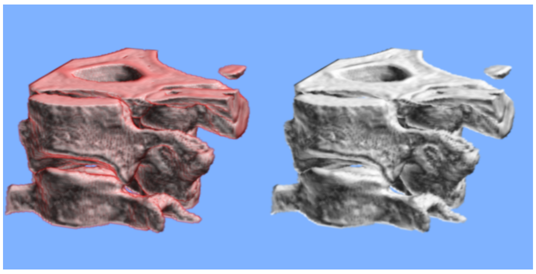


Figure 5: *Color bleeding: Independent interpolation of color and opacity values (left) and opacity weighted color interpolation (right).*

Figure 6 illustrates another example where darkening artifacts can be noticed. This example is a volumetric dataset from image based rendering that originates as color (red, green, and blue) at each grid position. Therefore, it illustrates what would happen when visualizing the visible human with and without opacity weighted color interpolation. The artifacts are quite severe and disturbing.

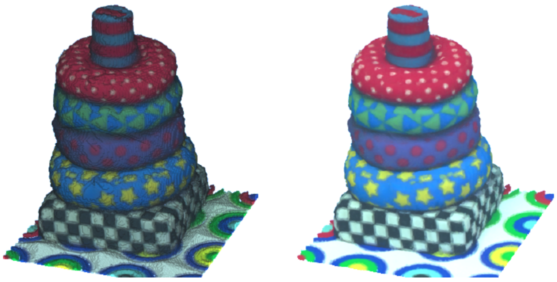


Figure 6: *Darkening and aliasing: Independent interpolation of color and opacity values (left) and opacity weighted color interpolation (right).*

2.11. Summary

Within this section, we provided an overview of the different types of grids as well as sources of volume data. Using classification, gradient estimation, shading, and compositing, extremly different visualization results can be achieved. Also the selection of the filter used to interpolate data or color has a strong influence on the resulting image. Therefore, one has to carefully choose depending on the application requirements which of the described techniques schemes should be integrated and which interpolation scheme used.

3. Volume Viewing Algorithms

The task of the rendering process is to display the primitives used to represent the 3D volumetric scene onto a 2D screen. Rendering is composed of a viewing process which is the subject of this section, and the shading process. The projection process determines, for each screen pixel, which objects are seen by the sight ray cast from this pixel into the scene. The viewing algorithm is heavily dependent on the display primitives used to represent the volume and whether volume rendering or surface rendering are employed. Conventional viewing algorithms and graphics engines can be utilized to display geometric primitives, typically employing surface rendering. However, when volume primitives are displayed directly, a special volume viewing algorithm should be employed. This algorithm should capture the contents of the voxels on the surface as well as the inside of the volumetric object being visualized. This section surveys and compares previous work in the field of direct volume viewing.

3.1. Introduction

The simplest way to implement viewing is to traverse all the volume regarding each voxel as a 3D point that is transformed by the viewing matrix and then projected onto a Z-buffer and drawn onto the screen. Some methods have been suggested in order to reduce the amount of computations needed in the transformation by exploiting the spatial coherency between voxels. These methods are described in more details in Section 4.1.

The back-to-front (BTF) algorithm is essentially the same as the Z-buffer method with one exception that is based on the observation that the voxel array is presorted in a fashion that allows scanning of its component in an order of decreasing or increasing distance from the observer. Exploiting this presortedness of the voxel arrays, traversal of the volume in the BTF algorithm is done in order of decreasing distance to the observer. This avoids the need for a Z-buffer for hidden voxel removal considerations by applying the painter's algorithm by simply drawing the current voxel on top of previously drawn voxels or by compositing the current voxel with the screen value [24].

The front-to-back (FTB) algorithm is essentially the same as BTF only that now the voxels are traversed in increasing distance order. Front-to-back has the potential of a more efficient implementation by employing a dynamic data structure for screen representation [87] in which only un-lit pixels are processed and newly-lit pixels are efficiently removed from the data structure. It should be observed that while in the basic Z-buffer method it is impossible to support the rendition semi-transparent materials since voxels are mapped to the screen in arbitrary order. Compositing is based on a computation that simulates the passage of light through several materials. In this computation the order of materials is crucial. Therefore, translucency can easily be realized in both BTF and FTB in which objects are mapped to the screen in the order in which the light traverses the scene.

Another method of volumetric projection is based on first transforming each slice from voxel-space to pixel-space using 3D affine transformation (shearing) [33, 90, 52] and then projecting it to the screen in a FTB fashion, blending it with the projection formed by previous slices [22]. Shear-warp rendering [52] is currently the fastest software algorithm. It achieves 1.1 Hz on a single 150MHz R4400 processor for a $256 \times 256 \times 225$ volume with 65 seconds of pre-processing time [51]. However, the 2D interpolation may lead to aliasing artifacts if the voxel values or opacities contain high frequency components [50].

Westover [107, 108] has introduced the splatting technique in which each voxel is transformed into screen space and then shaded. Blurring, based on 2D lookup tables is performed to obtain a set of points (footprint) that spreads the voxels energy across multiple pixels. These are then composited with the image array. Sobierajski et al. have described [94] a simplified splatting for interactive volume viewing in which only voxels comprising the object's surface are maintained. Rendering is based on the usage of a powerful transformation engine that is fed with multiple points per voxel. Additional speedup is gained by culling voxels that have a normal pointing away from the observer and by adaptive refinement of image quality.

The ray casting algorithm casts a ray from each pixel on the screen into the volume data along the viewing vector until it accumulates an opaque value [4, 99, 100, 56]. Levoy [59, 57] has used the term ray tracing of volume data to refer to ray casting and compositing of even-spaced samples along the primary viewing rays. However, more recently, ray tracing is referred to as the process where reflected and transmitted rays are traced, while ray casting solely considers primary rays, and hence, does not aim for "photorealistic" imaging. Rays can be traced through a volume of color as well as data. Ray casting has been applied to volumetric datasets, such as those arising in biomedical imaging and scientific visualization applications (e.g., [22, 100]).

We now turn to classify and compare existing volume viewing algorithms. In section 4 we survey recent advances in acceleration techniques for forward viewing (section 4.1), backward viewing (section 4.2) and hybrid viewing (section 4.3).

3.2. Classification of Volume Viewing Methods

Projection methods differ in several aspects which can be used for a their classification in various ways. First, we have to observe whether the algorithm traverses the volume and projects its components onto the screen (called also forward, object-order, or voxel-space projection) [24, 107, 29], does it traverse the pixels and solve the visibility problem for each one by shooting a sight ray into the scene (called also backward, image-order, or pixel-space projection) [46, 54, 88, 99, 100, 120, 124], or does it perform some kind of a hybrid traversal [40, 100, 52, 50].

Volume rendering algorithms can also be classified according to the partial voxel occupancy they support. Some algorithms [86, 35, 87, 99, 125, 124] assume uniform (binary) occupancy, that is, a voxel is either fully occupied by some object or it is devoid of any object presence. In contrast to uniform voxel occupancy, methods based on partial voxel occupancy utilize intermediate voxel values to represent partial voxel occupancy by objects of homogeneous material. This provides a mechanism for the display of objects that are smaller than the acquisition grid or that are not aligned with it. Partial volume occupancy can be used to estimate occupancy fractions for each of a set of materials that might be present in a voxel [22]. Partial volume occupancy is also assumed whenever gray-level gradient [36] is used as a measure for the surface inclination. That is, voxel values in the neighborhood of a surface voxel are assumed to reflect the relative average of the various surface types in them.

Volume rendering methods differ also in the way they regard the material of the voxels. Some methods regarded all materials as opaque [27, 29, 37, 87, 98, 99] while others allow each voxel to have an opacity attribute [22, 54, 88, 100, 107, 120, 124, 52]. Supporting variable opacities models the appearance of semi-transparent jello and requires composition of multiple voxels along each sight ray.

Yet another aspect of distinction between rendering methods is the number of materials they support. Early methods supported scenes consisting of binary-valued voxels while more recent methods usually support multi-valued voxels. In the first case objects are represented by occupied voxels while the background is represented by void voxels [24, 35, 87, 99]. In the latter approach, multi-valued voxels are used to represent objects of non-homogeneous material [27, 36, 98]. It should be observed that given a set of voxels having multiple values we can either regard them as fully occupied voxels of various materials (i.e., each value represents a different material) or we can regard the voxel value as an indicator of partial occupancy by a single material, however we can not have both. In order to overcome this limitation, some researchers adopt the multiple-material approach as a basis for a classification process that attaches a material-label to each voxel. Once each voxel has a material label, these researchers regard the original voxel values as partial occupancy indicators for the labeled material [22].

Finally, volume rendering algorithms can also be classified according to whether they assume constant value across the voxel extent [46] or do they assume (trilinear) variation of the voxel value[54].

A severe problem in the voxel-space projection is that at some viewing points, holes might appear in the scene. To solve this problem one can regard each voxel in our implementation as a group of points (depending on the viewpoint) [95] or maintain a ratio of $1 : \sqrt{3}$ between a voxel a pixel [13]. Another solution is based on a hybrid of voxel-space and pixel-space projections that is based on traversing the volume in a BTF fashion but computing pixel colors by intersecting the voxel with a scan line (plane) and then integrating the colors in the resulting polygon [100]. Since this computation is relatively time consuming it is more suitable to small datasets. It is also possible to apply to each voxel a blurring function to obtain a 2D footprint that spreads the sample's energy onto multiple image pixels which are latter composed into the image [108]. A major disadvantage in the splatting approach is that it tends to blur the edges of objects and reduce the image contrast. Another deficiency in the voxel-space projection method is that it must traverse and project all the voxels in the scene. Sobierajski et al. have suggested the use of a normal based culling in order to reduce (possibly by half) the amount of processed voxels [95]. On the other hand, since voxel-space projection operates in object-space, it is most suitable to various parallelization schemes based on object space subdivision [28, 79, 107].

The main disadvantages of the pixel-space projection scheme are aliasing (specially when assuming uniform value across voxel extent) and the difficulty to parallelize it. While the computation involved in tracing rays can be performed in parallel, memory becomes the bottleneck. Since rays traverse the volume in arbitrary directions it seems to be no way to distribute voxels between memory modules to guarantee contention free access [55].

Before presenting a side by side comparison of the four most popular volume rendering algorithms, we will introduce general acceleration techniques that can be applied to forward and backward viewing algorithms.

4. Acceleration Techniques

Either forward projection or backward projection requires the scanning of the volume buffer which is a large buffer of size proportional to the cubic of the resolution. Consequently, volume rendering algorithms can be very time-consuming algorithms. This section focuses on techniques for expediting these algorithms.

4.1. Expediting Forward Viewing

The Z-buffer projection algorithm, although surprisingly simple, is inherently very inefficient and when naively implemented, produces low quality images. The inefficiency attribute of this method is rooted in the N^3 vector-by-matrix multiplications it calculates and the N^3 accesses to the Z-buffer it requires. Inferior image quality is caused by this method's inability to support compositing of semitransparent materials, due to the arbitrary order in which voxels are transformed. In addition, transforming a set of discrete points is a source for various sampling artifacts such as holes and jaggies.

Some methods have been suggested to reduce the amount of computations needed for the transformation by exploiting the spatial coherency between voxels. These methods are: recursive "divide and conquer" [27, 69], pre-calculated tables [24], and incremental transformation [44, 65].

The first method exploits coherency in voxel space by representing the 3D volume by an octree. A group of neighboring voxels having the same value (or similar, up to a threshold value) may, under some restrictions, be grouped into a uniform cubic subvolume. This aggregate of voxels can be transformed and rendered as a uniform unit instead of processing each of its voxels. In addition, since each octree node has eight equally-sized octants, given the transformation of the parent node, the transformation of its sub-octants can be efficiently computed. This method requires, in 3D, three divisions and six additions per coordinate transformation.

The table-driven transformation method [24] is based on the observation that volume transformation involves the multiplication of the matrix elements with integer values which are always in the range $[1 \ldots N]$ where N is the volume resolution. Therefore, in a short preprocessing stage each matrix element t_{ij} is stored in table $tab_{ij}[N]$ such that $tab_{ij}[k] = t_{ij} \times k, 1 \leq k < N$. During the transformation stage, coordinate by matrix multiplication is replaced by table lookup. This method requires, in 3D, nine table lookup operations and nine additions, per coordinate transformation.

Finally, the incremental transformation method is based on the observation that the transformation of a voxel can be incrementally computed given the transformed vector of the voxel. To begin the incremental process we need one matrix by vector multiplication to compute the updated position of the first grid point. The remaining grid points are incrementally transformed, requiring three additions per coordinate. However, to employ this approach, all volume elements, including the empty ones, have to be transformed. This approach is therefore more suitable to parallel architecture where it is desired to keep the computation pipeline busy [65].

So far we have been looking at methods that ease the computation burden associated with the transformation. However, consulting the Z-buffer N^3 times is also a source of significant slow down. The back-to-front (BTF) algorithm is essentially the same as the Z-buffer method with one exception the order in which voxels are scanned. It is based on the observation that the voxel array is spatially presorted. This attribute allows the renderer to scan the volume in an order of decreasing distance from the observer. By exploiting this presortedness of the voxel arrays, one can draw the volume in a back-to-front order, that is, in order of decreasing distance to the observer. This avoids the need for a Z-buffer for hidden voxel removal by applying the painter's algorithm. That is, the current voxel is simply drawn on top of previously drawn voxels. If compositing is performed, the current voxel is composited with the screen value [23, 24]. The front-to-back (FTB) algorithm is essentially the same as BTF, only that now the voxels are traversed in increasing distance order.

As mentioned above in the basic Z-buffer method it is impossible to support the rendition of semitransparent materials because voxels are mapped to the screen in an arbitrary order. In contrast, translucency can easily be realized in both BTF and FTB because in these methods objects are mapped to the screen in viewing order.

Another approach to forward projection is based on first transforming the volume from voxel-space to pixel-space by employing a decomposition of the 3D affine transformation into five 1D shearing transformations [33]. Then, the transformed voxel is projected onto the screen in an FTB order, which supports the blending of voxels with the projection formed by previous (farther) voxels [22]. The major advantage of this approach is its ability (using simple averaging techniques) to overcome some of the sampling problems causing the production of low quality images. In addition, this approach replaces the 3D transformation by five 1D transformations which require only one floating-point addition each.

Another solution to the image quality problem mentioned above is splatting [108], in which each voxel is transformed into screen space and then it is shaded. Blurring, based on 2D lookup tables, is performed to obtain a set of points (a cloud) that spreads the voxel's energy across multiple pixels called footprint. These are then composited with the image array. However this algorithm which requires extensive filtering is time consuming.

Sobierajski et al. have described [94] a simplified approximation to the splatting method for interactive volume viewing in which only voxels comprising the object's surface are maintained. Each voxel is represented by several 3D points (a 3D footprint). Rendering is based on the usage of a contemporary geometry engine that is fed with those multiple points per voxel. Additional speedup is gained by culling voxels that have a normal pointing away from the observer. Finally, adaptive refinement of image quality is also supported: when the volume is manipulated only one point per voxel is rendered, interactively producing a low quality image. When the volume remains stationary and unchanged, for some short period, the rendering system renders the rest of the points to increase image quality.

Another efficient implementation of the splatting algorithm, called hierarchical splatting [53] uses a pyramid data structure to hold a multiresolution representation of the volume. For volume of N^3 resolution the pyramid data structure consists of a sequence of $\log N$ volumes. The first volume contains the original dataset, the next volume in the sequence is half the resolution of the previous one. Each of its voxels contains the average of eight voxels in the higher resolution volume. According to the desired image quality, this algorithm scans the appropriate level of the pyramid in a BTF order. Each element is splatted using the appropriate sized splat. The splats themselves are approximated by polygons which can efficiently be rendered by graphics hardware.

4.2. Expediting Backward Viewing

Backward viewing of volumes, based on casting rays, has three major variations: parallel (orthographic) ray casting, perspective ray casting, and ray tracing. The first two are variations of ray casting, in which only primary rays, that is, rays from the eye through the screen, are followed. These two methods have been widely applied to volumetric datasets, such as those arising in biomedical imaging and scientific visualization applications (e.g., [22, 100]). Levoy [57, 58] has used the term ray tracing of volume data to refer to ray casting and compositing of even-spaced samples along the primary viewing rays.

Ray casting can further be divided into methods that support only parallel viewing, that is, when the eye is at infinity and all rays are parallel to one viewing vector. This viewing scheme is used in applications that could not benefit from perspective distortion such as biomedicine. Alternatively, ray casting can be implemented to support also perspective viewing.

Since ray casting follows only primary rays, it does not directly support the simulation of light phenomena such as reflection, shadows, and refraction. As an alternative, Yagel et al. have developed the 3D raster ray tracer (RRT) [122] that recursively considers both primary and secondary rays and thus can create "photorealistic" images. It exploits the voxel representation for the uniform representation and ray tracing of sampled and computed volumetric datasets, traditional geometric scenes, or intermixing thereof.

The examination of existing methods for speeding up the process of ray casting reveals that most of them rely on one or more of the following principles: (1) pixel-space coherency (2) object-space coherency (3) inter-ray coherency and (4) space-leaping.

We now turn to describe each of those in more detail.

1. **Pixel-space coherency:** There is a high coherency between pixels in image space. That is, it is highly probable that between two pixels having identical or similar color we will find another pixel having the same (or similar) color. Therefore it is observed that it might be the case that we could avoid sending a ray for such obviously identical pixels.
2. **Object-space coherency:** The extension of the pixel-space coherency to 3D states that there is coherency between voxels in object space. Therefore, it is observed that it should be possible to avoid sampling in 3D regions having uniform or similar values.
3. **Inter-ray coherency:** There is a great deal of coherency between rays in parallel viewing, that is, all rays, although having different origin, have the same slope. Therefore, the set of steps these rays take when traversing the volume are similar. We exploit this coherency so as to avoid the computation involved in navigating the ray through voxel space.
4. **Space-leaping:** The passage of a ray through the volume is two phased. In the first phase the ray advances through the empty space searching for an object. In the second phase the ray integrates colors and opacities as it penetrates the object (in the case of multiple or concave objects these two phases can repeat). Commonly, the second phase involves one or a few steps, depending on the object's opacity. Since the passage of empty space does not contribute to the final image it is observed that skipping the empty space could provide significant speed up without affecting image quality.

The adaptive image supersampling, exploits the pixel-space coherency. It was originally developed for traditional ray-tracing [7] and later adapted to volume rendering [57, 60]. First, rays are cast from only a subset of the screen pixels (e.g., every other pixel). "Empty pixels" residing between pixels with similar value are assigned an interpolated value. In areas of high image gradient additional rays are cast to resolve ambiguities.

Van Walsum et al. [104] have used the voxel-space coherency. In his method the ray starts sampling the volume in low frequency (i.e., large steps between sample points). If a large value difference is encountered between two adjacent samples, additional samples are taken between them to resolve ambiguities in these high frequency regions. Recently, this basic idea was extended to efficiently lower the sampling rate in either areas where only small contributions of opacities are made, or in regions where the volume is homogeneous [20]. This method efficiently detects regions of low presence or low variation by employing a pyramid of volumes that decode the minimum and maximum voxel value in a small neighborhood, as well as the distance between these measures.

The template-based method [120, 124] utilizes the inter-ray coherency. Observing that, in parallel viewing, all rays have the same form it was realized that there is no need to reactivate the discrete line algorithm for each ray. Instead, we can compute the form of the ray once and store it in a data structure called ray-template. All rays can then be generated by following the ray template. The rays, however, differ in the exact positioning of the appropriate portion of the template, an operation that has to be performed very carefully. For this purpose a plane that is parallel to one of the volume faces is chosen to serve as a base-plane for the template placement. The image is projected a by sliding the template along that plane emitting a ray at each of its pixels. This placement guarantees complete and uniform tessellation of the volume. The regularity and simplicity of this efficient algorithm make it very attractive for hardware implementation [121].

So far we have seen methods that exploit some type of coherency to expedite volumetric ray casting. However, the most prolific and effective branch of volume rendering acceleration techniques involve the utilization of the fourth principle mentioned above – speeding up ray casting by providing efficient means to traverse the empty space.

The hierarchical representation (e.g., octree) decomposes the volume into uniform regions that can be represented by nodes in a hierarchical data structure. An adjusted ray traversal algorithm skips the (uniform) empty space by maneuvering through the hierarchical data structure [57, 89]. It was also observed that traversing the hierarchical data structure is inefficient compared to the traversal of regular grids. A combination of the advantages of both representations is the uniform buffer. The "uniformity information" decoded by the octree can be stored in the empty space of a regular 3D raster. That is, voxels in the uniform buffer contain either a data value or information indicating to which size empty octant they belong. Rays which are cast into the volume encounter either a data voxel, or a voxel containing "uniformity information" which instructs the ray to perform a leap forward that brings it to the first voxel beyond the uniform region [16]. This approach saves the need to perform a tree search for the appropriate neighbor – an operation that is the most time consuming and the major disadvantage in the hierarchical data structure.

When a volume consists of one object surrounded by empty space, a common and simple method to skip most of this empty space uses the well known technique of bounding-boxes. The object is surrounded by a tightly fit box (or other easy-to-intersect object such as sphere). Rays are intersected with the bounding object and start their actual volume traversal from this intersection point as opposed to starting from the volume boundary. The PARC (Polygon Assisted Ray Casting) approach [3] strives to have a better fit by allowing a convex polyhedral envelope to be constructed around the object. PARC utilizes available graphics hardware to render the front faces of the envelope (to determine, for each pixel, the ray entry point) and back faces (to determine the ray exit point). The ray is then traversed from entry to exit point. A ray that does not hit any object is not traversed at all.

It is obvious that the empty space does not have to be sampled – it has only to be crossed as fast as possible. Therefore, Yagel et al. have proposed [123, 122] to utilize one fast and crude line algorithm in the empty space (e.g., 3D integer-based 26-connected line algorithm) and another, slower but more accurate (e.g., 6-connected integer or 3D DDA floating point line algorithm), in the vicinity and interior of objects. The effectiveness of this approach depends on its ability to efficiently switch back and forth between the two line algorithm, and its ability to efficiently detect the proximity of occupied voxels. This is achieved by surrounding the occupied voxels by a one-voxel-deep "cloud" of flag-voxels, that is, all empty voxels neighboring an occupied voxel are assigned, in a preprocessing stage, a special "vicinity flag". A crude ray algorithm is employed to rapidly traverse the empty space until it encounters a vicinity voxel. This flags the need to switch to a more accurate ray traversal algorithm. Encountering later an empty voxel (i.e., unoccupied and not carrying the vicinity flag) can signal a switch back to the rapid traversal of empty space.

The proximity-clouds method [16, 128] is based on the extension of this idea even further. Instead of having a one-voxel-deep vicinity cloud this method computes, in a preprocessing stage, for each empty voxel, the distance to the closest occupied voxel. When a ray is sent into the volume it can either encounter an occupied voxel, to be handled as usual, or a "proximity voxel" carrying the value . This suggests that the ray can take a -step leap for-n n ward, being assured that there is no object in the skipped span of voxels. The effectiveness of this algorithm is obviously dependent on the ability of the line traversal algorithm to efficiently jump arbitrary number of steps [16].

Yagel and Shi [127] have reported on a method for speeding up the process of volume rendering a sequence of images. It is based on exploiting coherency between consecutive images to shorten the path rays take through the volume. This is achieved by providing each ray with the information needed to leap over the empty space and commence volume traversal at the vicinity of meaningful data. The algorithm starts by projecting the volume into a C-buffer (Coordinate-buffer) which stores, at each pixel location, the object-space coordinates of the first non empty voxel visible from that pixel. For each change in the viewing parameters, the C-buffer is transformed accordingly. In the case of rotation the transformed C-buffer goes through a process of eliminating coordinates that possibly became hidden [30]. The remaining values in the C-buffer serve as an estimate of the point where the new rays should start their volume traversal.

4.3. Hybrid Viewing

The most efficient rendering algorithm uses a ray-casting technique with hybrid object/image-order data traversal based on the shear-warp factorization of the viewing matrix [124, 91, 52]. The volume data is defined in object coordinates (u,v,w), which are first transformed to isotropic object coordinates by a scale and shear matrix L. This allows to automatically handle anisotropic data sets, in which the spacing between voxels differs in the three dimensions, and gantry tilted data sets, in which the slices are sheared, by adjusting the warp matrix. A permutation matrix P transforms the isotropic object to permuted coordinates (x,y,z). The origin of permuted coordinates is the vertex of the volume nearest to the image plane and the z axis is the edge of the volume most parallel to the view direction. A shear matrix S represents the rendering operation that projects points in the permuted volume space onto points on the base plane, which is the face of the volume data that is most parallel to the viewing plane.

In the shear-warp implementation by Lacroute and Levoy [52], the volume is stored three times, run-length encoded along the major viewing direction. The projection is performed using bi-linear interpolation and back-to-front compositing of volume slices parallel to the base plane. Pfister et al. [83] perform the projection using ray-casting. This prevents view-dependent artifacts when switching base planes and accommodates supersampling of the volume data. Instead of casting rays from image space, rays are sent into the data set from the base plane. This approach guarantees that there is a one-to-one mapping of sample points to voxels [124, 91].

The base plane image is transformed to the image plane using the warp matrix $W = M \times L^{-1} \times P^{-1} \times S^{-1}$. To resample the image, one can use 2D texture mapping with bi-linear interpolation on a companion graphics card. The additional 2D image resampling results in a slight degradation of image quality. It enables, however, an easy mapping to an arbitrary user-specified image size.

The main advantage of the shear-warp factorization is that voxels can be read and processed in planes of voxels, called slices, that are parallel to the base plane. Slices are processed in positive z direction. Within a slice, scanline of voxels (called voxel beams) are read from memory in top to bottom order. This leads to regular, object-order data access. In addition, it allows parallelism by having multiple rendering pipelines work on several voxels in a beam at the same time.

4.4. Progressive Refinement

One practical solution to the rendering time problem is the generation of partial images that are progressively refined as the user interacts with the crude image. Both forward and backward approach can support progressive refinement. In the case of forward viewing this technique is based on a pyramid data structure. First, the smaller volume in the pyramid is rendered using large-footprint splats. Later, higher resolution components of the pyramid are rendered [53].

Providing progressive refinement in backward viewing is achieved by first sampling the screen in low resolution. The regions in the screen where no rays were emitted from receive a value interpolated from some close pixels that were assigned rays. Later more rays are cast and the interpolated value is replaced by the more accurate result [60]. Additionally, rays that are intended to cover large screen areas can be traced in the lower-resolution components of a pyramid [57].

Not only screen-space resolution can be progressively increased. Sampling rate and stopping criteria can also be refined. An efficient implementation of this technique was reported by Danskin and Hanrahan [20].

5. The four most popular Approaches

As we have seen in the previous sections, there are numerous approachs that can be taken in volume visualization. A side by side comparison of all these approaches would cover many pages and would probably not give many insights due to the overwhelming amount of information and the large parameter set. Generally, there are two avenues that can be taken:

1. The volumetric data are first converted into a set of polygonal iso-surfaces (i.e., via Marching Cubes [63]) and subsequently rendered with polygon rendering hardware. This is referred to as indirect volume rendering (IVR).
2. The volumetric data are directly rendered without the intermediate conversion step. This is referred to as direct volume rending (DVR) [20, 88, 100].

The former assumes (i) that a set of extractable iso-surfaces exists, and (ii) that with the infinitely thin surface the polygon mesh models the true object structures at reasonable fidelity. Neither is always the case, as illustrative examples may serve: (i) amorphous cloud-like phenomena, (ii) smoothly varying flow fields, or (iii) structures of varying depth (and varying transparencies of an isosurface) that attenuate traversing light corresponding to the material thickness. But even if both of these assumptions are met, the complexity of the extracted polygonal mesh can overwhelm the capabilities of the polygon subsystem, and a direct volume rendering may prove more efficient [81], especially when the object is complex or large, or when the isosurface is interactively varied and the repeated polygon extraction overhead must be figured into the rendering cost [5].

Within this section, we concern ourselves solely with the direct volume rendering approach, in which four techniques have emerged as the most popular: Raycasting [99, 54], Splatting [108], Shear-warp [52], and 3D texture-mapping hardware-based approaches [9].

5.1. Introduction

Over the years, many researchers have worked independently on refining these four methods, and due to this multifarious effort, all methods have now reached a high level of maturity. Most of this development, however, has evolved along separate paths (although some fundamental scientific progress has benefited all methods such as advances in filter design [10, 6, 73] or efficient shading [103, 105]). A number of frequently used and publicly available datasets exists (e.g., the UNC CT / MRI heads or the CT lobster), however, due to the large number of parameters that were not controlled across presented research, it has so far been difficult to assess the benefits and shortcomings of each method in a decisive manner. The generally uncontrolled parameters include (apart from hardware architecture, available cache, and CPU clock speed): shading model, viewing geometry, scene illumination, transfer functions, image sizes, and magnification factors. Further, so far, no common set of evaluation criteria exists that enables fair comparisons of proposed methods with existing ones. Within this section, we will address this problem, and present an appropriate setup for benchmarking and evaluating different direct volume rendering algorithms. Some work in this direction has already been done in the past: Bartz [5] has compared DVR using raycasting with IVR using marching cubes for iso-surface extraction, while Tiede [97] has compared gradient filters for raycasting and marching cubes. However, a clear answer to which algorithm is best cannot be provided for the general case but the results presented here are aimed at providing certain guidelines to determine under what conditions and premises each volume rendering algorithm is most adequately chosen and applied.

5.2. Common Theoretical Framework

We can write all four investigated volume rendering methods as approximations of the well-known low-albedo volume rendering integral, VRI [8, 48, 41, 66]. The VRI analytically computes I l (x,r), the amount of light of wavelength l coming from ray direction r that is received at location x on the image plane:

$$I_\lambda(x,r) \ = \ \int_o^L C_\lambda(s)\mu(s)e^{(-\int_0^s \mu(t)dt)}ds \qquad (1)$$

Here, L is the length of ray r. If we think of the volume as being composed of particles with certain densities (or light extinction coefficients [66]) μ, then these particles receive light from all surrounding light sources and reflect this light towards the observer according to their specular and diffuse material properties. In addition, the particles may also emit light on their own. Thus, in (1), C_λ is the light of wavelength l reflected and/or emitted at location s in the direction of r. To account for the higher reflectance of particles with larger densities, we must weigh the reflected color by the particle density. The light scattered at s is then attenuated by the densities of the particles between s and the eye according to the exponential attenuation function.

At least in the general case, the VRI cannot be computed analytically [66]. Hence, practical volume rendering algorithms discretize the VRI into a series of sequential intervals i of width Δs:

$$I_\lambda(x,r) \ = \ \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\mu(s_i)\Delta s \prod_{j=0}^{i-1} e^{-\mu(s_j)\Delta s)} \qquad (2)$$

Using a Taylor series approximation of the exponential term and dropping all but the first two terms, we get the familiar compositing equation [57]:

$$I_\lambda(x,r) \ = \ \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\alpha(s_i) \prod_{j=0}^{i-1}(1-\alpha(s_j)) \qquad (3)$$

We denote this expression as discretized VRI (DVRI), where $\alpha = 1.0 - transparency$. Expression 3 represents a common theoretical framework for all surveyed volume rendering algorithms. All algorithms obtain colors and opacities in discrete intervals along a linear path and composite them in front to back order or back to front order, see section 2.8. However, the algorithms can be distinguished by the process in which the colors $C(s_i)$ and opacities $\alpha(s_i)$ are calculated in each interval i, and how wide the interval width Δs is chosen. The position of the shading operator in the volume rendering pipeline also affects C(s_i) and a(s_i). For this purpose, we distinguish the pre-shaded from the post-shaded volume rendering pipeline. In the pre-shaded pipeline, the grid samples are classified and shaded before the ray sample interpolation takes place. We denote this as Pre-DVRI (pre-shaded DVRI) and its mathematical expression is identical to formula 3. Pre-DVRI generally leads to blurry images, especially in zoomed viewing, where fine object detail is often lost [39, 77].

The blurriness is eliminated by switching the order of classification/shading and ray sample interpolation. Then, the original density volume f is interpolated and the resulting sample values $f(i)$ are classified, via transfer functions, to yield material, opacity, and color. All blurry parts of the edge image can be clipped away using the appropriate classification function [77]. Shading follows immediately after classification and requires the computation of gradients from the density grid. The resulting expression is termed Post-DVRI (post-shaded DVRI) and is written as follows:

$$I_\lambda(x,r) \ = \ \sum_{i=0}^{L/\Delta s} C_\lambda(f(s_i))\alpha(f(s_i)) \prod_{j=0}^{i-1}(1-\alpha(f(s_j))) \qquad (4)$$

C and α are now transfer functions, commonly implemented as lookup-tables. Since in Post-DVRI the raw volume densities are interpolated and used to index the transfer functions for color and opacity, fine detail in these transfer functions is readily expressed in the final image. One should note, however, that Post-DVRI is not without problems: Due to the partial volume effect, a density might be interpolated that is classified as a material not really present at the sample location, which can lead to false colors in the final image. This can be avoided by prior segmentation, which, however, can add severe staircasing artifacts due to introduced high-frequency. Based on formulas 3 and 4, we will now present the four surveyed algorithms in detail.

5.3. Distinguishing Features of the different algorithms

Our comparison will focus on the conceptual differences between the algorithms, and not so much on ingenious measures that speed runtime. Since numerous implementations for each algorithm exist – mainly providing acceleration – we will select the most general implementation for each, employing the most popular components and parameter settings. More specific implementations can then use the benchmarks introduced later to compare the impact of their improvements. We have summarized the conceptual differences of the four algorithms in Table 1.

5.3.1. Raycasting

Of all volume rendering algorithms, Raycasting has seen the largest body of publications over the years. Researchers have used Pre-DVRI [57, 54] as well as Post-DVRI [2, 38, 97]. The density and gradient (Post-DVRI), or color and opacity (Pre-DVRI), in each DVRI interval are generated via point sampling, most commonly by means of a trilinear filter from neighboring voxels (grid points) to maintain computational efficiency, and subsequently composited. Most authors space the ray samples apart in equal distances Δs, but some approaches exist that jitter the sampling positions to eliminate patterned sampling artifacts, or apply space-leaping [20, 127] for accelerated traversal of empty regions. For strict iso-surface rendering, recent research analytically computes the location of the iso-surface, when the ray steps into a voxel that is traversed by one [81]. But in the general case, the Nyquist theorem needs to be followed which states that we should choose $\Delta s < 1.0$ (i.e., one voxel length) if we do not know anything about the frequency content in the sample's local neighborhood. Then, for Pre-DVRI and Post-DVRI raycasting, the $C(s_i), \alpha(s_i), and f(s_i)$ terms in equations 3 and 4, respectively, are written as:

$$\begin{aligned} C_\lambda(s_i) &= C_\lambda(i\Delta s) \\ \alpha(s_i) &= \alpha(i\Delta s) \\ f(s_i) &= f(i\Delta s) \end{aligned} \qquad (5)$$

Note that a needs to be normalized for $\Delta s \neq 1.0$ [62]. In the used implementation, we use early ray termination, which is a powerful acceleration method of raycasting where rays can be terminated when the accumulated opacity has reached a value close to unity. Furthermore, all samples and corresponding gradient components are computed by trilinear interpolation of the respective grid data.

5.3.2. Splatting

Splatting was proposed by Westover [108], and it works by representing the volume as an array of overlapping basis functions, commonly Gaussian kernels with amplitudes scaled by the voxel values. An image is then generated by projecting these basis functions to the screen. The screen projection of these radially symmetric basis function can be efficiently achieved by the rasterization of a precomputed footprint lookup table. Here, each footprint table entry stores the analytically integrated kernel function along a traversing ray. A major advantage of splatting is that only voxels relevant to the image must be projected and rasterized. This can tremendously reduce the volume data that needs to be both processed and stored [78]. However, depending on the zooming factor, each splat can cover up to hundreds of pixels which need to be processed.

The preferred splatting approach [108] summed the voxel kernels within volume slices most parallel to the image plane. This was prone to severe brightness variations in animated viewing and also did not allow the variation of the DVRI interval distance Δs. Mueller [76] eliminated these drawbacks by processing the voxel kernels within slabs of width Δs, aligned parallel to the image plane – hence the approach was termed image-aligned splatting: All voxel kernels that overlap a slab are clipped to the slab and summed into a sheet buffer, followed by compositing the sheet with the sheet before. Efficient kernel slice projection is achieved by analytical pre-integration of an array of kernel slices and using fast slice footprint rasterization methods [78]. Both Pre-DVRI and Post-DVRI [77] are possible, and the $C(s_i), \alpha(s_i), and f(s_i)$ terms in equations 3 and 4 are now written as:

$$\begin{aligned} C_\lambda(s_i) &= \frac{\int_{i\Delta s}^{(i+1)\Delta s} C_\lambda(s)ds}{\Delta s} \\ \alpha(s_i) &= \frac{\int_{i\Delta s}^{(i+1)\Delta s} \alpha(s)ds}{\Delta s} \\ f(s_i) &= \frac{\int_{i\Delta s}^{(i+1)\Delta s} f(s)ds}{\Delta s} \end{aligned} \qquad (6)$$

We observe that splatting replaces the point sample of raycasting by a sample average across Δs. This introduces an additional low-pass filtering step that helps to reduce aliasing, especially in isosurface renderings and when $\Delta s > 1.0$. Splatting also typically uses rotationally symmetric Gaussian kernels, which have better anti-aliasing characteristics than linear filters, with the side effect of performing some signal smoothing. However, when splatting data values and not color, classification is an unsolved problem since the original data value, i.e. density, is smoothed, accumulated, and then classified. This aggravates the so-called partial volume effect and solving this remains future research.

Splatting can use a concept similar to early ray termination: early splat elimination, based on a dynamically computed screen occlusion map, that (conservatively) culls invisible splats early from the rendering pipeline [78]. The main operations of splatting are the transformation of each relevant voxel center into screen space, followed by an index into the occlusion map to test for visibility, and in case it is visible, the rasterization of the voxel footprint into the sheet-buffer. The dynamic construction of the occlusion map requires a convolution operation after each sheet-buffer composite, which, however, can be limited to buffer tiles that have received splat contributions in the current slab [78]. It should be noted that, although early splat elimination saves the cost of footprint rasterization for invisible voxels, their transformation must still be performed to determine their occlusion. This is different from early ray termination where the ray can be stopped and subsequent voxels are not processed.

5.3.3. Shear-Warp

Shear-warp was proposed by Lacroute and Levoy [52] and has been recognized as the fastest software renderer to date. It achieves this by employing a clever volume and image encoding scheme, coupled with a simultaneous traversal of volume and image that skips opaque image regions and transparent voxels. In a pre-processing step, voxel runs are RLE-encoded based on pre-classified opacities. This requires the construction of a separate encoded volume for each of the three major viewing directions. The rendering is performed using a raycasting-like scheme, which is simplified by shearing the appropriate encoded volume such that the rays are perpendicular to the volume slices. The rays obtain their sample values via bilinear interpolation within the traversed volume slices. A final warping step transforms the volume-parallel baseplane image into the screen image. The DVRI interval distance Δ s is view-dependent, since the interpolation of sample values only occurs in sheared volume slices. It varies from 1.0 for axis-aligned views to 1.41 for edge-on views to 1.73 to corner-on views, and it cannot be varied to allow for supersampling along the ray. Thus the Nyquist theorem is potentially violated for all but the axis-aligned views.

The Volpack distribution from Stanford (a volume render-

Sampling rate	freely selectable	freely selectable	fixed $[1.0, 1.73]$	freely selectable
Interpolation kernel	trilinear	Gaussian	bilinear	trilinear
Acceleration	early ray termination	early splat elimination	RLE opacity encoding	graphics hardware
Voxels considered	all	relevant	mostly relevant	all

Table 1: *Distinguishing features and commonly used parameters of the four different algorithms.*

ing package that uses the shear warp algorithm) only provides for Pre-DVRI (with opacity weighted colors), but conceptually Post-DVRI is also feasible, however, without opacity classification if shear-warp's fast opacity-based encoding is used. The $C(s_i)$, $\alpha(s_i)$, and $f(s_i)$ terms in equations 3 and 4 are written similar to raycasting, but with the added constraint that Δs is dependent on the view direction:

$$\begin{aligned} C_\lambda(s_i) &= C_\lambda(i\Delta s) \\ \alpha(s_i) &= \alpha(i\Delta s) \\ f(s_i) &= f(i\Delta s) \end{aligned} \tag{7}$$

$$\Delta s = \sqrt{\left(\frac{dx}{dz}\right)^2 + \left(\frac{dy}{dz}\right)^2 + 1} \tag{8}$$

where $[dx, dy, dz]^T$ is the normalized viewing vector, reordered such that dz is the major viewing direction. In Volpack, the number of rays sent through the volume is limited to the number of pixels in the base plane (i.e., the resolution of the volume slices in view direction). Larger viewports are achieved by bilinear interpolation of the resulting image (after back-warping of the base plane), resulting in a very low image quality if the size of the view-port is significantly larger than the volume resolution. This can be fixed by using a scaled volume with a higher volume resolution.

5.3.4. 3D Texture-Mapping Hardware

This is a short introduction to 3D texture mapping and more details are disclosed in section 7.// The use of 3D texture mapping was popularized by Cabral [9] for non-shaded volume rendering. The volume is loaded into texture memory and the hardware rasterizes polygonal slices parallel to the viewplane. The slices are then blended back to front, due to the missing accumulation buffer for a. The interpolation filter is a trilinear function (on SGI's RE 2 and IR architectures, quadlinear interpolation is also available, where it additionally interpolates between two mipmap levels), and the slice distance Δ s can be chosen freely. A number of researchers have added shading capabilities [19, 70, 26, 106], and both Pre-DVRI [26] and Post-DVRI [19, 70, 106] are possible. Usually, the rendering is brute-force, without any opacity-based termination acceleration, but some researchers have done this [19]. The drawbacks of 3D texture mapping is that larger volumes require the swapping of volume bricks in and out of the limited-sized texture memory (usually a few MBytes for smaller machines). Fortunately, 3D texture mapping recently became popular in PC based graphics hardware. Texture-mapping hardware interpolates samples in similar ways to raycasting and hence the $C(s_i)$, $\alpha(s_i)$, and $f(s_i)$ terms in equations 3 and 4 are written as:

$$\begin{aligned} C_\lambda(s_i) &= C_\lambda(i\Delta s) \\ \alpha(s_i) &= \alpha(i\Delta s) \\ f(s_i) &= f(i\Delta s) \end{aligned} \tag{9}$$

5.4. Comparison

For a fair comparison of the presented four algorithms, one needs to define identical viewing and rendering parameters. The first one can be accomplished using a common camera model, i.e. like OpenGL while the latter includes multiple parameters such as filtering method, classification, shading, blending, and so forth. Some of these properties are illustrated in table 1. Others such as shading, classification, and blending need to be adopted accross the algorithms such that all use the same operations.

It is difficult to evaluate rendering quality in a quantitative manner. Often, researchers simply put images of competing algorithms side by side, appointing the human visual system (HVS) to be the judge. It is well known that the HVS is less sensitive to some errors (stochastic noise) and more to others (regular patterns), and interestingly, sometimes images with larger numerical errors, e.g., RMS, are judged similar by a human observer than images with lower numerical errors. So it seems that the visual comparison is more appropriate than the numerical, since after all we produce images for the human observer and not for error functions. In that respect, an error model that involves the HVS characteristics would be more appropriate than a purely numerical one. But nevertheless, to perform such a comparison we still need the true volume rendered image, obtained by analytically integrating the volume via equation (1) (neglecting the prior reduction of the volume rendering task to the low-albedo case). As was pointed out by Max [66], analytical integration can be done when assuming that $C(s)$ and $\mu(s)$ are piecewise linear. This is, however, somewhat restrictive on our transfer functions, so in the following we employ visual quality assessment only.

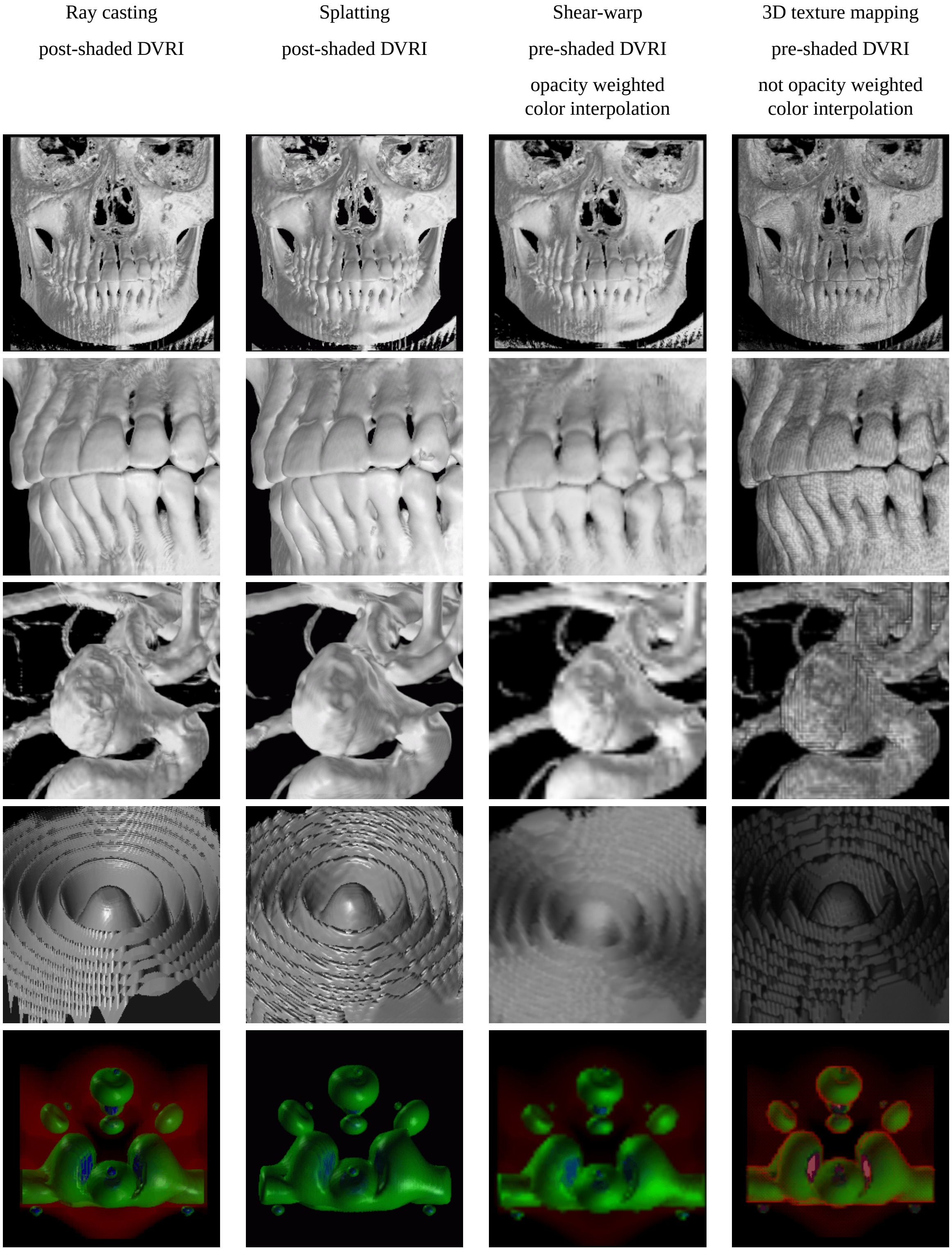


Figure 7: *Comparison of the four algorithms. Columns from left to right: Ray casting, splatting, shear-warp, and 3D texture mapping. Rows from top to bottom: CT scan of a human skull, zoomed view of the teeth, CT scan of brain arteries showing an aneurism, Marschner-Lobb function containing high frequencies, and simulation of the potential distribution of electrons around atoms.*

5.5. Results

All presented results were generated on the same platform (SGI Octane). The graphics hardware was only used by the 3D texture mapping approach. Figure 7 shows representative still frames of different datasets that we rendered. We observe that the image quality achieved with texture mapping hardware shows severe color-bleeding artifacts due to interpolation of colors independent from the α-value [119] (see section 2.10, as well as staircasing. Furthermore, highly transparent classification results in darker images, due to limited precision of the RGBA-channels of the hardware (8 bit).

Volpack shear-warp performs much better, with quality similar to raycasting and splatting whenever the resolution of the image matches the resolution of the baseplane. For the other images, the rendered baseplane image was of lower resolution than the screen image and had to be magnified using bilinear interpolation in the warping step. This leads to excessive blurring, especially for the Marschner-Lobb dataset, where the magnification is very high. A more fundamental draw-back can be observed in the 45 degree neghip view in Figure 7, where – in addition to the blurring – significant aliasing in the form of staircasing is present. This is due to the ray sampling rate being less than 1.0, and can be disturbing in animated viewing of some datasets but is less noticeable in still images. The Marschner-Lobb dataset renderings for raycasting and splatting demonstrate the differences of point sampling (raycasting) and sample averaging (splatting). While raycasting's point sampling misses some detail of the function at the crests of the sinusoidal waves, splatting averages across the waves and renders them as blobby rims. For the other datasets the averaging effect is more subtle, but still visible. For example, raycasting renders the skull and the magnified blood with crisper detail than splatting does, but can suffer from aliasing artifacts, if the sampling rate is not chosen appropriately (Nyquist rate). However, the quality is quite comparable, for all practical purposes.

5.6. Summary

Generally, 3D texture mapping and shear-warp have subsecond rendering times for moderately-sized datasets. While the quality of the display obtained with our mainstream texture mapping approach is limited and can be improved as demonstrated in section 7, the quality of shear-warp rivals that of the much more expensive raycasting and splatting when the object magnification is about unity. Handling higher magnifications is possible by relaxing the condition that the number of rays must match the resolution of the volume. Although higher interpolation costs will be the result, the rendering frame rate will most likely still be high (especially if view frustum culling is applied). A more serious concern is the degradation of image quality at off-axis views. In these cases, one could use a volume with extra interpolated slices, which is Volpack's standard solution for higher image resolutions. But the fact that shear-warp requires an opacity-encoded volume makes interactive transfer function variation a challenge. In applications where these limitations do not apply, shear-warp proves to be a very useful algorithm for volume rendering. The side-by-side comparison of splatting and raycasting yielded interesting results as well: We saw that image-aligned splatting offers a rendering quality similar to that of raycasting. It, however, produces smoother images due to the z-averaged kernel and the anti-aliasing effect of the larger Gaussian filter. It is hence less likely to miss high-frequency detail. Raycasting is faster than splatting for datasets with a low number of non-contributing samples. On the other hand, splatting is better for datasets with a small number of relevant voxels and sheetbuffers. Since the quality is so similar and the same transfer functions yield similar rendering results, one could build a renderer that applies either raycasting or splatting, depending on the number of relevant voxels and the level of compactness of the dataset. One could even use different renderers in different portions of the volume, or for the rendering of disconnected objects of different compactness.

More details on this comparison can be found in [71].

6. The VolumePro Real-Time Ray-Casting System

Software based volume rendering approaches can be accelerated such that interactive frame-rates can be achieved. However, this requires certain trade-offs in quality or parameters that can be changed interactively. In order to achieve interactive or even real-time frame-rates at highest quality and full flexibility, dedicated hardware is necessary. Within this section we will have a closer look at special purpose hardware, i.e. the VolumePro system.

6.1. Introduction

Special purpose hardware for volume rendering has been proposed by various researchers, but only a few machines have been implemented. VIRIM was built at the University of Mannheim, Germany [31]. The hardware consists of four VME boards and implements ray-casting. VIRIM achieves 2.5 frames/sec for 256^3 volumes.

The first PCI based volume rendering accelerator has been built by the University of Tübingen, Germany. Their VIZARD system implements true perspective ray-casting and consists of two PCI accelerator cards [47]. An FPGA-based system achieves up to 10 frames/sec for 256^3 volumes. To circumvent the lossy data compression and the limitations of changes in classification or shading parameters due to the pre-processing, a follow-up system is currently under development [72, 21]. This VIZARD II system will be capable of up to 20 frames per second for datasets of 256^3 voxels. The strength of this system is its ray traversal engine with an optimized memory interface that allows for fly throughs which is mandatory for immersive applications. The system uses a single ray processing pipeline (RPU) and exploits algorithmic optimizations such as early ray termination and space leaping.

Within this section, we will describe VolumePro, the first single-chip real-time volume rendering system for consumer PCs [83]. The first VolumePro board was operational in April 1999 (see Figure 8).

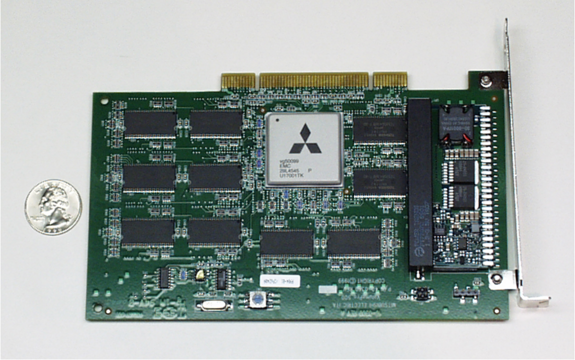


Figure 8: The VolumePro PCI card.

The VolumePro system is based on the Cube-4 volume rendering architecture developed at SUNY Stony Brook [84]. Mitsubishi Electric licensed the Cube-4 technology and developed the Enhanced Memory Cube-4 (EM-Cube) architecture [80]. The VolumePro system, an improved commercial version of EM-Cube, is commercially available since May 1999 at a price comparable to high-end PC graphics cards. Figure 9 shows several images rendered on the VolumePro hardware at 30 frames/sec.

6.2. Rendering Algorithm

VolumePro implements ray-casting [54], one of the most commonly used volume rendering algorithms. Ray-casting offers high image quality and is easy to parallelize. The current version of VolumePro supports parallel projections of isotropic and anisotropic rectilinear volumes with scalar voxels.

VolumePro is a highly parallel architecture based on the hybrid ray-casting algorithm shown in Figure 10 [124, 91, 52]. Rays are sent into the dataset from each pixel on a base plane, which is co-planar to the face of the volume data that is most parallel and nearest to the image plane. Because the image plane is typically at some angle to the base-plane, the resulting base-plane image is warped onto the image plane.

The main advantage of this algorithm is that voxels can be read and processed in planes of voxels (so called slices) that are parallel to the base-plane. Within a slice, voxels are read from memory a scanline of voxels at a time, in top to bottom order. This leads to regular, object-order data access.

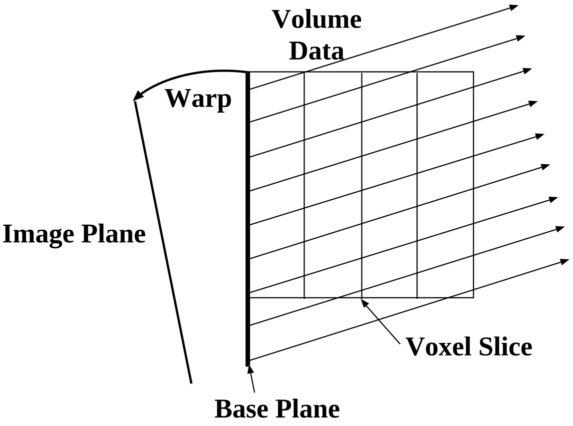


Figure 10: Template-based ray-casting.

In contrast to the shear-warp implementation by Lacroute and Levoy [52], VolumePro performs tri-linear interpolation and allows rays to start at sub-pixel locations. This prevents view-dependent artifacts when switching base planes and accommodates supersampling of the volume data.

6.3. VolumePro System Architecture

VolumePro is implemented as a PCI card for PC class computers. The card contains one volume rendering ASIC (called the *vg500*) and 128 or 256 MBytes of volume memory. The warping and display of the final image is done on an off-the-shelf 3D graphics card with 2D texture mapping. The vg500 volume rendering ASIC, shown in Figure 11, contains four identical rendering pipelines, arranged side by side, running at 125 MHz each. It is an application specific integrated circuit (ASIC) with approximately 3.2 million random logic transistors and 2 Mbits of on-chip SRAM. The vg500 also contains interfaces to voxel memory, pixel memory, and the PCI bus.

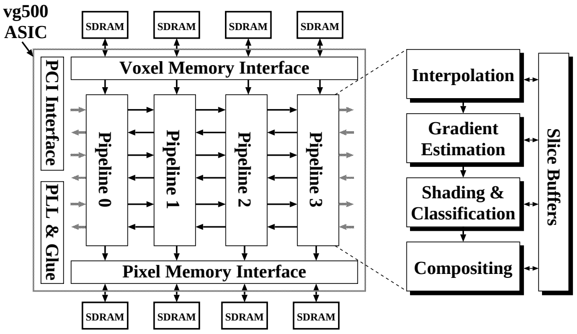


Figure 11: The vg500 volume rendering ASIC with four identical ray-casting pipelines.

Each pipeline communicates with voxel and pixel mem-

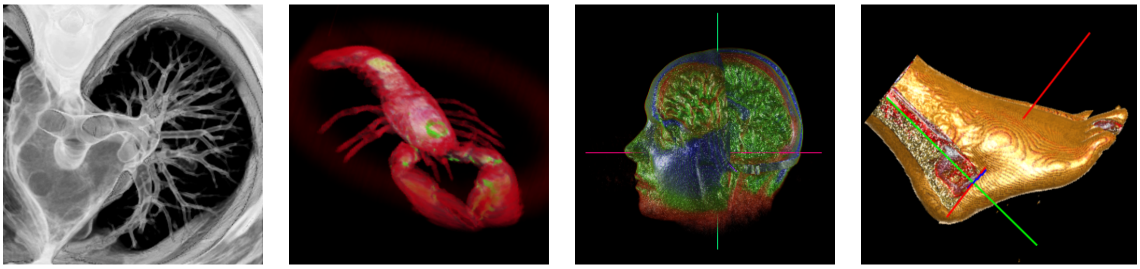


Figure 9: Several volumes rendered on the VolumePro hardware at 30 frames per second.

ory and two neighboring pipelines. Pipelines on the far left and right are connected to each other in a wrap-around fashion (indicated by grey arrows in Figure 11). A main characteristic of VolumePro is that each voxel is read from volume memory exactly once per frame. Voxels and intermediate results are cached in so called slice buffers so that they become available for calculations precisely when needed.

Each rendering pipeline implements ray-casting and sample values along rays are calculated using tri-linear interpolation. A 3D gradient is computed using central differences between tri-linear samples. The gradient is used in the shader stage, which computes the sample intensity according to the Phong illumination model. Lookup tables in the classification stage assign color and opacity to each sample point. Finally, the illuminated samples are accumulated into base plane pixels using front-to-back compositing.

Volume memory uses 16-bit wide synchronous DRAMs (SDRAMs) for up to 256 MBytes of volume storage. $2 \times 2 \times 2$ cells of neighboring voxels, so called miniblocks, are stored linearly in volume memory. Miniblocks are read and written in bursts of eight voxels using the fast burst mode of SDRAMs. In addition, VolumePro uses a linear skewing of miniblocks [45]. Skewing guarantees that the rendering pipelines always have access to four adjacent miniblocks in any of the three slice orientations. A miniblock with position $[xyz]$ in the volume is assigned to the memory module k as follows:

$$k = \left(\left\lfloor \frac{x}{2} \right\rfloor + \left\lfloor \frac{y}{2} \right\rfloor + \left\lfloor \frac{z}{2} \right\rfloor\right) \bmod 4. \tag{10}$$

6.4. VolumePro PCI Card

The VolumePro board is a PCI Short Card with a 32-bit 66 MHz PCI interface (see Figure 8). The board contains a single vg500 rendering ASIC, twenty 64 Mbit SDRAMs with 16-bit datapaths, clock generation logic, and a voltage converter to make it 3.3 volt or 5 volt compliant. Figure 12 shows a block diagram of the components on the board and the busses connecting them.

The vg500 ASIC interfaces directly to the system PCI-Bus. Access to the vg500's internal registers and to the off-chip memories is accomplished through the 32-bit 66 MHz PCI bus interface. The peak burst data rate of this interface is 264 MB/sec. Some of this bandwidth is consumed by image upload, some of it by other PCI system traffic.

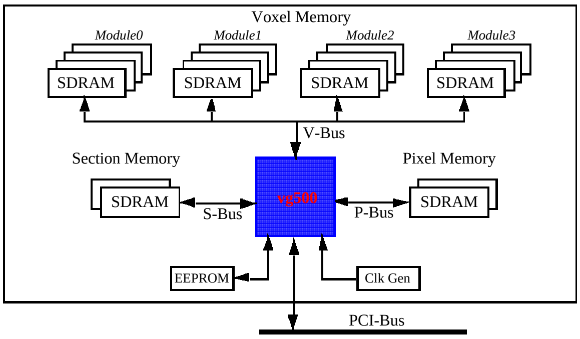


Figure 12: VolumePro PCI board diagram.

6.5. Supersampling

Supersampling [32] improves the quality of the rendered image by sampling the volume data set at a higher frequency than the voxel spacing. In the case of supersampling in the x and y directions, this would result in more samples per beam and more beams per slice, respectively. In the z direction, it results in more sample slices per volume.

VolumePro supports supersampling in hardware only in the z direction. Additional slices of samples are interpolated between existing slices of voxels. The software automatically corrects the opacity according to the viewing angle and sample spacing by reloading the opacity table.

Figure 13 shows the CT scan of a foot (152 x 261 x 200) rendered with no supersampling (left) and supersampling in z by 3 (right). The artifacts in the left image stem from the insufficient sampling rate to capture the high frequencies of the foot surface. Notice the reduced artifacts in the supersampled image. VolumePro supports up to eight times supersampling.

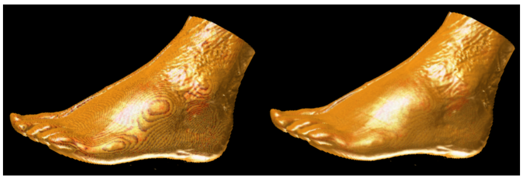


Figure 13: No supersampling (left) and supersampling in z (right).

6.6. Supervolumes and Subvolumes

Volumes of arbitrary dimensions can be stored in voxel memory without padding. Because of limited on-chip buffers, however, the VolumePro hardware can only render volumes with a maximum of 256 voxels in each dimension in one pass. In order to render a larger volume (called a supervolume), software must first partition the volume into smaller blocks. Each block is rendered independently, and their resulting images are combined in software.

The VolumePro software automatically partitions supervolumes, takes care of the data duplication between blocks, and blends intermediate base planes into the final image. Blocks are automatically swapped to and from host memory if a supervolume does not fit into the 128 MB of volume memory on the VolumePro PCI card. There is no limit to the size of a supervolume, although, of course, rendering time increases due to the limited PCI download bandwidth.

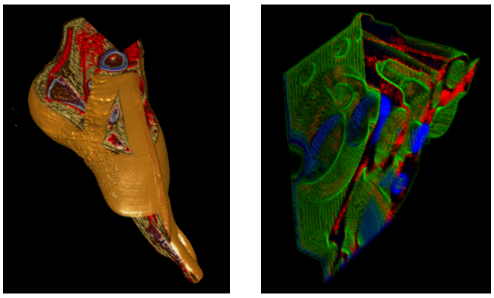
Volumes with less than 256 voxels in each dimension are called subvolumes. VolumePro's memory controller allows reading and writing single voxels, slices, or any rectangular slab to and from Voxel Memory. Multiple subvolumes can be pre-loaded into volume memory. Subvolumes can be updated in-between frames. This allows dynamic and partial updates of volume data to achieve 4D animation effects. It also enables loading sections of a larger volume in pieces, allowing the user to effectively pan through a volume. Subvolumes increase rendering speed to the point where the frame rate is limited by the base plane pixel transfer and driver overhead, which is currently at 30 frames/sec.

6.7. Cropping and Cut Planes

VolumePro provides two features for clipping the volume data set called cropping and cut planes. These make it possible to visualize slices, cross-sections, or other portions of the volume, thus providing the user an opportunity to see inside in creative ways. Figure 14(a) shows an example of cropping on the CT foot of the visible man. Figure 14(b) shows a cut plane through the engine data.

6.8. VolumePro Performance

Each of the four SDRAMs provides burst-mode access at up to 125 MHz, for a sustained memory bandwidth of $4 \times 125 \times 10^6 = 500$ million 16-bit voxels per second. Each rendering pipeline operates at 125 MHz and can accept a new voxel from its SDRAM memory every cycle. 500 million tri-linear samples per second is sufficient to render 256^3 volumes at 30 frames per second.



(a) (b)

Figure 14: (a) Cropping. (b) Cut plane.

6.9. VLI - The Volume Library Interface

Figure 15 shows the software infrastructure of the VolumePro system. The VLI API is a set of C++ classes that provide full access to the vg500 chip features. VLI does not replace an existing graphics API. Rather, VLI works cooperatively with a 3D graphics library, such as OpenGL, to manage the rendering of volumes and displaying the results in a 3D scene. Higher level toolkits (such as vtk – The Visualization Toolkit) and scene graphs on top of the VLI will likely become the primary interface layer to applications. The VLI classes can be grouped as follows:

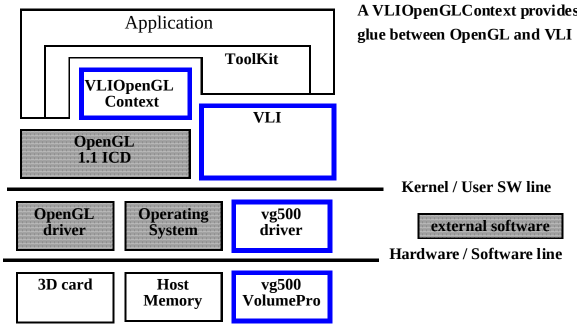


Figure 15: Software infrastructure of the VolumePro system.

- Volume data handling. VLIVolume manages voxel data storage, voxel data format, and transformations of the volume data such as shearing, scaling, and positioning in world space.
- Rendering elements. There are several VLI classes that provide access to the VolumePro features, such as color

and opacity lookup tables, cameras, lights, cut planes, clipping, and more.

- Rendering context. The VLI class VLIContext is a container object for all attributes needed to render the volume. It is used to specify the volume data set and all rendering parameters (such as classification, illumination, and blending) for the current frame.

The VLI automatically computes reflectance maps based on light placement, sets up α-correction based on viewing angle and sample spacing, supports anisotropic and gantry-tilted data sets by correcting the viewing and image warp matrices, and manages supervolumes, supersampling, and partial updates of volume data. In addition, there are VLI functions that provide initialization, configuration, and termination for the VolumePro hardware.

6.10. Summary

This section describes the algorithm, architecture, and features of VolumePro, the world's first single-chip real-time volume rendering system. The rendering capabilities of VolumePro – 500 million tri-linear, Phong illuminated, composited samples per second – sets a new standard for volume rendering on consumer PCs. Its core features, such as on-the-fly gradient estimation, per-sample Phong illumination with arbitrary number of light sources, 4K RGBA classification tables, α-blending with 12-bit precision, and gradient magnitude modulation, put it ahead of any other hardware solution for volume rendering. Additional features, such as supersampling, supervolumes, cropping and cut planes, enable the development of feature-rich, high-performance volume visualization applications.

Some important limitations of VolumePro are the restriction to rectilinear scalar volumes, the lack of perspective projections, and no support for intermixing of polygons and volume data. Mixing of opaque polygons and volume data can be achieved by first rendering geometry, transferring z buffer values from the polygon card to the volume renderer, and then rendering the volume starting from these z values. Future versions of the system will support perspective projections and several voxel formats, including pre-classified material volumes and RGBA volumes. The limitation to rectilinear grids is more fundamental and hard to overcome.

7. 3D Texture Mapping

So far, we have seen different volume rendering techniques and algorithmic optimizations that can be exploited to achieve interactive frame-rates. Real-time frame-rates can be accomplished by special purpose hardware such as presented in the previous section (VolumePro). Another avenue that can be taken is based on 2D and 3D texture mapping hardware which currently migrates into the commodity PC graphics hardware and allows for mixing polygons and volumes.

7.1. Introduction

With fast 3D graphics hardware becoming more and more available even on low end platforms, the focus in developing new algorithms is beginning to shift towards higher quality rendering and additional functionality instead of simply higher performance implementations of the traditional graphics pipeline.

Graphics libraries like OpenGL and its extensions provide access to advanced graphics operations in the geometry and the rasterization stage and therefore allow for the design and implementation of completely new classes of rendering algorithms. Prominent examples can be found in realistic image synthesis (shading, bump/environment mapping, reflections) and scientific visualization applications (volume rendering, vector field visualization, data analysis).

In this respect, the goal of this session is twofold: To give both a state-of-the-art overview of volume rendering algorithms using the extended OpenGL graphics library and to present a number of selected and advanced volume rendering algorithms in which access to dedicated graphics hardware is paramount. The first part of this section summarizes the most important fundamentals and features of the graphics library OpenGL with respect to the practical and efficient design of volume rendering algorithms. The second part is dedicated to the efficient use of multi-texture register combiners as available on low-cost PCs in volume rendering applications. In each of both parts hardware accelerated graphics operations are used thus allowing interactive, high quality rendering and analysis of large-scale volume data sets.

7.2. Advanced volume rendering techniques

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.

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, 75], 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 graphics researchers are actively investigating possibilities to accelerate expensive visualization algorithms by using these extensions.

In this session we will summarize various approaches that make extensive use of graphics hardware for the rendering of volumetric data sets. In particular, the goal of this session is to provide participants with dedicated knowledge concerning the application of 3D textures in volume rendering applications and to demonstrate how to exploit the processing power and functionality of the rasterization and texture subsystem of advanced graphics hardware. Although at this time hardware accelerated 3D texture mapping is only supported on a few particular architectures we expect the same functionality to be available on low-end architectures like PCs in the near future thus leading to an increasing need for hardware accelerated algorithms as will be presented. Nonetheless, we will also demonstrate how to efficiently exploit existing PC graphics hardware on which only 2D textures are available in order to achieve high image quality at interactive frame rates.

Hereafter we will first describe the basic concepts of volume rendering via 3D textures thereby focusing on the potential benefits and advantages compared to software based solutions. We will further outline extensions that enable flexible and interactive editing and manipulation of large scale volume data. We will introduce the concept of clipping geometries by means of stencil buffer operations, and we will review the use of 3D textures for the rendering of lighted and shaded iso-surfaces in real-time without extracting any polygonal representation. Additionally, we will describe novel approaches for the rendering of scalar volume data using 2D textures and multi-texture register combiners as available on Nvidia's GeForce 256 PC graphics processor. The intention here is to streamline general directions how to bring high quality volume rendering to the consumer market by exploiting dedicated but affordable graphics hardware.

Our major concern in this session is to outline techniques for 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 a basis for hardware accelerated interactive visualization of both iso-surfaces and direct volume rendering has been provided in [106].

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 [41, 49, 68, 112]. A few standard algorithms exist for computing the intensity contribution along a ray of sight, enhanced by a wide variety of optimization strategies [57, 68, 53, 20, 52]. But only recently, since hardware supported 3D texture mapping is available, has direct volume rendering become interactively feasible on graphics workstations [9, 18, 115]. This approach has been extended further on with respect to flexible editing options and advanced mapping and rendering techniques.

The major goal is the manipulation and rendering of large-scale volumetric data sets at interactive rates within one application on standard graphics architectures. In this session 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. The presented approach is pixel oriented, takes 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 it is possible to

- **extend volume rendering via 3D textures** with respect to arbitrary clipping geometries
- **render shaded iso-surfaces** at interactive rates combining 3D textures and fragment operations thus avoiding any polygonal representation
- **extend volume rendering via 2D textures** with respect to view independent texture resampling and advanced shading and lighting models

7.3. Volume rendering via 3D textures

When 3D textures became available on graphics workstations their benefit in volume rendering applications was soon recognized [18, 9]. The basic idea is to interpret the 3D scalar 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. The data is re-sampled on clipping planes that are oriented orthogonal to the viewing plane with the plane pixels trilinearly interpolated from the 3D scalar texture. This operation is successively performed for multiple planes that have to be clipped against the parametric texture domain (see Figure 16). These polygons are rendered from front-to-back or back-to-front and the resulting texture slices are blended appropriately into the frame buffer thereby approximating the continuous volume rendering integral.

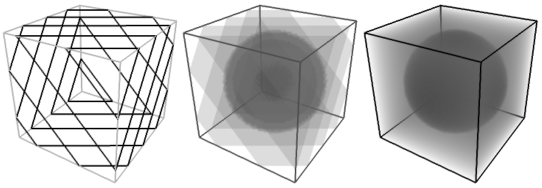


Figure 16: *Volume rendering by 3D texture slicing.*

Dedicated graphics hardware is exploited for trilinearly interpolating within the texture and for blending the generated fragments on a per-pixel basis. However, the real potential of volume rendering via 3D textures just turned out after texture lookup tables became available. Scalar samples that are reconstructed from the 3D texture are converted into RGAα pixels by a lookup-up table prior to their drawing. The possibility to directly manipulate the transfer functions necessary to perform the mapping from scalar values to RGBα values without the need for reloading the entire texture allows the user to interactively find meaningful mappings of material values to visual quantities. In this way arbitrary parts of the data can be highlighted or suppressed and visualized using different colors and transparencies.

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 can be modified in order to extract portions of the data, the use of additional clipping geometries often allows separating the relevant structures in a much more convenient and intuitive way. Planar clipping planes available as core OpenGL mechanisms may be utilized, but from the user's point of view more complex geometries are necessary.

7.4. Clipping geometries and Stenciling

A straightforward approach which is implemented quite often is the use of multiple clipping planes to construct more complex geometries. However, notice that even the simple task of clipping an arbitrarily scaled box cannot be realized in this way. More flexibility and ease of manipulation can be achieved by taking advantage of the per-pixel operations provided in the rasterization stage. As will be 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 for all slicing planes those pixels which are covered by the cross-section between the object and this plane (see Figure 17). Then, these pixels are locked, thus preventing the textured polygon from getting drawn to these locations. The locking mechanism is implemented by exploiting the OpenGL stencil 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 the clipping object is rendered in polygon mode. However, since one is only interested in setting the stencil buffer none of the frame buffer values altered. 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 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.

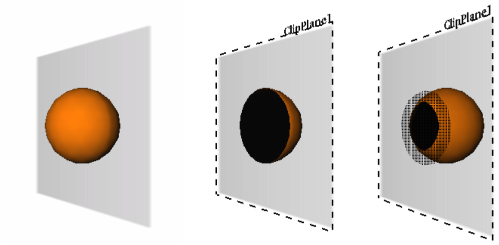
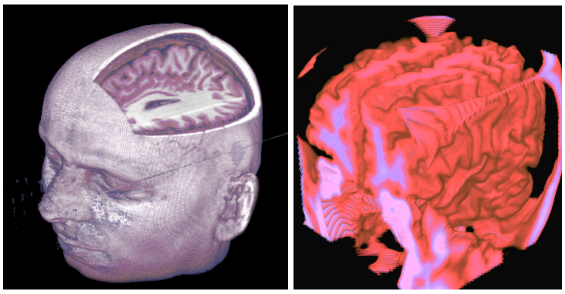


Figure 17: *The use of arbitrary clipping geometries is demonstrated for the case of a sphere. In regions where the object intersects the actual slice the stencil buffer is locked. The intuitive approach of rendering only the back faces might result in the patterned erroneous region.*

In general, however, depending on the clipping geometry this procedure fails in determining the cross-section exactly (see rightmost image in Figure 17). 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. Clearing the stencil buffer each time a new slice is to be rendered can be avoided by using different stencil planes. Then the number of slices that can be processed without clearing the buffer depends on the number of stencil bits provided by the current visual.

Since this approach is independent of the used geometry it allows arbitrary shapes to be specified. In particular it turns out that transformations of the geometry can be handled without any additional overhead, thus providing a flexible tool for carving portions out of the data in an intuitive way.

In Figure 18 two images are shown, which should demonstrate the extended functionality of 3D texture based volume rendering. In the first image a simple box was used to mask the interior of a MRI-scan by means of the stencil buffer approach. The second images was generated by explicitly clipping the slicing planes against the box and by tesselating the resulting contours. Note that only the region of interest needs to be textured in this way.



(a) Box clipping with the stencil buffer.

(b) Inverse box clipping with OGL tesselation.

Figure 18: *Box clipping using the stencil test (left) and the OGL tesselation (right).*

7.5. Rendering iso-surfaces via 3D textures

So far we described extensions to texture mapped direct volume rendering that have been introduced in order to define a general hardware accelerated framework for adaptive exploration of volumetric data sets. 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 [63, 74, 92, 111], but none of these approaches can effectively be used in interactive applications. This is due to the effort that has to be made to fit the surface and also to the enormous amount of triangles produced. For realistically sized data sets interactively manipulating the iso-value seems to be quite impossible, and also rendering the surface at acceptable frame rates can hardly be achieved. In contrast to these polygonal approaches, in [106] an algorithm was designed that completely avoids any polygonal representation by combining 3D texture mapping and advanced pixel transfer operations in a way that allows the iso-surface to be rendered on a per-pixel basis.

Recently, first approaches for combining hardware accelerated volume rendering via 3D texture maps with lighting and shading were presented. In [102] 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 [34] 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) [102], and the difficulty to achieve smoothly shaded surfaces due to the limited quantization of the normal orientation and the intrinsic hardware interpolation problems [34].

Basically, the non-polygonal 3D texture based 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 of sight 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 texture interpolation is already exploited to re-sample the data, all that needs to be evaluated is how to capture those texture samples above the iso-value that are nearest to the image plane. Therefore the OpenGL **alpha test** can be employed, 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 its alpha component. Then, texture mapped volume rendering is performed as usual, but pixel values are only drawn if they pass the depth 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 two different approaches should be outlined:

- **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 19). For the surface shading to proceed properly, pixel values have to be scaled and translated back to the range [-1,1]. In order to account for changes in the orientation of the object 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.

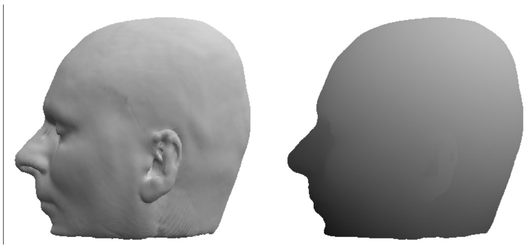


Figure 19: *On the left, for an iso-surface the gradient components are displayed in the RGB pixel values. On the right, for the same iso-surface the coordinates in texture space are displayed in the RGB components.*

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:

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

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

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

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

Figure 20 illustrates the quality of the described rendering technique for shaded iso-surfaces. The surface on the left image was rendered in roughly 9 seconds using a software based ray-caster. 3D texture based gradient shading was run with about 6 frames per second on the next image. The distance between successive slices was chosen to be equal to the sampling intervals used in the software approach. The surface on the right appears somewhat brighter with a little less contrast due to the limited frame buffer precision, but basically there can hardly be seen any differences.

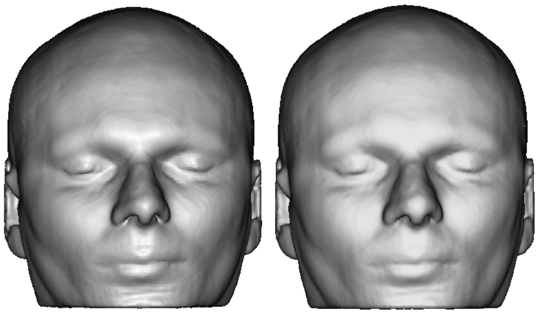


Figure 20: *Iso-surface rendering by direct ray-casting (left) and by using a gradient texture (right).*

To circumvent the additional amount of memory that is needed to store the gradient texture a second technique can be employed which applies concepts borrowed from [82] but in an essentially different scenario. The diffuse shading term can be simulated by simple frame buffer arithmetic if the surface is assumed to be 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:

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

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

In order to obtain the coordinates of the surface points it is taken advantage of the alpha test as proposed and pixel textures are applied 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.

By considering that to each vertex the computed texture coordinate (u,v,w) is assigned as RGB color value. Texture coordinates are supposed to be within the range [0,1] since they are computed in parametric texture space. 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 the same texture format is used as in traditional texture slicing. Each element comprises a single-valued color entry which is mapped via a RGBα lookup table. This allows one to temporarily set all RGB values in the lookup table to one thus avoiding any modulation of color values.

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. By changing the blending equation appropriately all values get subtracted from those already in the frame buffer thus yielding the approximated diffuse lighting.

In Figure 21 illustrates the difference 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 the presented approach.

However, only one fourth of the memory needed in gradient shading is used in gradientless shading but the rendering times, on the other hand, only differ insignificantly. The only difference lies in the way the shading is finally computed. In gradient shading the whole frame buffer is copied once. In gradientless shading the pixel data has to be read and written twice with enabled pixel texturing. On the other hand, since the overhead does not depend on the data resolution but on the size of the viewport, its relative contribution to the overall rendering time can be expected to decrease rapidly with increasing data size.

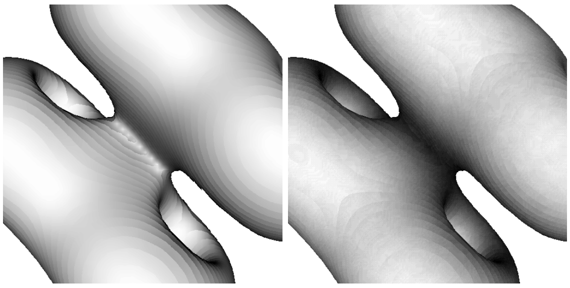


Figure 21: *Comparison of iso-surface rendering using a gradient texture (left) and frame buffer arithmetic (right).*

7.6. Volume rendering via 2D textures using register combiners

In order to exploit 2D textures for volume rendering, the volume data set is represented by three object-aligned texture stacks. The stack to be used for rendering has to be chosen according to the view direction in order to avoid degenerated projected polygons (see Figure 22). For a particular stack, the textured polygons are rendered in back-to-front order, and the generated fragments are composited with the pixels already in the frame buffer by α-blending. Due to varying view direction the distance between consecutive sample points that are blend into a certain pixel varies accordingly. Consequently, the opacity of slices should be adapted with respect to this distance in order to account for the thickness these samples represent. In addition, only bilinear interpolation within the original slices is performed thus leading to visually less pleasant results. Both drawbacks, however, can be avoided quite efficiently using multi-texture register combiners as available in the Nvidia GeForce 256 graphics processor.

Multi-texturing is an optional extension available in OpenGL 1.2, which allows one polygon to be texture mapped using color and opacity information from multiple textures. OpenGL 1.2 specifies multi-texturing as a sequence of stages, in which the result of one stage can be combined with the result of the previous one.

Unfortunately, in general this concept turns out to be too static for many desired applications. Therefore, recent PC graphics boards support multi-stage rasterization in order to explicitly control how color-, opacity- and texture-components are combined to form the resulting fragment. By means of this extension rather complex calculations can be performed in a single rendering pass.

Although multiple rasterization stages are supported by PC graphics boards from different vendors, until now these features are optional extensions to the OpenGL standard and thus hardware-dependent. Since every manufacturer of graphics hardware defines its own extensions, we will restrict our description to graphics boards with NVidia's *GeForce 256* processor. In this respect we strictly focus on the work of [85], where most of the ideas that will be presented hereafter have been introduced.

As an OpenGL extension that enables one to gain explicit control over per-fragment information, NVidia has provided the `NV_register_combiners` [96] which completely bypasss the standard OpenGL texturing units (see Fig. 23). It consists of two flexible general rasterization stages and one final combiner stage. The general combiner stage is divided into an RGB-portion and a separate Alpha-portion as displayed in Figure 24.

A number of input registers can be programmed and combined with each other quite flexibly (see Fig. 24). The output registers of the first combiner stage are then used as the in-

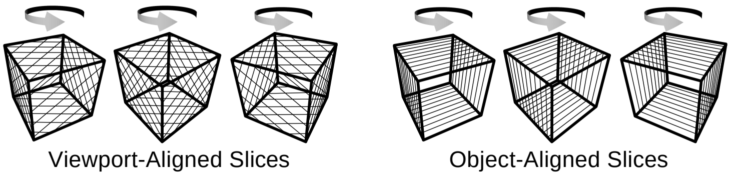


Figure 22: *Viewport-aligned slices (left) in comparison to object aligned slices (right) for a spinning volume object.*

put registers for the next stage. Fixed point color components that are usually clamped to a range of $[0,1]$ can internally be expanded to a signed range of $[-1,1]$. Vector components, for example, can be handled in this way, which significantly simplifies the computation of local diffuse illumination for the methods described below.

The output registers of the second general stage are directed into a final combiner stage with restricted functionality. Once multi-stage rasterization is performed in the described way the standard OpenGL per-fragment operations are performed. We should note here, that in contrast to the SGI texture color tables the GeForce architecture supports paletted textures that allow for interpolation of pre-shaded texture values.

7.7. Multi-Texture Interpolation

In order to achieve view independent distances between sample points intermediate slices are trilinearly interpolated on the fly from the original slices. The missing third interpolation step is performed within the rasterization hardware using multi-textures, as outlined in Figure 26).

Any intermediate slice $S_{i+\alpha}$ can be obtained by blending between adjacent slices S_i and S_{i+1} from the original stack:

$$S_{i+\alpha} = (1-\alpha)\cdot S_i + \alpha \cdot S_{i+1}. \tag{11}$$

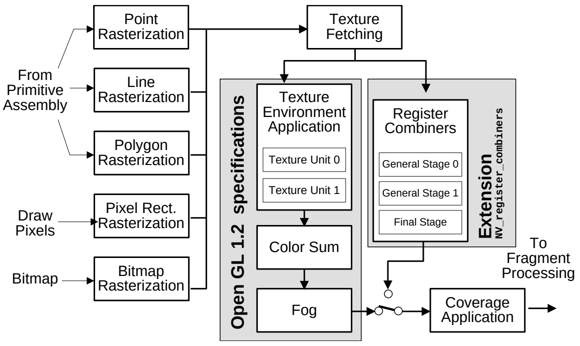


Figure 23: *Since the multi-texture model of OpenGL 1.2 turns out to be too limiting, NVidia's GeForce 256 processor provides multi-stage register combiners that completely bypass the standard texturing unit.*

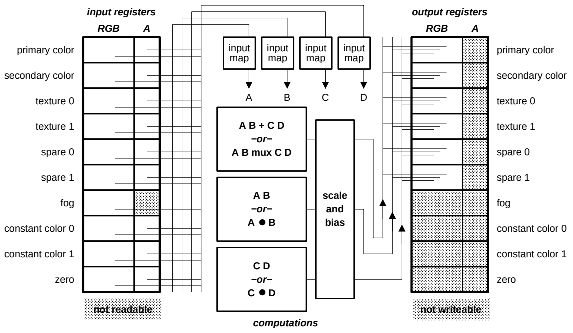


Figure 24: *The RGB-portion of the general combiner stage supports arbitrary register mappings and complex computation like dot products and component-wise weighted sum.*

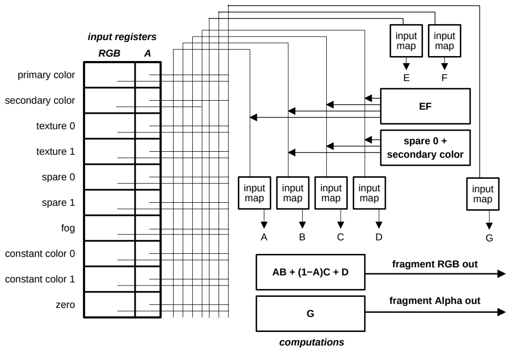


Figure 25: *The final combiner stage is used to compute the resulting fragment output for RGB and Alpha.*

With each slice image stored in a separate 2D-texture, the final interpolation between bilinearly interpolated samples is computed by blending the results. As displayed in Figure 26, blending is computed by a single general combiner stage, where the original slices S_i and S_{i+1} are specified as multi-textures `texture 0` and `texture 1`. The combiner is setup to compute a component-wise weighted sum $AB+CD$ with the interpolation factor α stored in one of the constant color registers. The contents of this register is mapped to input variable A, and at the same time it is inverted and mapped to variable C. In the RGB-portion, variables B and C are as-

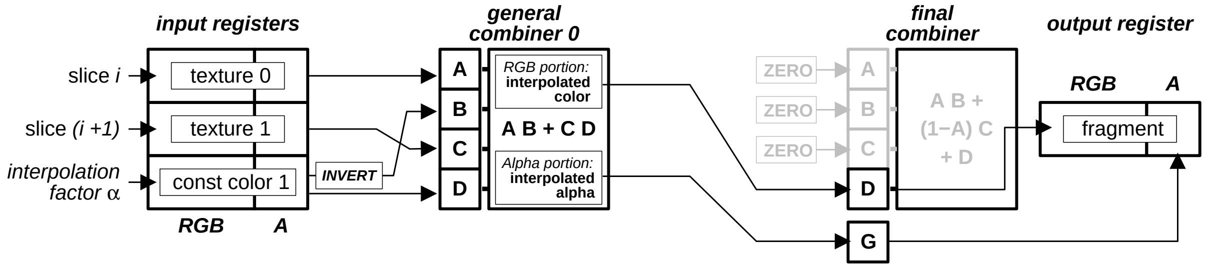


Figure 26: *Combiner setup for interpolation of intermediate slices.*

signed the RGB components of `texture 0` and `texture 1` respectively. Analogously, the Alpha-portion interpolates between the alpha-components. Now, the output of this first combiner stage is combined in with the pixel values already in the frame buffer using α-blending.

All that remains to be done is to choose the location of intermediate slices depending on the actual view direction in order to guarantee äquidistant sample distances. Thus, we completely avoid adapting the opacity for every view by selecting the number of slices to be reconstructed appropriately.

7.8. Shaded iso-surfaces

As mentioned in the first part of this section, in [106] an efficient algorithm was introduced that exploits rasterization hardware to display shaded iso-surfaces using a 3D gradient map. Using multi-stage rasterization, this method can be efficiently adapted to the GeForce graphics processor. The voxel gradient is computed as before and written into the RGB components of a set of 2D-textures that represent the volume. Analogously, the material is coded in the alpha-component. The register combiner are then programmed as illustrated in Fig. 27. The first general combiner stage is applied as described in Section 7.7 to interpolate intermediate slices. The second general combiner now computes the dot product $A \bullet B$ between , where variable A is mapped to the RGB output of the first combiner stage (the interpolated gradient $\vec{n}$) and variable B is mapped to the second constant color register, that contains the light vector $\vec{l}$. The alpha-component is not modified by the second combiner stage. Note that the general combiner stages support signed fixed point values, so there is no need to scale and bias the vector components to positive range.

Since the final combiner is capable of computing $AB + (1-A)C + D$, when storing the color of diffuse and ambient light in the registers for primary and secondary color, the final combiner can be used for compositing all involved terms. Therefore variable A is assigned to primary color (I_d) and is multiplied with variable B which is mapped to the dot product, computed by the RGB-portion of the second general combiner. Variable C is set to zero and variable D is mapped to secondary color (I_a).

Note that this particular implementation is a single-pass rendering technique, since all computations are performed in the register combiners before fragments are going to be combined. In this way, the copy operation in order to multiply pixel values with the properly initialized color matrix can be avoided.

Moreover, by using the same combiner setup multiple transparent iso-surfaces can be displayed. Therefore, the α-lookup-table is set up in such a way, that the material ranges to be displayed are represented by α-values larger than zero. For all other materials the alpha value is set to zero. Now, the alpha test discards everything equal to zero, the depth test is disabled and α-blending is performed. Rendering the slices in back-to-front order yields the semi-transparent iso-surfaces correctly accumulated with respect to the selected attenuation.

8. Unstructured Volume Rendering

Figure 28 shows four types of volume datasets: regular, rectilinear, curvilinear, and unstructured. Interactive volume rendering has always been a challenging problem. So far, we have seen how a combination of algorithm developments and hardware advances can created interactive rendering solutions for regular rectilinear datasets, see previous sections and [117]. These solutions provide changing viewpoints with interactivity 1-30 frames/second. In this portion of the tutorial, we focus on unstructured data. Such datasets are computed in computational fluid dynamics and finite element analysis.

8.1. Introduction

Providing interactivity requires applying optimally tuned graphics hardware and software to accelerate the features of interest. Many researchers in the past [93] have commented that creation of special purpose hardware would solve the slowness of handling unstructured grids. And, general purpose hardware is not adequate, because the operations per

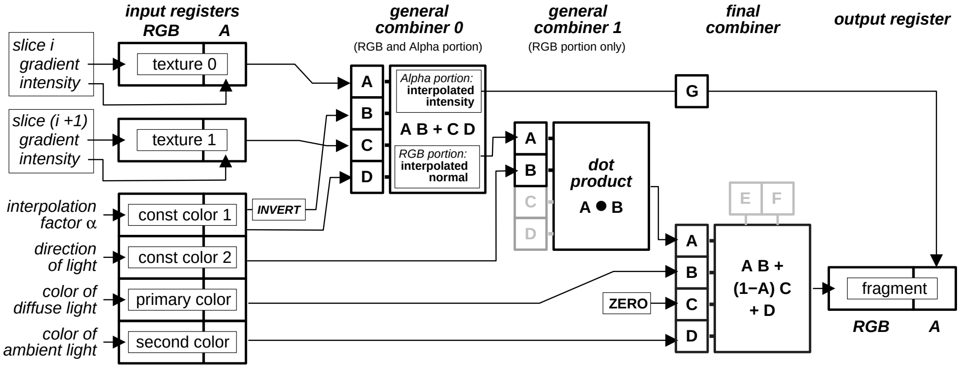


Figure 27: *Combiner setup for fast rendering of shaded isosurfaces*

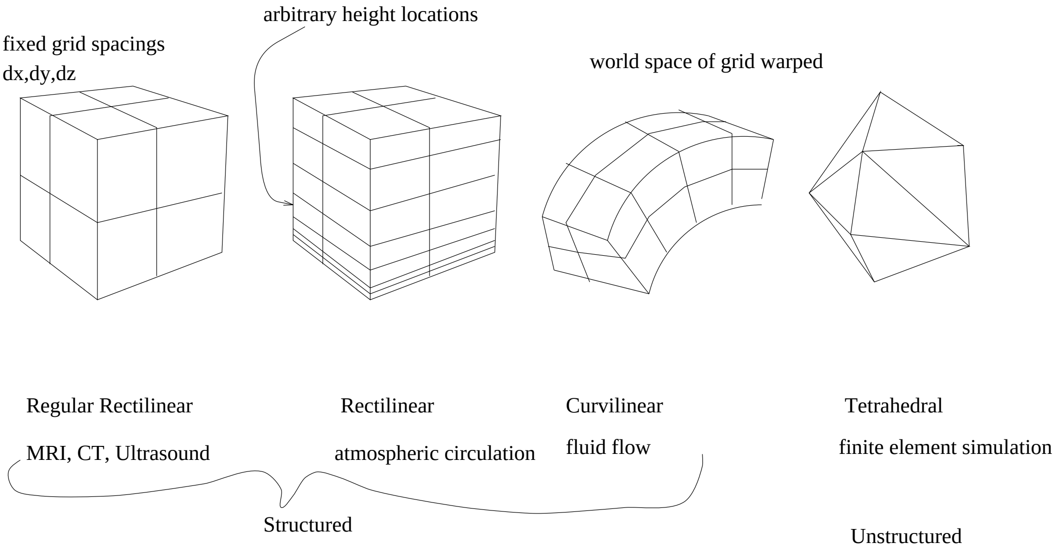


Figure 28: *Example grid types.*

second requirement is simply too high. Within the last year special purpose graphics hardware is now fast enough to make the existing algorithms interactive – if only the proper optimizations are made. Current graphics hardware is interactive, using our optimizations, for moderate sized datasets of thousands to millions of cells. Special purpose volume hardware or reprogrammable graphics hardware are additional ways to further accelerate unstructured volume rendering algorithms.

In order to achieve the highest possible performance for interactive rendering of unstructured data, many software optimizations are necessary. In this section of the tutorial, we explore software optimizations using OpenGL triangle fans, customized quicksort, memory organization for cache efficiency, display lists, tetrahedral culling, and multithreading. The optimizations can vastly improve the performance of projected tetrahedra rendering to provide interactive rendering on datasets of hundreds of thousands of tetrahedra. These results are an order of magnitude faster than the best in the literature for unstructured volume rendering [126, 101]. The sorting is also an order of magnitude faster than the fastest sorting timing reported for Williams MPVONC [17].

Momentum for unstructured rendering, cellular based rendering, and ray tracing of irregular grids was created at the Volume Visualization Symposium [67, 93, 25]. Peter Williams developed extensions to Shirley et al. [93] tetrahedral renderer, and provided simplified cell representations to give greater interactivity [113]. Ray tracing has been used for rendering of irregular data as well [25]. Most recently there has been work on multiresolutional irregular volume representations [11] distributed volume rendering algorithms on massively parallel computers [64], And, optimization using texture mapping hardware [126]. Software scan conversion implementations are also being researched [110].

The rendering of unstructured grids can be done by supporting cell primitives, at a minimum tetrahedra, which when drawn in the appropriate order, can be accumulated into the frame buffer for volume visualization. There is a disadvantage of using only tetrahedral cells to support the hexahedral and other cells, mainly a 5x explosion in data as a hexahedral cube requires 5 tetrahedra at a minimum to be represented, and the tiling of adjacent cells when subdividing may cause problems in matching edges including cracking, where a small crack may appear between hexahedral cells that have been subdivided into tetrahedra [93].

For proper compositing either the cells viewpoint ordering can be determined from the grid or a depth sorting is performed. For regular grids the viewpoint ordering is straightforward. For unstructured data, the viewpoint ordering must be determined for each viewpoint by comparing the depth of cells. Schemes that use cell adjacency data structures can help to reduce the run time complexity of the sort [114, 93].

The basic approach to compute cell's contributions is three dimensional scan conversion. Several variants were investigated in Wilhelms and Van Gelder [109]. The first variant is averaging opacity and color of front and back faces then using a nonlinear approximation to the exponent. The second variant is averaging opacity and color of front and back faces, then exactly computing the exponential. And, the third variant is, using numerical linear integration of color and opacity between front and back faces, and exactly computing the exponentials. Gouraud shading hardware is used by interpolating colors and opacities across faces which is possible if the exponential approximation is evaluated at the vertices [93].

8.2. Projected Tetrahedra

This tutorial shows how to make unstructured rendering as interactive as possible on available hardware platforms. The projected tetrahedra algorithm of Shirley and Tuchman [93] uses the geometry acceleration and rasterization of polygon rendering to maximum advantage. The majority of the work, the scan conversion and compositing, are accelerated by existing graphics hardware. In earlier work, we investigated hardware acceleration with parallel compositing on PixelFlow [116]. OpenGL hardware acceleration is now widely available in desktop systems, and using the earlier implementation as a starting point we investigated further optimizations to improve desktop rendering performance. Figure 29 provides pseudo-code of Shirley and Tuchman's [93] projected tetrahedra algorithm, where tetrahedra are projected at the screen plane, and subdivided into triangles. Figure 31 shows the four classes of projections that result in one to four triangles.

```
I.  Preprocess dataset
for a new viewpoint:
II. Visibility sort cells
for every cell (sorted back-to-front)
  III. Test plane equations to determine
       class (1,2,3,4)
  IV.  Project and split cell unique
       frontback faces
  V.   Compute color and opacity for thick
       vertex
  VI.  (H) Scanconvert new triangles
```

Figure 29: *Projected tetrahedra pseudo-code*

Preprocessing is necessary to calculate colors and opacities from input data, setup for visibility sorting of primitives, and creation of plane equations used for determining the class a tetrahedra belongs to for a viewpoint. Cells are visibility sorted (step II) for proper compositing for each viewpoint [114]. Figure 31 shows that new vertices are introduced during steps III and IV. The new vertex requires a color and opacity to be calculated in step V. Then the triangles are drawn and scan converted in step VI. Figure 30 shows the output of our renderer for the Phoenix, NASA Langley Fighter, and F117 datasets.

8.3. Acceleration Techniques

8.3.1. Triangle strips.

Triangle strips can greatly improve the efficiency of an application since they are a more compact way of describing, storing, and transferring a set of neighbouring triangles. Creating triangle strips from triangular meshes of data can be done through greedy algorithms. But, in projected tetrahedra, the split cases illustrated in Figure 31 can directly create triangle fans. Each new vertex given with *glVertex* specifies a new triangle, instead of redundantly passing vertices. Figure 31 gives triangle strips for the four classes of projections.

In experiments on the NASA Langley fighter dataset, with 70,125 tetrahedra, there are an average of 3.4 triangles per tetrahedra (a sum of the percentage of Class 1–60% 3 triangles, 2–40% 4 triangles, 3, and 4). Fewer vertices are transmitted for the same geometry and many fewer procedure calls are made to OpenGL. For n tetrahedra there are 10.8n procedure calls with fans versus 20.4n procedure calls without fans, a factor of 2 reduction. Williams also investigated triangle stripping using Iris-GL.

8.3.2. Display Lists For Static Geometry.

Display lists allow the graphics drivers to optimize vertices, colors, and triangle strips for the hardware. I converted all static geometry into display lists. Figure 32 shows examples of vertices, surfaces, and a background mesh. The primary impact of these changes is to eliminate any slowdown when these auxiliary data are also rendered. Unfortunately,

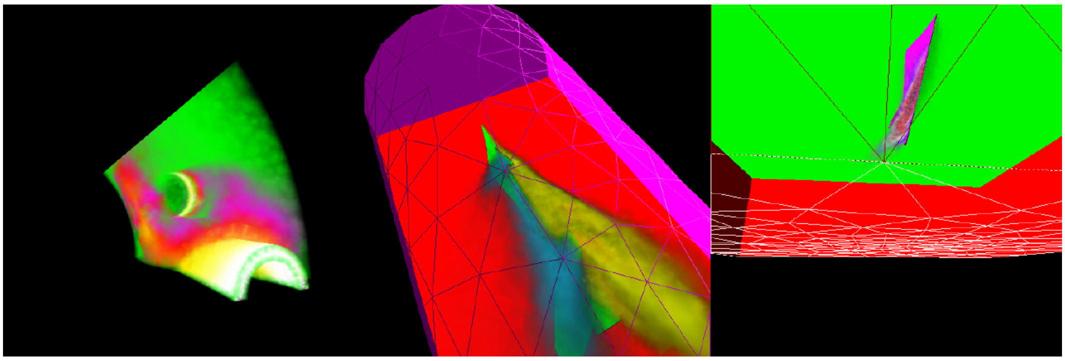


Figure 30: *Unstructured volume rendering of Phoenix (left), NASA Langley Fighter (middle), and F117 (right).*

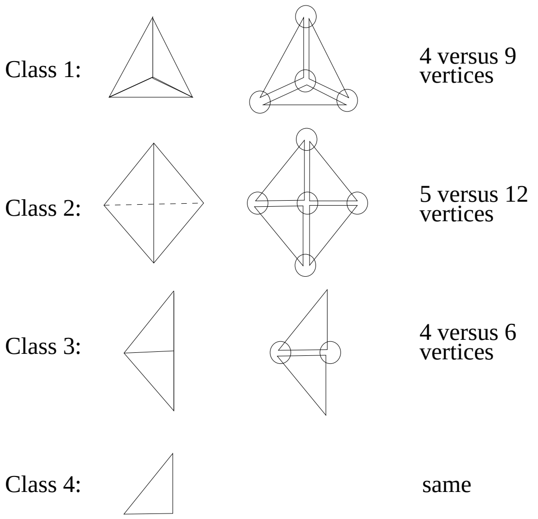


Figure 31: *Number of vertices for fan versus triangles.*

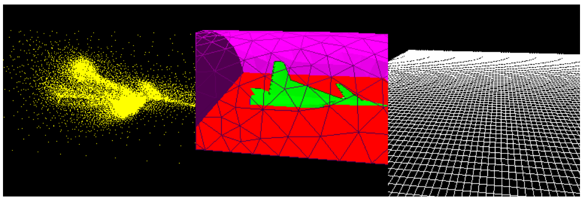


Figure 32: *Points (left) surfaces (middle) and mesh (right) stored in display lists.*

the projected tetrahedra recomputes the thick vertex for every new viewpoint so that the volume data cannot be placed in display lists. Also, because the thick vertex is recalculated for each new view, vertex arrays cannot be used.

8.3.3. Visibility Sorting—Customized Quicksort.

Visibility sorting of cells is done through Cignoni et al. and Karasick et al.'s [12, 42] technique of sorting the tangential distances to circumscribing spheres. We have compared a custom coded quicksort to the C library utility *qsort()* an improvement of 75% to 89%. A generic sorting routine cannot as efficiently handle the data structures that are to be sorted. Two values, the tangential squared distance (float) and the tetrahedral index are moved.

8.3.4. Taking advantage of view coherence.

The proper choice of pivots gives an efficient sorting of sorted and nearly sorted lists. We previously resorted the same input for each view. But, using the sorted values from the previous view speeds up the sorting. The program was also modified to use smaller viewpoint changes, and run times were improved by an additional 18%. Both sorts are much faster on sorted data. This property is exploited for view coherence. The rate for the custom quicksort varies between 600,000 to 2 million cells per second depending on how sorted the list is. For comparison, recently reported results for MPVONC [114] are from 185,000 to 266,000 cells per second [17]. In our earlier work, we showed that quicksort achieved 109,570 cells/second on a PA RISC 7200 (120 MHz) [116], without the the view coherence and data structure optimizations discussed here.

8.3.5. Cache coherency.

Because the tetrahedral data structures are randomly accessed, a high percentage of time is spent in the first fetch of each tetrahedra's data. Tetrahedra are accessed by their view and not memory storage order. Reordering the tetrahedra when performing the view sort eliminated cache stalls when rendering data, but the sorting routine was slowed down by moving more data. There was an overall slowdown, so future work is needed to find effective caching strategies.

8.3.6. Culling.

Tetrahedra whose opacity are zero are removed from sorting and rendering. This is classification dependent, but yields 20% and 36% reduction in tetrahedra for the Langley Fighter and F117 datasets. The runtime decreases accordingly.

8.3.7. Multithreading.

Many desktop systems now have two CPU's. We multithreaded the renderer on Windows NT to explore additional speedup available. Profiling of the code showed that spare CPU cycles were available, and that the graphics performance was not yet saturated. Two threads for the compute intensive floating point work of testing and splitting cells, were used, and the sorting and OpenGL calls were separated into separate threads. The sorting was done for the next frame, for a pipeline parallelism. The partitioning for the cell sorting and splitting was done on a screen based partition using the centroid of the visible or nonculled tetrahedra. Such a dynamic partitioning provided a fast decision, yet gave good load balancing in most cases. The multithreading provided an additional 14% to 25% speedup.

8.4. Summary

Unstructured volume rendering is now truly interactive due to a combination of advances in software algorithms for sorting of cells in depth and graphics hardware improvements. To implement an interactive unstructured renderer requires understanding how to exploit the special purpose graphics hardware, and the CPU's available in today's desktop machines. We have shown the key elements needed to optimize a renderer including use of triangle fans for reducing the bandwidth bottleneck, culling of unseen data, multithreading for lowly parallelism, and making OpenGL state changes as infrequently as possible. Display lists are possible only for the nonvolume data, because of the nature of the projected tetrahedra, requires recomputing different triangles from the tetrahedra for every new viewpoint. We did show that placing surfaces, points, and auxilliary visualization planes in display lists literally adds these capabilities for free. For rendering performance results, please see our papers [116, 117, 118]. The next step in reaching even higher visualization rates over what commodity graphics hardware and desktop systems provide will be to add important extensions and capabilities to the hardware to directly support volume primitives.

9. Acknowledgments

Data sets are thankfully acknowledged: for the skull Siemens, Germany; for the blood vessels Philips Research, Hamburg; for the NASA Langley Fighter, Neely and Batina; for the Super Phoenix Nuclear reactor, Bruno Nitrosso, Electricite de France; for the F117, Robert Haimes, MIT;

We also would like to thank Bruce Culbertson, Tom Malzbender, Greg Slabaugh, Jian Huang, Klaus Mueller, Roger Crawfis, Dirk Bartz, Dean Brederson, Claudio Silva, Vivek Verma, and Peter Williams for help in getting images, data, and classifications.

References

1. K. Akeley. RealityEngine graphics. In *Computer Graphics*, Proceedings of SIGGRAPH 93, pages 109–116, August 1993.
2. R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: A diversified system for volume visualization research and development. In *Proceedings of Visualization 94*, pages 31–38, Washington, DC, October 1994.
3. R. Avila, L. Sobierajski, and A. Kaufman. Towards a comprehensive volume visualization system. In *Proceedings of Visualization 92*, pages 13–20, Boston, MA, October 1992.
4. R. Bakalash and A. Kaufman. 3D visualization of biomedical data. In *Proc. 6th Annual International Electronic Imaging Conference*, pages 1034–1036, Boston, MA, October 1989.
5. D. Bartz and M. Meißner. Voxels versus Polygons: A Comparative Approach for Volume Graphics. Proc. of 1st Workshop on Volume Graphics, March 1999.
6. M. Bentum. Frequency analysis of gradient estimators in volume rendering. *IEEE Transactions of Visualization and Computer Graphics*, 2(3):242–254, 1996.
7. L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. In *Computer Graphics*, pages 29–37. Proceedings of SIGGRAPH 86, November 1986.
8. J. F. Blinn. Light refelction functions for simulation of clouds and dusty surfaces. *Computer Graphics*, 16(3):21–29, July 1982.
9. B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Workshop on Volume Visualization*, pages 91–98, Washington, DC, October 1994.
10. I. Carlbom. Optimal filter design for volume reconstruction. In *Proc. of IEEE Visualization*, San José, CA, USA, October 1993. IEEE Computer Society Press.
11. P. Cignoni, L. De Floriani, C. Montani, E. Puppo, and R. Scopigno. Multiresolution modeling and visualization of volume data based on simplicial complexes. In

Proceedings of the Symposium on Volume Visualization, pages 19–26, 1994.

12. Paolo Cignoni, Claudio Montani, D. Sarti, and Roberto Scopigno. On the optimization of projective volume rendering. In R. Scaneni, J. van Wijk, and P. Zanarini, editors, *Proceedings of the Eurographics Workshop, Visualization in Scientific Computing'95*, pages 59–71, Chia, Italy, May 1995.

13. H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. C. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, May 1988.

14. D. Cohen. 3D scan conversion of geometric objects. Doctoral Dissertation, Department of Computer Science, SUNY at Stony Brook, December 1991.

15. D. Cohen and A. Kaufman. Scan conversion algorithms for linear and quadratic objects. In A. Kaufman, editor, *Volume Visualization*, pages 280–301. IEEE Computer Society Press, Los Alamitos, CA, 1990.

16. D. Cohen and Z. Shefer. Proximity clouds - an acceleration technique for 3D grid traversal. Technical Report FC 93-01, Department of Mathematics and Computer Science, Ben Gurion University of the Negev, February 1993.

17. J. Comba, J. T. Klosowski, N. Max, J. S.B. Mitchell, C. T. Silva, and P. L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. In *Proceedings of Eurographics'99*, Milan, Italy, September 1999.

18. 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.

19. F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-Quality Volume Rendering Using Texture Mapping Hardware. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 69–76, Lisboa, Portugal, August 1998.

20. J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In A. Kaufman and W. L. Lorensen, editors, *Workshop on Volume Visualization*, pages 91–98, Boston, MA, October 1992.

21. M. C. Doggett, M. Meißner, and U. Kanus. A low-cost memory architecture for volume rendering. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 7–14, August 1999.

22. R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, August 1988.

23. E. J. Farrell and R. A. Zappulla. Three-dimensional data visualization and biomedical applications. *CRC Critical Reviews in Biomedical Engineering*, 16(4):323–363, 1989.

24. G. Frieder, D. Gordon, and R. A. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics & Applications*, 5(1):52–60, January 1985.

25. M. P. Garrity. Raytracing irregular volume data. In *Symposium on Volume Visualization*, San Diego, CA, November 1990.

26. A. Van Gelder and K. Kim. Direct Volume Rendering With Shading via Three-Dimensional Textures. In *Symposium on Volume Visualization*, pages 23–30, San Francisco, CA, USA, October 1996.

27. S. M. Goldwasser and R. A. Reynolds. *Techniques for the Rapid Display and Manipulation of 3D Biomedical Data*. Dept. of Computer and Info. Science, Univ. of Pennsylvania Report MS-CIS-86-14, GRASP LAB 60., Philadelphia, July 1986.

28. S.M. Goldwasser, R.A. Reynolds, D.A. Talton, and E.S. Walsh. High performance graphics processors for medical imaging applications. In *Proc. International Conference on Parallel Processing for Computer Vision and Display*, Leeds, UK, January 1988.

29. D. Gordon and R. A. Reynolds. Image space shading of 3-dimensional objects. *Computer Vision, Graphics, and Image Processing*, 29:361–376, 1985.

30. B. Gudmundsson and M. Randen. Incremental generation of projections of CT-volumes. In *Proceedings of the First Conference on Visualization in Biomedical Computing*, pages 27–34, Atlanta, GA, May 1990. IEEE Computer Society Press, Los Alamitos, California.

31. T. Guenther, C. Poliwoda, C. Reinhard, J. Hesser, R. Maenner, H.-P. Meinzer, and H.-J. Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. In *Proceedings of the 9th Eurographics Workshop on Graphics Hardware*, pages 103–108, Oslo, Norway, September 1994.

32. P. Haeberli and K. Akeley. The accumulation buffer; hardware support for high-quality rendering. In *Computer Graphics*, volume 24 of *Proceedings of SIGGRAPH 90*, pages 309–318, Dallas, TX, August 1990.

33. P. Hanrahan. Three-pass affine transforms for volume rendering. *Computer Graphics*, 24(5):71–78, November 1990.

34. 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.

35. G. T. Herman and H. K. Liu. Three-dimensional display of human organs from computed tomograms. *Computer Graphics and Image Processing*, 9:1–21, 1979.

36. K. H. Höhne and R. Bernstein. Shading 3D-images from CT using gray-level gradients. *IEEE Transactions on Medical Imaging*, MI-5(1):45–47, March 1986.

37. K. H. Höhne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, and G. Wiebecke. 3D-visualization of tomographic volume data using the generalized voxel model. *The Visual Computer*, 6(1):28–37, February 1990.

38. K. H. Höhne, B. Pfiesser, A. Pommert, M. Riemer, T. Schiemann, R. Schubert, and U. Tiede. A virtual body model for surgical education and rehearsal. *IEEE Computer*, 29(1):45–47, 1996.

39. J. Huang, K. Mueller, N. Shareef, and R. Crawfis. Edge preservation in volume rendering using splatting. pages 63–69, Research Triangle Park, NC, USA, October 1998.

40. D. Jackel and W. Straßer. Reconstructing solids from tomographic scans - the PARCUM II system. In A.A.M. Kuijk and W. Straßer, editors, *Advances in Computer Graphics Hardware II*, pages 209–227. Springer-Verlag, Berlin, 1988.

41. J. T. Kajiya and T. Von Herzen. Ray tracing volume densities. *Computer Graphics*, 18(3):165–173, July 1984.

42. M.S. Karasick, D. Lieber, L.R. Nackman, and V.T. Rajan. Visualization of three-dimensional delaunay meshes. *Algorithmica*, 19(1-2):114–128, Sept.-Oct. 1997.

43. A. Kaufman. An algorithm for 3D scan-conversion of polygons. *Proc. EUROGRAPHICS'87*, pages 197–208, August 1987.

44. A. Kaufman. The *voxblt* engine: A voxel frame buffer processor. In A.A.M. Kuijk and W. Straßer, editors, *Advances in Graphics Hardware III*. Springer-Verlag, Berlin, 1989.

45. A. Kaufman and R. Bakalash. Memory and processing architecture for 3D voxel-based imagery. *IEEE Computer Graphics & Applications*, 8(6):10–23, November 1988.

46. A. Kaufman and R. Bakalash. Parallel processing for 3D voxel-based graphics. In *Proc. International Conference on Parallel Processing for Computer Vision and Display*, Leeds, UK, January 1988.

47. G. Knittel and W. Straßer. Vizard – visualization accelerator for real-time display. In *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*, pages 139–146, Los Angeles, CA, August 1997.

48. W. Krueger. The application of transport theory to the visualization of 3d scalar fields. *Computers in Physics 5*, pages 397–406, 1991.

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

50. P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform.* PhD thesis, Stanford University, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford, CA 94305–4055, 1995. CSL-TR-95-678.

51. P. Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):218–231, September 1996.

52. P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transform. In *Computer Graphics*, Proceedings of SIGGRAPH 94, pages 451–457, July 1994.

53. D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume renderi ng. *Computer Graphics*, 25(4):285–288, July 1991.

54. M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(5):29–37, May 1988.

55. M. Levoy. Design for real-time high-quality volume rendering workstation. In C. Upson, editor, *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 85–92, Chapel Hill, NC, May 1989. University of North Carolina at Chapel Hill.

56. M. Levoy. Display of surfaces from volume data. *Ph.D. Dissertation, Department of Computer Science, The University of North Carolina at Chapel Hill*, May 1989.

57. M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

58. M. Levoy. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics & Applications*, 10(2):33–40, March 1990.

59. M. Levoy. A taxonomy of volume visualization algorithms. In *SIGGRAPH 90 Course Notes*, pages 6–12, 1990.

60. M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, pages 2–7, July 1990.

61. B. Lichtenbelt. Design of a High Performance Volume Visualization System. In *Proc. of the Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 111–119, Los Angeles, CA, USA, July 1997.

62. B. Lichtenbelt, R. Crane, and S. Naqvi. *Introduction to volume rendering.* Hewlett-Packard Professional Books, Prentice-Hall, Los Angeles, USA, 1998.

63. W. E. Lorensen and H. E. Cline. Marching–cubes: A high resolution 3D surface construction algorithm. In *Computer Graphics*, Proceedings of SIGGRAPH 87, pages 163–169, 1987.

64. Kwan-Liu Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of the Parallel Rendering Symposium*, pages 23–30, Atlanta, GA, Oct 1995. IEEE.

65. R. Machiraju, R. Yagel, and L. Schwiebert. Parallel algorithms for volume rendering. OSU-CISRC-10/92-R29,, Department of Computer and Information Science, The Ohio State University, October 1992.

66. N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

67. N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. *ACM Computer Graphics (Proceedings of the 1990 Workshop on Volume Visualization)*, 24(5):27–33, 1990.

68. 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.

69. D. Meagher. Efficient synthetic image generation of arbitrary 3D objects. In *Proceedings of IEEE Computer Society Conference on Pattern Recognition and Image Processing*, June 1982.

70. M. Meißner, U. Hoffmann, and W. Straßer. Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering using OpenGL and Extensions. In *Proc. of IEEE Visualization*, pages 207–214, San Franisco, CA, USA, October 1999. IEEE Computer Society Press.

71. M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of four popular volume rendering algorithms. In *Symposium on Volume Visualization*, Salt Lake City, UT, USA, October 2000.

72. M. Meißner, U. Kanus, and W. Straßer. VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 61–68, Lisboa, Portugal, August 1998.

73. T. Moeller, R. Machiraju, K. Mueller, and R. Yagel. Evaluation and design of filters using a taylor series expansion. *IEEE Transactions of Visualization and Computer Graphics*, 3(2):184–199, 1997.

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

75. 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.

76. K. Mueller and R. Crawfis. Elminating popping artifacts in sheet buffer-based splatting. In *Proc. of IEEE Visualization*, pages 239–246, Triangle Research Park, NC, USA, October 1998. IEEE Computer Society Press.

77. K. Mueller, T. Moeller, and R. Crawfis. Splatting without the blur. In *Proc. of IEEE Visualization*, pages 363–371, San Franisco, CA, USA, October 1999. IEEE Computer Society Press.

78. H. Müller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.

79. T. Ohashi, T. Uchiki, and M. Tokoro. A three-dimensional shaded display method for voxel-based representation. In *Proceedings EUROGRAPHICS '85*, pages 221–232, Nice, France, September 1985.

80. R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibson, W. Hiatt, and T. Ohkami. EM-Cube: An architecture for low-cost real-time volume rendering. In *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*, pages 131–138, Los Angeles, CA, August 1997.

81. S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive ray tracing for isosurface rendering. In *Proc. of IEEE Visualization*, pages 233–238, Triangle Research Park, NC, USA, October 1998. IEEE Computer Society Press.

82. M. Peercy, J. Airey, and B. Cabral. Efficient bump mapping hardware. In *Computer Graphics*, Proceedings of SIGGRAPH '97, pages 303–306, August 1997.

83. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Computer Graphics*, SIGGRAPH 99 Proceedings, pages 251–260, Los Angeles, CA, August 1999.

84. H. Pfister and A. Kaufman. Cube-4 – A scalable architecture for real-time volume rendering. In *1996 ACM/IEEE Symposium on Volume Visualization*, pages 47–54, San Francisco, CA, October 1996.

85. C. Reszk-Salama, K. Engel, T. Bauer, T. Ertl, and G. Greiner. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. *EG Hardware Workshop'2000*, 2000.

86. R. A. Reynolds, G. Frieder, and D. Gordon. Back-to-front display of voxel-based objects. *IEEE Computer Graphics & Applications*, pages 52–59, January 1985.

87. R. A. Reynolds, D. Gordon, and L.-S. Chen. A dynamic screen technique for shaded graphics display of slice-represented objects. *Computer Vision, Graphics, and Image Processing*, 38(3):275–298, 1987.

88. P. Sabella. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics*, 22(4):59–64, August 1988.

89. H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, June 1984.

90. P. Schröder and J. B. Salem. Fast rotation of volume data on data parallel architectures. In *Proceedings of Visualization '91*, pages 50–57, San Diego, CA, October 1991. IEEE CS Press.

91. P. Schröder and G. Stoll. Data parallel volume rendering as line drawing. In *1992 Workshop on Volume Visualization*, pages 25–31, Boston, MA, October 1992.

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

93. P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *1990 Workshop on Volume Visualization*, pages 63–70, San Diego, CA, December 1990.

94. L. Sobierajski, D. Cohen, A. Kaufman, R. Yagel, and D. Acker. Trimmed voxel lists for interactive surgical planning. TR 90.05.22, SUNY Stony Brook, May 1990.

95. L. Sobierajski, D. Cohen, A. Kaufman, R. Yagel, and D. Acker. Fast display method for interactive volume rendering. *The Visual Computer*, 1993.

96. J. Spitzer. GeForce 256 and RIVA TNT Combiners. http://www.nvidia.com/Developer.

97. U. Tiede, K. H. Höhne, M. Bomans, A. Pommert, M. Riemer, and G. Wiebecke. Investigation of medical 3D-rendering algorithms. *IEEE Computer Graphics & Applications*, 10(2):41–53, March 1990.

98. Y. Trousset and F. Schmitt. Active-ray tracing for 3D medical imaging. In G. Marechal, editor, *Proceedings of EUROGRAPHICS'87*, pages 139–149. Elsevier Science Publishers, 1987.

99. H. K. Tuy and L. T. Tuy. Direct 2-D display of 3D objects. *IEEE Computer Graphics & Applications*, 4(10):29–33, November 1984.

100. C. Upson and M. Keeler. V-BUFFER: Visible volume rendering. *Computer Graphics*, 22(4):59–64, August 1988.

101. A. Van Gelder, V. Verma, and J. Wilhelms. Volume decimation of irregular tetrahedral grids. In *Proceedings of Computer Graphics International*, pages 222–230,247, Canmore, Alta., Canada, June 1999.

102. A. Van Geldern 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.

103. J. Terwisscha van Scheltinga, J. Smit, and M. Bosma. Design of an on Chip Reflectance Map. In *Proc. of the 10th EG Workshop on Graphics Hardware*, pages 51–55, Maastricht, The Netherlands, August 1995.

104. T. van Walsum, A. J. S. Hin, J. Versloot, and F. H. Post. Efficient hybrid rendering of volume data and polygons. *Second EUROGRAPHICS Workshop on Visualization in Scientific Computing*, April 1991.

105. D. Voorhies and J. Foran. State of the art in data visualization. pages 163–166, July 1994.

106. R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 169–177, Orlando, FL, USA, August 1998.

107. L. Westover. Interactive volume rendering. In C. Upson, editor, *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 9–16, Chapel Hill, NC, May 1989. University of North Carolina at Chapel Hill.

108. L. Westover. Footprint evaluation for volume rendering. In *Computer Graphics*, Proceedings of SIGGRAPH 90, pages 367–376, August 1990.

109. J. Wilhelms. Decisions in volume rendering. In *State of the Art in Volume Visualization*, volume 8, pages I.1–I.11. ACM SIGGRAPH, Las Vegas, NV, Jul./Aug. 1991.

110. J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *Proceedings of Visualization*, pages 57–64, San Francisco, CA, October 1996.

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

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

113. P. L. Williams. Interactive splatting of nonrectilinear volumes. In *Proceedings Visualization*, pages 37–44, Boston, October 1992.

114. P. L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

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

116. C. M. Wittenbrink. Irregular grid volume rendering with composition networks. In *Proceedings of IS&T/SPIE Visual Data Exploration and Analysis V*, volume 3298, pages 250–260, San Jose, CA, January 1998. SPIE. Available as Hewlett-Packard Laboratories Technical Report, HPL-97-51-R1.

117. C. M. Wittenbrink. Cellfast: Interactive unstructured volume rendering. Technical Report HPL-1999-81(R1), Hewlett-Packard Laboratories, July 1999. Appeared in 1999 IEEE Visualization-Late Breaking Hot Topics.

118. C. M. Wittenbrink, M. E. Goss, H. Wolters, and T. Malzbender. Interactive unstructured volume rendering and classification. Technical Report HPL-2000-13, Hewlett-Packard Laboratories, January 2000.

119. C. M. Wittenbrink, T. Malzbender, and M. E. Goss. Opacity-Weighted Color Interpolation For Volume Sampling. In *Symposium on Volume Visualization*, pages 135–142, Research Triangle Park, NC, USA, October 1998.

120. R. Yagel. *Efficient Methods for Volumetric Graphics*. PhD thesis, State Univeristy of New York at Stony Brook, December 1991.

121. R. Yagel. The flipping CUBE: A device for rotating 3D rasters. *6th Eurographics Hardware Workshop*, September 1991.

122. R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics & Applications*, pages 19–28, September 1992.

123. R. Yagel, D. Cohen, A. Kaufman, and Q. Zhang. Volumetric ray tracing. TR 91.01.09, Computer Science, SUNY at Stony Brook, 1991.

124. R. Yagel and A. Kaufman. Template-based volume viewing. *Computer Graphics Forum, Proceedings Eurographics*, 11(3):153–167, September 1992.

125. R. Yagel, A. Kaufman, and Q. Zhang. Realistic volume imaging. In *Proceedings Visualization '90*, pages 226–231, San Diego, CA, October 1991. IEEE Computer Society Press.

126. R. Yagel, D. M. Reed, A. Law, Po-Wen Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *ACM/IEEE Symposium on Volume Visualization*, pages 55–62, San Francisco, CA, October 1996.

127. R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. OSU-CISRC-3/93- R10,, Department of Computer and Information Science, The Ohio State University, March 1993.

128. K. Z. Zuiderveld, A. H. J. Koning, and M. A. Viergever. Acceleration of ray casting using 3D distance transform. In R. A. Robb, editor, *Proceedings of Visualization in Biomedical Computing*, pages 324–335, Chapel Hill, NC, October 1992. SPIE, Vol. 1808.