Occlusion Culling and Hardware Accelerated Volume Rendering

Dissertation

der Fakultät für Informatik
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. **Michael Meißner**
aus Tübingen

Tübingen
2000

Tag der mündlichen Qualifikation:	20.12.2000
Dekan:	Prof. Dr. A. Zell
1. Berichterstatter:	Prof. Dr. W. Straßer
2. Berichterstatter:	Prof. Dr. W. Rosenstiel

Zusammenfassung

Das Moor'sche Gesetz besagt, daß sich die Leistung von CPUs und die Speicherdichte alle 24 bzw. 18 Monate verdoppeln. Obgleich dieses Gesetz sich über viele Jahre hinweg bewahrheitet hat, sagt es meist nur sehr wenig über die Leistungssteigerung von Gesamtsystemen aus. Unter anderem hat sich in den letzten Jahren ein wesentlich steilerer Leistungsanstieg bei Graphiksystemen gezeigt. Ermöglicht wurde dies durch die enorme Steigerung der Integrationsdichte[1], die Einzelchiplösungen erlaubte, den intensiven Einsatz von SDRAM Speichern und optimiertem Speicherzugriff. Dadurch hat sich die Hardware der Graphiksysteme von teuren Großsystemen zu einem Massenprodukt entwickelt.

Im Gegensatz dazu ist die Volumengraphik ein reiner Nischenmarkt mit sehr hohen Anforderungen an Bildqualität, Flexibilität und Speicherbandbreite. Bis heute ist es nicht möglich, diese Anforderungen mit allgemeiner Hardware oder Oberflächengraphikhardware in akzeptabler Weise zu erfüllen. Daher bedarf es noch immer speziell entwickelter Hardware, um qualitativ hochwertige Bilder bei interaktiver Bildwiederholrate oder in Echtzeit zu erstellen.

In dieser Doktorarbeit werden algorithmische Optimierungen vorgestellt, die es erlauben, bedeutende Leistungssteigerung durch eine wesentlich verbesserte Ausnutzung der vorhandenen Bandbreiten zu erreichen. Darüberhinaus werden zum einen Verbesserungen in den Datenpfaden vorhandener Architekturen vorgestellt, die deren Engpässe umgehen bzw. reduzieren, zum anderen neue dedizierte Architekturen vorgestellt.

Für die Oberflächengraphik werden Mechanismen zur Integration von Verdeckungsanfragen vorgestellt, die es ermöglichen, unsichtbare Geometrie zu verwerfen, ehe man sie an die Bildgenerierungspipeline weiterleitet, und somit Bandbreite auf Seiten des "Front bus" spart. Als eine logische Ergänzung dazu wird ein neuartiges Rasterisierungsverfahren vorgestellt, das nur Fragmente in sichtbaren Bildbereichen erzeugt und die verdeckten Bereiche überspringt. Auf diese Weise wird die Pipeline effizienter genutzt und es können entweder mehr Objekte dargestellt oder mehr Zyklen pro potentiell sichtbarem Objekt verwendet werden (multi-pass).

Für die Volumengraphik werden erstmalig die meist verwendeten Algorithmen im direkten Vergleich analysiert und deren Stärken und Schwächen gegenübergestellt. Desweiteren werden neue Techniken präsentiert, die unter Verwendung von Oberflächengraphikhardware und Multi-pass Verfahren Beleuchtungsrechnung und Klassifizierung von Volumendaten ermöglichen. Ergänzend dazu werden Veränderungen am Datenpfad vorgeschlagen, um Beleuchtungsrechnung und Klassifizierung in einem einzigen Durchgang zu erreichen. Ein weiterer Beitrag ist der effiziente Einsatz einer neuartigen allgemeinen Einzelchip SIMD Architektur für die Volumenvisualisierung. Die-

[1] Board-to-chip Integration.

se Lösung bietet einen höheren Grad an Flexibilität als Oberflächengraphikhardware und ist dennoch in der Lage, mehrere Bilder pro Sekunde bei vergleichbarer Qualität darzustellen. Als logischer Konsequenz aus den obigen Lösungsansätzen wird eine neue und kostengünstige Architektur (Spezialhardware) für die Volumenvisualisierung präsentiert. Diese verbindet überlegene Bildqualität mit einem hohen Grad an Flexibilität durch die Verwendung rekonfigurierbarer Hardwarebausteine. Abschließend wird die Softwareumgebung, mit deren Hilfe die überwiegende Anzahl der in dieser Dissertation vorgestellten Resultate erarbeitet wurde, vorgestellt.

Abstract

Moore's Law states that the processing power of CPUs and the capacity of memory chips doubles every 24/18 months. While this law has proven to be valid for many years, the overall system performance does not necessarily adhere to this law. But computer graphics hardware has broken this law by surpassing performance growth over the past years. This has been due to the dramatic board-to-chip integration, the intensive use of SDRAM, and improved memory access patterns. At the same time, dedicated polygon graphics hardware has evolved from expensive large scale systems to a single chip commodity product.

In contrast, volume rendering is a niche market with high demands in image quality, flexibility, and memory bandwidth. So far, general purpose graphics hardware has not been capable of satisfying these demands to an acceptable degree. Therefore, special purpose hardware is required to accomplish high-quality images at interactive or real-time frame-rates.

Within this dissertation, a set of algorithmic optimizations are developed, enabling significant performance improvements due to a much better utilization of the available bandwidth. Additionally, new architectural concepts circumventing the bottle-necks of currently available general purpose graphics hardware are presented.

In the field of polygon rendering, a unique mechanism for hardware supported occlusion queries to cull geometry prior to geometry transformation — saving bandwidth on the front bus — is presented. As an orthogonal addition to this, a novel visibility driven rasterization scheme is presented, saving processing cycles within the pipeline by culling occluded geometry prior to rasterization. Thus, more objects can be rendered or more cycles can be spent on multi-pass rendering of the potentially visible objects.

With respect to volume rendering, this dissertation contributes the first side by side comparison of different volume rendering algorithms identifying each algorithm's strengths and weaknesses. Furthermore, new techniques for using polygon graphics hardware and multi-pass rendering are presented, enabling the combination of shading and classification of volume data. Additionally, minor modifications to the data path are proposed such that multi-pass rendering can be avoided, thus increasing the overall achievable frame-rate. Furthermore, we demonstrate how to efficiently use general purpose hardware (a single-chip SIMD architecture) for volume rendering, providing much more flexibility than dedicated polygon graphics hardware. As a summary of the above described work, a novel low-cost special purpose hardware architecture that achieves superior image quality while providing an incomparable degree of flexibility is presented. Last but not least, Appendix A presents the rendering environment used to produce most of the results of this dissertation.

Acknowledgements

The work described in this dissertation would not have been possible without the support and encouragement of many people. First of all, I would like to thank my advisor professor Straßer and professor Ruder head of the SFB 382[2] for the tremendous degree of freedom I received to work on different topics and the support to cooperate with various industrial partners. I would also like to thank my colleagues at the WSI/GRIS for the collaborations and helpful discussions, namely Dirk Bartz, Edelhard Becker, Michael Doggett, Johannes Hirche, Tobias Hüttner, Urs Kanus, Anders Kugler, Gregor Wetekam, as well as our secretary Adelheid Ebert, and our system administrators Helga Mayer and Jürgen Fechter for providing the necessary "backbone".

Nevertheless, only few of this would still have been possible without the numerous master thesis and student projects of highly motivated graduate and undergraduate students, as there were: Sven Fleck, Richard Günther, Michael Guthe, Soeren Grimm, Ulrich Hoffman, Markus Janich, Alexander Ott, Marcus Schiesser, and many others. It has always been an exciting experience to cooperate with students and get them involved in interesting projects.

Furthermore, I would like to thank all industrial cooperation partners, my international tutorial co-organizers, as well as other people who gave me support and helpful hints for my work; Phil Atkin (PixelFusion), Roger Crawfis (Ohio State University), Urs Kanus (DD&T), David Kirk (nVidia), Jian Huang (Ohio State University), Tom Malzbender (HP labs), Klaus Mueller (Ohio State University), Hanspeter Pfister (MERL), Roland Proksa (Philips Research), Rüdiger Westermann (University of Aachen), and Craig Wittenbrink (HP labs).

Last but not least, I would like to express my gratitude for all kinds of non measurable support and experiences in life to my family as well as to my girl friend Vanja Kvarnstroem who sacrificed numerous weekends such that I could work overtime or attend conferences abroad.

[2]Special research grant for methods and algorithms for simulation of physical processes on high performance computers of the german research council.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *Graphics provides one of the most natural means of communicating with a computer, since our highly developed 2D and 3D pattern-recognition abilities allow us to perceive and process pictorial data rapidly and efficiently. In many design, implementation, and construction processes today, the information pictures can give is virtually indispensable. (...)*
> *Interactive computer graphics is the most important means of producing pictures since the invention of photography and television; it has the added advantage that, with the computer, we can make pictures not only of concrete, "real-world" objects but also of abstract, synthetic objects, such as mathematical surfaces of 4D, and of data that have no inherent geometry, such as survey results. Furthermore, we are not confined to static images. Although static pictures are a good means of communicating information, dynamically varying pictures are frequently even better — to coin a phrase, a moving picture is worth ten thousands static ones.*
>
> James Foley and Andries van Dam, Introduction to *Fundamentals of Interactive Computer Graphics*, 1982.

Despite the fact, that this statement was made 20 years ago, it still holds for what computer graphics is all about today. Then, interactive computer graphics was limited to expensive machines while now it is widely available on almost every PC. To create the perfect illusion of virtual three dimensional worlds or to visualize abstract information as meaningful images, image synthesis aims for interactivity and highest image quality.

Within this chapter, the basic concepts of polygon and volume rendering are summarized and common terminology used throughout this dissertation is introduced. Thereafter, the chapter outline is presented.

1.1 Polygon Rendering

Polygon rendering is the synthesis of images from three dimensional scenes which are represented as surfaces. Instead if using an implicit object description, surfaces are approximated by a set of polygons. The advantage of a polygonal approximation is that it enables the development of dedicated graphics hardware capable of generating images at interactive frame-rates. Indistinguishable and appealing images can be generated using a sufficiently detailed approximation of the surface.

In computer graphics, the goal of image synthesis is photo-realism to generate a true illusion. However, there are a number of issues in polygon rendering which prevents it from reaching this goal. The most difficult one is the true simulation of global light effects such as refraction, transmission, reflection, and resulting shadows. Therefore, many approaches have been developed to approximate this behavior trading accuracy for interactivity.

1.1.1 Illumination and Shading

Illumination and shading is a technique to greatly enhance the appearance of a geometric object that is being rendered. Shading tries to model effects like shadows, light scattering, and atmospheric attenuation. Generally, there are local, direct, and global illumination models. Local illumination models perform shading considering only the position of light source(s) and observer relative to the surface. Hence, occluded light sources have the same impact as visible ones. Even though this is far from photo-realism, it produces fairly good results and is the most frequently used illumination technique in polygon rendering. In contrast to local illumination, global illumination simulates the exchange of light between all objects in a scene. Hence, a point appears dark if the light is occluded from this location and bright if the light is directly visible or in case light is reflected onto it. Unfortunately, global illumination is quite expensive and not yet feasible at interactive frame-rates. A good compromise between local and global illumination are direct illumination methods. Instead of examining the global exchange of light, for each point all light sources are probed for the directly incoming light. This technique allows the integration of shadow effects while usually being less costly than solving the global illumination[1].

There are two shading models used within polygon rendering: Gouraud and Phong Shading. While Gouraud shading performs the illumination calculation on a per vertex base interpolating intensities across a triangle, Phong shading interpolates the surface normal and performs illumination on a per pixel base. The problem with Phong shading is that the interpolated normal needs to be normalized and mapped back into the world coordinate system before the illumination can be computed. This is quite expensive and the reason why current graphics hardware supports Gouraud shading. Gouraud and Phong Shading use local illumination consisting of an ambient, a diffuse and a specular component. While the ambient component is present at each position in the scene, the diffuse component can be computed using the angle between the normal vector at the given position and the vector to the light, simulating a lambertian reflector. In contrast, the specular component depends on the angle between the light and the eye position and specifies how much of a light source's intensity is reflected. All three components can be combined by weighting each of them differently, using material properties. Additionally, light attenuation can be integrated to provide depth cueing.

[1] Direct illumination can be accomplished e.g. using projective textures.

1.1.2 Geometry Transformation

The geometric part of the rendering process consists of several coordinate systems. While objects are usually modeled in local coordinates, they are assembled to an entire model or scene in world coordinates. Both steps are performed during the modeling of a scene which is performed using CAD programs. Furthermore, light sources and surface attributes such as texture, color, etc. are defined in world space.

In contrast to the local and world coordinate systems, the camera coordinate system defines which part of the scene is displayed. It can be positioned anywhere in world space and in any direction and orientation. However, graphics hardware does not use the camera coordinate system to synthesize images, simply because it would require complex floating point operations which would be extremely expensive. Therefore, the image synthesis is performed in yet another coordinate system, the 3D or 2D screen space coordinates. Generally, viewing transformations are performed in homogeneous coordinates since all transformations on points and vectors can be handled as matrix multiplication, including translation. To obtain a final 3D screen coordinate, the homogeneous coordinates need to be converted into Cartesian coordinates performing the division by W. However, per vertex lighting needs to be performed before the perspective viewing transformation.

Geometry transformation also includes the clipping of the transformed triangles. This can be done in the final 3D screen space coordinates but there are also approaches to perform this in homogeneous coordinates. Polygons which do not intersect with the view frustum can be eliminated using trivial reject mechanisms. The ones which are partially inside/outside the view frustum need to be clipped properly. Finally, the triangle setup needs to be performed by means of computing the partial increments in x and y of all parameters needed across the triangle. This includes the increment of the color components (Gouraud shading), the opacity, the depth value, and the texture coordinates. The values of the triangle setup are passed on to the rasterization in fixed-point format which suffices due to the axis aligned screen space coordinate system (image plane is equal to the x/y-plane).

Until very recently, geometry transformation was done by the CPU and not supported by the graphics hardware. This has been due to the required floating point precision, however, modern graphics hardware also performs geometry transformation.

1.1.3 Rasterization

Rasterization is the process of converting transformed primitives (triangles) into pixel values. To keep rasterization simple with respect to the required hardware, it is implemented as incremental process avoiding expensive multiplications. Rasterization is usually referred to as scan conversion since the primitives are traversed and converted in scanline order.

Scan Conversion

While scan-converting a triangle, the color-, alpha-, and depth-value as well as texture coordinates are generated for each pixel. Different triangle conversion schemes exist, keeping the number of idle cycles as low as possible when moving from one scanline to the next. Recently, scanline based rasterization has been extended to stamp based rasterization; instead of moving from pixel to pixel, e.g. 2×2 pixels are processed in parallel moving the stamp across the triangle in scanline based order. This process

exploits the texture and framebuffer coherence better than simple scanline based rasterization. Furthermore, it works well with modern SDRAM devices where graphics hardware exploits the fast access within the opened page before moving to the next page.

Texture Mapping

Texture mapping is the process of mapping a texture (e.g., an image) onto a surface. By defining texture coordinates on a per vertex base, interpolated texture coordinates are used to determine the footprint of the pixel in the texture[2]. Using the texture footprint, a color- and possibly alpha-value can be filtered from the texture. To prevent aliasing artifacts, textures can be generated at different resolutions and the appropriate texture level is selected by the hardware, depending on the partial increments of the texture coordinates. In addition to the color which is interpolated across the triangle (Gouraud shading), texture mapping generates a second color. Different blending operations can be selected to combine the two colors to a final fragment color.

While texture mapping was primarily introduced to map textures onto the surface such that objects made of wood or marble can be displayed realistically, many other applications have evolved. One can use textures to add shadow effects to an object by pre-computing shadow textures. Reflections can be simulated by rendering a scene from one camera — e.g. the floor — and using the resulting image as texture for the floor such that it looks like a shiny and reflecting mirror.

Mainly driven by game developers exploiting texture mapping to achieve more realistic images, computer graphics hardware developers have added multi-texture capabilities such that more than one texture can be applied per vertex. In extension to two dimensional textures, three and four dimensional textures are now supported. E.g., three dimensional textures can be used to carve a surface object out of a texture block which can be used for volumetric effects.

1.1.4 Framebuffer Operations

The result of the rasterization stage are fragments which consist of color and alpha information as well as the two dimensional pixel coordinates. Within the framebuffer stage, several per fragment operations can be applied to the fragments and tests are performed to determine whether a fragment should be discarded. These test include scissor-, alpha-, stencil-, and depth-test. Only in case that all these tests are passed, the fragment color is combined with the color stored in the framebuffer.

The *Scissor test* can be used to define a rectangular screen space area and only fragments within this area pass. More important is the *Alpha test* allowing to discard fragments which have an alpha-value that does not pass the alpha function (usually a threshold). In contrast to the scissor test, the *stencil test* can be used to enable or disable individual pixels from being processed. The most important test is the *depth test* which enables hidden surface removal. In case the depth value of a fragment passes the depth test — e.g. being smaller than the previously stored value — then the color-, alpha, and depth-value are replaced. However, instead of replacing color and alpha, one can also combine the values using any of the possible blending operations. This is useful to enable semi-transparent surfaces. Finally, there is a *dithering* stage as well as some other logic and masking operations before the fragments color is written into the actual framebuffer.

[2] Generally, texture coordinates need to be corrected to account for perspective distortion.

1.1.5 Summary

Interactive computer graphics based on polygon rendering has emerged into numerous application fields from where it is indispensable. The goal of future graphics hardware development is two-fold. On the one hand, the overall rendering bandwidth is to be increased such that higher resolution displays can be driven as well as more complex scenes be rendered. This includes the need for better algorithms to determine the potentially visible polygons since for depth complex scenes the hidden surface removal performed in the hardware is very expensive. On the other hand, it is important to develop algorithms which can easily be mapped into hardware allowing a higher degree of realism such as displacement mapping or per pixel lighting.

1.2 Volume Rendering

Using surface primitives for image synthesis is a very powerful approach but when looking beyond the objects surface, nothing will be behind or inside simply because only the hull of the object is available and modeling all interior structures would be a highly challenging task.

In contrast to polygon rendering, volume rendering is the synthesis of images from structures which are represented as three dimensional sampled volume. Since volume rendering deals with scanned, simulated, or measured data, photo-realism is not the goal. In fact, the task is to synthesize meaningful images from volumetric data such that the resulting two-dimensional image reveals useful insights to the user. Hence, the difference between polygon and volume rendering are the underlying primitives as well as the rendering approaches. Despite these differences, volume rendering borrows rendering techniques such as shading and blending.

1.2.1 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 be applied to color the usually scalar data and to assign a certain opacity to the different structures (transparent, semi-transparent, or opaque) and hence, it 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 are other examples 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 these acquisition devices is the missing color information since scalar values represent 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 can not be captured by the earlier mentioned acquisition devices. A well-known example is the visible human project where this technique has been applied to a male and a female cadaver. However, color is usually added during the visualization process.

Geoseismic data is probably one of the sources that generates the largest amount 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 reduced tremendously by finding the right location where to drill the hole.

Another source of volumetric data are 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.

1.2.2 Grid Structures

Depending on the source from which the volumetric data origins, it can be given as a Cartesian rectilinear grid, curvilinear grid, or maybe even completely unstructured. While scanning devices mostly generate rectilinear grids (isotropic or anisotropic),

physical simulations mostly generate unstructured data. Figure 1.1 illustrates these different grid types for the 2D case. Depending on the underlying grid topology, dif-

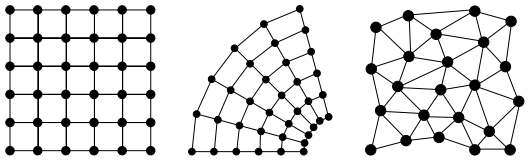


Figure 1.1: Different grid structures: (a) Rectilinear. (b) Curvilinear. (c) Unstructured.

ferent algorithms can be used for rendering. While rendering on rectilinear grids is referred to as volume rendering, rendering of curvilinear and unstructured grids is referred to as scientific visualization. Within this dissertation, the focus is on rectilinear grids.

1.2.3 Classification

Classification is the stage that 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 polygon rendering.

In the classification stage, certain properties are assigned to a sample such as color and opacity. Additionally, shading parameters indicating how shiny a structure should appear can be assigned (material properties). Finding the right transfer function for the opacity can be a very complex operation and has a major impact on the final 2D image. In order to find the right transfer function(s), it is usually helpful to use histograms illustrating the distribution of voxel values within the dataset.

The actual assignment of color, opacity, and other properties can be based on the voxel value only, but other values can be taken as input parameters as well. Using the gradient magnitude as further input parameter, samples within homogeneous space can be interpreted differently than the ones within heterogeneous space [Lev88]. This is a powerful technique when visualizing geoseismic data where the scalar values only change noticeably in between different layers of the ground.

1.2.4 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 — which similarly absorb X-rays — are mapped onto the same scalar value. Therefore, no classification of density values can be found such that structures of similar absorption properties can be properly separated. To separate such structures, the voxels need to be labeled to possibly differentiate them during the rendering process, requiring higher order knowledge. This process is referred to as segmentation and for each segment, a different classification can be applied.

Depending on the acquisition method and the scanned object, it can be relatively easy, hard, or even impossible to segment some of the structures automatically. Most

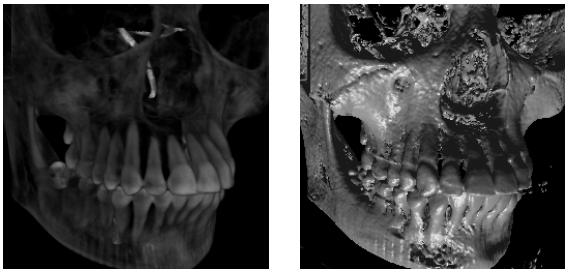
algorithms are semi-automatic or optimized for segmenting a specific structure where higher order knowledge about the shape can be exploited.

1.2.5 Illumination and Shading

Illumination and shading within volume rendering refers to the same illumination models and shading techniques as used in polygon rendering. The goal is to enhance the appearance of rendered objects by simulating the effects of light interacting with the object (see Section 1.1.1). Therefore, volume rendering borrows these techniques from polygon rendering applying them to volumetric objects instead of surface elements.

As mentioned earlier, ambient, diffuse, and specular light components are computed and combined by weighting each of them differently. The weighting depends on the material properties. While tissue is less likely to result in reflected light, teeth might reflect more light. Thus, it is important not to use material properties as global constants but to include them in the classification such that the different structures in the volume can be assigned different material properties.

To show the importance of illumination for the perception of rendered images, Figure 1.2 shows a skull without and with local illumination. In Figure 1.2 (b), the bone is classified white using a strong specular component.



(a) (b)

Figure 1.2: Comparison of shading: (a) No illumination. (b) Local illumination.

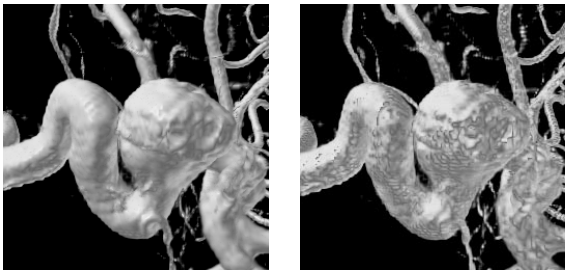
1.2.6 Gradient Estimation

In order to perform shading, a normal is required to compute the diffuse and specular components. 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, which can be used for shading.

A frequently used gradient operator is the central difference operator. For each dimension of the volume, the central difference of two neighboring voxels of a voxel is computed which is an approximation of the local change of the gray value. Its filter kernel can be written as $Gradient_{x,y,z} = [-1 \quad 0 \quad 1]$. Generally, the central difference operator is a good low-pass filter and relatively cheap to compute since it requires only six voxel values and three subtractions. However, very narrow structures can be missed due to the central difference [LCN98]. Furthermore, the central difference gradient

operator does not produce isotropic gradients which can be troublesome when using the gradient magnitude as further input parameter to assign opacities.

The intermediate difference operator is very 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 the intermediate difference operator is that it detects high frequency detail which can be lost when using the central difference operator. However, this also leads to less appealing images when rendering datasets with a lot of noise. Similar to the central difference gradient, the intermediate difference gradient is not isotropic. Figure 1.3 illustrates the difference of central and intermediate difference operator for a noisy dataset containing an aneurism.



(a) (b)

Figure 1.3: Two renderings of an aneurism dataset with high frequency noise using different gradient estimation schemes: (a) Central difference gradient. (b) Intermediate difference gradient.

A much better gradient operator is the Sobel operator which uses all 26 voxels that surround one voxel. The Sobel gradient operator was developed for 2D imaging but can easily be extended to 3D and applied to volume rendering. A nice property of the Sobel gradient operator is that it produces nearly isotropic gradients but is expensive to compute. It requires 27 voxel values, 54 multiplications, and 51 subtractions [Lic97].

A more detailed comparison of the resulting image quality of different gradient estimation schemes can be found in [THB+90] and different gradient estimation schemes are compared in [Ben95].

1.2.7 Compositing

Compositing is the stage where all contributions to a pixel are combined into one final pixel value. This can be expressed as an approximation of the well-known low-albedo volume rendering integral, VRI [Bli82, Kru91, KH84, Max95]. The VRI analytically computes $I_\lambda(x,\vec{r})$, the amount of light of wavelength λ coming from ray direction $\vec{r}$ that is received at location x on the image plane:

$$I_\lambda(x,\vec{r}) \quad = \quad \int_o^L C_\lambda(s)\mu(s)e^{(-\int_0^s \mu(t)dt)}ds \tag{1.1}$$

Here, L is the length of ray $\vec{r}$. If thinking of the volume as being composed of particles with certain densities (or light extinction coefficients [Max95]) 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 Equation 1.1, C_λ is the light of wavelength λ reflected and/or emitted at location s in the direction of $\vec{r}$. To account for the higher reflectance of particles with larger densities, one must weigh the reflected color by the particle density. The light scattered at location 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 can not be computed analytically [Max95]. Hence, practical volume rendering algorithms discretize the VRI into a series of sequential intervals i of width Δs.

$$I_\lambda(x,\vec{r}) \quad = \quad \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\mu(s_i)\Delta s \prod_{j=0}^{i-1} e^{(-\mu(s_j)\Delta s)} \tag{1.2}$$

Using a Taylor series approximation of the exponential term and dropping all but the first two terms results in the familiar compositing equation [Lev90].

$$I_\lambda(x,\vec{r}) \quad = \quad \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\alpha(s_i) \prod_{j=0}^{i-1} (1-\alpha(s_j)) \tag{1.3}$$

This is generally denoted as discretized VRI (DVRI), where the opacity is given as $\alpha = 1.0 - \textit{transparency}$. Due to the non linear behavior of the DVRI, all steps need to be performed in sorted order.

For the front to back case, the discrete volume rendering integral can be written as:

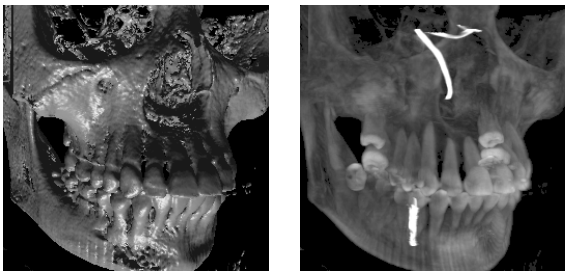
```
Trans = 1.0;  -- full
Inten = I[0]; -- initial value
for (i=1; i<n; i++) {
   Trans *= T[i-1];
   Inten += Trans * I[i];
}
```

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

For back to front order, compositing is much less work since it is not necessary to keep track of the remaining transparency. However, it requires that all samples are 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 volume rendering integral, one can chose other compositing operators. Another famous operator simply takes the maximum density value of all samples of a pixel and is known as maximum intensity projection (MIP). This is frequently used in medical applications dealing with MRI data (magnetic resonance angiography) visualizing arteries that have been acquired using contrast agents. Two further common operators are (i) averaging all values which contribute to one pixel or (ii) finding the first value which is above a given threshold. The former one results in X-ray like images and the latter can be accomplished by applying a binary opacity classification. Figure 1.4 illustrates the volume rendering integral with almost binary classification versus maximum intensity projection.



(a) (b)

Figure 1.4: Compositing operators: Volume rendering integral with almost binary classification (a) and maximum intensity projection (b).

1.2.8 Filtering

Many volume rendering algorithms resample the volumetric data in a certain way using rays, planes, or random sample points. These sample points rarely coincide with the actual grid positions and require the interpolation of a value based on the neighboring values at grid positions. There are numerous different interpolation methods and each of them is controlled by an interpolation kernel. The shape of the interpolation kernel provides the coefficients for the weighted interpolation sum. One dimensional interpolation kernels can be applied to interpolate in two, three, and even more dimensions if the kernel is separable. In the following, the most frequently used interpolation kernels — which are all separable — are presented.

The nearest neighbor interpolation is the simplest and crudest method. The value of the closest of all neighboring voxel values is assigned to the sample which results more in a selection than filtering. Therefore, when using nearest neighbor interpolation, the image quality is fairly low and when using magnification, a blobby structure appears.

Trilinear interpolation assumes a linear relation between neighboring voxels. The achievable image quality is much higher than with nearest neighbor interpolation. However, when using large magnification factors, three dimensional crosses (diamonds) 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 [Ben95]. However, there is a trade-off between quality and computational cost as well as memory bandwidth. Cubic convolution and B-spline interpolation require a neighborhood of 64 voxels and a significant larger amount of computations than trilinear interpolation. Thus, trilinear interpolation is usually a good trade-off with respect to achievable image quality and computational costs.

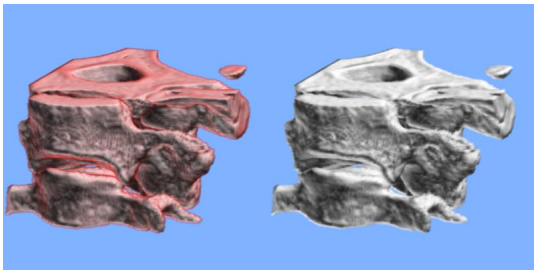
1.2.9 Color filtering

The previously described classification and filtering steps can be performed in different order, resulting in different image quality and characteristic artifacts. Generally, both approaches — interpolation of data or color — are prone to aliasing if no appropriate sampling frequency is applied. Additionally, the different processing order can possibly introduce artifacts.

Interpolating scalar values can result in interpolated values which could be classified as structures which are not at all present in the data. This can only be circumvented with a proper segmentation of the data but introduces high frequencies. On the other hand, color interpolation by 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 independently from each other [WMG98]:

$$\begin{aligned} C_\lambda &= \sum_{i=0}^{N-1} w_i * C_{\lambda,i} \\ \alpha &= \sum_{i=0}^{N-1} w_i * \alpha_i \end{aligned} \tag{1.4}$$

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 the corresponding color volume, the interpolation of Equation 1.4 leads to color bleeding, as illustrated in Figure 1.5(a). To obtain the correct color, one needs to multiply each color with the



(a) (b)

Figure 1.5: Color bleeding: (a) Independent interpolation of color and opacity values. (b) Opacity weighted color interpolation. Images are courtesy of [WMG98].

corresponding opacity value before interpolating the color:

$$C_\lambda = \sum_{i=0}^{N-1} w_i * \alpha_i * C_{\lambda,i} \tag{1.5}$$

The result is shown in Figure 1.5(b). While the color bleeding effect can be noticed quite easily in colored images, it appears less obvious in grey-scale images where it introduces darkening artifacts.

1.2.10 Summary

Within this section, an overview of the core terminology and elements of volume rendering have been presented. Varying classification, gradient estimation, shading, and compositing results in extremely different visualizations. Also the selection of the filter used to interpolate data or color has a strong influence on the resulting image. Depending on the target application, different combinations might be better than others and one needs to be aware of the potential artifacts and misinterpretations.

1.3 Outline

The remainder of this thesis is divided into two parts, one for polygon rendering and another for volume rendering.

Part A presents approaches for the efficient utilization of graphics hardware for rendering. With the growing complexity and popular multi-pass rendering approaches, the overall rendering bandwidth needs to be utilized efficiently. Besides view frustum culling and visibility determination, occlusion culling has evolved as a popular technique. While the available mechanisms are fairly limited, a simple low-cost mechanism is presented, extending OpenGL and the graphics hardware such that sophisticated occlusion queries can be performed (Chapter 2). Furthermore, a visibility driven rasterization scheme using a visibility mask is presented. It extends conventional rasterization by discarding objects and parts of objects which belong to occluded screen space regions (Chapter 3). The advantages of visibility driven rasterization are its simplicity and the fact that it fits well into stamp or tile based rasterization, requiring only few modifications.

In part B, a side by side comparison and analysis of the four most popular volume rendering algorithms is presented (Chapter 4). Since interactivity is mandatory for most volume rendering applications, different avenues can be taken to trade quality versus frame-rate. Thereafter, the limitations of texture mapping based volume rendering are surveyed and it is shown how on-the-fly shading and classification can be accomplished within the available graphics hardware using OpenGL and extensions only (Chapter 5). Since, the required multi-pass rendering reduces the frame-rate, slight modifications of the datapath within polygon graphics hardware are proposed such that true and high-quality volume rendering could be accomplished in a single pass.

Another possible avenue for interactive volume rendering is parallel computing. While most parallel systems are expensive, an optimized parallel ray caster has been implemented on a low-cost single-chip SIMD architecture (Chapter 6). The programmability enables higher flexibility than using dedicated graphics hardware (OpenGL), but the memory interface remains one of the most troublesome issues.

To circumvent the limitations of general purpose hardware approaches, we developed our own special purpose volume rendering accelerator with superior image quality and flexibility (Chapter 7). Based on the results of Chapter 4, ray casting has been selected as the algorithm of choice and a hardware architecture for high quality image synthesis has been designed. The architecture is very flexible due to the utilization of reconfigurable hardware elements (field programmable gate arrays) and fits into a low-cost PCI card which can be plugged into any off the shelve PC. In contrast to other approaches, the architecture supports true ray casting which is mandatory for immersive applications and stereoscopic display.

Finally, Appendix A presents a cross-platform rendering environment based on OpenGL and the Qt library. This library has been used for most of the work presented throughout this dissertation.

Part A:

Polygon Rendering

Chapter 2

Occlusion Culling Hardware

Hidden-line-removal and visibility are among the classic topics in computer graphics and a large variety of algorithms are known to solve these visibility problems. While the z-buffer approach is very simple to implement and supported by any state-of-the-art graphics card, it is extremely costly to solve overall visibility since it operates on a per pixel base. For each pixel, all stages of the graphics pipeline need to be passed — e.g. geometry transfer, geometry transformation, rasterization, texturing, etc. — before finally performing the depth test. Especially when rendering scenes with high occlusion depth[1], the z-buffer approach performs poor since only a small part of the overall used bandwidth is spend on finally visible pixel.

Over the past years, many techniques have been presented to cull geometry prior to sending it to the graphics hardware. The most obvious one is view-frustum culling which culls geometry outside the view frustum. Other approaches cull geometry that is not visible. Generally, this can be accomplished using an aspect graph but for large scenes this graph is fairly complex and expensive to compute. It is also troublesome to update such a visibility graph for dynamic scenes. Therefore, another avenue that can be taken is to not attempt to compute visibility but to determine objects which are definitely occluded. Even though interactive frame-rates have been reported for specific cases, real-time frame-rates are hard to accomplish for the general case. For interactive rendering of large polygonal objects, fast visibility queries are necessary to quickly decide whether polygonal objects are occluded or need to be rendered. None of the numerous published algorithms provide visibility performance for interactive rendering of large models.

Within this chapter, a hardware mechanism for efficient occlusion culling support is presented, providing more detailed query information then other approaches. Furthermore, extensions to the OpenGL API are presented to ensure the wide availability of such a hardware mechanism.

[1] Number of triangles contributing to a screen pixel. A high number indicates a high occlusion depth and vice versa.

2.1 Introduction

Visibility has been of special interest for walkthroughs of architectural scenes [ARB90, TS91] and rendering of large models [L. 97, GBW90, ZMHH97]. Unfortunately, most of these approaches are limited to cave-like scenes [L. 97] or do not provide interactive rendering (more than 10 frames/second) of large models on mid-range graphics hardware [ZMHH97]. There are several papers which provide a survey of visibility algorithms. In [ZMHH97], Zhang provides a brief recent overview with some comparison. Brechner surveys methods for interactive walk-throughs [Bre96]. In the following, previous work on visibility algorithms that can possibly be implemented in hardware is summarized.

Occlusion culling is a technique to cull geometry in order to accelerate the rendering process by removing redundant work from the graphics hardware. One of the first hardware supported mechanisms of this type has been the PixelFlow system [MEP92]. Even though all geometry needs to be processed with respect to transformation, rasterization, and texturing, shading is performed for visible fragments only. This process is called *deferred shading* and enables the hardware to perform the expensive shading operation on visible fragments only[2]. However, all other pipeline stages still need to process all fragments.

Greene et al. proposed the hierarchical z-buffer algorithm [GKM93, Gre95]. After subdividing the scene into an octree, each of the octants is culled to the view-frustum as proposed in [GBW90]. Thereafter, the silhouettes of the remaining octants are scan-converted into the framebuffer to check if these blocks are visible. If they are visible, their content is assumed to be visible too; if they are not visible, nothing of their content can be visible. The visibility query itself is performed by checking a z-value-image-pyramid for changes. Usually, the respective levels of the z-value-image-pyramid are searched for z-value changes, a feature which is commonly not supported in hardware. In [GKM93], a hardware implementation of this query on a Kubota Pacific Titan 3000 workstation using a Denali GB graphics hardware is discussed. Still, most time of the visibility query is spent performing this "Z query".

Xie and Shantz suggested a simplified two-level hierarchical z-buffer approach that is more suitable for implementation in hardware [XS99]. The update of the hierarchy is performed only once per frame. However, all the hierarchical z-buffer related approaches increase the hardware complexity significantly.

Zhang et al. presented hierarchical occlusion maps [ZMHH97]. An occluder database is selected from the scene database. Using these occluders, screen bounding boxes of the potential occludees of the scene database are tested for overlaps, using an image hierarchy of the projected occluders (hierarchical occlusions maps). Basically, two features of this algorithm are supported in hardware. First, the construction process of the hierarchical occlusion maps can be supported by modern texture-mapping hardware. Second, the alternative use of a z-buffer as the depth estimation buffer for the overlap test.

In December 1997, Hewlett-Packard proposed an OpenGL extension for occlusion

[2] PixelFlow is based on SIMD arrays of processing elements which perform most operations. Programmable shading can therefore require many processing cycles which makes it an *expensive operation*.

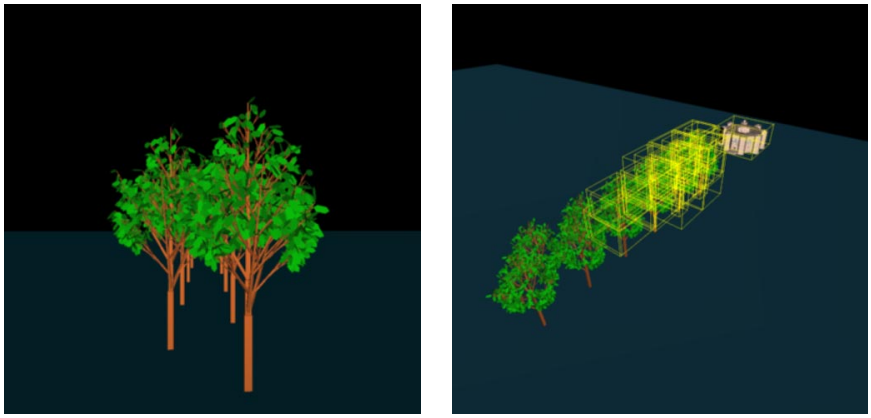
culling [HP97]. Similar to the hierarchical z-buffer approach, graphic primitives, which represent a more complex geometry, are rendered within an occlusion test mode to determine their visibility. Depending on the result, all underlying geometry is rendered or skipped. Shortly after this announcement, an almost identical approach has been presented by SGI and is supported on their Visual PC [SGI99]. However, no hardware details were disclosed.

The work presented in this chapter is based on the same idea as Hewlett-Packard's OpenGL extension for occlusion culling [HP97] and therefore very similar. However, it has been developed independently from what was proposed by Hewlett-Packard, goes much further, and was published slightly after the announcement of Hewlett-Packard [HMB98, BMH98, BMH99].

2.2 General Approach

Given a three dimensional scene of polygons, one can find a spatial partitioning scheme such that each of the partition entities contains a subset of polygons. The partitioning scheme can either be disjoint (octree) or allow for spatial overlaps of the partition entities (sloppy partitioning [MBH+99]). Instead of testing whether the actual polygons are occluded, one can use the conservative assumption that polygons contained within a partitioning entity must be occluded in case the partitioning entity itself is occluded [GKM93, Gre95]. The advantage of this approach is the reduced complexity of the problem, potentially resulting in an overall slightly reduced culling efficiency due to the chosen granularity.

In summary, the rendering process consists of the following steps: First, the partitioning entities are processed by performing view-frustum culling. Second, the partitioning entities are tested for occlusion. Finally, all polygons contained in the remaining not occluded entities are sent to the graphics hardware for rendering. Figure 2.1



(a) (b)

Figure 2.1: Occlusion culling in a tree alley: (a) Initial view. (b) Birds view of (a), occluded objects for view (a) are marked with yellow outlined bounding boxes.

illustrates the impact of occlusion culling: (a) shows the initial view of a tree alley and (b) is a bird's view of what can be seen from (a). All yellow outlined boxes in (b) indicate occluded objects in view (a).

When using such an approach, one needs to be able to quickly determine whether a partitioning entity is occluded. For complex scenes, this is hard to accomplish in software while current graphics hardware does not support such queries[3]. However, visibility information is available in the graphics hardware at the depth test where the comparison result indicates visibility or occlusion.

In earlier experiments, the stencil buffer was used to render partitioning entities searching for footprints of the partitioning entity in the stencil buffer [HMB98, BM99]. For further culling, a visibility ratio was additionally calculated by counting all footprints and dividing this number by the amount of total pixels of the 2D screen space bounding box of the projected entity. This allows to apply *Adaptive Occlusion Culling* which can be used to determine an appropriate level of detail of the geometry contained in the partitioning entity. Due to the expensive access to the stencil buffer[4], it was obvious that dedicated hardware is required for interactive and real-time frame-rates. Hardware support for occlusion culling should provide information about the general visibility of the rendered objects but also more quantitative information such as number of pixels of the projection of the geometry, number of visible pixels, minimum and maximum of *z*-values, and many more [BMH98].

In the following, the necessary extensions to the hardware and to the OpenGL API supporting quantitative queries is presented. The queries include the number of visible pixels and the number of pixels within the projection of the object.

- **Projection Hit Counter (PHC)**. This is used to quantify all pixels which are within the projection of the partitioning entity.
- **Visible Hit Counter (VHC)**. This counts the number of pixels of the projection of the object which pass the depth test.

Furthermore, the queries are restrictable to a certain screen space area such that a screen space subdivision can be realized. The proposed hardware and API extension is not limited to these two counters and could easily be extended for even more sophisticated queries.

2.3 Hardware Implementation

Each triangle that is rendered passes through the pipeline stages of the graphics subsystem. First, a triangle is transformed, clipped, and per vertex lighting information computed. Second, the triangle is scan-converted interpolating texture coordinates, color, alpha, and depth values for each individual pixel. Finally after texturing, the individual pixels are passed through the per fragment pipeline. This stage performs the final operations on the data before the fragments are stored as pixels in the framebuffer. Since the framebuffer update depends on some conditions, some tests which evaluate arriving and previously stored z-values (for z-buffering) have to be carried out. Also, blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values are done in this stage of the pipeline. Figure 2.2 illustrates the per fragment pipeline of OpenGL.

[3] Parallel to this work, Hewlett-Packard announced the occlusion culling flag which provides such a query. This is compared and discussed in Section 2.7.

[4] For this experiment, an SGI O$_2$ has been used. The O$_2$ uses a unified memory architecture where reading of any buffer is relatively fast compared to other graphics hardware. Furthermore, the stencil buffer is the buffer which can be read quickest of all framebuffers.

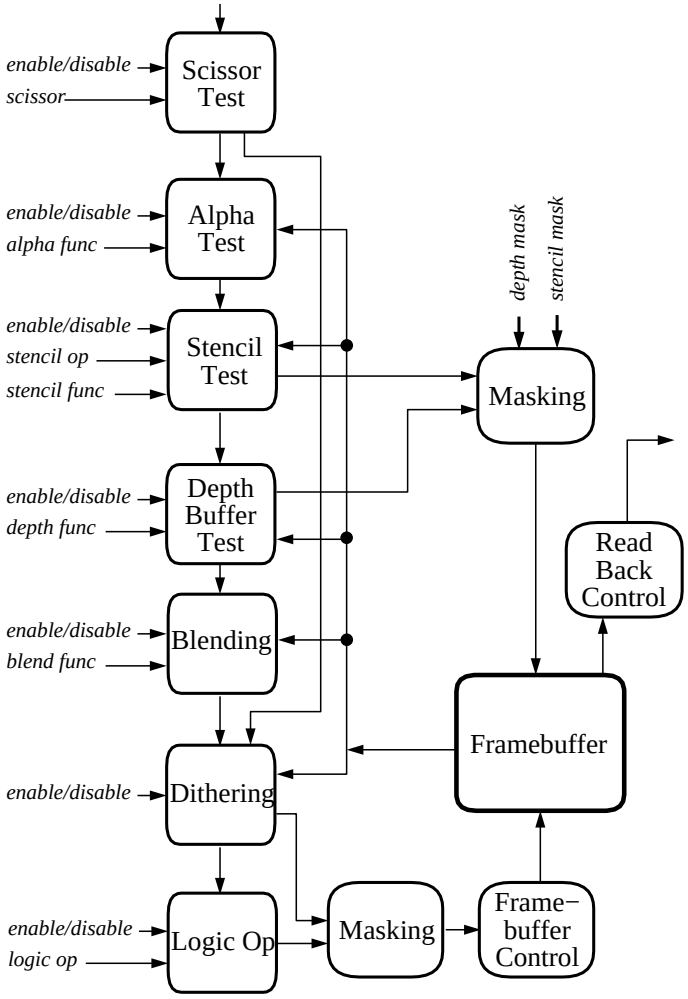


Figure 2.2: Per fragment operations of OpenGL.

To establish the previously described quantitative occlusion information, it is necessary to access the per fragment data. The PHC (projection hit counter) is incremented whenever a fragment enters the per fragment pipeline. However, the VHC (visible hit counter) requires the result of the depth test to be able to determine whether a pixel is visible. In the following, the unit which generates the quantitative occlusion information will be referred to as *occlusion unit*. This occlusion unit needs to be located in the per fragment pipeline such that it can access the required data. Implementing the occlusion unit is very cheap. All it takes are two registers, two counters, and a little bit of logic, as illustrated in Figure 2.3. For each fragment, its x and y address are compared to determine whether the fragment resides within a user specified screen space bounding box. In case the fragment belongs to a pixel within this bounding box, the PHC is incremented. Furthermore, the VHC is incremented in case the fragment passes the depth test, e.g. it would alter the frame-buffer. To ensure correct counting of the projection area and visible pixels, backface culling must be enabled, otherwise

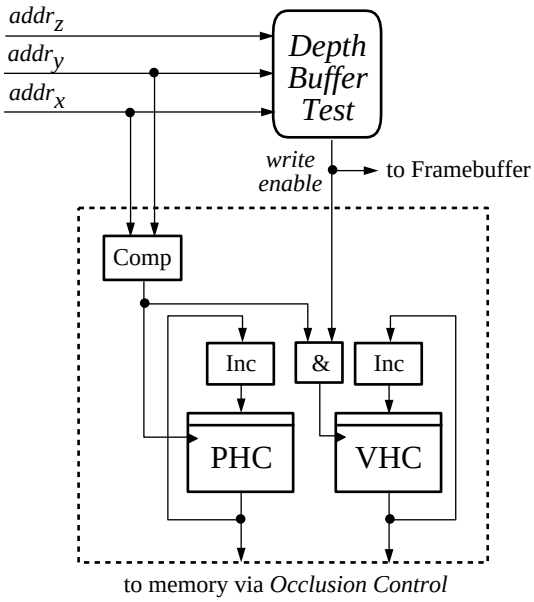


Figure 2.3: Schematic description of the occlusion unit.

the amount of projection hits would be doubled while the amount of visible hits would depend on the order the triangles are rendered[5]. Furthermore, the triangles of the partitioning entity need to be representing a convex object since concave objects would again result in multiple projection and possibly multiple visible hits for a single pixel (x,y). One could certainly design hardware which can handle the correct processing of concave objects but would result in a more complex design and the restriction to convex objects certainly is a good trade-off between functionality and hardware consumption.

Figure 2.3 schematically illustrates the implementation of an occlusion unit, capable of handling the quantitative query with respect to a certain screen space area (Comp). In case the graphics hardware consists of more than one pixel pipeline, each pixel pipeline will require an occlusion unit. Thus, when querying the information from the occlusion unit, the collected data needs to be combined.

2.4 Extending OpenGL for Occlusion Queries

In order to exploit hardware extensions as proposed in the previous section, it is necessary to provide a common interface to the user (API). In the following, this is shown extending OpenGL but could be done equivalently for DirectX. The integration of occlusion queries in OpenGL requires to extend the API by adding new types and calls, but also reusing existing calls.

[5] In case the polygons are rendered in back to front order, the number of visible hits would double while it would remain unchanged for rendering in front to back order. Since the rendering order depends on the viewing direction, this would result in undesirable random results.

Using already existing OpenGL calls with new parameters

In order to provide an occlusion mechanism, a new type needs to be provided.

```
GLenum GL_OCCLUSION_QUERIES
```

Occlusion queries can be enabled extending the available OpenGL calls `glEnable` and `glDisable`.

```
void glEnable(GL_OCCLUSION_QUERIES)
void glDisable(GL_OCCLUSION_QUERIES)
```

This is similar to the way light sources are handled in OpenGL. On enabling the test `GL_OCCLUSION_QUERIES`, the pipeline will be flushed to ensure that all available data has been processed and is correctly rendered into the frame-buffer. Furthermore, all fragments of geometry rendered with enabled `GL_OCCLUSION_QUERIES` pass all fragment tests but will not alter the frame-buffer content, as described earlier. Individual occlusion queries are enabled or disabled correspondingly.

```
GLenum GL_OCCLUSION_QUERYi

void glEnable(GL_OCCLUSION_QUERYi)
void glDisable(GL_OCCLUSION_QUERYi)
```

Again, this is similar to the way light sources are handled in OpenGL. *i* denotes the index of the occlusion query which should be enabled/disabled. When enabling an occlusion query, the counters of this query will be set to zero. Multiple occlusion queries are possible at the same time using different screen space bounding boxes. The maximum number of possible occlusion queries is specified by another type and the maximum number of supported queries can be obtained using an existing call.

```
GLenum GL_MAX_OCCLUSION_QUERIES

void glGet(GL_MAX_OCCLUSION_QUERIES)
```

The maximum number of occlusion queries is important in case multiple occlusion queries are used in parallel. How many queries are support depends on the implementation of OpenGL which is a matter of how many counters can be implemented at a reasonable cost.

Adding new OpenGL calls

Besides enabling and disabling the previously described occlusion queries, it must be possible to specify the screen space bounding box in which the occlusion query will be sensitive to fragments. Therefore, a new type is necessary.

```
GLenum GL_2DBOX
```

Adding a new call which is again similar to specifying light sources, the screen space bounding box can be specified.

```
void glOcclusionQueryiv(GL_OCCLUSION_QUERYi,
                        GL_2DBOX, values)
```

where *values* is an array of four ints needed to specify an axis aligned screen space bounding box.

```
GLint values[] = {Xmin, Ymin, Xmax, Ymax}
```

With the so far described calls, multiple occlusion queries can be specified as well as enabled and disabled. Acquiring the data from the hardware can be done using two new types, one for each counter.

```
GLenum GL_PROJECTION_HITS
GLenum GL_VISIBLE_HITS
```

To obtain the query result, a new call needs to be added.

```
glGetOcclusionQueryiv(GLenum query,
                      GLenum name,
                      GLint *params)
```

`query` is a symbolic name of type `GL_OCCLUSION_QUERY`*i* where *i* can be any value of $0 \leq i <$ `GL_MAX_OCCLUSION_QUERIES`. `name` specifies the query parameter type which can be either `GL_PROJECTION_HITS` or `GL_VISIBLE_HITS`. Finally, `params` contains an array of values but for the so far presented parameter types (`GL_PROJECTION_HITS` and `GL_VISIBLE_HITS`) only one parameter is returned, indicating the number of determined hits. However, other query results providing a list of values could generally be supported.

Overall, the following sequence of calls enables determining the visibility of an object (set of polygons):

```
 1: GLint VHC, PHC;
 2: GLint values[] = {0,0,1023,1023}; // full screen
 3: glOcclusionQueryiv(GL_OCCLUSION_QUERY0,
                       GL_2DBOX, values);
 4: glEnable(GL_OCCLUSION_QUERIES);
 5: glEnable(GL_OCCLUSION_QUERY0);
 6: Render()          // geometry approximating
                      // the partitioning entity
 7: glDisable(GL_OCCLUSION_QUERY0);
 8: glDisable(GL_OCCLUSION_QUERIES);
 9: glGetOcclusionQueryiv(GL_OCCLUSION_QUERY0,
                          GL_VISIBLE_HITS, VHC);
10: if (VHC > 0)
       Render();      // geometry contained in
                      // the partitioning entity
```

Starting with the presented framework for occlusion queries, further query parameters and results can easily be added. E.g., the minimum and maximum depth value of the encountered hits could be collected and returned, as proposed in Section 2.2. This would simply require to add another type.

```
GLenum GL_MINMAX_DEPTH
```

With this type, the depth information could be integrated into the API, extending the parameters an occlusion query can return. Adding the minimum and maximum *z*-value to the functionality of the occlusion unit (see Figure 2.3) is very simple and does not require much hardware.

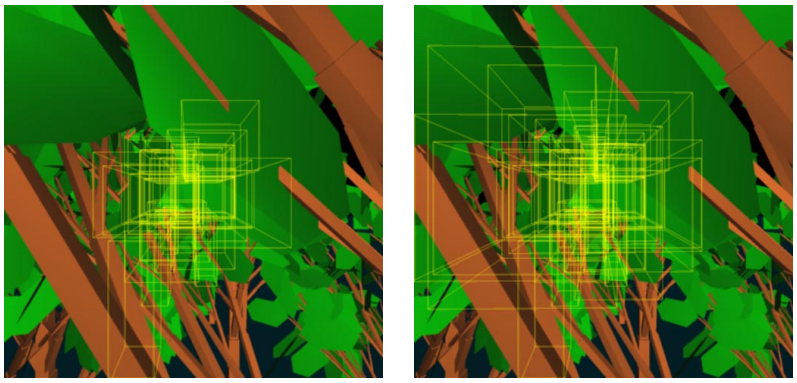
2.5 Adaptive Occlusion Culling

Adaptive occlusion culling was first proposed by Zhang in [ZMHH97]. The basic idea is that objects which only have a small number of not occluded pixels have a small visual contribution to the final image. Therefore, if those objects are skipped, the visual impression of the rendered scene will not be jeopardized. In their approach, the user can specify a certain threshold used to determine which geometry is discarded and which rendered. Using quantitative occlusion query information, it is possible to determine the accurate percentage of an object's visibility.

$$Visibility = \frac{VHC}{PHC} \tag{2.1}$$

Instead of simply discarding geometry which has a visibility that is below a given threshold, the visibility percentage can be used as additional parameter to determine the level of detail (LoD) at which the geometry should be rendered. If only a small percentage is visible, a simple representation of the object should suffice.

However, the relative visibility can be misleading since an object might be fully visible while the partitioning entity used to perform the query is much larger. Hence, the set of polygons used to describe the partitioning entity should be a good approximation of the actual geometry contained in the entity. Nevertheless, in contrast to previous approaches where the degree of visibility is determined using the minimal 2D screen space bounding box [HMB98], the presented approach here is more accurate because the actual pixels of the projection of the object (PHC) are used. Figure 2.4(a) shows a zoomed view of Figure 2.1(a). While Figure 2.4(a) is rendered culling entirely occluded objects only, (b) shows the image rendered culling all objects which have a degree of visibility which is below 0.02 (see Equation 2.1).



(a) (b)

Figure 2.4: Adaptive occlusion culling can be used to further accelerate the rendering by using the relative visibility to choose an appropriate level of detail. Partitioning entities of culled objects are marked yellow: (a) Shows an image of the tree alley culling only objects which are fully occluded. (b) Geometry with a relative visibility of less than 0.02 is assumed to be occluded.

Overall, adaptive occlusion culling can be a good trade-off to further reduce the number of rendered objects. Furthermore, it can be used for frame to frame coherency to predict how the visibility of an object might change in the subsequent frame.

2.6 Results

The presented hardware for occlusion queries is not yet implemented in any available polygon graphics hardware and hence, it is difficult to give accurate performance estimates. However, Hewlett-Packard provides a simplified version of this occlusion query returning a plain visible or occluded information for polygons being rendered in a dedicated occlusion mode [HP97, HP97]. Therefore, the performance improvements possible with the here presented detailed occlusion queries are expected to be better. Nevertheless, tests using HP's graphics accelerators can serve as a measure of the lower bound.

In general, the hierarchical representation of a scene and its elements is of crucial importance for the efficiency of such hardware supported screen space based occlusion tests. The grainer a bounding box approximates the actual geometry contained in the bounding box, the higher the risk to detect it as being visible even though the content might be occluded. Finding good bounding boxes as well as a good spatial hierarchy is an ongoing research topic [MBH+99]. In the following, summarized results are presented using SGI's commercially available Optimizer tool. It can be used to automatically generate a hierarchy of a given scene and produces a hierarchical representation of groups polygons using an octree based subdivision scheme. For the evaluation, two scenes were selected; a large city model and a surface representation of a human ventricular system. Images of the two scenes are given in Figure 2.5 and the polygon count is shown in Table 2.1. The experiments were conducted on an HP

Scene	Polygons	Raw	Vfc	Occ
Ventricle	270,882	4.6	5.3	13.6
City	1,408,152	0.9	1.4	11.8

Table 2.1: Performance improvements due to Hewlett-Packard's occlusion culling. Frame-rates are denoted *Raw* for no culling, *Vfc* for view frustum culling, and *Occ* for occlusion culling.

B180/fx4 graphics workstation using a camera path leading through each scene. The numbers given in Table 2.1 are averaged along the used camera paths. In average, the performance improvements range from 150 to 740%. Using the more detailed occlusion queries as proposed in this chapter could yield further performance improvement since object with a lower visibility could be rendered at a lower level of detail.

2.7 Summary

A dedicated hardware supporting occlusion queries within the graphics subsystem has been presented. By extending the OpenGL API in a straight forward manner, users can be empowered to exploit this quantitative visibility information. Further extensions of the possible return parameters can easily be integrated. Nevertheless, such an occlusion query comes at a certain price. When enabling occlusion queries, all elements in buffers (DisplayLists) and the pipeline (triangles, fragments) need to be processed until completion (flush). Depending on the graphics subsystem, this needs some finite time. After executing the rendering of the polygons approximating the partitioning entity, one needs to disable the occlusion queries causing yet another buffer and pipeline

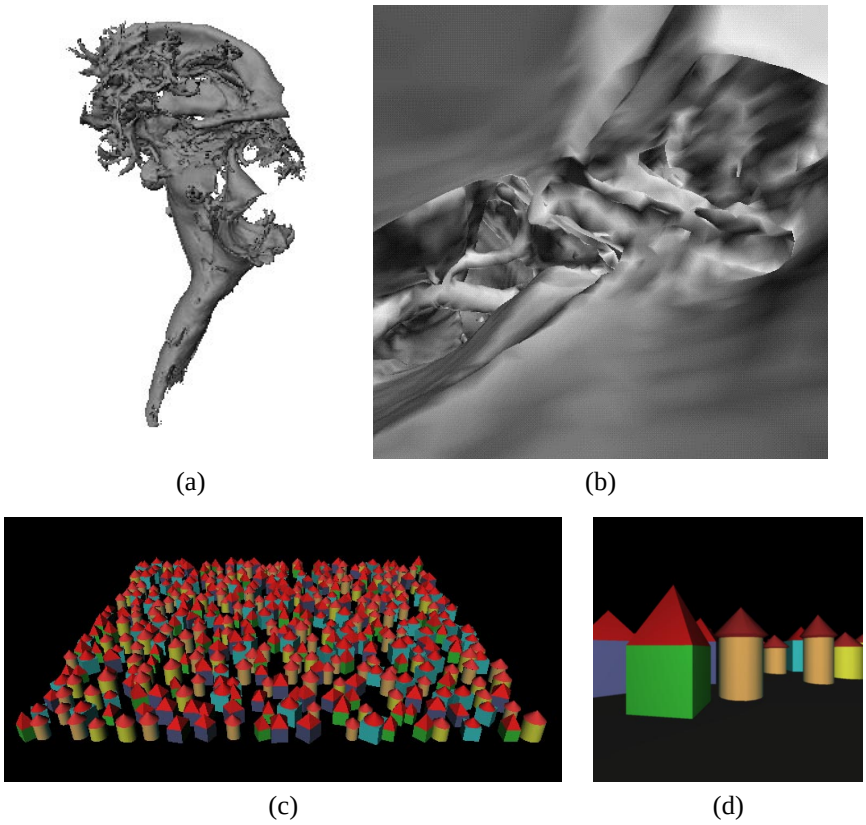


Figure 2.5: Ventricular system: (a) Overview. (b) Inside view. City model: (c) Overview. (d) Inside view.

flush. Overall, during these two flushes operations, rendering bandwidth is wasted and one needs to carefully choose which partitioning entities are tested.

As mentioned earlier, Hewlett-Packard also proposed an occlusion test and a corresponding extension to OpenGL [HP97, HP97]. However, this test is based on a simple visible/occluded query which does not include such counters or more detailed query information. Very recently, Hewlett-Packard also introduced to possibly extend this to multiple occlusion tests within one pass which is also discussed within the OpenGL ARB[6]. However, returning more detailed query information — as proposed in this chapter — has not yet been proposed.

What can be concluded from Hewlett-Packard's occlusion test is the cost due to the pipeline flushes. Hewlett-Packard estimates that an occlusion test is equivalent to rendering 190 (65) 25-pixel triangles on an fx6+ (fx2+) [Sev99]. The cost of the here proposed quantitative occlusion query will be the same as for Hewlett-Packard's occlusion test.

[6] Architecture review board.

Chapter 3

Visibility driven Rasterization

In the previous chapter, occlusion culling hardware enabling to cull geometry prior to sending it to the graphics hardware has been presented. Within this chapter, an additional mechanism establishing local visibility information during the rendering of the occlusion query is introduced. In case the partitioning entity has been detected visible, the local visibility information can be exploited for accelerated rendering of the contained geometry.

Visibility driven rasterization is capable of significantly increasing the rendering performance of modern graphic subsystems. Instead of scan converting, texturing, lighting, and depth-testing each individual pixel, a two-level visibility mask is integrated within the rasterization stage enabling the removal of groups of pixels and triangles from rasterization and the subsequent pipeline stages. Local visibility information is stored within the visibility mask that is updated several times during the generation of a frame. The update can easily be accomplished by extending already (in hardware) available occlusion culling mechanisms (e.g., those of HP and SGI or the mechanism which was proposed in the previous chapter), where it is possible to integrate the additional functionality without any additional delay cycles. In addition to these existing hardware based occlusion culling approaches — which cull only geometry contained in bounding primitives determined as *occluded* — visibility driven rasterization is able to significantly accelerate the rendering of the geometry determined as *visible*. However, the approach does not specifically rely on such occlusion culling hardware.

3.1 Introduction

Over the last few years, the complexity and overall rendering bandwidth of graphics subsystems has increased dramatically. Starting from large scale multi-chip systems [Ake93, MBDM97], single chip solutions [Inc99, MMG+98a], and even full graphics processing units (GPUs, which also integrate transformation, lighting, and setup [nVi99]) are available. This trend is based on increased gate counts in integrated circuits and by new memory technologies. However, memory bandwidth and memory access efficiency remain one of the most troublesome issues.

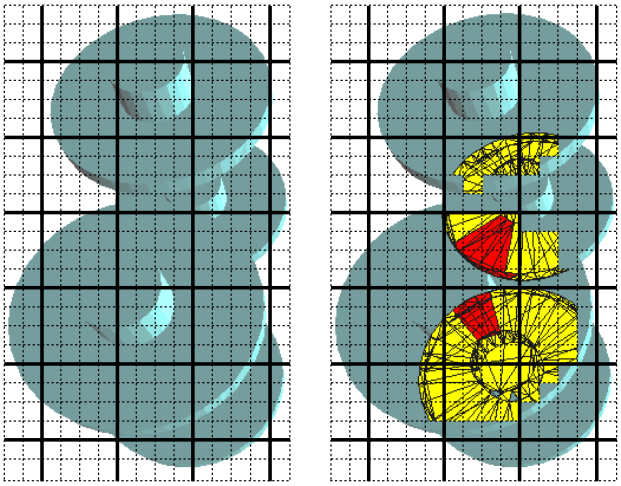
Numerous approaches have been presented to hide memory latency by caching [AMSW97], prevent stall cycles by prefetching [IEP98], and interleave memory such that stalls appear as infrequent as possible [MMG+98b]. Nevertheless, each pixel can cause *memory stalls* due to memory accesses. Moreover, the dramatic board-to-chip integration, enables more complex dynamic geometry and *richer pixels* operations [Kir98]. Richer pixels comprise better filtering techniques, multi-texturing, and per pixel shading models. The associated iterations over multiple light sources and application of multi-texturing increases computational complexity as well as the overall memory inefficiency per pixel, since the multiple memory accesses potentially cause memory stalls. Thus, the processing of pixels becomes increasingly expensive and it is exceedingly important to keep the amount of redundantly processed pixels as small as possible.

Approaches to reduce the bandwidth and rendering problems include mesh simplification and compression, as well as visibility and occlusion culling. While the former reduce the overall geometry load, the pixel complexity remains almost unchanged. In contrast, visibility and occlusion culling address geometry and pixel complexity by culling non-visible geometry. However, these algorithms either introduce high computational costs or tend to be of limited efficiency in scenes with low depth complexity. Therefore, the remaining rasterization and subsequent pixel processing load is frequently beyond the interactive rendering capabilities of current graphics hardware. Our novel visibility driven rasterization significantly reduces the remaining rasterization load (after “traditional” occlusion culling [SGI99, SOG98] or what has been presented in the previous chapter), introducing a two-level visibility mask. This mask enables the graphics hardware to reject triangles and groups of pixels in non-visible areas and requires only incremental modifications of the graphics subsystem. In analogy to PixelFlow [MEP92], this can be referred to as *deferred fragment processing* with a certain granularity determined by the number of times the visibility information is established.

3.2 Rasterization

Visibility driven rasterization represents a new scheme which extends current rasterization. Based on a few polygons — e.g. representing the bounding box of many polygons — binary screen-space oriented visibility information is generated and stored within the rasterizer in a *visibility mask*. In case any portion of these polygons left traces in the visibility mask, the established visibility information is exploited to cull subsequently rendered triangles and portions of triangles which reside within not visible

screen-space areas, as depicted in Figure 3.1.



(a) (b)

Figure 3.1: Four wheel hubs of cotton picker scene (see Figure 3.6). (a) Two of the wheel hubs are partially occluded by the front wheel hubs. (b) The culled triangles of the partially occluded wheel hubs are colored in red, the culled pixel groups are colored in yellow. The black grid indicates the applied two-level visibility mask.

To accomplish a good visibility culling performance, the visibility information represented in the visibility mask should be updated several times. This is achieved by first clearing the visibility mask and rendering a set of polygons — e.g., a bounding box or other kinds of a convex hull — used to determine visibility. Therefore, visibility driven rasterization can easily be combined with occlusion culling mechanisms as presented in the previous chapter which already performs such a pipeline synchronization step.

The fragments of the rendered polygons which pass the depth test will cause the corresponding bit of the visibility mask to be set. Thereafter, this visibility information can be exploited during rendering of geometry that is located behind the polygons used to establish the visibility. Hence, the visibility mask contains only coverage information based on a set of polygons, but no depth information[1]. The depth information is inherently given, since the geometry (to which the visibility mask is applied) is contained or at least visually blocked by the geometry used to establish the visibility mask. A simple example is to render a bounding box to establish the visibility mask and subsequently render the contained geometry. As we do not store depth but pure visibility information, the visibility mask is very compact.

3.2.1 Visibility Mask

The visibility mask is the core element of visibility driven rasterization. Each bit of it represents a certain area of the screen space. A set bit indicates that the geometry used to establish the visibility information is at least partially visible within this area, otherwise it is occluded. A certain granularity has to be selected for the subdivision

[1] Note that for best culling performance, geometry should be rendered sorted front to back.

of the screen space into areas and the efficiency of a subdivision scheme depends on the scenes to be rendered. Generally, selecting very large areas per bit of the visibility mask results in a high probability to detect mostly visible areas where no geometry can be culled from rasterization. On the other hand, selecting a bit for each pixel of the screen space does not achieve significant performance improvement since at best only idle pipeline cycles can be gained. Furthermore, the finer the visibility mask, the more storage is required (e.g., a viewport of 1024×1024 pixels and one bit per area of 8×8 pixels would require 2 KByte to store such a visibility mask). Since the visibility mask has to be accessed at the processing speed of the rasterizer without introducing stall cycles, it must be implemented either as a large register file or as an SRAM module, possibly on-chip. All things considered, the granularity is a trade-off between storage (chip real estate) and culling efficiency.

In the following, rectangular areas (tiles) of size $n \times m$ are employed for each bit of the visibility mask. The optimal values for n (in x) and m (in y) are evaluated in Section 3.4.1, using different polygonal scenes from "real world" applications. To facilitate an efficient implementation in hardware, n and m are chosen as powers of two. Furthermore, since register files and SRAMs are organized as addressable space of four, eight, 16, or 32 bit entries, 16 bit entries are selected for simplified matters. By storing the information of a group of tiles in a single entry of the visibility mask (here $4 \times 4 = 16$ neighboring tiles), a two-level hierarchy is obtained. This two-level hierarchy can be exploited to cull triangles more efficiently. The visibility mask is empty in case the visibility hit counter (VHC) — as presented in the previous chapter — is zero. In case of VHC > 0, the contained geometry would be rendered possibly using an appropriate LoD and the content of the visibility mask can be exploited during rendering of the contained geometry.

3.2.2 Culling Triangles

The screen space nature of the visibility mask requires that triangles are first transformed and clipped. Once screen space coordinates are available, it can be determined whether a triangle resides within a single tile of the visibility mask by testing the address of the vertices of the triangle. In case the triangle is entirely contained and the corresponding bit in the visibility mask indicates non-visibility, it can safely be culled since it resides within a non-visible tile (*trivial reject I*). Depending on the size of tiles and triangles, it occurs more or less frequently that triangles extend over two or more tiles. In this case, the trivial reject I mechanism will fail. To further increase the culling efficiency, a second test can be added culling triangles which reside within a group of tiles (4×4 tiles = *tile group*) exploiting the previously described two-level hierarchy of the visibility mask. Thus, if the entire tile group is non-visible then the triangle is culled (*trivial reject II*). Figure 3.1(b) shows triangles culled due to trivial reject I and II colored red. Please note that for both trivial reject mechanisms, the perspective division of the x and y components is necessary. Only if the triangle can not be culled, the perspective division for the z component is required. Please note that trivial reject I and II are performed in "parallel" and a triangle is discarded in case the logical *or* of the two test results is "1".

Figure 3.2 illustrates the two-level hierarchy and the trivial reject mechanisms. In this example, none of the triangles are culled by testing them against single tiles (trivial reject I). However, if all four tile groups are non-visible, all but triangle "B", which emerges over two tile groups, are culled (trivial reject II). All triangles which can not

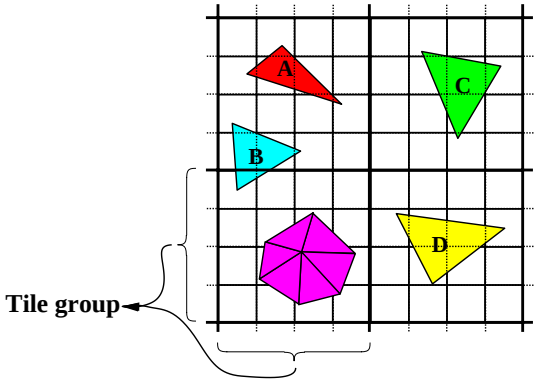


Figure 3.2: Four tile groups: Each tile represents the area of $n \times m$ pixels on the screen. The visibility information of a tile group (16 tiles, 4×4) is stored in one entry of the visibility mask.

be culled are sent to the rasterization stage, including some partially or entirely non-visible triangles which are addressed in the following section.

Generally, culling of triangles from the graphics subsystem results in improved performance if the stages from which load is removed can exploit this additionally available bandwidth. Therefore, to achieve good performance using visibility driven rasterization, the graphics subsystem must be designed with respect to this culling mechanism, e.g., accommodate the bandwidth of the rasterizer and the subsequent pixel processing stages and integrate look-ahead logic for the skipping. Nevertheless, for current graphics subsystems, performance improvements are already achievable for (i) larger triangles (by reducing fill-rate limitations) and (ii) when using multi-texturing (by reducing memory stalls due to page misses).

3.2.3 Culling Groups of Pixels

The triangles which pass trivial reject I and II are sent to the rasterization stage, even though they are not necessarily fully visible. Further bandwidth can be saved on the subsequent pipeline stages by removing groups of pixels of these triangles. Figure 3.3 illustrates one of the frequent cases where a remaining triangle covers one or more non-visible tiles. During rasterization, the groups of pixels associated with those tiles can safely be culled (see yellow pixels of triangles in Figure 3.1(b)). To exploit the resulting idle cycles in existing graphics subsystems, the rasterization continues within visible tiles, after skipping the non-visible tiles. Depending on the implemented rasterization scheme (scanline or stamp based), this requires more subtlety and cleverness which is discussed in the following.

3.3 Hardware Issues

Generally, a rasterizer performs several setup calculations per triangle (such as edge increments for color, texture coordinates, depth value, etc.) and subsequently generates pixels in a certain order. For many years, this was performed in a scanline

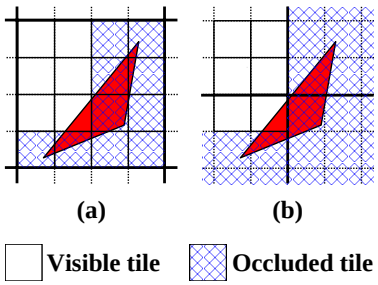


Figure 3.3: The triangles covering several non-visible tiles of a single tile group (a), of multiple tile groups (b).

based order by stepping along the edges and incrementally generating all pixels between two edge points. Partially driven by the experiences gained in the PixelFlow project [MEP92, EMPG97], stamp based rasterization recently became popular. A stamp of e.g., 2×2 pixels is moved across a triangle, potentially generating up to four pixels in each cycle by evaluating three edge equations for each pixel of the stamp [MMG+98b, MM00]. Unfortunately, graphics chip companies do not publish details on such implementations. One of the few valuable sources are [MMG+98b, MM00] indicating that tile based rasterization and visibility driven rasterization could be nicely combined.

Scanline and stamp based rasterization finally generate pixels including interpolated color, texture coordinates, depth value, etc. In contrast to *visibility driven rasterization*, the above described rasterization is referred to as as *current rasterization*. Figure 3.4 illustrates the schematic components of a current rasterization scheme. Additionally, the extensions needed for visibility driven rasterization are included.

3.3.1 Establishing Visibility Information

From the set of polygons used to establish visibility information, all pixels which pass the depth buffer test are used to set their corresponding bit of the visibility mask. The address of the tile group and the mask to obtain the corresponding bit can easily be generated. Furthermore, accessing, setting, and writing a bit works well in architectures where at best one pixel passes the depth test per cycle. However, in graphics architectures consisting of multiple pixel pipelines, access conflicts need to be resolved. An obvious solution would be to replicate the visibility mask for each pixel pipeline but this increases the overall hardware implementation costs. Alternatively, the accesses can be synchronized by adding a small FIFO buffer to each pixel processing pipeline which stores the tile ID[2] and applying a processing priority given by the number of entries in each FIFO. When reading the value from a FIFO, all other FIFOs having an ID that belongs to the same visibility mask entry need to be combined by performing a logical OR operation of the mask bits. To prevent stall cycles due to a full FIFO, each FIFO has a controller that impedes IDs from entering the FIFO, if one of the entries in the FIFO already has the same ID. This mechanism works well as long as the number of pixels per tile is larger than the number of pixels generated per cycle by the

[2]The ID is a composition of address of the visibility mask entry and the corresponding mask for the specific individual bit.

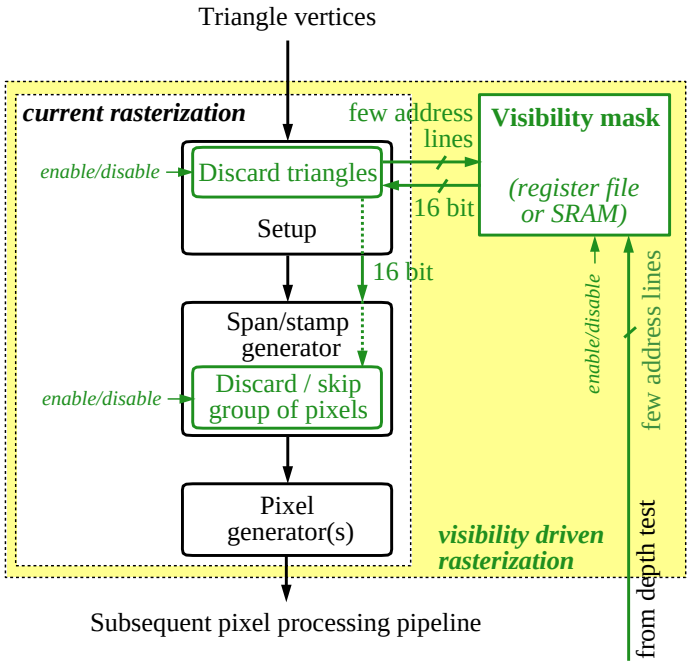


Figure 3.4: Current and visibility driven rasterization. Yellow areas and green blocks and arrows indicate additionally needed resources for visibility driven rasterization.

rasterizer.

Generally, modern pixel pipelines operate on neighboring pixels of the same triangle which prevents conflicts while addressing the visibility mask as long as the screen space alignment of the pixel pipelines does not conflict with the tile borders of the visibility mask. Since powers of two are very common, conflicts should be entirely avoidable without requiring FIFOs.

3.3.2 Culling Triangles

Implementing the two described trivial reject mechanisms in hardware is straight forward. Trivial reject II requires the address of the tile group which is computed by $log\ n$ shift operations of $x_{address}$, and $log\ m$ shift operations of $y_{address}$ for each vertex of the triangle. If the resulting bit patterns are identical for all three vertices, the triangle resides within one tile group and assembling the bit patterns generates the address of the corresponding entry in the visibility mask. Finally, non-visibility is given in case the entry is zero and trivial reject II succeeds. Trivial reject I is evaluated by decoding the corresponding tile of the tile group and checking the resulting bit for non-visibility.

A schematic implementation of this mechanism requiring few hardware components is shown in Figure 3.5. Here, tiles of 16×16 pixels and tile groups of 4×4 are used for a viewport of 1024×1024 pixels. This results in a visibility mask of 256 entries and an overall size of 4 Kbit which can either be realized as a large register file or an SRAM memory module, possibly on-chip.

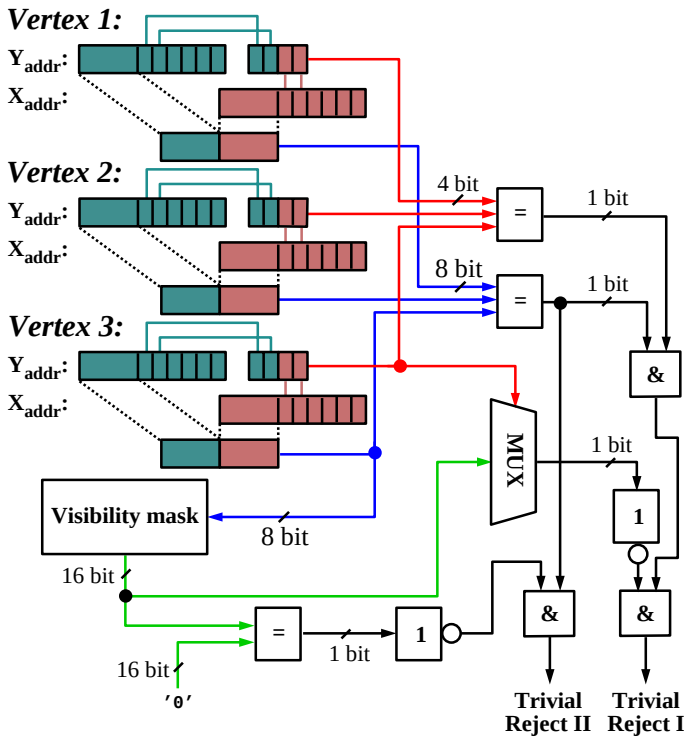


Figure 3.5: Implementation of the trivial reject I and II mechanisms. Logic for tiles of 16×16 pixels and tile groups of 4×4 tiles is shown. Only very basic and cheap bit operations are required: Test if triangle resides in one tile group (blue arrows), in a single tile (red arrows and subsequent AND), and compare with bit and bit pattern of corresponding visibility mask entry (green arrows).

3.3.3 Culling Groups of Pixels

Removing groups of pixels from the subsequent processing stages during rasterization can be integrated by checking the bit of the corresponding tile in the visibility mask of the pixels to be generated. However, besides avoiding memory stalls possibly due to processing of these pixels, only moderate additional performance improvements are likely. To achieve significant additional performance improvement, the rasterization process must be modified such that skipping non-visible tiles can be accomplished without latency such that the pipeline is always processing potentially visible pixels. This skipping mechanism is not trivial, since rasterization is usually performed in an incremental manner and skipping is dependent on the implemented rasterization approach.

In a scanline based rasterization approach [Kug96, WEWL99], multiplications are avoided as often as possible, except in the setup phase. Therefore, while generating pixels between two edge points along a scanline, color, depth, and other increments $(\Delta R, \Delta G, \Delta B, \Delta Z)$ are added to the current values stored in registers (span iteration). A step of a power of two can easily be achieved by shifting the corresponding Δ increment. Steps of $3, 5, 6, 7$ etc. are more difficult to implement. By generating an array of

increment values ($\Delta, 2*\Delta, 3*\Delta$, etc.) during the rasterization setup phase, the according increment could be chosen to quickly skip non-visible tiles.

For stamp based rasterization, McCormack et al. mention that the stamp may also be constrained to generate all fragments in a 2^n by 2^m rectangular "chunk" before moving to the next chunk[3] [MMG+98b]. Very recently, the authors presented a tile based rasterization scheme [MM00] which indicates that visibility driven rasterization based on tiles would nicely fit together. Therefore, it seems to be feasible to integrate skipping of an occluded tile without introducing any idle cycles.

In summary, it is not straight forward to incorporate the required skipping mechanism into common scanline based rasterization. However, it fits perfectly into more recent stamp based rasterization and thus, enabling additional performance improvements.

3.4 Results

For the simulation of the potential benefits of visibility driven rasterization, the MESA 3D graphics library — an Open Source implementation of the OpenGL graphics API — has been extended. The extensions follow the terminology used in the previous chapter. For visibility driven rasterization, three new types are necessary.

```
GLenum GL_VISIBILITY_MASK
GLenum GL_GENERATE_VISIBILITY_MASK
GLenum GL_APPLY_VISIBILITY_MASK
```

To ensure proper visibility information, the visibility mask needs to be cleared before it is established.

```
void glClear(GL_VISIBILITY_MASK)
```

Establishing the visibility information can be enabled or disabled reusing existing OpenGL calls.

```
void glEnable(GL_GENERATE_VISIBILITY_MASK)
void glDisable(GL_GENERATE_VISIBILITY_MASK)
```

Fragments of subsequently rendered polygons do affect the content of the visibility mask. For each fragment which passes the depth test, the corresponding bit of the visibility mask is set. Note that `GL_GENERATE_VISIBILITY_MASK` could automatically be enabled while performing an occlusion test as presented in the previous chapter (`GL_OCCLUSION_QUERIES`).

Once the visibility mask is established, it can be used during rendering by enabling `GL_APPLY_VISIBILITY_MASK`. Thus, the difference between rendering without and with enabled `GL_APPLY_VISIBILITY_MASK` is that the latter one employs visibility driven rasterization.

```
void glEnable(GL_APPLY_VISIBILITY_MASK)
void glDisable(GL_APPLY_VISIBILITY_MASK)
```

[3] The authors state that *"(...) chunking is not cheap due to three additional 600-bit save states and associated multiplexers."* but in a later publication refine this statement *"(...) we recently discovered that we could have used a single additional wait state."*

Please note that the rendering of the bounding box, as presented in the previous chapter, could potentially be accelerated in case `GL_GENERATE_VISIBILITY_MASK` and `GL_APPLY_VISIBILITY_MASK` are both enabled.

The following code excerpt illustrates the use of these two calls. Alpha, depth, and stencil test have to be set appropriately such that all fragment reach the occlusion unit (see previous chapter) and the visibility mask. Lines 6 through 9 are used to establish the visibility mask of the geometry approximating the partitioning entity. In case anything is visible (line 12), the visibility mask can be applied while rendering the geometry contained in the partitioning entity (lines 13 through 15).

```
 1: GLint VHC;
 2: GLint values[] = {0,0,1023,1023}; // full screen
 3: glOcclusionQueryiv(GL_OCCLUSION_QUERY0,
                       GL_2DBOX, values);
 4: glEnable(GL_OCCLUSION_QUERIES);
 5: glEnable(GL_OCCLUSION_QUERY0);
 6: glEnable(GL_ESTABLISH_VISIBILITY_MASK);
 7: Render();          // geometry approximating
                       // the partitioning entity
 8: glDisable(GL_ESTABLISH_VISIBILITY_MASK);
 9: glDisable(GL_OCCLUSION_QUERY0);
10: glDisable(GL_OCCLUSION_QUERIES);
11: glGetOcclusionQueryiv(GL_OCCLUSION_QUERY0,
                          GL_VISIBLE_HITS, VHC);
12: if (VHC > 0) {
13:     glEnable(GL_APPLY_VISIBILITY_MASK)
14:     Render()       // geometry contained in
                       // the partitioning entity
15:     glDisable(GL_APPLY_VISIBILITY_MASK)
16: }
```

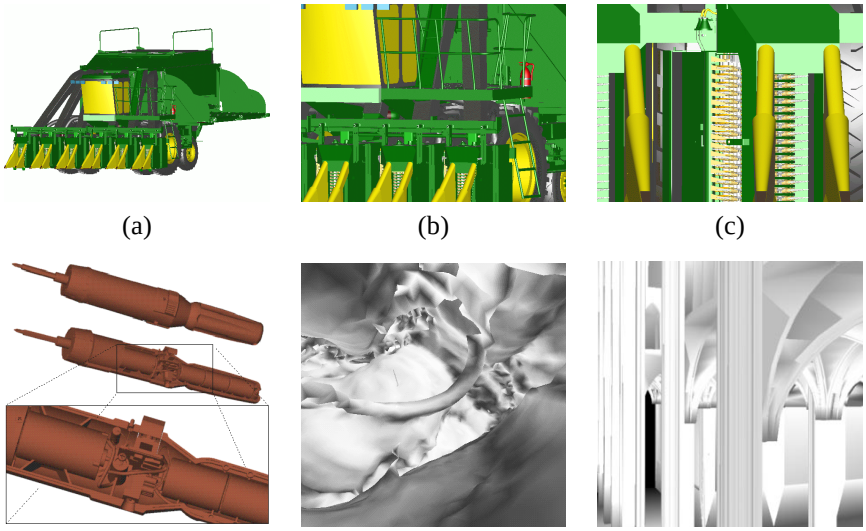
3.4.1 Experiments

To measure the potential efficiency of visibility driven rasterization using the above described extension of MESA, a set of diverse scenes has been selected (see Table 3.1 and Figure 3.6). Each scene is organized in a subdivision tree where the geometry is stored in the leaf nodes of the tree. Thus, the leaf nodes represent the partitioning entities which are tested.

Scenes	Source	#Triangles	#Leaf Nodes
Cotton picker	CAGD	10,605,158	13,257
Screwdriver	CAGD	156,424	83
Ventricular System	MRI	270,882	266
Cathedral	CAGD	391,868	133

Table 3.1: Set of scenes used to evaluate the potential benefits of visibility driven rasterization.

In a first step, view-frustum culling is applied to all entities (leaves) of the subdivision tree and the remaining entities are sorted in front to back order. Thereafter, these



(a) (b) (c)

(d) (e) (f)

Figure 3.6: Images of different scenes: (a-c) Cotton picker. (b) Close-up of (a). (c) Further close-up of (a) showing the spindles drums. (d) Screwdriver; top: screwdriver with complete chassis; middle: one chassis part removed; bottom: close-up of middle image. (e) Interior view of ventricular system. (f) Interior view of cathedral.

entities are tested for occlusion by rendering their axis aligned bounding box. The entire geometry of an entity (leaf) is culled, if no contribution to the visibility mask is detected (VHC is zero). Otherwise, the geometry is rendered applying the visibility mask. In this step, the three earlier described rejection mechanisms are performed (trivial reject I, II, and pixel groups) and the culled triangles and pixels are measured to obtain the potential culling efficiency. The average of the remaining triangles and pixels is shown in Table 3.2. Note that all reported culling performance is achieved additionally to "traditional" occlusion culling approaches like [GKM93, ZMHH97]. Furthermore, the numbers in Table 3.2 and Figures 3.7- 3.9 are referring to the number of triangles and pixels after clipping of the triangles which belong to the geometry nodes that are only partially located within the view-frustum.

	Cotton picker	Screwdriver	Ventricle	Cathedral
tris	547.596	43.874	10.054	10.933
pix	2.572.241	1.227.566	3.214.565	5.020.633
pix1	2.429.600	1.088.365	2.971.471	4.972.707
pix2	1.271.809	537.875	1.758.939	3.224.297

Table 3.2: Number of triangles and pixels remaining after view-frustum culling, occlusion culling, and clipping (tris and pix). Furthermore, remaining pixels after trivial reject I and II (using a tile size of 16×16 pixels (pix1)) and the finally remaining pixels after culling of pixel groups (pix2).

Cotton Picker

The cotton picker is a “real world” model from an industrial CAGD modeling package containing 13,257 individual parts in its assembly list (see Figure 3.6(a-c)). Based on this assembly list, the subdivision tree has been generated. Most of the geometric complexity is located in the six spindle compartments (1.633,137 triangles each) containing the spindle drums (694,113 triangles each) which collect the cotton flakes. The drums are usually occluded by covers and chassis parts. However, from a frontal point of view, all spindle drums are at least partially visible which decreases the potential culling benefits for traditional occlusion culling approaches. All results are averaged over a sequence of frames of twelve arbitrary views (on a sphere centered around the datasets), where the cotton picker is always completely contained in the view-frustum.

Screwdriver

Similar to the cotton picker, the screwdriver scene is a “real world” model from an industrial CAGD modeling package. It contains 83 individual parts in its assembly list, where the chassis consists of two parts, occluding most of the geometry of this model (see Figure 3.6(d)). Based on the assembly list, the subdivision tree has been generated. The sequence of frames uses twelve arbitrary views (on a sphere centered around the datasets), where the screwdriver is always completely contained in the view-frustum.

Ventricular System

The ventricular system (ventricles) is extracted from a pre-segmented MRI volume dataset using Marching Cubes (see Figure 3.6(e)). The associated subdivision tree has been generated by an octree decomposition of the volume dataset. It is a typical iso-surface model generated from a volumetric representation with a high occlusion depth for interior views and very regular shaped triangles of similar (object space) size. The sequence of frames consist of 150 individual viewpoints which are located inside of the ventricular system along a camera path.

Cathedral

The cathedral is an architectural CAGD scene modeled with a customized modeling package (see Figure 3.6(f)). It contains numerous long and narrow triangles of various sizes. The subdivision tree has been generated from the unordered triangles of the model by an automatic subdivision generator. The tall nave and transept of the building impede efficient results using traditional bounding box based occlusion culling approaches due to missing of suitable occluders. The sequence of frames consist of 100 individual viewpoints which are located inside the cathedral.

3.4.2 Trivial Reject I

Using trivial reject I, all triangles residing within a single tile are culled. Figure 3.7 shows the improvements possible due to trivial reject I for different tile sizes, after view-frustum and occlusion culling. While up to only 20% of the remaining triangles are culled for the cathedral, almost 40% are culled for the other scenes. The relatively low cull-rate for the cathedral is due to the columns and arches, which are poor occluders. Nevertheless, the triangles behind these occluders are culled using visibility driven

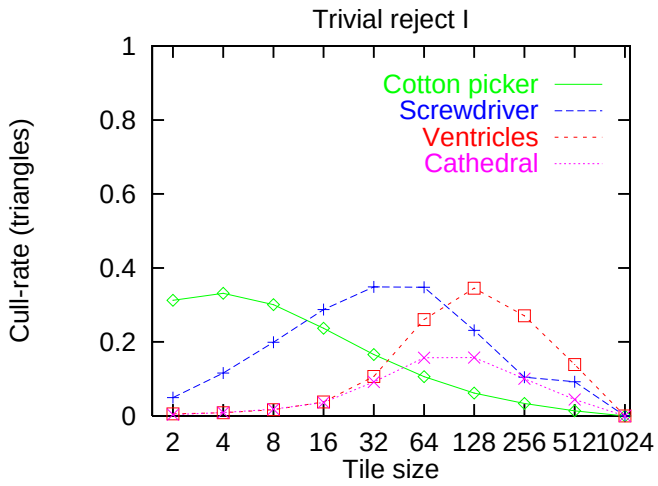


Figure 3.7: Cull-rate for trivial reject I relative to the triangles which remain after view-frustum and occlusion culling depending on tile size.

rasterization, while traditional occlusion culling can only cull them by applying a finer subdivision of the scene. However, only a certain amount of occlusion culling tests can be performed interactively, due to the relative high costs of the occlusion query (see Summary of Chapter 2).

Generally, the peak performance of trivial reject I depends on the size of the triangles of each scene. On average, the cotton picker has very small triangles (2.4 pixels) — due to the spindle drums within the spindle compartments — and therefore, small tiles achieve the best cull-rates. In contrast, the cathedral and the ventricles consist of large triangles (on average 180 and 465 pixels) and for small tiles these triangles reside frequently across multiple tile groups and hence, can not be culled with trivial reject I only. The screwdriver consists of triangles with on average 149 pixels and achieves best results with tiles of 16×16 to 64×64.

3.4.3 Trivial Reject I and II

While trivial reject I can only be applied to single tiles, trivial reject II exploits the second level of the visibility mask by testing 16 entries for non-visibility. The results of trivial reject I and II are shown in Figure 3.8. Compared to trivial reject I, the maximum cull-rates hardly increase[4]. However, for the screwdriver, cathedral, and ventricles the cull-rate function is stretched, achieving better cull-rates for a broader set of tile sizes. The stretched cull-rate function is mandatory to determine tile sizes which are efficient for a wide range of different polygonal scenes (e.g., the screwdriver achieves good results already with tiles of 8×8 pixels, the cathedral with 16×16, and the ventricles with 32×32). Overall, between 20% and 40% of the remaining triangles can be culled using trivial reject I and II.

It can also be observed that for the cotton picker model, visibility driven rasterization performs best with small tile sizes. This is due to the large majority of very

[4]Most of the culled triangles are quite small, hence only limited pixel savings are obtained (see Table 3.2).

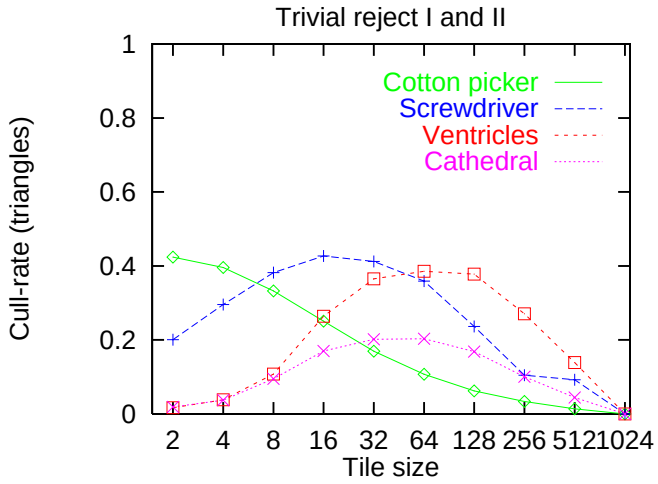


Figure 3.8: Cull-rate for trivial reject I and II relative to the triangles which remain after view-frustum and occlusion culling depending on tile size.

small triangles modeling the highly detailed elements (e.g., spindle drums in the spindle compartments). However, industrial CAGD rendering applications (e.g., EAI Vis MockUp) use a lower level of detail of a multi-resolution representation of the model, which significantly increases the average triangle size. This larger average triangle size results in a cull-rate function drop off at larger tiles, resulting in similar culling curves as for the other scenes.

3.4.4 Culling Groups of pixels

In contrast to trivial reject I and II — which reduce the number of triangles and hence, the setup in the rasterizer — culling groups of pixels removes only load from the pixel processors (fill-rate). Figure 3.9 shows the additional cull-rate of pixels remaining after view-frustum and occlusion culling, and trivial reject I and II.

Not surprisingly, culling pixel groups works best for smallest tiles and decreases as the size of the tiles increases. For the screwdriver, cathedral, and ventricles, tiles of up to 16×16 pixels achieve almost the same cull-rate as tiles of 2×2 pixels. Only for the cotton picker, tiles of at most 8×8 should be used to maintain a high culling efficiency. This is due to the large number of very small triangles rendered, which does not exploit level-of-detail selection.

3.4.5 Discussion

Generally, tiles of 16×16 and 32×32 achieve good cull-rates, while maintaining a good cost/performance ratio. For these two tiles sizes 4Kbit and 1Kbit (512 and 128 Bytes) respectively are needed to store the visibility mask. This is a size which can still be integrated into graphics subsystems using either a large register file or an on-chip SRAM. For the presented scenes, such visibility masks facilitate the culling of up to 40% of all remaining triangles (after “traditional” view-frustum and occlusion culling and clipping) and hence, no rasterization setup needs to be performed for these trian-

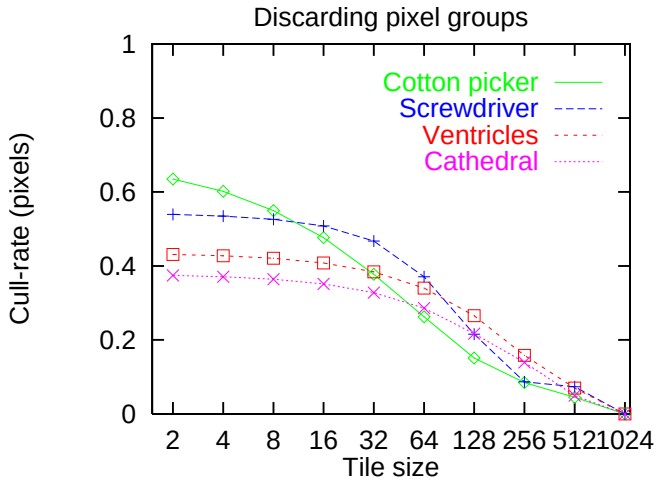


Figure 3.9: Cull-rate for culling pixel groups relative to the triangles which remain after view-frustum and occlusion culling, trivial reject I, and trivial reject II depending on tile size. Absolute values are given in Table 3.2 (pix2).

gles. Additionally, visibility driven rasterization culls between 40% and 55% percent of all further remaining pixels (after both trivial rejects).

Overall, integrating visibility driven rasterization into graphics subsystems has the ability to significantly improve and potentially double the frame-rates for the presented datasets. Moreover, the ability to cull triangles and pixels from the pixel processors is increasingly important when using multi-texturing, where systems are mostly fill-rate limited.

Further improvements for trivial reject II can be accomplished by extending the two-level hierarchy to a three- or four-level hierarchy with a few more logic operations (e.g., with the proposed 16 bit, a three-level visibility mask can be realized by additionally checking whether the triangle resides entirely within one of the four disjoint 2×2 tile subgroups). This could naturally be extended to a four-level hierarchy, using a register file with 64 bit entries. The advantage would be that starting out with each bit of the visibility mask representing tiles of 8×8 pixels, even tiles of 64×64 pixels would be covered with the same visibility mask. Hence, the impact of trivial reject II can be increased even more by investing a little bit more logic.

Tiles consisting of pixel spans are feasible as well choosing $m = 1$. However, additional experiments showed that spans result in an inefficient average triangle and pixel culling performance, and therefore, squared tiles which are a power of two are suited best.

3.5 Summary

A novel visibility driven rasterization scheme, capable of accelerating the rendering of scenes with a high depth complexity has been presented. A small and compact two-level visibility mask has been introduced to represent visibility information, enabling

a rasterization scheme which can remove a significant number of triangles before rasterization setup and pixel groups during rasterization, in addition to “traditional” view-frustum and occlusion culling techniques.

For a visibility mask of 16×16 pixels, storing the information of 4×4 tiles in one entry of the visibility mask, only 4 Kbit of memory are required, while being able to cull up to 40% of the geometry and 55% of the pixels in “real world” datasets. The saved bandwidth can either be invested into richer pixel operations such as multi-texturing, or in rendering more polygons, if there is sufficient bandwidth available on the vertex bus. Removing triangles from the rasterization and subsequent pipeline stages will be of even stronger importance once higher order primitives, e.g. NURBS or subdivision surfaces, are sent right to the graphics subsystem. Higher order primitives will reduce the traffic on the front bus (vertex transfer) and hence, the per fragment operations are a potential bottleneck.

Future work should focus on how to avoid enabling and disabling the visibility mask (occlusion queries) without the need of flushing the pipeline to ensure proper processing of triangles. The associated synchronization costs could be eased, interleaving the rendering of the bounding geometry and the actual geometry.

Part B:

Volume Rendering

Chapter 4

A Comparison of Volume Rendering Algorithms

Despite of the long history of volume rendering, there has not been a side by side comparison of different volume rendering methods so far, even though such a comparison could reveal some guidelines on which method performs best under certain conditions of different applications. The reason for this missing comparison might be that there are simply too many approaches and agreeing on a common set of core techniques might lack generality. Furthermore, different institutions have certain expertise but not necessarily access to implementations of other techniques, simply because there are only few available packages that provide source code. Yet another difficulty is defining a common framework such that the methods can really be compared side by side.

In order to determine the pros and cons, as well as avenues for future research, a comparison for the volume rendering algorithms which have become rather popular for rendering datasets described on uniform rectilinear grids is performed. The four algorithms are: ray casting, splatting, shear-warp, and hardware assisted texture mapping. For a direct side by side comparison, a common viewing framework has been developed, allowing to generate identical views for each method (see Appendix A). Furthermore, all algorithm-independent image synthesis parameters such as light sources, transfer functions, and optical model are kept constant to enable a fair comparison of the rendering results.

4.1 Introduction

Numerous volume rendering methods exist and for each of these methods a large variety of optimizations has been presented. A side by side comparison of all the existing approaches could easily fill an entire book but 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 is first converted into a set of polygonal iso-surfaces (e.g., via Marching Cubes [LC87]) and subsequently rendered with polygon graphics hardware. This is referred to as indirect volume rendering (IVR).

2. The volumetric data is directly rendered without the intermediate conversion step. This is referred to as direct volume rending (DVR) [DH92, Sab88, UK88].

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 accuracy. However, this is not always the case and some examples are: (i) amorphous cloud-like phenomena, (ii) smoothly varying flow fields, or (iii) structures of varying depth (and varying transparencies of an iso-surface), 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 to be more efficient [PSL+98], especially when the object is complex or large, or when the iso-surface is interactively varied and the repeated polygon extraction overhead must be figured into the rendering cost [BM99].

Within this chapter, the comparison focuses on the direct volume rendering approach, in which four techniques have emerged as the most popular: Ray casting [TT84, Lev88], splatting [Wes90], shear-warp [LL94], and 3D texture-mapping hardware-based approaches [CCF94]. However, a direct comparison of the Marching Cubes approach and ray casting has been performed earlier, examining the required computations, storage, as well as the resulting image quality [BM99]. Furthermore, others compared different gradient filters for ray casting and Marching Cubes [THB+90].

4.2 Common Theoretical Framework

All four surveyed algorithms obtain colors and opacities in discrete intervals along a linear path and composite them in front to back or back to front order, computing the DVRI which was presented earlier (see Section 1.2.7).

$$I_\lambda(x,\vec{r}) \quad = \quad \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\alpha(s_i)\prod_{j=0}^{i-1}(1-\alpha(s_j)) \tag{4.1}$$

However, the algorithms can be distinguished by the order in which the colors $C_\lambda(s_i)$ 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 and classification operator in the volume rendering pipeline also affects $C_\lambda(s_i)$ and $\alpha(s_i)$. For this purpose, the pre-shaded and pre-classified volume rendering pipeline can be distinguished from the post-shaded and post-classified volume rendering pipeline. In the pre-shaded and pre-classified

pipeline, the grid samples are shaded and classified before the ray sample interpolation takes place. In the following, this is denoted as Pre-DVRI (pre-shaded and pre-classified DVRI) and the computation of $C_\lambda(s_i)$ and $\alpha(s_i)$ are performed following Equation 4.1 by generating colors based on the colors at the surrounding grid locations. The left column of Figure 4.1 shows images of the fuel dataset[1] using Pre-DVRI. Pre-DVRI generally leads to blurry images for zoomed views where excessive

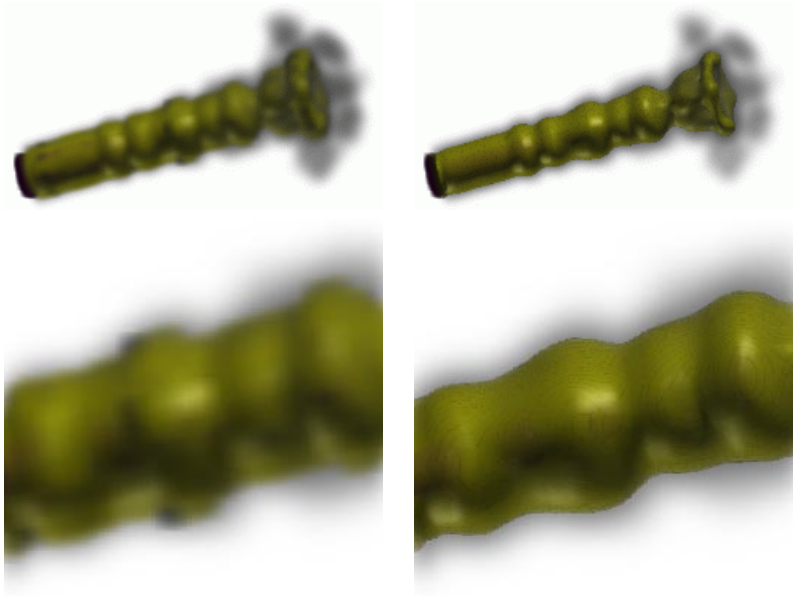


Figure 4.1: Images of the fuel dataset using Pre- (left column) and Post-DVRI (right column). Excessive blurring can be noticed for the Pre-DVRI, especially for zoomed views.

color interpolation takes place. The blurriness can be eliminated by switching the order of shading and classification which influences the ray sample interpolation. In this case, the original density volume f is interpolated and the resulting sample values at location s_i is $f(s_i)$. The resulting expression is termed Post-DVRI (post-shaded and post-classified DVRI).

$$I_\lambda(x,\vec{r}) \quad = \quad \sum_{i=0}^{L/\Delta s} C_\lambda(\text{Shade}(f(s_i)))\alpha(f(s_i))\prod_{j=0}^{i-1}(1-\alpha(f(s_j))) \tag{4.2}$$

where C_λ and α are the classification results and *Shade* is the shading function.

The resulting Post-DVRI images are shown in the right column of Figure 4.1. Since in Post-DVRI the raw volume densities are interpolated and the result used to index the classification for color and opacity, fine detail in the classification is readily expressed in the final image on a per sample (pixel) base.

However, one should note that Post-DVRI is not without problems. Due to the interpolation of density values, one might obtain interpolated values being classified as a structure that is not present at this location (partial volume effect). This can be

[1] The dataset is described in Section 4.4.4.

avoided using segmentation but can add severe stair casing artifacts due to introduced high-frequency.

4.3 Distinguishing Features of the Algorithms

The comparison focuses on the conceptual differences between the algorithms, and not so much on absolute performance. Since numerous implementations for each algorithm exist — mainly providing acceleration — the most general implementation for each was selected, employing the most popular components and parameter settings. The conceptual differences of the four algorithms are summarized in Table 4.1.

	Ray casting	**Splatting**
Rendering	Post-DVRI	
Sampling rate	freely selectable	
Sample evaluation	point sampled	averaged across Δs
Filter kernel	trilinear	Gaussian
Precision	floating point	
Voxels considered	all	relevant
Acceleration	early ray termination	early splat elimination

	shear-warp	**3D Texture Mapping**
Rendering	Pre-DVRI	
Sampling rate	fixed $[1.0, 1.73]$	freely selectable
Sample evaluation	point sampled	
Filter kernel	bilinear	trilinear
Filtering	opacity-weighted colors	no opacity-weighted colors[2]
Precision	floating point	8 – 12 bits
Voxels considered	mostly relevant	all
Acceleration	RLE opacity encoding	graphics hardware

Table 4.1: Distinguishing features of ray casting, splatting, shear-warp, and 3D texture mapping.

4.3.1 Ray Casting

Of all volume rendering algorithms, ray casting has seen the largest body of publications over the years. Researchers have used Pre-DVRI [Lev90, Lev88] as well as Post-DVRI [AHH+94, HPP+96, THB+90]. 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 of neighboring voxels (grid points) to maintain computational efficiency, and subsequently composited. Ray samples are mostly spaced apart in equal distances Δs, but one can also jitter the sampling positions

[2] Due to semi-transparent rendering and the limited precision of the hardware, opacity weighted colors are frequently too small when using low α-values. Hence, opacity weighted colors are not used within this comparison.

to eliminate patterned sampling artifacts, or apply space-leaping [DH92, YS93] 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 [PSL+98]. But in the general case, the Nyquist theorem which states that one should choose $\Delta s < 1.0$ (e.g., one voxel length) if the frequency content in the sample's local neighborhood is unknown needs to be taken into account. The $C(s_i)$, $\alpha(s_i)$, and $f(s_i)$ terms in Equations 4.1 and 4.2 are written as:

$$\begin{aligned} C_\lambda(s_i) &= C_\lambda(i\Delta s) \\ \alpha(s_i) &= \alpha(i\Delta s) \\ f(s_i) &= f(i\Delta s) \end{aligned} \tag{4.3}$$

The used implementation of ray casting performs Post-DVRI and follows Equations 4.2 but Pre-DVRI could easily be accomplished. Note that α needs to be normalized for $\Delta s \neq 1.0$ [LCN98]. The only algorithmic optimization exploited in the implementation is early ray termination, where rays can be terminated once the accumulated opacity has reached a value close to unity. All samples and corresponding gradient components are computed on the fly by trilinear interpolation of the respective grid data.

4.3.2 Splatting

Splatting was first proposed by Westover [Wes90] and represents voxels as overlapping basis functions. These basis functions are commonly Gaussian kernels with amplitudes scaled by the voxel values. An image is 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[3] must be projected and scan converted. This can tremendously reduce the volume data that needs to be both processed and stored [MSHC99]. Nevertheless, depending on the zooming factor, each splat can cover up to hundreds of pixels which need to be processed.

The preferred splatting approach [Wes90] accumulated the voxel kernels within volume slices most parallel to the image plane by sorting the kernel centers. This was inaccurate due to the overlapping kernels, did not allow for the variation of the DVRI interval distance Δs, and prone to severe variations of brightness in animated viewing. Image aligned splatting [MC98] eliminates most of these drawbacks by processing the voxel kernels within slabs of width Δs, aligned parallel to the image plane: all voxel kernels that overlap a slab are clipped to the slab and summed into a sheet buffer, followed by compositing the current sheet with the previous sheet. Efficient kernel slice clipping is achieved by analytical pre-integration of an array of kernel slices. Even though kernels do still overlap within sheet buffers, their sheet buffer sections are correctly integrated and the inaccuracy due to the kernel overlap gets smaller with decreasing slab width Δs. Both Pre-DVRI and Post-DVRI [MMC99] are possible, and the $C(s_i)$, $\alpha(s_i)$, and $f(s_i)$ terms in equations 4.1 and 4.2 are written as:

$$C_\lambda(s_i) = \frac{\int_{i\Delta s}^{(i+1)\Delta s} C_\lambda(s)ds}{\Delta s}$$

[3] Relevant voxels are all voxels which are classified as not transparent. These are identified by checking all voxels which needs to be done every time the classification changes.

$$\begin{aligned} \alpha(s_i) &= \frac{\int_{i\Delta s}^{(i+1)\Delta s} \alpha(s)ds}{\Delta s} \\ f(s_i) &= \frac{\int_{i\Delta s}^{(i+1)\Delta s} f(s)ds}{\Delta s} \end{aligned} \tag{4.4}$$

Splatting replaces the point sample of ray casting by a sample average across Δs. This introduces additional low-pass filtering reducing aliasing, especially in iso-surface renderings and when $\Delta s > 1.0$. However, when splatting data values and not color, classification is an unsolved problem since the original data value, e.g. density, is smoothed, accumulated, and then classified. This aggravates the so-called partial volume effect and solving this is still a topic of future research.

The used splatting implementations uses a concept similar to early ray termination to reduce the amount of redundant processing. This is referred to as early splat elimination and can be accomplished using a conservative screen occlusion map [MSHC99] which can be established convolving the opacity values. For efficient culling, the occlusion map is update every time a sheet buffer is completed. 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 sheetbuffer. The dynamic construction of the occlusion map requires a convolution operation after each sheet-buffer composite, which is a costly operation. Although early splat elimination reduces the cost of footprint rasterization for invisible voxels, their transformation must still be performed to determine their coordinates to perform the occlusion test. This is different from early ray termination where the ray can be stopped and subsequent voxels are not processed.

4.3.3 Shear-Warp

Shear-warp was proposed by Lacroute and Levoy [LL94] and is still one of the fastest software volume renderers. It achieves its speed by employing a clever encoding scheme in object and screen space. In a pre-processing step, the volume is RLE-encoded based on pre-classified opacities. Since this encoding scheme is axis aligned, it requires the construction of a separate encoded volume for each axis. Depending on the largest component of the viewing vector, the appropriate encoded volume is used. The rendering is performed using a ray casting-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 using bilinear interpolation within the traversed volume slices. During rendering, a screen space RLE-encoding is exploited and updated whenever a pixel does not alter its value any further[4]. In a final warping step, the image on the volume-parallel baseplane is transformed onto the screen plane. Since the shear-warp algorithm performs bilinear interpolation within volume slices only, the DVRI interval distance Δs is view-dependent. It varies from 1.0 for axis-aligned views to $\sqrt{2}$ for edge-on views to $\sqrt{3}$ to corner-on views, and can not be varied to allow for super-sampling along the ray. Thus, the Nyquist theorem is potentially violated.

The Volpack distribution from Stanford[5] (a volume rendering package using the shear-warp algorithm) only provides Pre-DVRI (with opacity weighted colors), but conceptually Post-DVRI is also feasible, however, without opacity classification if

[4] This occurs when either the pixel opacity reaches the maximum value or the corresponding ray leaves the volume.

[5] http://www-graphics.stanford.edu/software/volpack

shear-warp's fast opacity-based encoding is used. With respect to Equations 4.1 and 4.2, the $C(s_i)$, $\alpha(s_i)$, and $f(s_i)$ are written similar to ray casting, but with the added constraint that Δs is dependent on the view direction.

$$\begin{aligned} C_\lambda(s_i) &= C_\lambda(i\Delta s) \\ \alpha(s_i) &= \alpha(i\Delta s) \\ f(s_i) &= f(i\Delta s) \end{aligned} \tag{4.5}$$

$$\Delta s = \sqrt{\left(\frac{dx}{dz}\right)^2 + \left(\frac{dy}{dz}\right)^2 + 1} \tag{4.6}$$

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 baseplane (e.g., 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 baseplane), 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 resolution.

4.3.4 3D Texture Mapping Hardware

Within this section, only a short introduction to 3D texture mapping is provided and more details are disclosed in Section 5. The use of 3D texture mapping was popularized by Cabral [CCF94] for non-shaded volume rendering. The volume is loaded into texture memory and the hardware scan converts polygonal slices parallel to the viewplane. Usually, slices are blended in back to front order, due to the missing accumulation buffer for α. The interpolation filter is a trilinear function[6] and the slice distance Δs can be chosen arbitrarily. A number of researchers have added shading capabilities [GK96, DKC+98, MHS99, WE98]. Pre-DVRI [GK96] and Post-DVRI [DKC+98, MHS99, WE98] are possible but Post-DVRI sacrifices speed and accuracy. The rendering itself is brute-force, without any opacity-based termination acceleration. The drawback of 3D texture mapping is that larger volumes require 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 and increasingly more texture memory is available. Texture mapping hardware interpolates samples in similar ways to ray casting and hence the $C(s_i)$, $\alpha(s_i)$, and $f(s_i)$ terms in Equations 4.1 and 4.2 are written as:

$$\begin{aligned} C_\lambda(s_i) &= C_\lambda(i\Delta s) \\ \alpha(s_i) &= \alpha(i\Delta s) \\ f(s_i) &= f(i\Delta s) \end{aligned} \tag{4.7}$$

4.4 Common Experimental Framework

Unfortunately, not all parameters influencing the rendering process can be equally set for all four algorithms, e.g. texture mapping hardware operates on fixed-point data

[6] On SGI's RE 2 and IR architectures, also quadlinear interpolation is available to perform interpolation between mipmap levels.

while the software implementations use floating point. In the following, the general setup for the comparison as well as the differences which can not be circumvented are described.

4.4.1 Viewing

Comparing images of different volume rendering algorithms requires a common camera model and parameter set to be used across all rendering methods. A view is characterized by the type of projection which is either parallel or perspective, the location of the camera position, the view direction, the view-up vector as well as the opening angle of the camera. For the comparison, a common volume rendering framework has been developed (see Appendix A), allowing to provide identical view points for all algorithms. Only the shear-warp algorithm is not able to handle a viewing described by *gluLookAt*() but requires rotational angles[7]. This is the reason why some of the images generated using the shear-warp algorithm do not perfectly match.

Both rendering quality and expense are likely to be dependent on viewpoint, magnification, as well as image size. Within this chapter, the focus is on viewports of 256^2 pixels only. More detail of this comparison including other viewport sizes, magnifications, and animations can be found in [MHB+00].

4.4.2 Shading

For the comparison, the Phong illumination model has been chosen belonging to the category of local illumination models (see Section 1.1.1). The Phong illumination model can be written as:

$$C_\lambda = \text{Class}_\lambda(v) * (k_a I_a + k_d N L I_L) + k_s (NH)^{ns} * I_L \tag{4.8}$$

where C_λ is the resulting color of wavelength λ, v is the density value (grid or interpolated), Class_λ is the classification, I_L is the color of the light source, N is the gradient, H is the so-called half vector, ns the exponential factor to determine the size of the reflection highlight, and k_a, k_d, and k_s are the ambient, diffuse, and specular material properties.

Ray casting, splatting, and shear-warp use the Phong illumination model. However, the shear-warp implementation uses a look-up table based shading technique which does not allow for specular highlights, as described above. A specular highlight reflects the light of the light source but shear-warp replaces I_L by $\text{Class}_\lambda[f]$, as it can be seen in Figure 4.4(middle column) when looking at the specular highlights on the neghip[8].

For 3D texture mapping, the entire specular term is dropped because it depends not only on the light source(s) but also on the view point. Since the view point changes frequently, it would require to re-compute the three dimensional texture for every frame, which would prevent interactivity.

4.4.3 Compositing

As described earlier in Section 1.2.7, compositing can be performed in either front to back or back to front order. While the order does not affect the image quality, it certainly influences the rendering time because back to front compositing does not

[7] At least the publicly available implementation of VolPack.

[8] The neghip dataset is described in Section 4.4.4.

enable early ray termination or early splat elimination. However, front to back does require an accumulated α-channel which is not available in OpenGL. Therefore, the 3D texture mapping based approach performs back to front compositing while all other algorithms perform front to back compositing.

4.4.4 Datasets

A meaningful comparison requires highlighting different aspects of the algorithms from different perspectives. Thus, an appropriate selection of datasets is of crucial importance for the comparison. Five real-world datasets of different origin as well as one synthetic dataset were selected for the comparison (see Table 4.2).

Dataset	Size	Relevant voxels [1K]	([%])	Compactness	Pixel content
Vessels	256^3	79	(0.5)	low	low
Neghip	64^3	208	(79.3)	high	medium
Skull	256^3	1,385	(8.2)	medium	low
Fuel	64^3	33	(12.5)	medium	medium
Shockwave	$64^2 \times 512$	1,245	(59)	high	high
MLob	41^3	35	(51)	high	low

Table 4.2: Real-world datasets of comparison. MLob denotes Marschner-Lobb.

A volumetric object can be characterized by the amount of relevant material contained in its convex hull, which is referred to as compactness. A highly compact object, such as a brain or an engine block, fills a large percentage of its enclosing space. On the other hand, a highly dispersed object, such as a blood vessel tree, has relatively few relevant voxels within its convex extent. Thus, the compactness of a volume is not always pre-defined, but can be altered by the opacity transfer function. E.g., a formerly compact MRI head, may turn into a sparse and dispersed blood vessel tree, depending on the classification.

Apart from compactness, another useful measure is the amount of voxel material that contributes to a pixel. This will be referred to as the pixel content. Depending on the selected classification, the pixel content can be completely different. For semi-transparent renderings of a particular object significantly more object voxels are considered than for opaque renderings. The pixel content as described in Table 4.2, results from the classification applied during rendering (see Figures 4.2, 4.3, and 4.4) and denotes the average amount of contributions per pixel.

Using the definition of compactness and pixel content, the different datasets are introduced:

Fuel injection: Physical simulation of fuel being injected into a cylinder of an engine filled with air. This is a semi-transparent, but compact, representation that requires many samples to be taken for each pixel.

Neghip: Physical simulation of a high potential protein representing the electron probability around each atom (blue is high, green is medium, and red is low). This dataset is highly compact.

Skull: Rotational biplane X-ray (rotational angiography) scan of a human head. Bones and teeth are well scanned. The classification of the data into skull and teeth yields moderate compactness. The opacity classification also enables early ray termination, and many voxels are occluded.

Blood vessel: Rotational biplane X-ray (rotational angiography) scan of a human brain where a contrast agent has been injected into the blood to capture the blood vessel. This dataset is characterized by its very low compactness and pixel content.

Shockwave: Simulation of an unsteady interaction of a planar shockwave with a randomly perturbed contact discontinuity, rendered with a highly translucent opacity (α). All voxels potentially contribute to the display.

Marschner-Lobb [ML94]: High frequency test dataset, rendered as an iso-surface. This dataset is synthetic and used to assess rendering (filtering) quality in a quantitative manner.

4.4.5 Assessment of Image Quality

It is difficult to evaluate rendering quality in a quantitative manner. Often, images of competing algorithms are simply put 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 small numerical errors, e.g., RMS, are judged as worse by a human observer than images with larger numerical errors. So it seems that the visual comparison is more appropriate than the numerical, since after all images are generated 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 one still needs the true volume rendered image, obtained by analytically integrating the volume via Equation 1.3 (neglecting the prior reduction of the volume rendering task to the low-albedo case). As was pointed out by Max [Max95], analytical integration can be done when assuming that $C(s)$ and μ(s) are piecewise linear. This is, however, somewhat restrictive on classification. Hence, visual quality assessment are employed only.

Additional to the real-world datasets, a particularly challenging dataset was chosen for visual quality assessment: the Marschner-Lobb function [ML94]. This three-dimensional function is made of a combination of sinusoids and contains very high frequencies. However, 99.8% of these are frequencies which are just below the Nyquist limit (half the Nyquist rate). It is extremely sensitive to filter and sampling inaccuracies and has been used at many occasions for reconstruction error evaluations [MMMY97].

4.5 Results

Before one goes ahead and compares rendering times and quality, one needs to realize that not all evaluated volume renderers were created with identical priorities in mind. While shear-warp and 3D texture mapping hardware were devised to maximize framerates on the expense of rendering quality, ray casting and image-aligned splatting have been devised to achieve images of high quality, not to be compromised by acceleration strategies employed. To account for this, the four renderers need to be subdivided into two groups:

High-performance volume renderers: Shear-warp and 3D texture mapping hardware. These renderers use the Pre-DVRI optical model. Shear-warp with and 3D texture mapping without opacity weighted color interpolation.

High-quality volume renderers: Splatting and ray casting. These renderers use the Post-DVRI optical model.

All presented results were generated on a SGI Octane (R10000 CPU, 250MHz) with 250 MBytes main memory and MXE graphics with 4MBytes of texture memory. The graphics hardware is only used by the 3D texture mapping approach.

Figure 4.2, 4.3, and 4.4 show a comparison of different images of the selected datasets varying the applied magnification levels. For the high-performance renderers, we observe that the image quality achieved with 3D texture mapping shows severe color-bleeding artifacts — as expected —, due to the non-opacity weighted colors [WMG98]. The limited precision in the hardware, prevents the usage of opacity weights colors when using semi-transparent classification. Furthermore, 3D texture mapping shows stair casing artifacts which can be reduced by increasing the number of slices. Volpack's shear-warp performs much better, with quality similar to ray casting and splatting whenever the resolution of the image matches the resolution of the baseplane (full view on vessels and skull in Figure 4.2). For the other images, the rendered baseplane image is of lower resolution than the screen image and magnified using bilinear interpolation in the warping step, which leads to excessive blurring. A draw-back which is not visible in the orthogonal views is the angle dependent sampling distance which can result in significant aliasing in the form of stair casing.

Looking at the high-quality renderers, the Marschner-Lobb dataset renderings for ray casting and splatting demonstrate the differences of point sampling (ray casting) and sample averaging (splatting). While ray casting'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 (right column of Figure 4.3). For the other datasets the averaging effect is more subtle, but still visible. For example, ray casting renders the skull and the magnified vessel with somewhat crisper detail than splatting does. Even though the quality of ray casting and splatting is comparable, there are still subtle differences in the images rendered by each algorithm. These differences are most apparent for the fuel, neghip, and shockwave datasets. For example, the red cloud is completely missing in the image rendered by splatting and the purple core of the fuel injection is much larger when using splatting, see middle column in Figure 4.4. This is due to the Gaussian filtering and accumulation within sheet buffers, resulting in a slight shift in the density values[9].

Even though the implementations of the algorithms have not been optimized for speed, some analysis can be performed. For each dataset, an animation has been generated measuring the overall rendering time. Table 4.3 shows the results averaged per frame (more details on the rendering times can be found in [MHB+00]).

Generally, 3D texture mapping and shear-warp take sub-second rendering times. While texture mapping is back to front and hence brute force, shear-warp exploits run-length encoding and early ray termination. Thus, shear-warp performs better on datasets with low pixel content and/or few relevant voxels than on others: it takes shear-warp roughly three times longer to render the translucent shockwave than the

[9] In Post-DVRI splatting, classification is an unsolved problem since the original data value is smoothed, accumulated, and then classified. This aggravates the so-called partial volume effect and is still topic of future research.

	Fuel	Neghip	Skull	Vessel	Shockwave
Ray casting	4.96	8.15	7.78	12.31	3.02
Splatting	1.41	7.35	11.09	1.87	21.77
Shear-warp	0.09	0.24	0.27	0.09	0.91
Texture Mapping	0.06	0.04	0.71	0.71	0.14

Table 4.3: Rendering times for each dataset given in seconds. The numbers are averaged over a set of screen fitting views.

opaque skull, although both have about the same number of relevant voxels. Note that 3D Texture mapping is slower for large datasets (skull, vessel) which exceed the 4 MBytes of texture memory of the SGI Octane/MXE and require texture swaps.

In contrast, ray casting and splatting are in the order of seconds for all presented images. However, since splatting processes only relevant voxels, it is quite fast when rendering the vessel dataset. In this case, it is roughly half the speed of 3D texture mapping. Generally, compactness does not have a major effect on splatting but a high pixel content can be very expensive in case the early splat elimination can not be exploited which corresponds to semi-transparent rendering as for the shockwave dataset.

Ray casting shows its strength for medium and highly compact datasets with a high pixel content, e.g. the shockwave dataset. In contrast, a high number of non-relevant samples can dominate the rendering time of ray casting, as observed with the vessel and fuel datasets. In both cases, most of the rays are cast solely through non-relevant voxels, wasting render time. This is different with the skull dataset, where early ray termination skips most of the volume. The associated costs of early ray termination are low, since it is a simple comparison of the α-value with a specified threshold. In contrast, splatting's early splat elimination has high associated costs, and that is why splatting takes considerably longer to render the skull dataset.

The shockwave dataset — where the low opacity of all voxels prevents both early ray termination and early splat elimination – exposes the differences in cost for trilinear interpolation vs. footprint mapping. Since the rendering time for ray casting is almost seven times lower than that of splatting, it can be concluded that the mapping of footprint kernels is costlier than trilinear interpolation, at least at moderate screen sizes. When moving to larger screen size, splatting scales better because the amount of relevant voxels is independent of image dimension [MHB+00]. The increases in rendering time are due to (i) larger footprints to be rasterized and (ii) larger sheetbuffers to be convolved to update the occlusion map. Thus, datasets with many sheets (large depth) and many visible footprints will be more sensitive to screen size increases. For ray casting, the number of trilinear interpolations and compositings are linearly related to screen size. No further major dependencies exist.

4.6 Summary

Generally, 3D texture mapping and shear-warp have sub-second rendering times for moderately-sized datasets. While the quality of the display obtained with the mainstream 3D texture mapping approach is limited and can be improved as demonstrated in Chapter 5, the quality of shear-warp rivals that of the much more expensive ray casting and splatting when the object magnification is about unity. Handling higher mag-

nifications 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 classification 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 ray casting yielded interesting results as well: image-aligned splatting offers a rendering quality similar to that of ray casting. It 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. However, ray casting 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 ray casting or splatting, depending on the number of relevant voxels and the level of compactness of the dataset.

For ray casting, the problem areas are the transparent regions in front of the opaque object portions, while for splatting the problem areas are the non-transparent regions behind the opaque object portions. This may be a compromising fact for splatting, since large datasets may have a lot more material hidden in the back than empty regions in front. In addition, a great number of powerful techniques exist for ray casting to guide rays quickly through irrelevant volume regions: bounding volumes, space-leaping, multi-resolution grid traversal, and others. It has yet to be determined if these techniques also work well with highly irregular objects such as the blood vessel dataset, where splatting performs better.

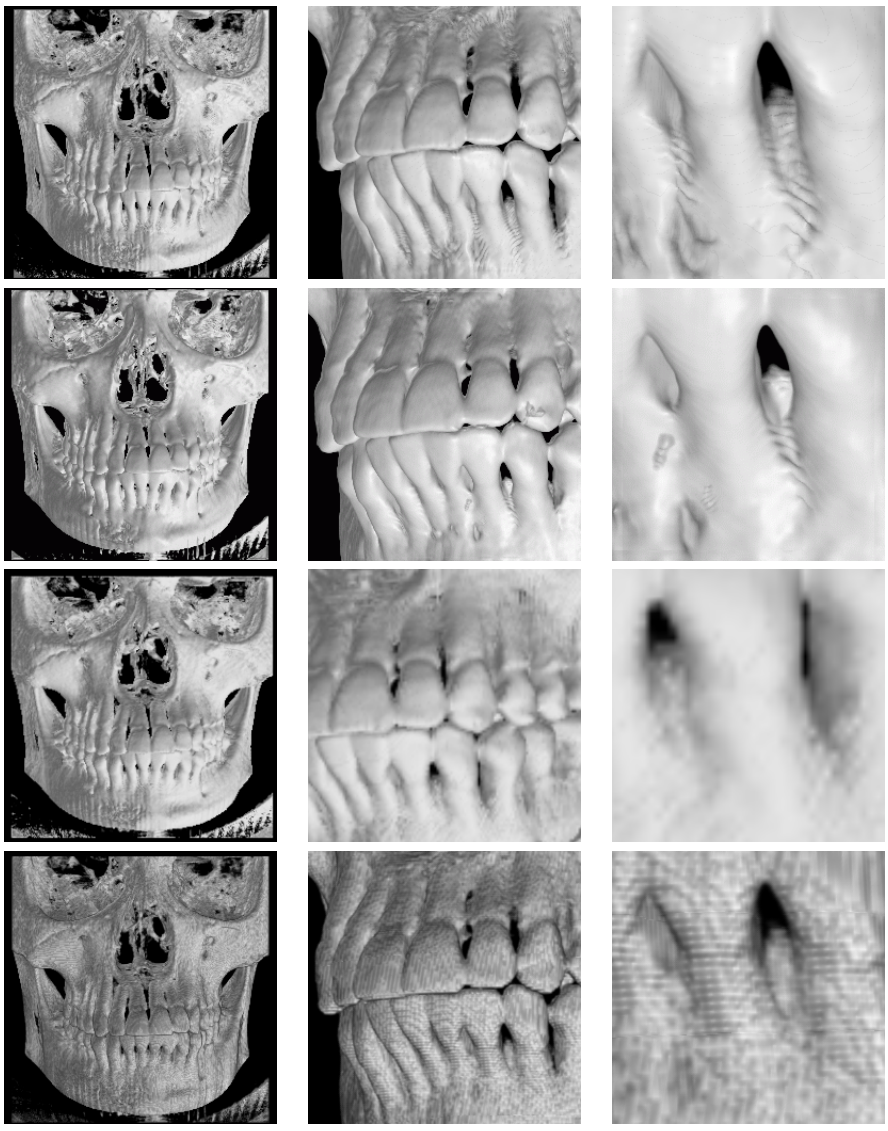


Figure 4.2: First through fourth row are ray casting, splatting, shear-warp, and 3D texture mapping showing the skull dataset at different magnifications.

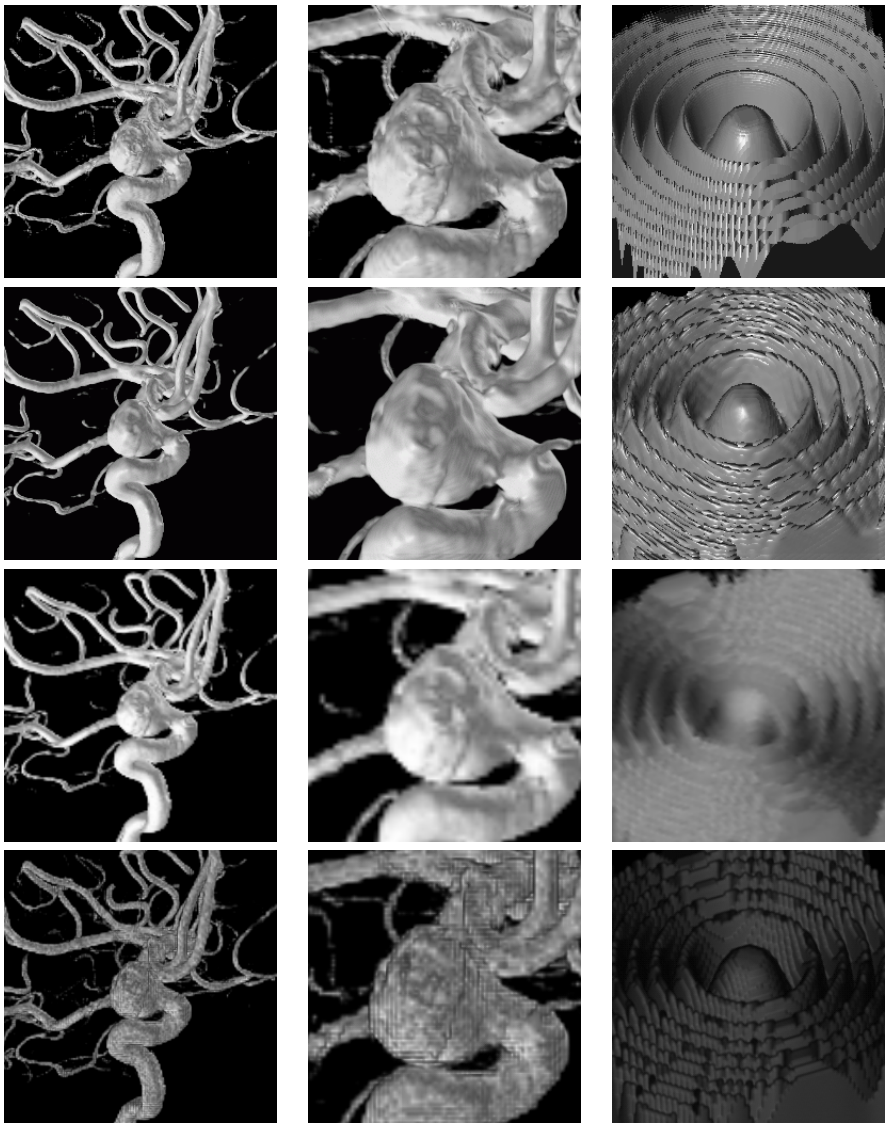


Figure 4.3: First through fourth row are ray casting, splatting, shear-warp, and 3D texture mapping showing the blood vessel dataset at different magnifications (column 1 and 2) and the Marschner-Lobb dataset (column 3).

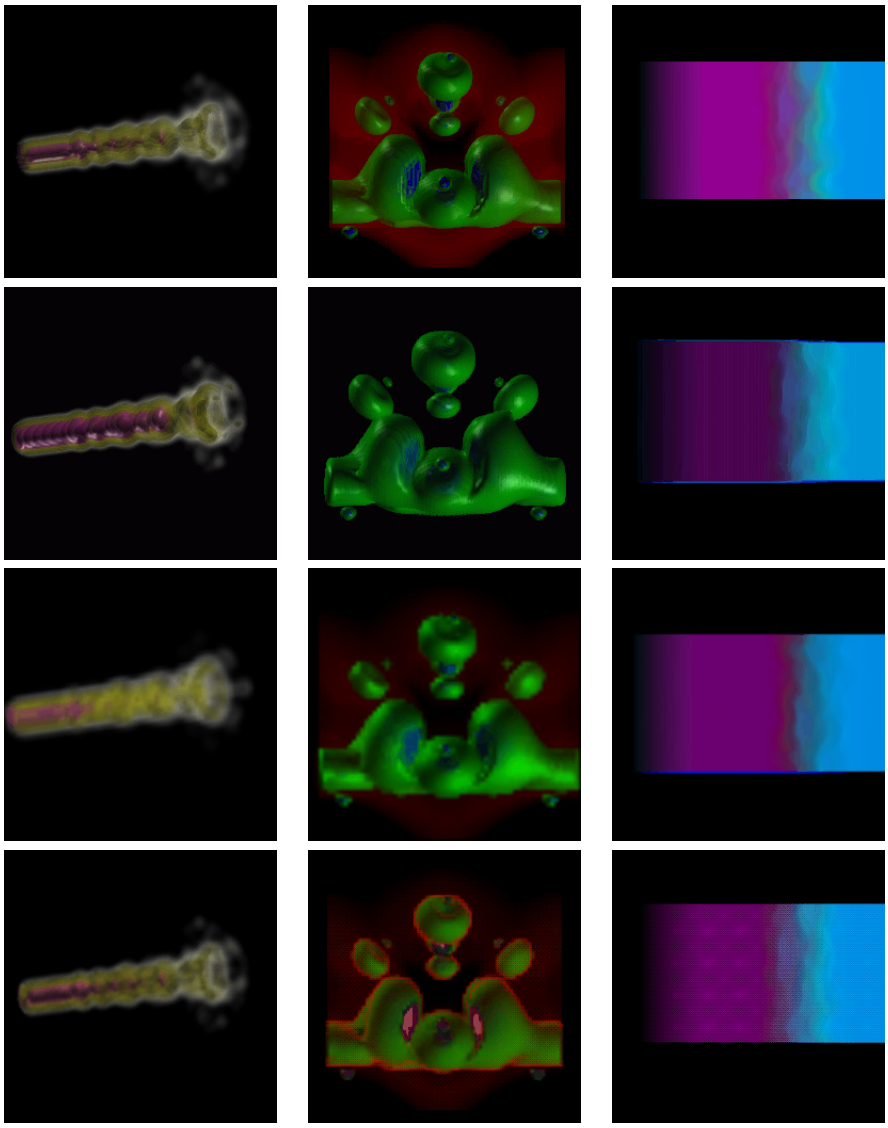


Figure 4.4: First through fourth row are ray casting, splatting, shear-warp, and 3D texture mapping showing the fuel (column 1), neghip (column 2), and shockwave dataset (column 3). It can clearly be noticed that for splatting there is a difference in the classification which is due to the classification performed on smoothed and accumulated voxel values in the sheet buffers. For 3D texture mapping, the color bleeding is fairly obvious and for shear-warp crosses appear (as a result of the bilinear filtering).

Chapter 5

Enabling Shading and Classification for 3D Texture Mapping Based Volume Rendering

3D texture mapping has become one of the most popular techniques used for volume rendering. Soon after the release of the first graphics workstation with hardware support for three dimensional textures (RealityEngine), their potential application for volume rendering was presented. Its favorite advantage is the achievable interactivity which is very important for many volume rendering applications and due to the hardware support.

Despite of its high frame-rates, 3D texture mapping has certain disadvantages since it is primarily designed for polygon graphics and not for volume graphics. While sample interpolation is very similar, shading is performed on a per vertex base but volume rendering requires shading on a per sample base. Furthermore, classification requires a lookup to obtain color and α values for interpolated volume samples and is available on a few architectures only.

In the following, the current state of the art is reviewed and an approach presented enabling the integrating of shading functionality into 3D texture mapping based volume rendering. Furthermore, it is demonstrated how multiple classification spaces can be enabled to apply “intelligent lenses”. Finally, the ideal datapath for graphics hardware to enable accurate shading including multiple light sources is proposed.

5.1 Introduction

Using standard OpenGL 3D texture mapping functionality, any geometry defined using vertices and corresponding texture coordinates is rendered such that values on the geometry surface are trilinearly interpolated from the 3D texture data. To ensure that the regions of the geometry within the 3D texture are processed correctly, clipping planes can be applied [GK96]. Operating on transparent data requires that planes intersecting with the 3D texture are correctly blended into the framebuffer. Usually, planes are processed in back-to-front order, however, front-to-back order can be used as well but the accumulated opacity needs to be stored for each pixel[1]. Figure 5.1 depicts the overall process of volume rendering using 3D texture mapping.

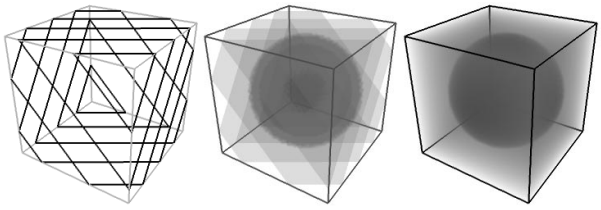


Figure 5.1: Basic slicing mechanism applied in 3D texture mapping based volume rendering (image is courtesy of [WE98]).

Utilizing 3D texture mapping for volume rendering is very fast due to the hardware support. Any three-dimensional grid data can be classified, possibly shaded, and stored as a three dimensional texture [CCF94, CN93, GK96, WGW94]. Alternatively, density values [SDWE98] and gradients can be stored as three-dimensional texture [EMPG97, WE98, DKC+98], which requires scaling the gradient components ($[-1,1]$) to fit into the range of the texture data ($[0,1]$). This is achieved by normalizing the gradients at grid positions, adding one, and dividing them by two.

5.2 Classification and Shading

Classification and shading are two of the most important elements in volume rendering. Classification is the stage of assigning color values and possibly other material properties to a sample based upon its scalar value (see Section 1.2.3). Shading can be used to enhance the visual quality of the rendered images by providing another cue to the human visual system (see Section 1.1.1). While shading is already available in polygon graphics hardware, it has so far only been applied to vertices[2] but not on a per sample base as it would be needed for volume rendering.

5.2.1 Classification

The classification of volume data is used to assign a color and an α-value to each scalar and is generally achieved using a lookup where the scalar value serves as index. When

[1] This can be circumvented when using colors being pre-multiplied with α but introduces inaccuracies for semi-transparent rendering due to the limited precision.

[2] Per pixel shading has recently become popular and can be accomplished combining textures and the color matrix [Hei99]. Unfortunately, the normals used for shading are not yet the ones which are interpolated from a three dimensional texture nor can this be combined with classification.

using 3D texture mapping for volume rendering, volume samples are available right after the texturing stage. To accomplish classification, one can use a SGI extension of OpenGL (GL_TEXTURE_COLOR_TABLE_SGI) which allows to treat the value of each channel (RGBα) as an index into a color table. The value of each channel is replaced by the corresponding entry of the color table and passed on to the fragment pipeline, as indicated in Figure 5.2.

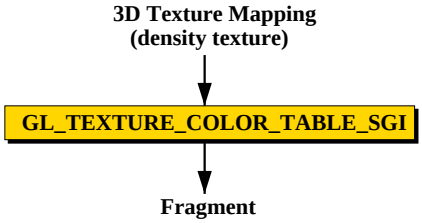


Figure 5.2: Classification using the OpenGL extension of SGI.

Despite of the simplicity and usefulness of this extension, it has certain limitations. First, the extension is not part of core OpenGL and currently supported only by SGI, HP[3], and very recently on ATI's Radon chip. Second, due to the circumstances that each channel is treated independently, one needs to replicate the scalar value four times and store it as RGBα texture. This tremendously increases the amount of memory needed. Furthermore, this method can not be combined with shading since no gradient information can be computed from the volume stored as three dimensional texture.

5.2.2 Shading Iso-Surfaces

Shading iso-surfaces of volumetric data has been first presented by Westermann et al. [WE98]. In their approach, scalar values and gradients are stored as three dimensional texture. During rendering of slicing planes, for each pixel the first interpolated sample above the iso-value is stored in the framebuffer using the iso-value as a threshold for the α-channel. After rendering all slicing planes, the framebuffer holds a full image containing the density values (in the α-channel) and the corresponding gradients (in the RGB-channels). In a final step, the content of the framebuffer is copied onto itself. During this copying process, the values are read from the framebuffer and sent down the pixel pipeline where the color matrix is applied to the pixel values. Additionally, the post color matrix scale and bias is used to obtain a diffuse as well as an ambient intensity.

In order to obtain the shading intensity, the color matrix needs to be initialized containing the light vector components, as illustrated in the following equation:

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

The matrix to the left of M_{rot} is required to calculate the scalar product of gradient (stored in RGB) and light vector. M_{rot} represents the current viewing transformation, needed to keep the light relative to the observer. Multiplying the matrix with a vector

[3]The fx10 graphics system of HP provides this extension.

containing the interpolated gradient calculates the diffuse shading intensity (I_d) for all three color channels. Evaluating $I = I_a + k_d * I_d$ results in the final intensity, which can be accomplished using the *post-color matrix scale and bias* stage (scale by k_d and bias by I_a), which comes right after the color matrix, see Figure 5.3.

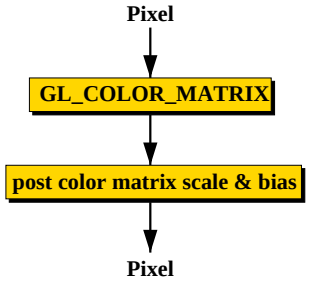


Figure 5.3: Shading using color matrix.

The above described method achieves impressive results at interactive frame-rates but has some limitations. The method requires that the α value of the fragment multiplied with the color matrix is set to one, otherwise no correct result is obtained since the gradient needs to be scaled back from $[0,1]$ to $[-1,1]$. This prevents semi-transparent representation because the α channel can not be used for other purposes. Furthermore, this shading technique is limited to one directional light source only and monochrome shading since all three color channels are needed for the gradient components.

5.2.3 Shading and Classifying Volume Data

The previously described classification and shading techniques can not be combined since classification using GL_TEXTURE_COLOR_TABLE_SGI requires the scalar value to be present in all four channels which prevents storing gradients together with the scalar value. Furthermore, semi-transparent rendering — one of the strength of volume rendering — is not possible due to the requirement of setting the α value to one.

To circumvent these limitations, a new approach has been developed. Similarly to [WE98], gradients and density values are stored in a three dimensional texture map. However, the restriction of being limited to opaque rendering only is circumvented by re-scaling the gradient vector from $[0,1]$ to $[-1,1]$ right at the beginning of the pixel pipeline. This can be accomplished using the scale and bias functionality of the pixel transfer operations since values within the pipeline are not clamped back to $[0,1]$ until they enter the per fragment operation stage. Therefore, there is no need to set the α-value to one and hence, it can be used for semi-transparent rendering.

The computation of the diffuse shading intensity is again done using the color matrix. However, the color matrix needs to be initialized differently since it is necessary to compute the scalar product (diffuse shading intensity) but also keep the interpolated density values for classification. This is achieved by the following initialization of the color matrix:

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

The fragments containing the interpolated gradient and density are then — during copying the pixels — multiplied with the color matrix which results in:

$$M_{col}\begin{bmatrix} R \\ G \\ B \\ \alpha \end{bmatrix} = \begin{bmatrix} \alpha \\ < L, N_{rot} > \\ 0 \\ 0 \end{bmatrix} == \begin{bmatrix} D \\ I_d \\ 0 \\ 0 \end{bmatrix}$$

where $< L, N_{rot} >$ denotes the scalar product of the light vector and gradient. The density value of the red channel (D) could be used to perform the classification and obtain a RGBα quadruple which needs to be multiplied with $I_d * k_d$ intensity (diffuse term) plus an ambient term $I_a * k_a$. Unfortunately, evaluating the shading equation is not feasible in hardware, since the remaining pipeline stages do not provide the functional units. However, it can be achieved using a pixel texture.

Pixel textures are an extension to OpenGL (GL_SGIX_PIXEL_TEXTURE) which enables the interpretation of pixel-values as texture coordinates. R, G, B, and α can be used as texture coordinates s, t, r, and q. Pixel textures take place right at the conversion of pixels to fragments and in case pixel textures are enabled, the color of the fragment is replaced by the result of a texture mapping operation. This process is depicted schematically in Figure 5.4.

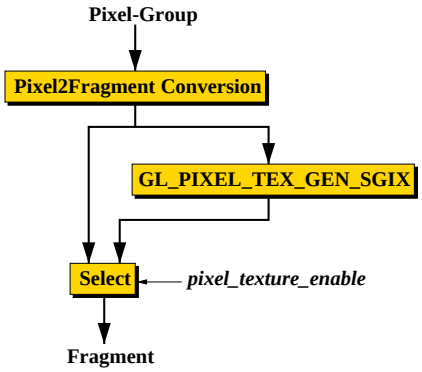


Figure 5.4: Pixel textures within the graphics subsystem.

Thus, the missing step of finally performing classification and evaluating the shading equation is accomplished applying a two dimensional pixel texture. The pixel texture can be addressed using the interpolated density (D) and the calculated diffuse intensity (I_d) as s and t texture coordinates respectively (R and G channel), which results in a two dimensional pixel texture. Despite of the fact that only the diffuse shading term is calculated, ambient intensity (I_a) can be included as well as different k_a and k_d factors, possibly for each density value D. This enables to include different material properties for the different density values. In one dimension the pixel texture represents the classification and in the second dimension the classification values are scaled and biased by an ambient term. Thus, for each D and k_d, the pixel texture contains a tuple $< RGB\alpha >$ and these tuples are computed using the following equation:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = k_d(D) * I_d * \begin{bmatrix} R(D) \\ G(D) \\ B(D) \end{bmatrix} + k_a(D) * I_a * \begin{bmatrix} R(D) \\ G(D) \\ B(D) \end{bmatrix}$$

$$\alpha = \alpha(D)$$

The pixel texture needs to be re-generated whenever the classification, or the factors k_a and k_d ($k_a + k_d = 1.0$) are changed by the user. However, due to its small size (2D, 65K entries), the pixel texture can be calculated interactively. Additionally, when changing the number of slicing planes, the opacity can be adjusted very quickly by simply updating the α-values of the 2D pixel texture. This is different to approaches storing a shaded and classified volume in texture memory because each time one of the above mentioned parameters or the light source changes, the entire 3D texture has to be recalculated. The latter can clearly not be accomplished at interactive frame-rates.

With this new pixel texture based approach, it is possible to generate (i) a colored and shaded iso-surface by applying the shading and classification step only to the final image or (ii) colored and shaded volume data by applying the shading and classification step to all slicing planes. Sample images are given in Figure 5.12.

5.3 Multiple Classification Spaces

As mentioned earlier, classification is one of the key features in volume rendering but the limitation of a single transfer function can be quite severe. Any material surrounded by a second material can only be visualized by classifying the surrounding material as transparent. However, when exploring the data, understanding the 3D relationship of both materials can be very important.

A simple way of realizing classification spaces are clipping planes and clipping geometry which have been introduced earlier [Ake93, GK96, WE98]. The major drawback of clipping planes or clipping geometry is that they simply clip fragments. This is equivalent to classifying the corresponding samples fully transparent but does not enable any other classification within this area/space. Clipping planes are enabled using the available hardware mechanisms and clipping geometry is realized by determining the cross-section of each plane with the geometry using the stencil buffer to enable or disable rendering of individual pixels.

This scheme can be extended to dual classification spaces using two different pixel textures. In the first pass, all pixels inside the cross-section are rendered using the first pixel texture and during a second pass the other pixels are rendered using the second pixel texture. This enables the application of *volumetric lenses* which can be extremely useful for certain volume rendering applications. Figure 5.5 shows a few examples of using simple clipping versus classification spaces. Figure 5.5(a) shows the neghip128 dataset using a spherical clipping geometry discarding the fragments which reside within the lens while Figure 5.5(b) uses different classification within the lens than outside. Figure 5.5 (c) shows the silicium dataset where the samples inside a cube have been clipped. In contrast, Figure 5.5 (d–f) show renderings where different classification is applied inside and outside of a cubic clipping geometry. Further examples of multiple classification spaces enabling “intelligent lenses” are depicted in Figure 5.12.

Generally, more than two classification spaces can be accomplished using more than two rendering passes. However, the use of multi-pass rendering sacrifices speed since each plane needs to be rendered more than once.

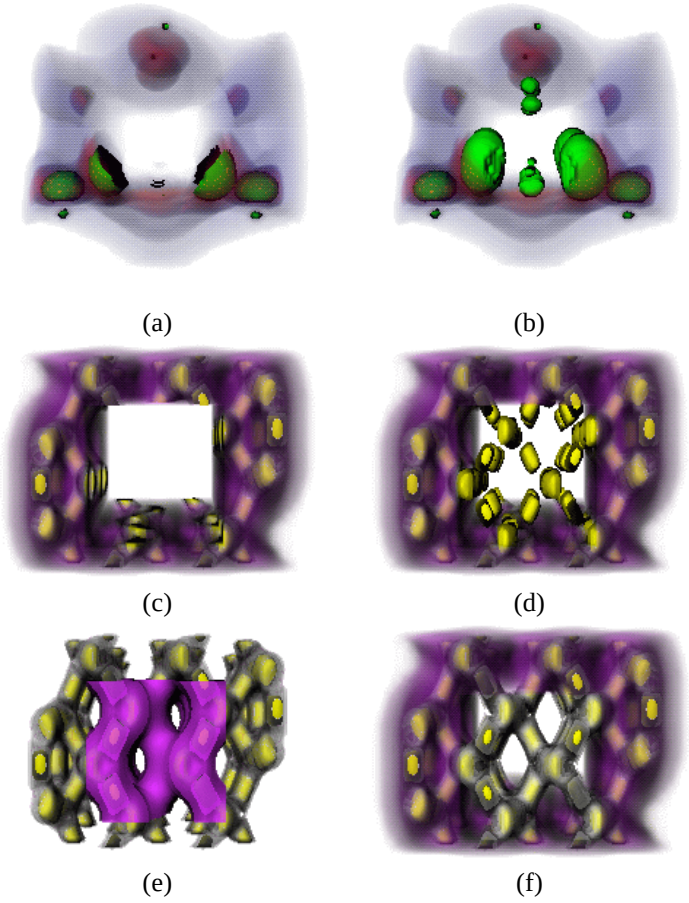


Figure 5.5: Clipping and multiple classification spaces for the neghip and the silicium dataset: (a,c) Simple clipping geometry. (b,d-f) Two classification spaces.

5.4 Results

In this section, different datasets used for rendering and resulting images with and without shading are presented. Furthermore, timing results including an analysis of the rendering time is given. All tests have been performed on a SGI Octane MXE with 4 MBytes of texture memory. Finally, a careful analysis of the error made in computing the scalar product from interpolated gradients is given.

5.4.1 Datasets

Several datasets of different size and characteristic have been chosen carefully to highlight the impact of the texture memory size. Since density values and gradients are stored in texture memory, datasets of up to $128 \times 128 \times 64$ voxels fit entirely into the texture memory of an Octane. Larger datasets require bricking and result in a reduced frame-rate. Due to the requirements of texture mapping, the datasets have to be ad-

justed in size such that each dimension is a power of two. The datasets are presented in Table 5.1, including their size and number of bricks. All datasets consist of 8 bit voxel data. The neghip64, silicium, and lobster fit within the 4 MBytes of texture memory available on the Octane while the neghip128 and engine need to be split into bricks. Except the engine dataset, all datasets have been taken from the VolVis package [AHH+94].

Dataset	MByte	bricks	Size
neghip64	1	1	$64 \times 64 \times 64$
silicon	2	1	$128 \times 64 \times 64$
lobster	4	1	$128 \times 128 \times 64$
neghip128	8	2	$128 \times 128 \times 128$
engine	32	8	$256 \times 256 \times 128$

Table 5.1: Test datasets. Due to the requirement of texture mapping each dataset has to be sized to a power of two. The size of each dataset includes voxel data plus gradient.

5.4.2 Images

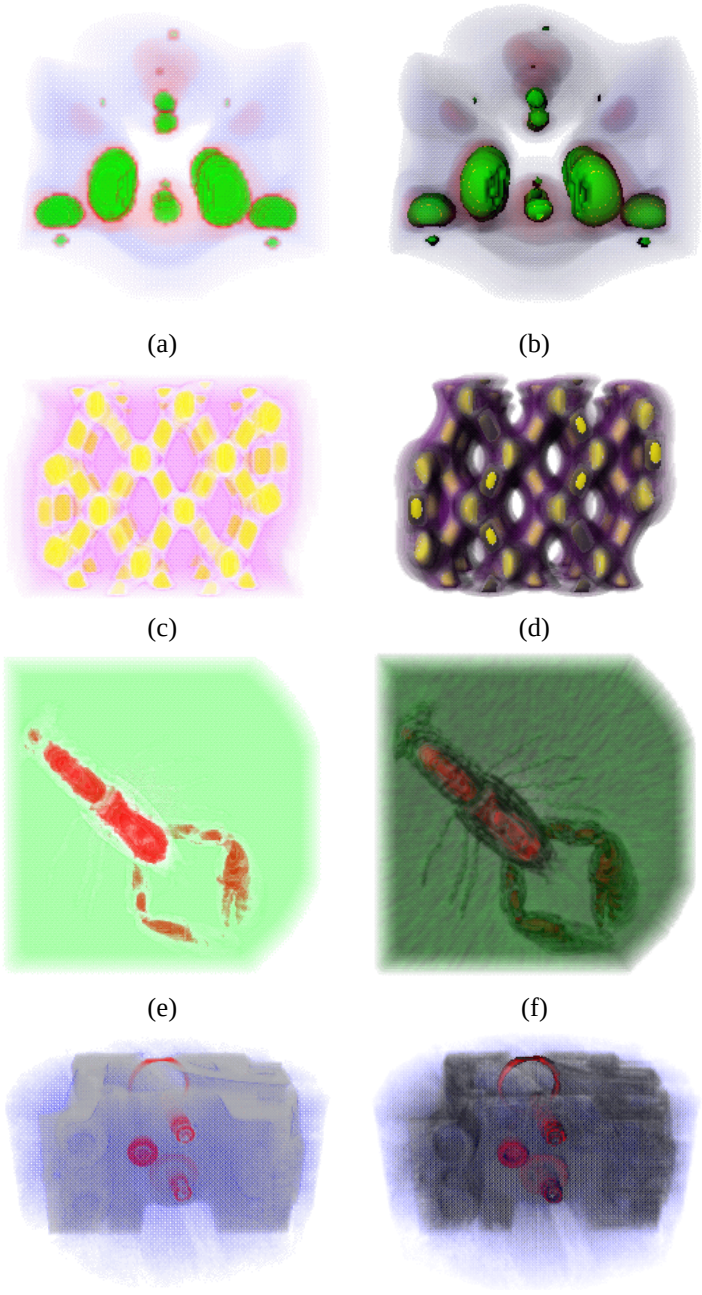
All datasets have been rendered using classification only as presented in Section 5.1 (referred to as the classical method) and using the color matrix and pixel texture combining shading and classification (referred to as ColMatPixTex).

Figure 5.6 (a) and (b) are images of the neghip64 dataset. While the three dimensional structure can be well understood using ColMatPixTex (b), no depth nor spatial extend can be extracted when using classification only (a). The images generated rendering the silicium dataset are shown in Figure 5.6 (c) and (d). Again, without shading (c) hardly any structure can be conceived. In contrast, using shading and classification reveals the structure comprehensively (d). Figure 5.6 (e) and (f) show images generated from the lobster dataset. Three different materials can be classified: Resin, shell, and meat of the lobster each classified green, white, and red respectively. Figure 5.6 (e) clearly lacks any illustration of the three dimensional character of the data, while Figure 5.6 (f) reveals the shape of the lobster and its shell. Finally, the engine dataset depicted in Figure 5.6 (g) and (h) contains an engine block, iron material, and noise around it. Using the classical approach (Figure 5.6 (g)), the structure of the engine block is exposed to a limited extend. In contrast, color matrix and pixel texture based shading and classification reveals the structure of the different materials much better (see Figure 5.6 (h)).

5.4.3 Timings

For a proper analysis of the bottle-necks, rendering time was measured for different settings. Furthermore, the rendering time has been split in two different steps:

Render-time I which is the time consumed to slice the three dimensional texture and to write the interpolated planes from the pixel buffer into main memory using *glReadPixels()*.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 5.6: Classical method (left column) and ColMatPixTex (right column). (a,b) Neghip64. (c,d) Silicon. (e,f) Lobster. (g,h) Engine.

Render-time II reveals the time needed to perform the shading and classification using *glDrawPixels()*[4].

Table 5.2 shows the timings for all datasets with varying viewport size. Since *glDrawPixels()* is required, the interpolated slices are copied into main memory and immediately sent back to the graphics pipeline without altering them. Between 60 and 80% of time I is spent reading the slices into main memory and only 20 to 40% are used for rendering (3D texture mapping). Even worse, sending the values back to the graphics subsystem takes over 90% of time II and only 10% are spent on the pixel texture and per fragment operations. Hence, efficient support of pixel textures using *glCopyPixels()* is mandatory to achieve interactive frame-rates ($\geq$ 10 frames). The color matrix is already part of the imaging subsystem of OpenGL 1.2 and pixel textures will be supported on future SGI platforms, not limited to IMPACT systems. It would certainly be a great feature if pixel textures would be supported by one of the next OpenGL releases, not only for volume rendering [Hei99].

Dataset	viewport	slices	time	time I	time II	Image
neghip64	256×256	64	0.34	0.07	0.25	Figure 5.6 (b)
neghip64	512×512	64	0.96	0.25	0.71	-
silicium	325×163	64	0.34	0.07	0.25	Figure 5.6 (d)
silicium	650×326	64	0.87	0.24	0.63	-
lobster	256×256	64	0.39	0.10	0.29	Figure 5.6 (f)
lobster	512×512	64	1.23	0.34	0.89	-
neghip128	256×256	128	0.73	0.23	0.51	-
neghip128	512×512	128	2.08	0.58	1.50	-
engine	256×256	128	1.05	0.48	0.57	Figure 5.6 (h)
engine	512×512	128	2.77	0.95	1.82	-

Table 5.2: Timing results of the different datasets. *time* is the overall time needed to render one frame. *time I* is the time spent slicing the 3D texture and writing the slices into main memory while *time II* reveals the time spent on color matrix and pixel textures (*glDrawPixels()*). *Num slices* indicates the number of slices taken for the presented views (column Image).

Another interesting property is the frame-time relative to the size of the viewport. Once enlarging the viewport by a factor of four, the rendering time only increases by a factor between 2.6 (engine and silicium) and 3.2 (lobster). Obviously, *glReadPixels()* and *glDrawPixels()* are more efficient the larger the selected area. Furthermore, texture mapping itself is more efficient for larger screen sizes, which is due to the higher cache hit ratio during texture memory accesses (fill-rate).

Despite the different volume size of neghip64, silicium, and lobster, the frame-rates are very similar which is due to the used views consisting of 64 slices (Z direction). When changing the view for the silicium or lobster, the number of required slicing planes increases (X or Y dimension) increasing the overall render time. Nevertheless, the texture memory access seems to be very efficient since frame-rate of neghip64 and

[4] In general, pixel textures are supported for *glCopyPixels()* and *glDrawPixels()* but their current implementation as OpenGL 1.1 extension only supports the latter one.

lobster hardly differ, despite the large difference in the size of the datasets (neghip64: $64 \times 64 \times 64$, lobster: $128 \times 128 \times 64$).

It can also be observed that the texture swapping mechanism — extensively using the system bus of the Octane — is very efficient. Comparing the frame-rate of the neghip128 and the engine (in both cases 128 slices are rendered), only a small difference can be noticed, despite the fact that the engine needs to be split into eight bricks while the neghip requires only two bricks.

In general, the results show that pixel textures are very valuable for texture mapping based volume rendering. However, interactive frame-rates can only be achieved if pixel textures are either applicable during the rendering process or if they can be applied using *glCopyPixels()* instead of a subsequent combination of *glReadPixels()* and *glDrawPixels()*. Since a large number of applications using pixel textures have been presented, chances are high that it will soon be supported by other graphics hardware vendors at a better performance.

5.4.4 Analysis

Despite its good visual results, the presented method to combine semi-transparent and diffuse shaded rendering of volumetric data has one drawback. Generally, the images look darker than images rendered in software using the same operations. There are two reasons for the difference:

1. Limited precision of the graphics hardware
2. Incorrect result of the scalar product

While the impact of the limited precision (8 bit) will diminish within the next generations of graphics subsystems[5], the computation of the scalar product is based on interpolated gradients. The basic assumption is that one can compute the scalar product using the interpolated gradient and a matrix multiplication. While the matrix multiplication is generally correct and performed at highest precision (floating point), the interpolation of the gradients frequently results in not normalized gradients. Therefore, the scalar product does not represent the correct angle between light vector and sample gradient. To keep the error in an acceptable range, the gradients need to be normalized before storing them as 3D texture. Nevertheless, the remaining error can still be high. Figure 5.7 illustrates a dataset as well as the resulting normalized central difference gradients. Even though Figure 5.7 uses a binary dataset for demonstration purposes, this would also apply for non binary datasets, i.e., assuming that red voxels represent ‘1‘ instead of ‘255‘ (for eight bit volume data). The same gradient field would be obtained since gradients need to be normalized before storing them as texture map. Note that the central difference gradient needs to be normalized not only because of the reason mentioned above but also to avoid gradients which are larger than one[6].

The resulting scalar product of the matrix multiplication is of range $[-1,1]$ but is clamped to the interval $[0,1]$ before the per fragment operation stages are reached. The clamping of negative values to zero corresponds naturally to no diffuse light being present on the backside of an object.

Figure 5.8 illustrates the resulting scalar product using trilinear interpolation across the red-plane depicted in Figure 5.7 (a). The correct result is shown in Figure 5.8 (a)

[5] SGI’s latest Octane systems (VPro graphics) operates on 12 bit data per channel.

[6] The central difference gradient operator is anisotropic and thus, the length of the resulting gradients are element of $\left[0,\sqrt{3}\right]$

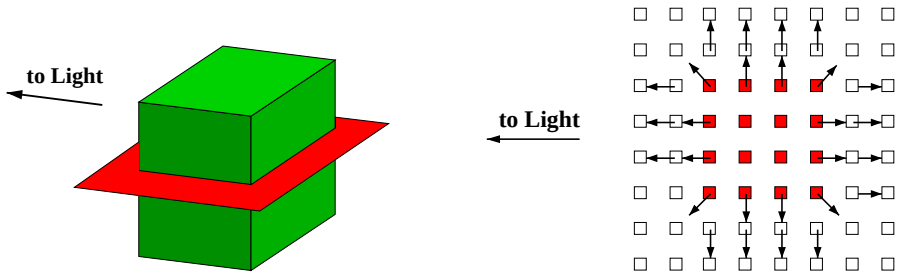


Figure 5.7: Setup for measuring the shading quality: (a) Dataset consisting of a cube (green). (b) Normalized central difference gradients for red plane in (a); empty voxels are displayed white and occupied voxels are displayed red.

and obtained by normalizing the interpolated gradient prior to computing the scalar product. Figure 5.8 (b), shows the result obtained without normalizing the interpolated gradient. Despite the fact that the two functions are somewhat similar, their absolute

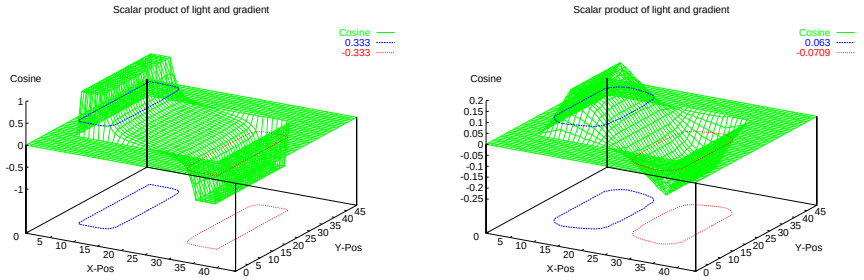
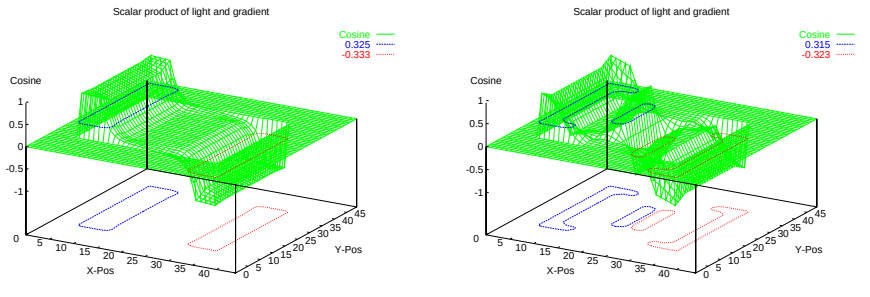


Figure 5.8: Trilinear interpolation of gradients which are (a) normalized before computing the scalar product (b) not normalized.

values differ significantly. The reasons for this are the high frequency change in gradients as well as the bilinear interpolation used. The actual error made is given by the difference of the two graphs, as shown in Figure 5.9 (a). While this case only covers two-dimensional gradients (Figure 5.7 (b)) being interpolated bilinearly, the case of true trilinear interpolation is given when moving the red plane in Figure 5.7 (a)) right between the topmost volume slice containing the cube and the one above. The resulting error graph is depicted in Figure 5.9 (b). Generally, the graphs in Figure 5.9 show that the error can be very high, in the worst case close to 100%. These cases occur only in areas where the gradient changes rapidly (high frequencies) but even low frequency datasets can result in high frequency gradients due to the required normalization of the gradients before storing them as a 3D texture map. Thus, the impact of computing the scalar product based on not normalized interpolated gradients depends on the dataset and the resulting normalized gradient field. For high frequency gradient fields, the error will be high which favors central difference gradients over intermediate difference



(a) (b)

Figure 5.9: Error made by not normalizing the gradients before computing the scalar product (cosine): (a) Across red plane as shown in Figure 5.7. (b) Assuming the red plane to be located between the topmost volume slice containing the cube and the slice above such such that real 3D gradients are interpolated.

gradients because it is the better low-pass filter.

Even though the potential error is quite high, this does not allow drawing conclusions on the final image quality since these cases might be rare in real world datasets. Thus, a software implementation with and without vector normalization prior to diffuse shading is used for analysis and allows to separate the shading error from the overall darkening, which happens additionally due to the frequent discretization in the graphics hardware. Figure 5.10 illustrates the resulting images for different datasets. The difference images are scaled for printing reasons but the individual pixel difference is quite high: the maximum pixel difference for the final pixel value is 82 for the neghip, 74 for the skull, and 128 for the arteries. Large differences can be observed in areas where neighboring grid gradients vary a lot and where small changes have a strong impact which depends on the light source direction. The neghip consists of an almost continuous 3D function with few high frequency areas and only in these areas a noticeable error is made. In contrast, the skull dataset consists of reconstructed bone and teeth surrounded by low frequency noise, which results in a strong gradient change around the bone and teeth surface. Hence, the largest error is noticeable along the edges of the bone and teeth. Finally, the artery dataset consists of high frequencies which results in the highest error of all three datasets.

Despite of the actual error, the rendered images look fairly good which is due to the mostly equal error distribution (skull and arteries). Thus, the goal of shading — which is an enhancement of the three dimensional character of the object — is still clearly accomplished and the images provide good quality. Therefore, shading using the scalar product of not normalized sample gradients is acceptable, as long as the three dimensional gradient field does not contain too many high frequencies. However, one should note that the images of Figure 5.10 are computed in software at floating point precision illustrating the impact of the wrong scalar product only. Images rendered with real hardware still look darker due to the frequent discretization of the values within the pipeline (see right column of Figure 5.6).

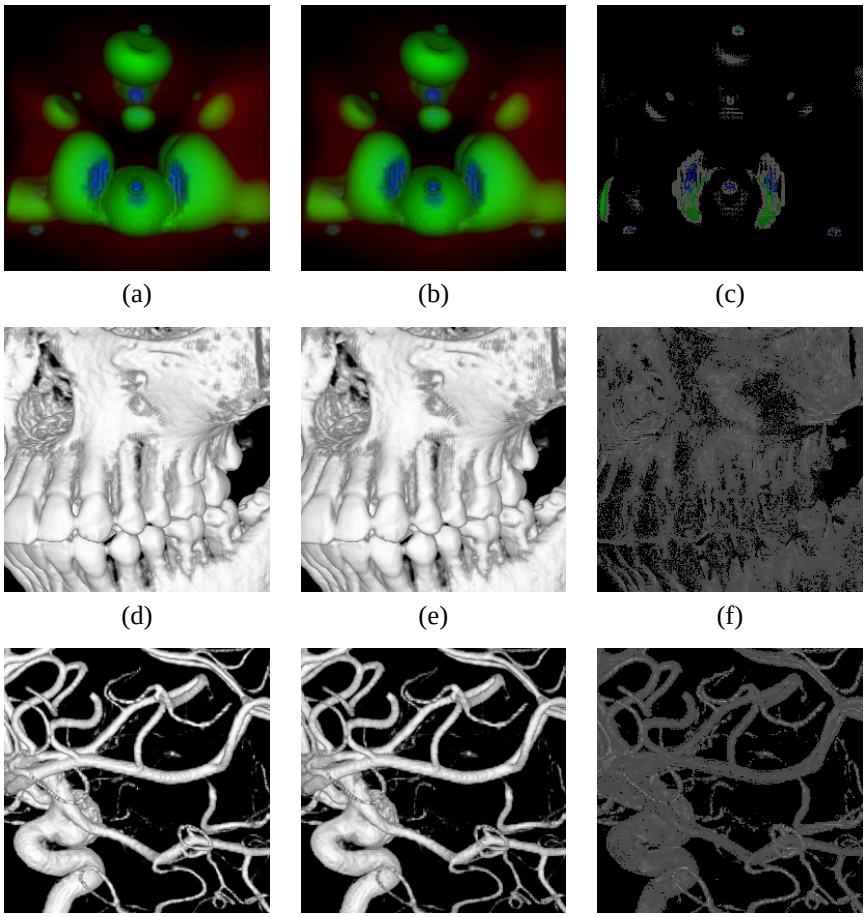


Figure 5.10: Error in the final images: Correct images (left column), images obtained without normalizing the interpolated gradients (middle column), and scaled difference images (right column). (a-c) Neghip. (d-f) Skull. (g-i) Arteries.

5.5 Summary

Polygon rendering and volume rendering have many things in common. Unfortunately, the datapath of polygon rendering hardware is different than what would be needed to efficiently use the same hardware for volume rendering. By exploiting OpenGL and different extensions, it is feasible to accomplish volume rendering with on the fly computation of shading effects but one sacrifices speed. While shaded and colored iso-surfaces can still be accomplished at interactive frame-rates, shaded semi-transparent rendering including classification reduces the frame-rate to a few frames per second due to the frequent copying of slices. The analysis of rendering times showed that this should improve when *glCopyPixels()* is supported for pixel textures but some limitations remain, e.g. one directional light source only.

Ideally, one would like to store a volume as a three dimensional texture of scalar

values and gradient components. Instead of using a matrix multiplication which introduces inaccuracies and is limited to one light source, it would be preferable to directly use the interpolated gradient to address a cube map (ARB_texture_cube_map). By initializing such an environment map containing the accumulated intensity of all light sources, accurate shading could be accomplished. At the same time, the density value would need to be used to address a one dimensional color table similar to GL_TEXTURE_COLOR_TABLE_SGI containing RGBα values. By multiplying the shading intensity with the classified RGB values, one could realize high quality volume rendering. A schematic of the needed dataflow is illustrated in Figure 5.11.

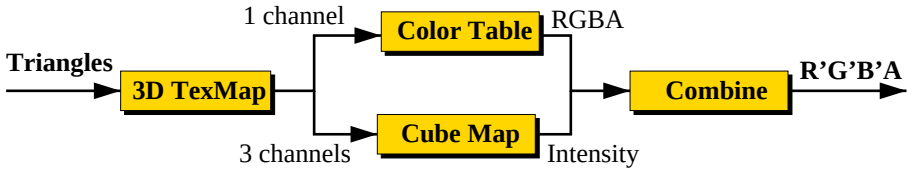


Figure 5.11: High quality volume rendering including shading and classification could be enabled in one pass if the datapath would allow to use different parts of the fragment to address different textures.

Looking at what is already available on different platforms, it appears to be only a matter of time. The cube map is already available on nVidia's GeForce chip but the texture coordinates s,t,u are used as input and not the content of the fragment channels. This is simply a matter of multiplexing the right data lines using the texturing output of one texturing unit as input for the second one (subsequent texturing as extension to multi-texturing). The texture color table is currently supported on SGI machines and HP's fx10 but each channel is treated individually. An extension that allows a *RGB*α lookup table addressed by one channel would solve this. Finally, multiplying the resulting shading intensity with the color and adding an ambient term could be done similar to the texture combiners available on nVidia's GeForce chip.

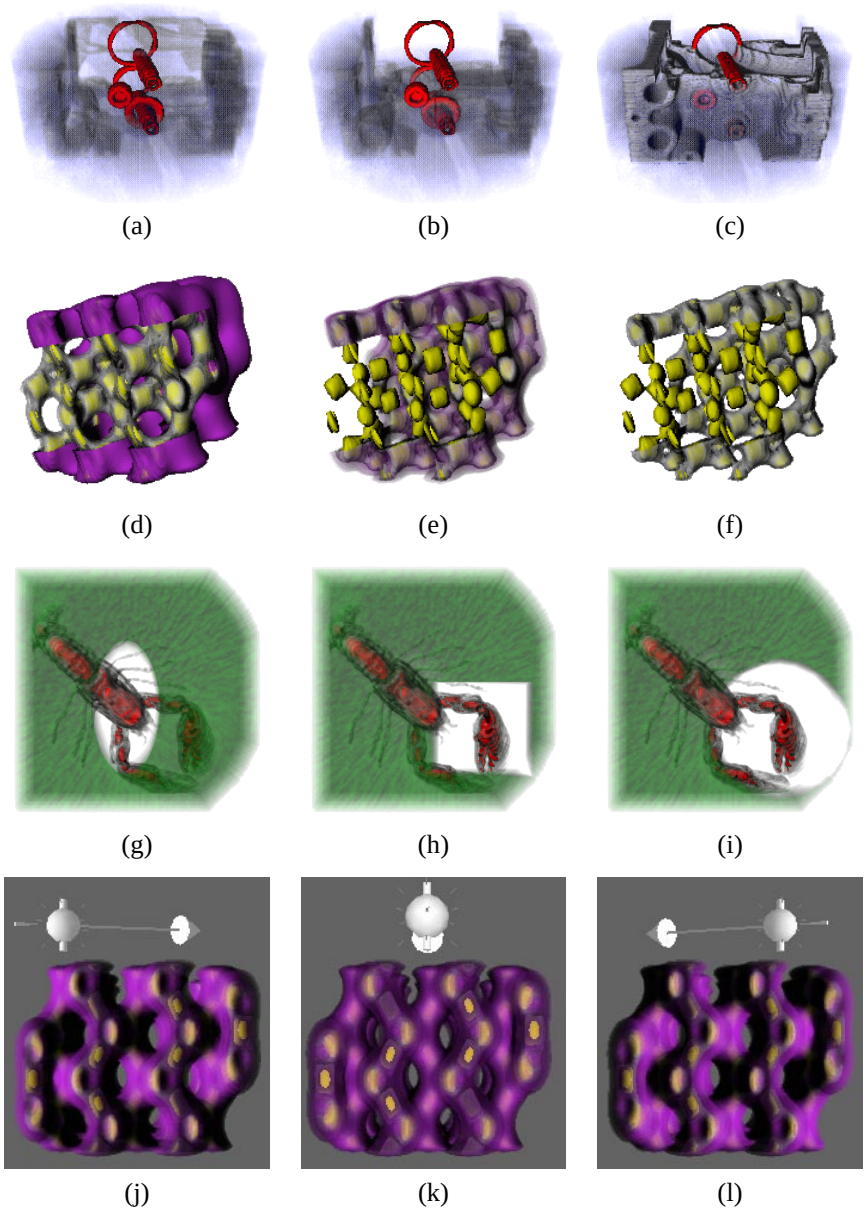


Figure 5.12: Color plate: (a–c) Engine block (grey), metal (red), and CT noise (blue). Cubic lens is used to reveal inside information, different classifications have been applied for the two classification spaces. (d–f) Silicon block; Center (yellow), hull (grey), and outer hull (purple). Cubic lens is used to enhance the relationship of the different areas. (g–i) Lobster consisting of meat (red) and shell (white), surrounded by resin (green). Lens reveals local information removing the resin by applying different classification. (j–l) Silicon block; Different light directions are selected to show the shading effect.

Chapter 6

A SIMD Approach for Volume Rendering

Software volume rendering systems usually lack the computational power to accomplish real-time or even interactive frame-rates for the general case. However, exploiting optimizations such as space leaping, early ray termination, smart memory organization, and others, noticeable speed-ups are feasible for certain datasets. One example is the shear-warp algorithm presented earlier in Chapter 4.

Parallel volume rendering has drawn a lot of attention and there are several different approaches. Distributed memory systems usually require replication of volume data or data partitioning schemes. This is not necessary for shared memory systems but shared memory systems usually do not scale well once a certain number of nodes is exceeded. A different avenue are SIMD machines which are usually distributed memory architectures but all processing elements perform the same task in every cycle, each having some local memory as well as a large shared memory. Generally, most of these systems are quite costly due to their high production costs.

Pixelfusion's FUZION^{TM}150 is a single-chip SIMD architecture with a total of 1536 processing elements, implemented on an AGP card. Each processing element runs at 200 MHz and has 2 KBytes of local memory and the entire architecture provides a high bandwidth memory bus which is connected to Rambus^{TM} memory modules. In the following, an optimized implementation of a parallel ray caster performing trilinear interpolation of data and gradients, Phong shading, and compositing is presented. The implemented parallel volume rendering algorithm on this machine provides flexibility with respect to the performed computations and is quite competitive to earlier presented large scale solutions.

6.1 Introduction

Due to the high complexity of volume rendering, parallel processing is very promising to accomplish significant speed-ups. Depending on the available system architecture, different approaches have to be taken. Generally, one can classify systems by their memory architecture resulting in shared memory, distributed memory, and distributed shared memory. When implementing a volume renderer, the following observations can be made: For distributed memory systems, object space partitioning is frequently used. Data replication and communication is prevented by assigning each node a certain part of the volume [Hsu93, Neu93, LWMT97b, LWMT97a]. On the other hand, image space partitioning schemes are more often used on shared memory architectures since the entire data resides within shared memory space [NL92, MPHK93, Lac95]. There have also been hybrid approaches were data is distributed across nodes using an image space partitioning scheme [CM93, AGS95, Lef93]. These approaches have different ways of keeping volume data communication low. Finally, Palmer et al. presented an object and image space partitioning scheme on a distributed shared memory architecture [PTT97].

SIMD systems (single instruction multiple data) usually belong to the category of distributed memory architectures and are well suited to implement volume rendering algorithms. Vézina et al. presented an iso-surface rendering approach running on a SIMD MasPar MP-1 [VFR92]. Using 16K processors — each equipped with 16 KBytes of memory —, their implementation achieved three frames per second for a 128^3 volume and a viewport of 128×128 pixels. Hsu et al. implemented a segmented ray casting approach on a DECmpp 12000/Sx [Hsu93]. On a 4K processor system with 4-bit ALUs, an image of 128×128 of a 128^3 dataset could be rendered in 0.82 seconds using using trilinear sample interpolation but no shading. Wittenbrink et al. presented a permutation warping providing almost constant run-time for all viewing angles [WS93]. On a SIMD MasPar MP-1 system of 16K processors, 2 frames per second were reported using trilinear interpolation. A refined implementation achieves 14 frames per second on a 128^3 dataset [Wit98]. Finally, Kreeger et al. proposed a SIMD programmable architecture for volume processing estimated to provide real-time frame-rates but the system has not been built and would be quite costly[KK98].

6.2 The FUZION^{TM}150 chip

The FUZION^{TM} architecture[1] is implemented on a single-chip — the FUZION^{TM}150 — which connects to several external interfaces, as depicted in Figure 6.1. The core of the FUZION^{TM}150 chip is a large SIMD array containing a total of 1536 processing elements (PEs). The PEs are split into six equally sized blocks containing 256 PEs each, so-called FUZION^{TM} blocks. Generally, instructions and data reside in the local off-chip memory which is realized as Rambus^{TM} memory modules. Instructions are fetched from the memory and then decoded (sequenced) into microcode which can be executed on the PEs. The FUZION^{TM} bus connects the SIMD array, the instruction fetch and decode, an embedded CPU, video input, and video output. The local Rambus^{TM} memory and the FUZION^{TM}150 chip are located on an AGP card which can be plugged into any PC. The FUZION^{TM}150 chip itself is controlled by a thread manager (see Figure 6.2) itself controlled by the embedded CPU (ARC). The thread

[1] Due to a non disclosure agreement with Pixelfusion, the description of the FUZION^{TM} architecture can only be of limited detail.

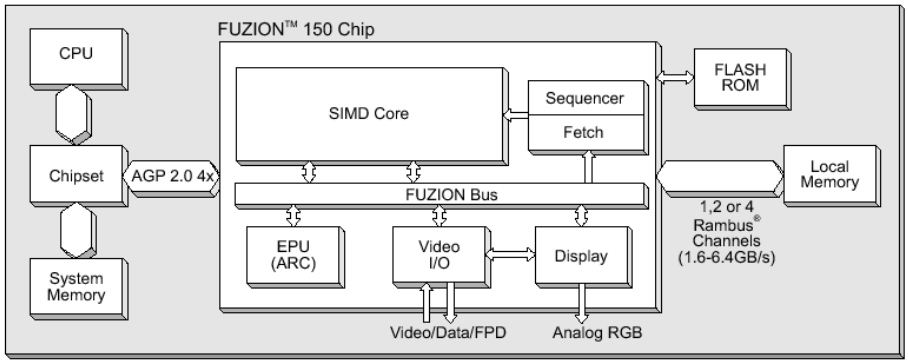


Figure 6.1: The FUZION^{TM}architecture.

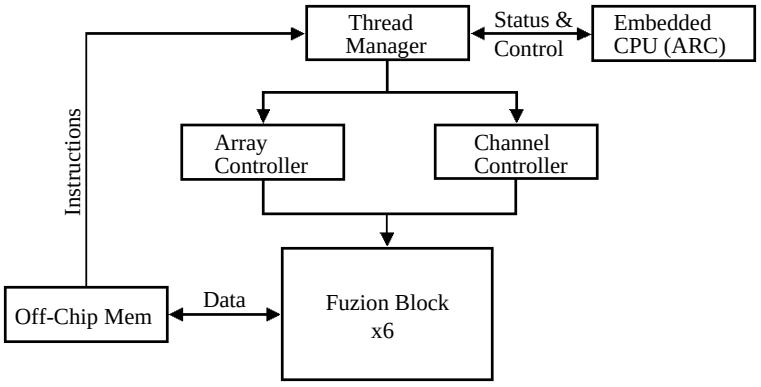


Figure 6.2: The thread manager.

manager can either access the local memory via the channel controller or perform operations on the SIMD array which is handled by the array controller. The array controller (see Figure 6.3) interprets incoming instructions using an instruction table. If the instruction is to be performed on the PEs, the instruction sequencer translates the instruction into a sequence of micro-instructions which are then executed by the PEs. Any other instruction that does involve data I/O is handled by the load/store controller issuing the necessary signals to access the main memory (Rambus^{TM}) and synchronizing the read/write operations of the PEs.

Consolidation of reads prevents redundant memory accesses when reading the same memory region from different PEs. This also implies that applications need to be written such that consolidation can be exploited saving bandwidth and yielding higher performance.

6.2.1 Processing Elements (PEs)

The core of the FUZION^{TM}150 chip is the SIMD array with its 1536 PEs. Each PE runs at 200 MHz internal clock frequency. The ALU is 8 bit wide and has a set of internal and external registers and in addition to fixed point operations on 16 bit and 32

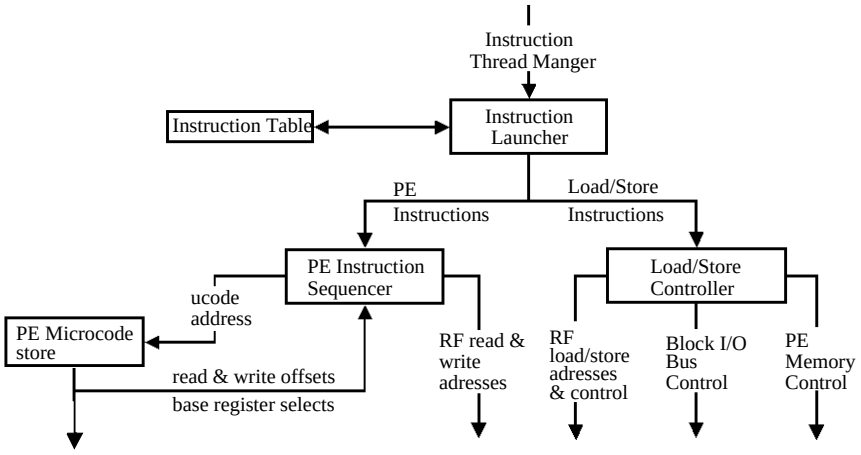


Figure 6.3: The Array controller (RF denotes register file).

bit, floating point are also supported. However, these operations are emulated by the 8 bit ALU which can require many cycles for completion. The number of cycles per instruction strongly depends on the instruction itself. Therefore, one needs to carefully determine the required data precision as well as the appropriate set of instructions.

The PEs are organized in a one dimensional array which is split into six blocks (FUZION^{TM} block) providing buffers between those blocks. The interconnect of the PEs is named swazzle path and enables direct communication and data exchange of neighboring PEs operating at full PE speed (200 MHz). Furthermore, each PE is connected to a data bus such that data can be transfered to and from other devices over the FUZION^{TM} bus. This is handled by the earlier mentioned array controller. The local 2 KByte of embedded DRAM memory of each PE is running at 50 MHz which is four times slower than the ALU. However, register to memory transfers can be setup and while the data is transfered, other operations can be performed.

6.2.2 Conditional branches

Due to the nature of SIMD, all PEs perform the identical operation in each cycle. However, for most applications conditional branches (`if ... else ...`) are unavoidable requiring different operations on different PEs. Since this is not feasible in a SIMD architecture, conditional branches are realized such that all PEs that do not fulfill the condition are deactivated. After processing the instruction sequence of the `if` branch, active PEs are deactivated and inactive PEs are re-activated before executing the `else` branch. Hence, all instructions in both branches are executed and performance is potentially wasted. Therefore, an algorithm should avoid conditional executions whenever it is possible. Generally, up to five nested `if`-conditions are supported on the FUZION^{TM}150 but further ones can easily be emulated.

6.2.3 Performance

The FUZION^{TM}150 chip provides a tremendous computational power. All 1536 PEs together are capable of performing 1.5 Teraops[2] or over 3 Gigaflops[3] per second. The on-chip system bus can deliver a bandwidth of up to 600 GBytes per second while the external memory bus to and from the Rambus^{TM} memory has a peak bandwidth of 6.4 GBytes per second. Despite this theoretical peak performance, it is not guaranteed that this performance can be reached. Algorithms need to be designed for this highly parallel architecture and reaching the peak performance can be hard to accomplish or might not be feasible at all, depending on the application and its implementation.

6.2.4 Development Environment

Pixelfusion provided cooperation partners a programming environment which is very similar to Microsoft's Visual Studio (C++). The debugger allows tracking the state of each PE as well as the content of its memory and the FUZION^{TM} compiler generates an instruction stream (C++ ostream) which is executed by the cycle accurate simulator translating it into microinstructions etc. For reasonably interactive simulation times, the simulation can be spawned on up to seven PCs. One PC simulates the instruction decoding and each FUZION^{TM} block (256 PEs) can be simulated on a different machine. For the presented implementation, two PCs were used — a Pentium III running at 600 MHz and one running at 700 MHz — simulating roughly 1400 cycles (7 μseconds) in one second. Thus, the simulation of one second of the FUZION^{TM}150 chip required approximately 40 hours real time.

Programming the SIMD array to perform the parallel ray casting algorithm was done strictly in assembler using the currently available set of instructions. Unfortunately, not all of the instructions were fully documented which made code development a tedious process but this project was Pixelfusion's first cooperation with an external partner. The tools used were a prototype development environment and Pixelfusion is currently developing a C compiler (based on ANSI C with extensions for SIMD data types) and an integrated development environment including a suite of tools such as assembler, linker, debugger, profiler, simulators and so on. This Software Development Kit (SDK) will be available with the next generation of IP products.

6.2.5 A Historical Note

Almost two decades ago, the well-known Pixel-Planes project was started by Fuchs et al. [FP81]. The first architecture of Pixel-Planes utilized fully customized VLSI-chips containing a SIMD array of compact pixel processors operating in parallel and allowing programmable shading. For better load balancing, more frame-buffer memory per pixel, higher quality, etc. successor architectures were built yielding to PixelFlow [MEP92]. PixelFlow consists of multiple boards, each containing 64 EMCs (logic-enhanced memory chips) as well as control and communication circuits. Each SIMD chip contains an array of 256 PEs and memory [MEP92]. Depending on the rendering workload, each board of the PixelFlow architecture does perform compositing

[2] Additional to the PE operations, there is a Linear Expression Evaluator (Ax+By+C) which performs another four operations per PE per cycle. Thus a total of 5 * 200MHz * 1.5K integer operations can be performed.

[3] This assumes 100 cycles for a floating point operation. Thus a total of $0.01 * 200$MHz * 1.5K floating point operations can be performed.

and either shading or rasterization. PixelFlow has been a joint project of Division Ltd and the university of North Carolina Chapel Hill. Later on, Hewlett-Packard joined the project and the first prototype was built [EMPG97][4].

PixelFusion Ltd. was set up in 1997 to take the Pixel-Planes architecture to the mass-market. The company has made many innovative improvements to the basic architecture in order to develop its single-chip SIMD solution, the FUZION^{TM}150 chip. The chip was targeted at the graphics market but, due to rapid changes in this market, Pixelfusion has decided not to take this part to production. However, the next generation architecture will include further enhancements, such as more flexible PE memory addressing and an improved ALU, which will address some of the shortcomings noted in this paper.

6.3 Parallel Ray Casting

The main design issues for the parallel implementation of a volume rendering algorithm is the equal load balance for all 1536 processing elements. Therefore, the algorithm of choice is a permutation of the shear-warp algorithm [LL94][5]. Starting from an intermediate baseplane — the face of the volume which is most perpendicular to the viewing direction — parallel viewing rays are traced through the volume. To improve the image quality compared to the common shear-warp implementation, trilinear interpolation is applied instead of bilinear interpolation. Figure 6.4 illustrates the relation of rays, volume, baseplane, and viewplane for two different viewing angles. The size of

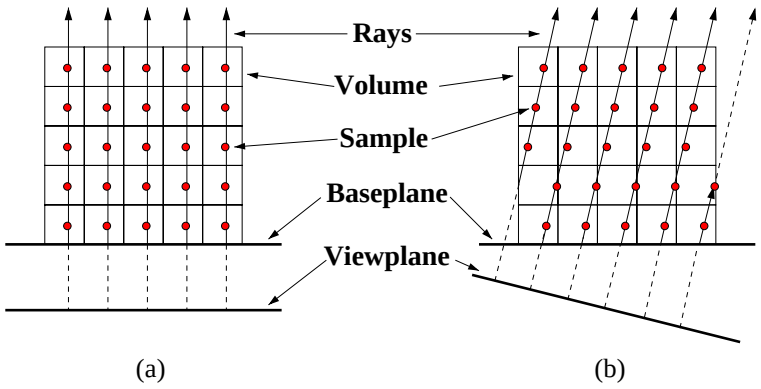


Figure 6.4: Baseplane oriented ray casting: (a) Orthogonal view. (b) Slanted view. Rays which leave the volume are automatically re-entering the volume in wrap around fashion.

the intermediate image on the baseplane depends on the viewing angle. As illustrated in Figure 6.4(b), for each ray that leaves the volume on one side, a new ray is cast into the volume on the other side. The one to one mapping of samples to voxels ensures that for each scanline, a total of n rays is active at any time, where n denotes the size of the volume (n^3 voxels). For a volume of 256^3 voxels, six scanlines are computed in parallel, one per FUZION block. However, the implementation is not limited to volume sizes which are a power of two.

[4] In this version, each board was equipped with 32 EMCs.

[5] A very similar permutation is used in the CUBE-4 architecture [PK96].

The ray casting process itself includes trilinear sample interpolation, gradient estimation, Phong shading using one light source, and blending. After processing one sample and computing the new sample location, one `if`-branch needs to be performed to check whether any PE traces a ray that left the volume. In this case, the pixel is completed and the PE will be initialized for the ray which enters the volume on the other side. Mapping the image from the baseplane onto the viewplane is done using texture mapping. A full OpenGL 1.1 implementation for the FUZION^{TM}150 has been developed by PixelFusion. Computer graphics was originally the primary target market but this has shifted to specialized co-processor markets providing high-performance data processing silicon and IP [6].

6.3.1 Analysis

So far, no AGP card with the FUZION^{TM}150 chip is available and first samples are expected by fourth quarter 2000. Nevertheless, an estimate of the overall performance can be made using the cycle accurate simulator of the FUZION^{TM}150 chip. In the following, the rendering stages are split and analyzed.

The processing of a sample includes trilinear interpolation of the sample value. Using fixed-point arithmetic, this requires 116 cycles for a full trilinear interpolation while it takes 2905 cycles using floating-point. The reason for this significant difference is the 8 bit ALU. Floating point operations as well as operations on 16 or 24 bit fixed-point operations need to be emulated by sequential microcode which can be quite expensive. Therefore, any code needs to be tuned avoiding floating point operations wherever possible. The developed implementation of the ray caster uses fixed point arithmetic only with a neglectable degradation of the image quality.

The gradient is computed using a modified version of the intermediate difference gradient estimation scheme [Kni94]. For a given sample position, the closest voxel on each face of the subcube is computed performing six bilinear interpolations. The local difference is taken from these six values. Even though this is not the ideal intermediate difference gradient, it still achieves good image quality at reasonable cost. Fetching the eight surrounding voxels takes 1840 cycles which is 40% of the overall time needed per sample. Other gradient estimation schemes could provide higher shading quality but would require a larger voxel neighborhood significantly increasing the overall rendering time.

Classification is performed on a per sample base and includes the assignment of an RGBα-tuple per 8 bit density sample. This requires to store a table of 1 KByte in the local memory of each processing element. Due to the addressing scheme of the local memory, the table needs to be arranged in eight tables, each containing 128 values. This is due to the limited addressing arithmetic available. On the FUZION^{TM}150 chip, one can only use a fixed base address with a relative offset which is 5 bit wide.

One of the most expensive computational operations is the evaluation of the Phong shading model due to the required normalization of the gradient vector. Additionally, two scalar products are required, one for the diffuse and one for the specular component. Even though this accounts for one directional light source only, it needs approximately 1500 cycles or 33% of the time spent per sample. This could be improved by implementing a faster square root computation because the existing one takes 862 cycles which is quite slow. While own instructions consisting of a set of micro instructions can potentially be developed, the required tools are not (yet) publicly accessible.

[6] http://www.pixelfusion.com

Finally, the blending of the color value with the already accumulated pixel color is performed using a special fixed-point arithmetic. Color and opacity values are 8 bit $[0...255]$ representing the interval $[0,1.0]$. Therefore, one needs to compute $(a*b)/255$ instead of $(a*b)/256$. While the latter can be simplified to $(a*b) >> 8$ and used to trilinearly interpolate sample values because the 8 bit weights represent the interval $[0,1.0[$, the former is more expensive[7].

Overall, the cycle count for each computation stage is summarized in Table 6.1. Fetching data from memory as well as the gradient normalization consume roughly

Task	Time [cycles]	[%]
Fetching data from external memory		**39.3**
All eight voxels	1840	
Trilinear sample interpolation		**2.5**
Seven linear interpolations	116	
Shading		**32.4**
Gradient computation	144	
Gradient normalization	862	
Diffuse scalar product	231	
Specular scalar product	231	
Phong evaluation	46	
Classification		**3.4**
Accessing classification data	159	
Compositing		**2.4**
Blending of color and α	110	
Test for exiting the volume:		**19.3**
Top	135	
Bottom	135	
Right	135	
Left	135	
Other tests	approx. 400	
Overall number of cycles per voxel:	4679	100

Table 6.1: Cycle per computation. The rightmost column denotes the percentage of the time needed per sample.

58% of the overall time. Shading could be accelerated using an environment map [vSSB95] but unfortunately, such an environment map would require 1.5 KByte. This is not feasible on the FUZION^{TM}150 chip since only 2 KBytes of memory are available and the look-up tables for the classification already consume 1 KByte. However, on a successor system with 3 or 4 KBytes of memory, this could reduce the overall time spend per sample by approximately 25%. The third most expensive part is the test whether a ray leaves the volume on one of the four side faces (each side 135 cycles) or on the back which includes some other tests (400 cycles). All these tests are of type `if...else...` which is expensive on a SIMD machine.

As mentioned in Section 6.3, the implemented permutation of the shear-warp al-

[7] The simplification of $(a*b)/255$ avoiding the division is presented in more detail in Section 7.2.3.

gorithm uses trilinear interpolation which is more costly than the original shear-warp. However, the costs are almost neglectable when looking at what can be saved when using bilinear interpolation only. Instead of fetching eight values, only four values would need to be fetched which saves 920 cycles. The trilinear interpolation could be reduced to a bilinear interpolation but would hardly be noticeable since it consumes only 2.5% of the overall render time. All other operations including the shading, classification, and compositing would not change. As a result, only the reduced data transfer saves cycles, translating into a performance gain of approximately 20% when using bilinear instead of trilinear interpolation.

6.4 Results

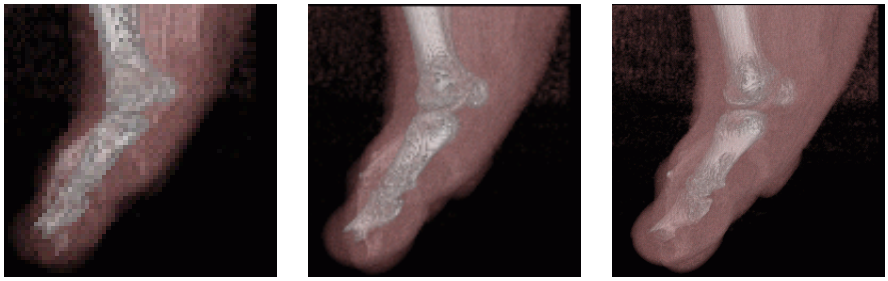
While software based shear-warp implementations use run-length encoding and early ray termination to speed-up the rendering, the presented parallel implementation does not exploit these optimizations. One reason is the difficulty to integrate such techniques into the SIMD concept. E.g. to keep the memory accesses regular, early ray termination could only be exploited if all PEs detected early ray termination, This could possibly be accomplished changing scanline based rendering to tile based rendering (16×16 rays). However, checking all PEs for their status is an expensive operation and due to the granularity of 256 rays, early ray termination is likely to be less efficient. Furthermore, performance gains due to early ray termination are only expectable for iso-surface rendering but not for semi-transparent classification.

Without exploiting any algorithmic optimizations, the presented implementation accomplishes a constant frame-rate which depends only on the size of the dataset. A total of 4679 cycles per sample per PE results in 1500 samples which are computed in parallel. Using the processing clock of 200 MHz, this translates into 26 frames per second for a volume data of 128^3 voxels which is 85% more than reported by [Wit98] using 16K processors. Table 6.2 summarizes the frame-rate for different dataset sizes.

Dataset size [voxel]	Memory [MByte]	Processing cycles [in thousand]	Frames [per second]
256^3	48	50.000	4.0
128^3	6	7.500	26.7
64^3	<1	900	222.2

Table 6.2: Frame-rate for different dataset sizes.

In order to demonstrate the visual quality of the trilinear permutation of the shear-warp algorithm, a dataset of a human foot has been rendered using different volume resolutions. The resulting baseplane images are presented in Figure 6.5. The baseplane image is the one that is finally be mapped onto the viewplane. Thus, the final image quality depends mainly on the quality of the baseplane image and the higher the dataset resolution, the higher the final image quality. However, there is a trade-off between quality and render time.



(a) (b) (c)

Figure 6.5: Baseplane image of differently sized datasets: (a) 64^2 (b) 128^2 (c) 256^2.

6.5 Summary

A trilinear permutation of the well-know shear-warp algorithm running on a highly parallel SIMD architecture containing 1536 processing elements has been presented. The implementation achieves four frames per second for medical datasets of 256^3 voxels. This frame-rate is constant because it does not depend on the dataset nor on the classification; neither run-length encoding nor early ray termination is used.

A careful analysis of the instruction set showed that floating point operations need to be avoided to accomplish a good performance. The analysis of cycles per volume rendering operation identified three main bottle-necks. First, the memory access to fetch eight voxels for each processing element consumes 39% of the time and can hardly be optimized any further. Second, the shading computation consumes 32% of the time due to the normalization of the gradient and the two scalar products. This could be reduced if more local memory would be available to store an environment map for shading but would require 3 or 4 KBytes of local memory instead of the available 2 KBytes. Finally, the evaluation whether a ray terminates and needs to re-enter the volume takes 19% of the time. This is also hard to optimize since these tests slow down the SIMD architecture.

Using bilinear interpolation instead of trilinear interpolation as well as assuming that more memory would be available to store an environment map, an upper limit of eight frames per second is to be expected with the current implementation. Compared to a software implementation as well as other earlier reported parallel implementations, this is quite competitive, especially since no algorithmic optimizations are exploited.

FUZION^{TM}150 is the first commercially available SIMD solution providing a tremendous number of processing elements on a single chip. Since it is a programmable system, it can be used for a large variety of other applications even though one needs to carefully map an algorithm onto such a system to accomplish a good performance. It would certainly be an enrichment if such systems would become more popular and widely available. With respect to volume rendering, it is capable to provide good image quality at a sustained frame-rate. In applications where interactive change of classification is mandatory, it is faster than an equivalent software based shear-warp implementation since the run length encoding exploiting the classification is generally a time consuming pre-processing step.

Chapter 7

VIZARD II: Special Purpose Hardware for Volume Rendering

As shown in the previous chapters, several available hardware systems can be used to accelerate volume rendering. Each of them has certain advantages such as broad availability (OpenGL graphics hardware) or massive parallelism (FUZION ^{TM} 150). However, both do not provide the full desired functionality nor are the memory interfaces ideal with respect to volume rendering. This is due to the fact that these systems have not been designed for volume rendering in the first place.

Real-time frame-rates at highest image quality and flexibility at moderate costs can only be accomplished using dedicated volume rendering hardware. *VIZARD II* is a special purpose volume rendering accelerator. Flexibility and moderate costs are guaranteed by using a DSP, reconfigurable hardware (FPGA), and SDRAM dual inline memory modules (DIMMs) which can be chosen appropriately. The flexibility — namely reconfigurable hardware — allows faster redesign cycles and lower costs than designing an ASIC. Furthermore, it uniquely enables the implementation of different algorithms, e.g. segmentation or iso-surface extraction using the same hardware platform.

In the following, previous work in the field of special purpose volume rendering hardware is reviewed. Thereafter, the VIZARD II system, its architecture, as well as the components are introduced and a analysis of the system bottle-necks is given. Finally, different enhancements are presented to accelerate the rendering using space leaping and to enable 16 bit voxel values.

7.1 Introduction

Despite its long history, volume rendering is still a challenging task due to its high bandwidth and computation requirements. Within this section, an overview on present previous work on special purpose volume rendering hardware is given.

The Cube architecture has certainly the longest history and the largest body of publications [BKX90, BKX92, KMS+96, PK96, BK97, OPL+97, PHK+99]. The architecture uses template based ray casting [YK92, SS92] and an optimized memory interface which ideally fetches each voxel exactly once per frame [CK95]. The architecture consists of a number of processing pipelines which perform buffering, sample interpolation, gradient estimation, shading, and compositing. The volumetric data is distributed over a set of memory modules. A memory interleaving scheme called skewing [CK95] is used to guarantee conflict free access to any axis-aligned beam of voxels. The skewing scheme assigns all voxels to memory module i which fullfill $x + y + z \ mod \ n = i$, where n is the number of operational pipelines of the system. While the Cube-3 architecture used a voxel bus to distribute the voxels from the voxel memories to the processing pipelines, Cube-4 removed this burden of having a global voxel bus by connecting each processing pipeline to exactly one memory module [KMS+96, PK96]. The only available implementation of the Cube family is the VolumePro system [PHK+99] which itself uses yet another memory scheme. SDRAM memory chips are used and skewing is applied to subcubes while voxels within the subcube are read in burst-mode, resorted, and assigned to the processing pipelines on-chip. All Cube architectures share the principle of using a baseplane to generate the image, similar to the shear-warp algorithm [LL94]. Using a one to one mapping of samples to voxels, each pipeline receives a single voxel in every cycle. The processing pipelines are locally connected such that the required voxel information is exchanged between neighboring pipelines which favors scalability. The final image is obtained by warping the baseplane image onto the view plane which can be done using 2D texture mapping. VolumePro is a single chip system implementing four parallel processing pipelines and achieves 30 frames per second for a dataset of 256^3 voxels[1], currently the fastest volume rendering accelerator available. Despite the excellent performance, the system has a number of disadvantages which are inherent to the architecture. Only parallel projections are possible since the local connectivity of the pipelines does not enable to access the necessary data of diverging rays. However, perspective projection is mandatory for immersive applications as frequently needed in the medical field. One could emulate perspective projection similar in the way it is achieved when using texture mapping[2] but this requires a tremendous bandwidth which would significantly reduce the frame-rate. Furthermore, oversampling in screen space (on the baseplane) causes a high penalty since multiple baseplane images with different offsets need to be rendered and then interleaved pixel by pixel before warping this higher quality baseplane image. Finally, scalability across multiple chips seems to be extremely costly due to the large amount of pins of each pipeline, which is roughly between 400 and 500.

Virtual Reality in Medicine (*VIRIM*) uses an object-order volume rendering architecture [GPR+94, dBHG+96]. The VIRIM architecture performs rendering in a two

[1] Generally, VolumePro delivers 500 million interpolated and shaded samples per second.
[2] Each polygon/slice is perspectively correct distorted.

steps. First, a geometry processor resamples the volume data and second, sample values are shaded and blended. The latter one is realized as a DSP board enabling high flexibility. Resampling the volume data is performed in top to bottom scanline order and for each scanline all samples of the corresponding plane are trilinearly interpolated. For better performance, an eight way interleaved memory scheme is used which has a maximum transfer rate of 640 MBytes per second. Each of the eight values passes a lookup to assign an opacity value before interpolation. While the sample value is trilinearly interpolated, the gradient consists only of two components which are determined using a local difference filter within the plane. This is possible since the light sources are directional and within the same plane (at 0 and 45 degrees). A maximum of 36 million trilinear interpolated samples and gradients are then passed to the DSP board which consists of several DSPs. A peak transfer rate of 240 MBytes per second can be achieved. The programmability of the DSPs allows the implementation of different algorithms such as MIP, ray casting, or ray tracing [MMSE91]. VIRIM has been the first operational interactive special purpose volume rendering accelerator. It achieves between one and four frames per second for a 256^3 volume. The disadvantages of the system are its large size and the high costs (approximately 100 K US$), Furthermore, monochrome images are generated and color is interpolated which results in blurry images for zoomed views and the shading quality is reduced due to the local difference gradient, limited to two dimensions. However, the architecture provides parallel and perspective projections and is therefore suited for immersive applications.

While the Cube architecture consists of an optimized memory interface with dedicated processing pipelines, VIRIM has been built for highest flexibility but has a less efficient memory interface than Cube. These two architectures represent the two main avenues that can be taken when designing volume rendering hardware. Many more architectures have been presented but only the most relevant of them are summarized:

Doggett et al. [Dog95] proposed the use of a warp array performing the viewing rotation and a ray array containing all rays to be processed. The idea is to implement the viewing rotation using process elements which reduce the 3D ray casting to a 2D ray casting requiring simple shift operations only. However, the required hardware is very complex and the implementation of a full ray array was expensive at that time. The volume memory is double buffered and for a HDL implementation 15 frames per second for 384^2 rays onto a 256^3 dataset were estimated.

Lichterman et al. [Lic95] proposed DIV^2A, a system of eight ray casting processors which are connected in a ring fashion. The volume data is split into subcubes which are distributed across the eight processors such that data required at the borders of subcubes is most likely available in one of the two connected processors. For 256^3 voxels, a performance of 20 frames per second has been estimated but the current status of the project is unknown.

Knittel et al. [Kni94] proposed an architecture that uses an eight way interleaved memory performing either fast (reduced intermediate difference gradient) or high quality rendering (central difference gradient). Furthermore, a custom implementation of a square root unit is used to implement high quality shading. An estimate of 20 frames per second for 256^3 voxels using eight parallel ray casting units running at 60 MHz has been estimated but would require complex custom hardware (ASIC).

Knittel et al. [Kni95] also proposed to build a PCI based volume rendering accelerator. The system uses the main memory of a PC class machine to store a pre-classified and pre-shaded volume dataset. The CPU computes the intersection of the ray and the

volume while the PCI-card performs the ray casting. To drastically reduce the DMA traffic on the PCI bus, a redundant block truncation coding (RBC) is used. This block truncation scheme uses 32 bit to represent a $3 \times 2 \times 2$ neighborhood of twelve voxels. Two representative intensity values are stored in 16 bit (8 bit each) and twelve bit are used to decode which of the two values belongs to the twelve voxel positions. The remaining four bit can be used for segmentation. Hence, for the interpolation of a sample, only one word needs to be fetched via DMA. Furthermore, due to the discretization of the voxels, the trilinear interpolation can be reduced to lookup and a single multiplication and subtraction. The lookup requires a table of 2^{20} entries for a 4 bit interpolation weight (two times eight bit voxel plus four bit). However, this simplified interpolation and reduced memory bandwidth has its price: Limited image quality and monochrome images. Since DMA introduces latency, an additional on-board cache has been added. The cache uses the Manhattan distance to address the cache line and the address in x,y, and z of the voxel position as cache tag to check for valid cache entries. The cache performs well for zoomed views exploiting ray to ray coherence. Furthermore, a space leaping mechanism based on distance coding using octants selects the appropriate multiple of the ray increment stored in a SRAM. To circumvent delays due to the latency introduced by space leaping, two visualization units operate in parallel, each handling one ray. Using a single visualization unit, the system was estimated to achieve 2.5 to 5 frames per second for a 256^3 dataset and 256^2 rays. Pre-processing time was in the order of 15 minutes and moderate image quality was achieved. The first implementation of this system [KS97] — named *VIZARD* — achieved for a similar view three frames per second using two acceleration boards, each equipped with two visualization engines. The bottle-necks of the system were identified as FPGA-technology (too small and slow), PCI-bus, and CPU (calculating the ray entry points).

All of the so far described architectures make use of SDRAM to store the volume data while only one approach using RDRAM (Rambus^{TM} memory modules) has been presented [dBGHM96]. Despite the fact that RDRAM could be used for volume rendering, it is too expensive and difficult to implement. Furthermore, modern SDRAM technology using double data rate or higher clocked memory modules performs equal or better than RDRAM because non regular access patterns can be handled more efficiently.

Many other architectures have been proposed but only VIRIM, VIZARD, and VolumePro were built. The discrepancy of proposals and real implementations seems to be based on two reasons: First, volume rendering has high bandwidth and computational requirements and a useful solution must provide features such as oversampling in each dimension, interactive classification, high image quality, cut planes, segmentation, etc. To incorporate all or most of these features into a hardware design is a challenging task and has yet not been solved to an satisfactory degree. Second, designing hardware has been a cost and labor intense process which has been difficult for universities. However, this has changed due to several reasons. First, the broad availability of high level hardware description language based design flows on the PC platform allow complex designs. Second, the capacity and speed of Field Programmable Gate Arrays (FPGA) has increased drastically, thus enabling the rapid prototyping of complex systems. Finally, ready to use PCI cores have relieve hardware designers from dealing with the time critical aspects of the PCI protocol, such that one can fully concentrate on the system functionality.

Our goal has been set to develop a PCI based volume rendering accelerator which

is based on reconfigurable hardware, thus enabling implementations of different algorithms (flexibility), not necessarily restricted to volume rendering applications.

7.2 System Overview

The VIZARD II is a special purpose PCI card consisting of several components. The main design goal has been the implementation of a ray casting algorithm as well as high flexibility to enable future changes and other implementations. In the following, the underlying algorithm, the architecture, and the implemented units are described.

7.2.1 Volume Rendering Algorithm

The first algorithm implemented on the VIZARD II system mainly follows the work presented in [Lev89], implementing a full ray casting pipeline. Rays are cast through a volumetric, possibly non-uniform, regular dataset. To ensure high image quality, sampling needs to be freely selectable in each dimension. Segmented datasets can be processed or arbitrary classification planes applied and high precision blending is used to guarantee highest image quality for semi-transparent rendering. Furthermore, different rendering modes such as MIP, unshaded, shaded, etc. are supported.

Starting with the viewing parameters such as eye position, view direction, view up vector, etc., the position of the view plane is calculated. For each point $P_{i,j}$ of the view plane, a ray is cast into the volume and tested if it hits the volume data. There are either no, one, or two intersection points depending on the position of $P_{i,j}$ and the increment vector. Further intersection points can exist, one for each classification plane. Classification planes allow the user to apply different classification functions on each half space introduced by a classification plane. Each classification plane doubles the amount of classification spaces, each possibly requiring its own classification table. Alternatively to classification planes, segmentation can be used to label voxels. The voxel label is used to determine the right classification table during rendering (see Section 1.2.4).

A sample is generated by trilinearly interpolating the eight neighboring voxels on the grid. In a similar fashion, the gradient at sample location is computed. Instead of computing gradients at voxel location on the fly — which would results in a 32 gradient neighborhood per sample for a central difference gradient operator — gradients are considered to be a voxel property. This is similar to surface rendering where a normal is a vertex property and not computed from neighboring triangles on the fly. Another reason for this is that numerous gradients operators exist but each of them would require a different memory interface to deliver the required data.

As it is described in the previous work section, memory access is a crucial aspect in all volume rendering architectures. Derived from [Kni93] and similar to VIRIM [GPR+94] and others, an eight-way interleaved memory is used for the VIZARD II system. In contrast to previous approaches, SDRAM DIMMs are used to allow different volume memory sizes without the need of fabricating a new PCI card. Since DIMMs come in modules providing a 64 bit data bus (72 bit including eight parity bits), four DIMMs are used spending 32 bit per voxel and replicating volume data in one dimension.

Classification is performed using the sample value for the look-up into the classification table and paging between different classification tables is realized using the

segmentation index or an index determined by the sample position relative to the classification planes. The result of addressing the two 32 bit SRAMs is a (r,g,b,α, k_a, k_d, k_s) tuple. Phong shading is performed using the gradient at sample position and a look-up table based shading technique as presented in [VF94, vSSB95]. The tables[3] require 3 KBytes of memory and only need to be computed when the illumination parameters change, e.g. the direction of the lights. The obtained diffuse and specular shading intensities are multiplied with the material properties and with the color of the classified sample. Currently, only white light sources are supported but this could be extended using RGB instead of intensity as table entries for the specular component.

As a last step, each classified and shaded sample needs to be composed with the previously accumulated color. The final pixel of a ray is obtained once the last pixel of the ray is composed or once the accumulated opacity is higher than a certain threshold, for example $\alpha \geq 0.98$. When using maximum intensity projection, all samples need to be processed until the highest value is found.

7.2.2 Architecture

A schematic overview of the VIZARD II system architecture is given in Figure 7.1. The VIZARD II system architecture has been designed for a ray casting algorithm

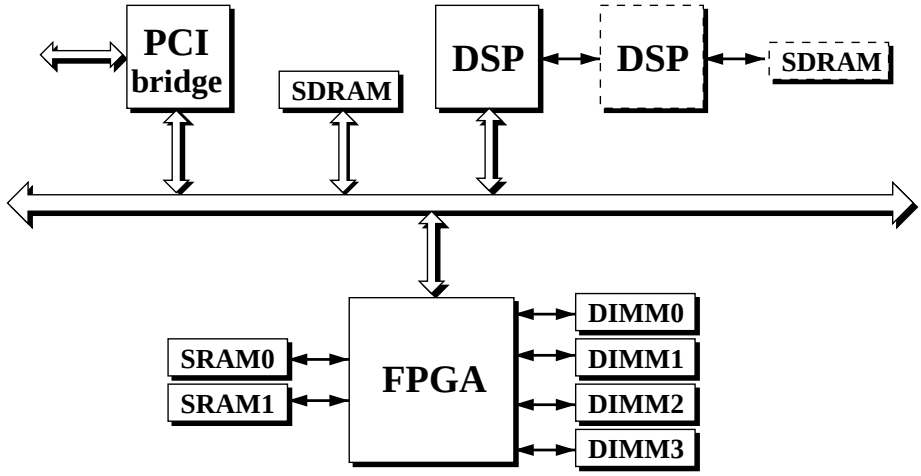


Figure 7.1: VIZARD II system architecture: The second DSP (dashed block) is optional.

and therefore has a dedicated memory interface to provide optimal voxel access for arbitrary rays. A local bus is used to transfer data within the system, but also to enable data transfer to and from the outside world (PCI bridge). The main component is the reconfigurable FPGA chip. It controls two SRAM and four DIMMs. Furthermore, there is one DSP and a SDRAM which is the external memory of the DSP. A second DSP and SDRAM are optional and not needed for the implementation of ray casting[4].

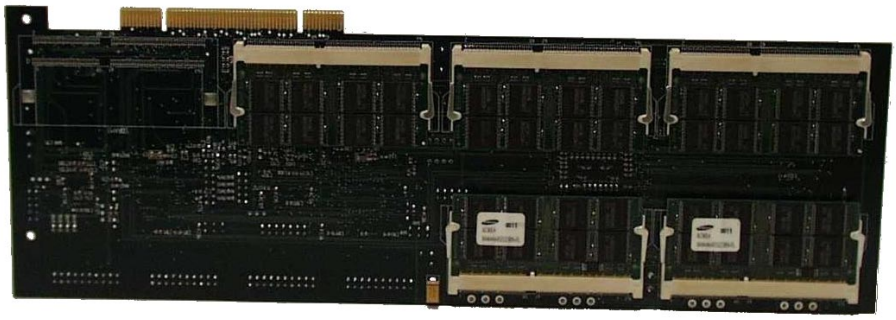
[3] One for the diffuse and one for the specular intensity which can be eye point independent.

[4] The VIZARD II board has been designed in a joint project with Phillips Research Hamburg including an implementation of a volume reconstruction algorithm which needs the second DSP.

7.2.3 PCI-Board

The VIZARD II system is implemented as a long PCI board with most components running at 100 MHz (FPGA, DIMMs, and SRAMs), the DSP runs at 80 MHz. While the backside is used to plug-in the DIMMs, the front side holds all logical units, as there are: one PCI interface chip (PLX), up to two DSPs (Analog Devices SHARC), one DIMM per DSP, one FPGA chip (XILINX Virtex XCV1000), and two 32 bit SRAM memories (see Figure 7.2). Additionally, there are pins on the board which are connected to the SHARC links, a specialty of the used DSPs. These links run at 80 MHz and can be connected to any other DSP to allow communication and data transfer between other DSPs. Thus, a system consisting of multiple boards and an additional video board could be built, such that the image can be delivered directly to the video controller without needing to transfer it over the PCI bus. Such a video board could also handle Gaussian filtering of the image before displaying it. Finally, there are connectors such that additional power can be supplied. This is necessary since the PCI bus does not supply sufficient power to drive all logic available on the board, mainly the SDRAM DIMMs which need a lot of power during refresh cycles.

(a)



(b)

Figure 7.2: The VIZARD II PCI card: (a) Front view (b) Back view.

PCI Interface

Instead of designing a PCI interface from scratch, an off-the shelf component is used. The two main options when buying a PCI bus interface are ASICs or PCI cores for FPGAs. The latter offers more flexibility but also requires FPGA space. Instead of using a second FPGA chip, a ready PCI interface chip of PLX Technologies has been

chosen. The PCI 9054 offers master and slave capabilities as well as simple driver models which can easily be extended. A reconfigurable PCI interface solution is not necessary since the additional buffering can be done on the FPGA or in the SDRAM of the DSP.

DSP

The DSP (digital signal processor) is a SHARC ADSP-21160 running at 80 MHz. With its two internal fully parallel pipelines, it provides a peak performance of 600 MFLOPS. It is a 32 bit core, capable of 32 bit fixed-point or 40 bit floating point computations. Furthermore, it has a dual-ported 4 Mbit on-chip SRAM, an integrated I/O processor, and six SHARC links running at 80 MHz which can be connected to other SHARC DSPs. The board comes with one DSP but a second one can be added.

The DSP is mainly used for setup calculations which includes the traversal of the viewplane as well as the computation of the intersection of each ray with the volume dataset (P_{entry}). To ensure highest precision, floating point arithmetic is used. An advantage of the ADSP 21160 are its SIMD capabilities and the completion of each floating point operation within a single cycle. This includes division as well as the computation of $\frac{1}{\sqrt{\ }}$ which is frequently needed. The algorithm used for the calculation of the intersection points of ray and volume is a modified version of the algorithm of Woo [Gla90, Woo90]. In short, for each ray the three potential volume faces are determined using basic compare operations. In each dimension, a division by the ray increment is needed to obtain the relative distance. The maximum of all three values determines the intersection point. Two further multiplications are needed to determine the intersection coordinates. The ray exit (P_{leave}) does not need to be computed since this is performed on the FPGA by testing the ray position against the volume boundaries. However, when using classification planes, additional intersection calculations are needed for each classification plane, as illustrated in Figure 7.3.

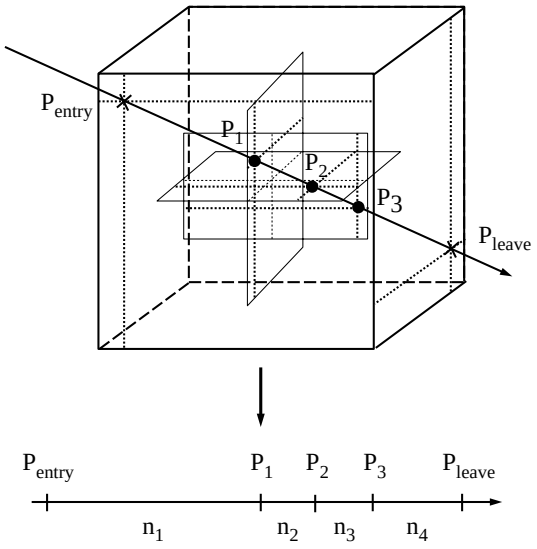


Figure 7.3: Ray intersecting the volume and three classification planes.

Since classification planes are not necessarily axis aligned, they can have arbitrary orientation. This aggravates the task of finding the intersection point but an optimized algorithm [Kir92, Geo92] requires nine MULT, eight ADD, one DIVIDE, and seven compares. The algorithm computes the orthogonal distance of a point to the plane and then scales this distance by the scalar product of ray direction and plane normal.

In a final step, the hit points are sorted and the number of samples to be taken for each ray segment is determined. Computing the Euclidean distance(s) can be done efficiently due to the SIMD capabilities and the fast square root instruction. It is estimated that up to three classification planes can be computed for roughly 15 frames per second using 256^2 rays. Higher frame-rates could be accomplished using both DSPs. However, the next generation VIZARD board will use a larger FPGA chip providing enough gates to perform the intersection calculations on-chip using division tables.

To be capable of distinguishing between the classification of samples without knowing about the position of the cut planes at the sample classification stage, it is necessary to parameterize the ray and forward the parameters with the data. Therefore a tuple $< c_i, n_i >$ is assigned to each ray segment, where c_i denotes the classification table and n_i the number of samples to be taken along ray segment i. A maximum of up to four tuples is send to the Ray Processing Unit (RPU). The use of ray segments removes 3D spatial computations from the RPU simplifying the hardware implementation. By incrementing the ray position and decrementing the sample counter, the corresponding classification table index can be fed through the pipeline to later on select the correct classification table. Up to three classification planes result in up to eight classification spaces. The first implementation deals with 8 bit voxel data requiring 2 KBytes per classification table and providing 64 bit per entry (R,G,B,α,k_a,k_d,k_s). The classification tables are stored in the external on-board SRAMs.

Memory Interface

The PCI board contains off the shelf SDRAM DIMMs in which the volume data is stored. DIMMs were the first choice since they are relatively cheap and allow to use different memory configurations on the same board. However, this comes at a certain price: First, the DIMM slots cover a large PCB area[5] and are therefore placed on the backside of the PCB, and second, they provide a 64 bit data bus. As a result, data is replicated such that each DIMM entry holds a 32 bit voxel and the subsequent voxel in z direction. In x and y, voxels are interleaved across four DIMMs. This allows to fetch eight voxels in parallel which is needed for trilinear interpolation. Looking at modern SDRAM devices, it could be worth removing the replication and making use of DDR SDRAM instead[6], but one would sacrifice flexibility.

Generally, ray casting suffers from arbitrary memory access due to the impredictable ray traversal. To still obtain a good performance, the volumetric data is stored in a cubic fashion. The goal is to keep the needed data in the caches of the SDRAMs while a ray is cast. The caches of four DIMMs can hold as much as 128 Kbit which allows to store a total of $16 \times 16 \times 16$ voxels of 32 bit within the available cache space. For a dataset of $64 \times 64 \times 64$ voxels, this translates into $4 \times 4 \times 4$ subcubes, as illustrated in Figure 7.4.

As long as samples are taken within such a subcube (16^3 voxels), the optimal memory access rate can be reached, which is 10 ns for the selected DIMMs. Only when

[5] Laptop SDRAM DIMMs are used and placed flat on the PCB board.

[6] XILINX already offers VHDL models of DDR SDRAM controllers for their Virtex FPGA series.

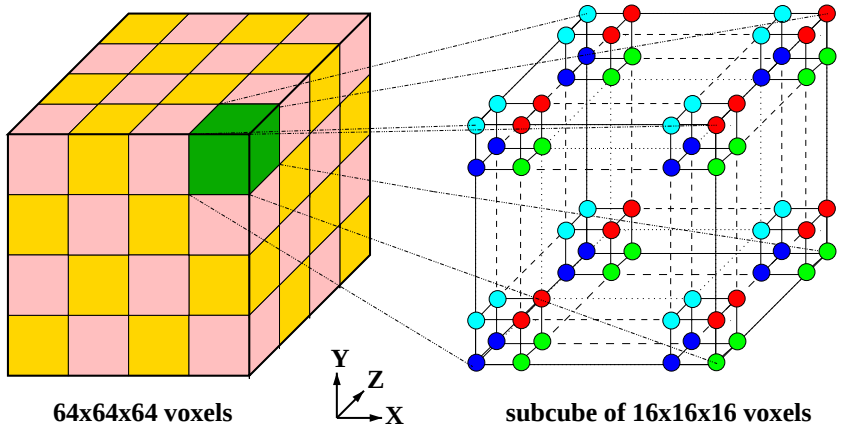


Figure 7.4: Memory organization scheme.

crossing the border of two subcubes, a cache miss will occur and cause a time penalty. This is illustrated in Figure 7.5 for the two dimensional case. While casting the ray

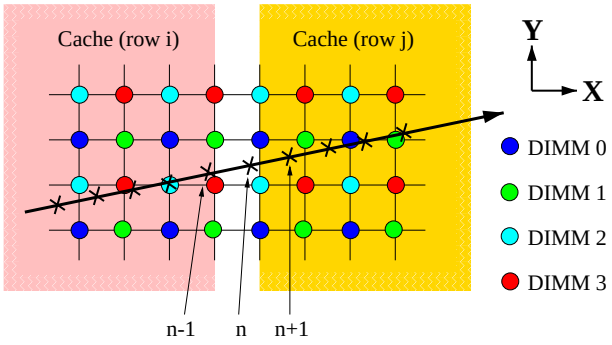


Figure 7.5: Crossing of cache borders.

until sample $n-1$, the values will be available across the cache lines of the DIMMs. However, for sample n the cache lines of DIMM 0 and 2 will need to be pre-charged and a new row can be activated. Depending on the used DIMM, this process takes between 30 and 90 ns. Once these rows have been activated sample n can be processed. Unfortunately, DIMM 1 and 3 will stall for sample $n+1$ since they need to pre-charge and activate a new row. Hence, crossing such a cache border results in two subsequent cache misses[7] which significantly increases the average sample access time from 10 ns to roughly 20 ns, but can vary depending on the sampling rate. The higher the sampling frequency, the less the relative penalty of stalling due to the higher cache efficiency. In the three dimensional case, up to three subsequent stalls can occur when crossing cache boundaries.

Fortunately, subsequent stalling of different memory modules can be hidden by interleaving the pre-charge and activate cycles using FIFO buffers. Figure 7.6 illustrates

[7] Depending on the sampling rate and the sample location this does not necessarily happen within two subsequent steps.

this schematically for the case depicted in Figure 7.5. To accomplish interleaving of

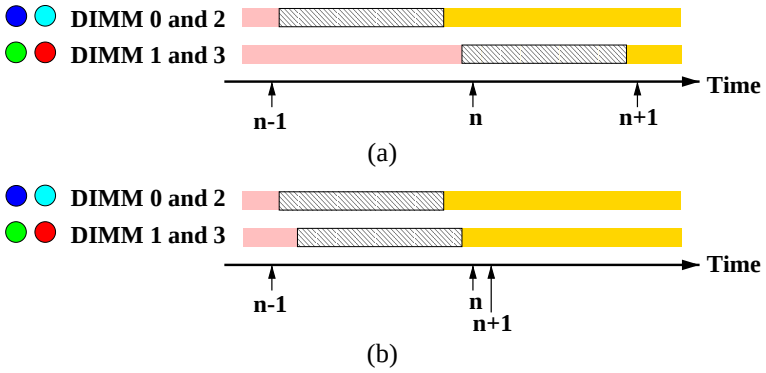


Figure 7.6: Cycles needed for crossing of cache borders without (a) and with interleaving of pre-charge and row activate (b).

stall cycles, address and data FIFOs are used for each of the four DIMMs. Hence, while one or more DIMMs stall, the address generation continues and voxels can be delivered from the data FIFOs. The address generation will stall when at least one of the address FIFOs is full and the ray processing will stall when at least one data FIFO is empty. With a FIFO depth of 16, an average memory access time of 12.7 ns can be achieved while the sampling rate is one. For a sampling rate of 0.5, this improves to 11.3 ns due to the better cache locality [DMK99].

FPGA (XILINX Virtex XCV1000)

The Field Programmable Gate Array (FPGA) is the core of the VIZARD II system. Most of the previously described memories (DIMMs and SRAMs) are connected with the FPGA. The entire RPU is implemented in the FPGA as well as further control units. While the I/O Control buffers in and outcoming data, the instruction decoder (InDer) controls the download of volume data, shader tables, classification tables, and ray data as well as the upload of finished pixels. A schematic overview of the FPGA, its logical units, and connections is given in Figure 7.7. In the following, the task of each unit, the required data exchange, and the control flow will be described in more detail.

I/O Control

The FPGA is not capable of controlling the local bus of the board. Instead, it is implemented as memory mapped device such that the DSP can write into it, similar to a memory. Therefore, communication between DSP and FPGA is accomplished using two DMA lines each consisting of a DMA request and a DMA acknowledge. The I/O Control is responsible for handling this protocol such that data can be transfered from the DSP to the FPGA and vice versa.

Status Control

The status control is fairly small but important for debugging of the design. It contains a full 32 bit register which stores information of the status of the design. This includes

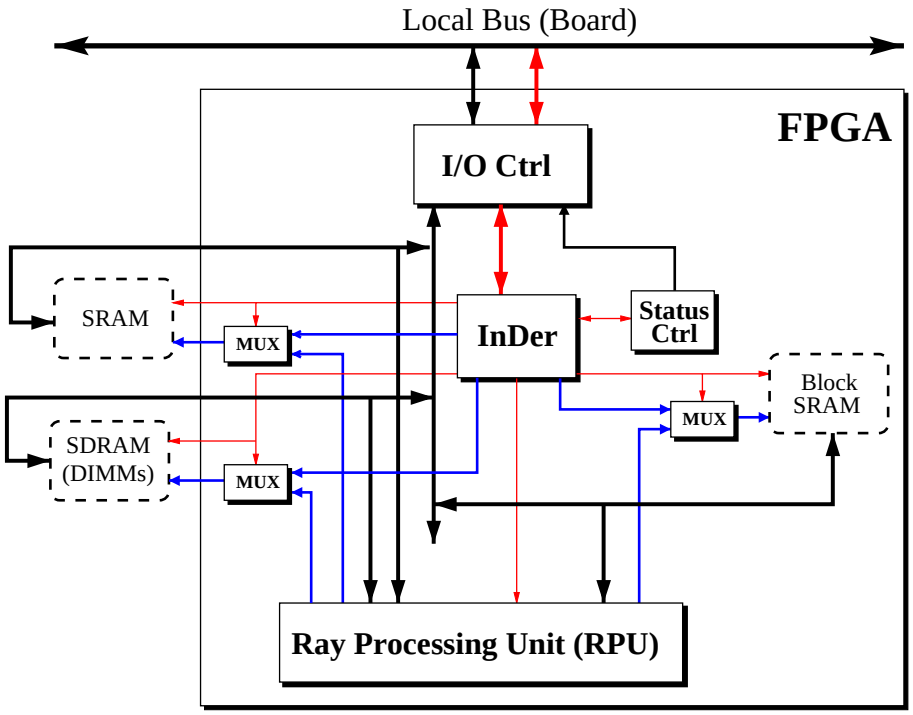


Figure 7.7: The FPGA toplevel components consisting of I/O Control (I/O Ctrl), Instruction Decoder (InDer), Status Control (Status Ctrl), and Ray Processing Unit (RPU). Data lines are black, address lines blue, and control lines red.

the current processed command, a flag in case a time-out has occurred, bits for refresh of DIMMs, and many others. In case of a dead-lock, one can still read this 32 bit register from the DSP to get a hint what went wrong.

Instruction Decoder (InDer)

The DSP needs to send different data down to the FPGA in order to provide all necessary information. Some of this data changes on a per frame base while other is valid for many frames. The current set of instructions includes the following operations:

- *Download data to DIMM:* Parameters are DIMM id, base address where to start, and number of voxels to download. Thus, parts of a volume can be replaced and multiple volumes stored.
- *Download shader tables:* Either the diffuse or the specular table is replaced.
- *Download classification table:* Parameter is index of the table to be downloaded.
- *Download dataset info:* Parameters are the minimum and maximum of the volume bounding box, the number of subcubes in all three dimensions, and render mode (shaded/unshaded, MIP, etc.).
- *Process ray:* The following words are interpreted as ray entry point, ray increment, and number of samples to be taken per ray segment.

All instructions are followed by a number of data words which are then interpreted correspondingly.

The InDer is realized as a state-machine that accepts an instructions if the instruction is valid, named *GrandCentral*. For each instruction, the responsible instruction state-machine is given control to process the subsequent incoming data. Once the instruction has been completed, control is given back to the GrandCentral, as illustrated in Figure 7.8. The individual instruction state machines control the corresponding

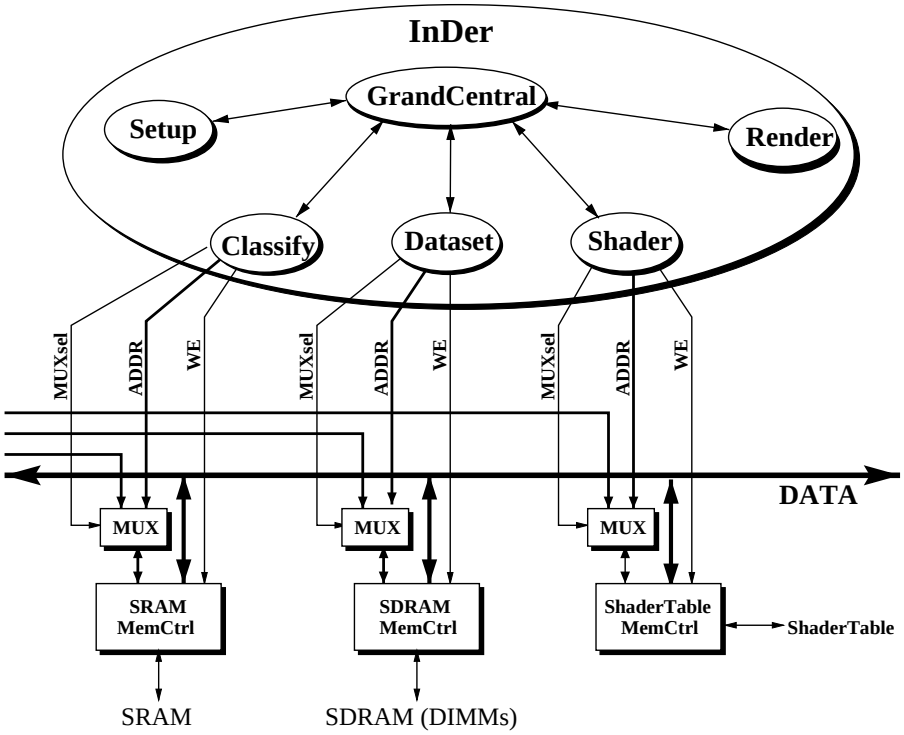


Figure 7.8: The Instruction Decoder (InDer): Depending on the incoming instruction, the corresponding state-machine receives control and returns control to GrandCentral after completion.

memory modules, and multiplex the read/write addresses as well as data lines.

Ray Processing Unit (RPU)

The ray processing unit traces a given ray through the volume and returns a final pixel value after the last sample within the min/max bounding box has been processed. It consists of several units, as illustrated in Figure 7.9. The *ray casting unit* (RC) generates the coordinates of the samples along a ray and the corresponding control information. Using the integer part of the sample coordinate, the individual addresses for the four DIMMs are generated in the *address unit* (AU). Since the DIMM access introduces latency, the corresponding control information is fed through a FIFO to ensure data alignment over time while the memories read the specified voxels. The *trilinear interpolation unit* (TIU) generates sample values and gradients which are sent to the

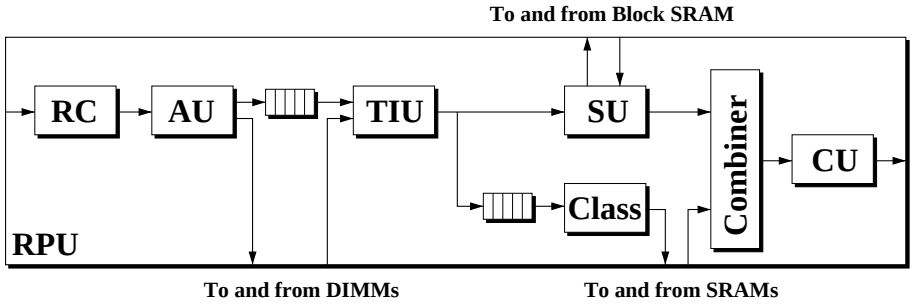


Figure 7.9: Ray processing unit (RPU) consisting of ray casting unit (RC), address unit (AU), trilinear interpolation unit (TIU), shader unit (SU), classification unit (Class), combiner, compositing unit (compos), and FIFOs.

shader unit (SU) to perform the computation of the diffuse and specular intensities. Parallel to this, the density value is fed through a shift-register before being interpreted as color and material properties (*classification unit* (Class)). The cycle aligned shading intensity and classification data is used in the *combiner unit* (CU) to perform Phong shading. Finally, all values along the ray are blended in the *compositing unit* which can be done using blending or MIP.

Ray casting Unit

A ray is specified by its entry point in the volume P_{enter}, the ray increment vector R_{dir}, and possibly up to four ray segments ($< c_i, n_i >$), as described in Section 7.2.3. Each ray segment is represented as a tuple containing the classification table c_i and the number of samples n_i. The position of P_{enter} is stored in a register and in each cycle the position is shifted by adding the ray increment until the position is not anymore inside the volume. To prevent stair-casing artifacts, the position tracking needs to be performed at a high precision (16 fractional bits) while only less precision needs to be used for the trilinear interpolation (8 fractional bits). Parallel to the position incrementing process, the number of samples for the current ray segment is decremented and the tuple of the subsequent ray segment fetched in case zero has been reached. The integer part of the current ray position, its fractional values, and the classification index are forwarded to the address unit.

Address Unit

The memory interleaving scheme — as described earlier in Section 7.2.3 — influences the address generation. Generally, the address unit receives the coordinates of the lower left corner voxel of the sample. The address x, y, z of the lower left corner voxel (P_{ref}) is used to determine the corresponding subcube address ($sc(x), sc(y), sc(z)$) and the offset inside the subcube ($sco(x), sco(y), sco(z)$).

$$\begin{aligned} sc(x) &= x \text{ slr } 4; \\ sc(y) &= y \text{ slr } 4; \\ sc(z) &= z \text{ slr } 4; \\ sco(x) &= (x \text{ slr } 1) \text{ and } 7; \end{aligned}$$

$$\begin{aligned} sco(y) &= (y \text{ slr } 1) \text{ and } 7; \\ sco(z) &= z \text{ and } 15; \end{aligned}$$

Due to the memory interleaving, the computation scheme results only on correct values for DIMM 3. For the other memory modules, the subcube and offset address in x and y need to be performed slightly different, requiring two additional incrementers.

$$\begin{aligned} DIMM2 &\rightarrow sc(x+1), \quad sco(x+1) \\ DIMM1 &\rightarrow sc(y+1), \quad sco(y+1) \\ DIMM0 &\rightarrow sc(x+1), \quad sco(x+1), \quad sc(y+1), \quad sco(y+1) \end{aligned}$$

Basically, for DIMM 2 $(x+1,y,z)$ is needed, for DIMM 1 $(x,y+1,z)$, and for DIMM 0 $(x+1,y+1,z)$ which affects the offset but possibly also the subcube address. Finally, the linear address for each DIMM can be computed using its subcube address and offset.

$$\begin{aligned} addr = {} & ((sc(z) * \text{SC\_IN\_X} * \text{SC\_IN\_Y} + sc(y) * \text{SC\_IN\_X} + sc(x)) \text{ sll } 10) \\ & + (sco(z) \text{ sll } 6) + (sco(y) \text{ sll } 3) + sco(x) \end{aligned}$$

where SC_IN_X (SC_IN_Y) is the number of subcubes in x (y).

Adding the offsets (DIMM row entry) is accomplished by concatenating the bits of the three offsets (sco_z, sco_y, sco) but the subcube address (DIMM row) requires a few multiplications and adders. The latter one can only be simplified for datasets where the size is restricted to a power of two. In this case, one can also simply concatenate the subcube addresses (sc_z, sc_y, sc_x) to obtain the DIMM address.

Trilin Unit

Different filter kernels can be used to obtain interpolated results. With respect to available logic and memory bandwidth, VIZARD II has been designed to perform trilinear interpolation since higher order interpolation schemes require more voxel values and more logical units (see Section 1.2.8). The trilinear interpolation has a separable kernel and can therefore be split into seven linear interpolations resulting in four linear interpolations in one dimension, two in the second dimension, and one in the third dimension. Figure 7.10 schematically illustrates the trilinear interpolation computing a sample and the corresponding gradient at position $P(x,y,z)$. The position can be split into an integer part P_{ref} which denotes the lower left corner of the interpolation cube and the fractional parts being represented as 8 bit weights w_x, w_y, w_z. A weight of 1.0 would result in another base address which means that the 8 bit of each weight represent the interval $[0.0, 1.0[$. This simplifies the implementation of the interpolation since voxel value and weight can simply be multiplied, shifting the result by 8 bit to the right (slr 8).

Due to the memory organization, data is replicated in z direction but interleaved in x and y direction. Hence, the hardwired datapaths requires that the linear interpolation in x and y are inverted depending on the address of $P(x,y,z)$. This can easily be detected checking the lowest bit of $P(x,y,z)$ in x/y ($P_{x(0)}/P_{y(0)}$). Multiplexing the weights and their inverse is one solution but since the weights are n bit representing a number in the interval $[0,1[$, the inverse would require $n+1$ bit to be able to correctly represent 1.0. As a result, the multipliers would be enlarged. Fortunately, there is another solution

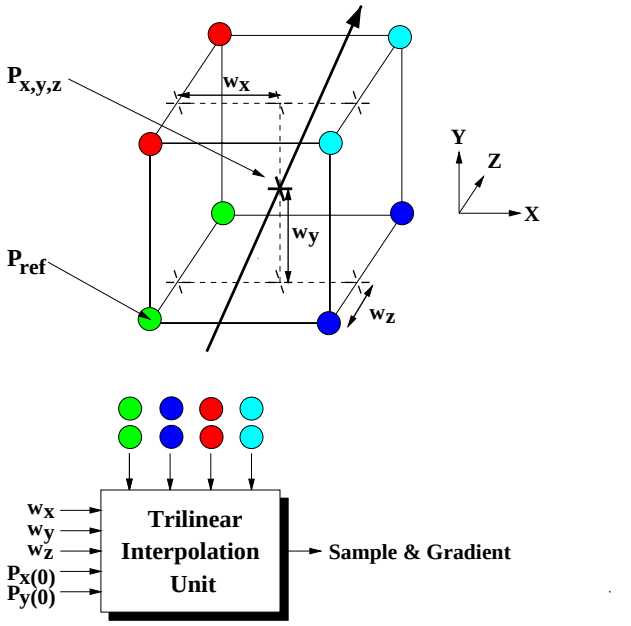


Figure 7.10: Trilinear interpolation.

which turns out to be cheaper because the linear interpolation can be written in two ways.

$$f_{linear}(a,b,weight) = a*(1.0-weight)+b*weight \quad (7.1)$$
$$= a+weight*(b-a) \quad (7.2)$$

While Equation 7.1 requires two multiplications and one adder (assuming $1.0-weight$ is available), Equation 7.2 requires only one multiplication but two adders. To circumvent multiplication of signed numbers, two cases can be differentiated.

```
if (a>=b)
   result = a - weight*(a-b);
else
   result = a + weight*(b-a);
```

This requires a simple compare operation ($\leq$) but ensures that only the final adder needs to handle signed numbers (one additional bit), but the result itself can never be a negative value. The earlier mentioned inversion of the linear interpolation can be incorporated by extending the multiplexing of the inputs using the lowest bit of the address of the samples position (flip).

```
if ((a>=b) xor (flip))
   ...
```

It might appear that all these optimizations are subtle and without much impact but it turned out that the optimized linear interpolation unit is almost 25% smaller than the straight forward implementation using signed representations computing $1.0-weight$.

Taking into account that the trilinear interpolation unit is the biggest of all units containing using a total of 28 linear interpolations[8], this is an important saving as shown in Section 7.3.1.

Shader Unit

The shader unit performs the correct computation of the diffuse and specular light intensities present at a given sample position. This could be performed by computing vector and scalar products but implementing a square root unit is expensive and not trivial. Therefore, it is performed using cube-mapping, a technique where the environment is mapped onto the six faces of a cube. By mapping the diffuse light intensities of all light sources onto the cube faces, the diffuse light intensity can be determined by computing the intersection of the gradient vector with one of the cube faces [vSSB95, Hir99]. The same mechanism can be applied for specular light sources but requires some more subtleness since it requires to compute the reflected vector to address a specular cube map [VF94, vSSB95], making the specular table independent from the eye position. Thus, instead of performing expensive vector operations, the corresponding sample on the cube map can be determined and used.

Figure 7.11 illustrates a Phong shaded sphere using a cube map of different size. The middle column is generated using 16×16 entries and achieves quite satisfactory results while consuming 1.5 KByte of memory [Hir99] only. However, when using a high exponent for the specular highlight, it is necessary to increase the resolution of the tables. The VIZARD II system implements cube map based shading using the BlockRAM available on the Virtex FPGA. For 16×16 entries per face, three to four BlockRAMs are needed[9].

Classification Unit

The interpolated sample value (8 bit) is taken as an index into a classification table stored in two SRAM chips. Each SRAM has a 32 bit data bus and both can be addressed individually. In the current design, the same address for both SRAMs resulting in 64 bit of classification data. This includes red, green, blue, α, k_a, k_d, k_s using 8 bit for each, except for α which is 16 bit for high precision when using semi-transparent rendering. Additional to the resulting classification data, the original density value is forwarded to the combiner unit and possibly used in the compositing in case maximum intensity projection is enabled.

Combiner Unit

While the classification unit delivers the color and material properties of a sample, the shader generates a diffuse and a specular intensity. The combiner multiplies these values implementing Phong shading.

$$C = k_a * I_a + k_d * I_d * C_{\text{classified}}(\text{sample}) + k_s * I_s \tag{7.3}$$

where k_a, k_d, and k_s are the material properties, I_a, I_d, and I_s are the ambient, diffuse, and specular intensities, and $C_{\text{classified}}(\text{sample})$ is the resulting classified color of the sample value. Currently, the specular light component is assumed to be white but could

[8] Four trilinear interpolations, one for the sample interpolation and one for each gradient component.

[9] A memory efficient implementation requires three and a logic efficient implementation four BlockRAMs.

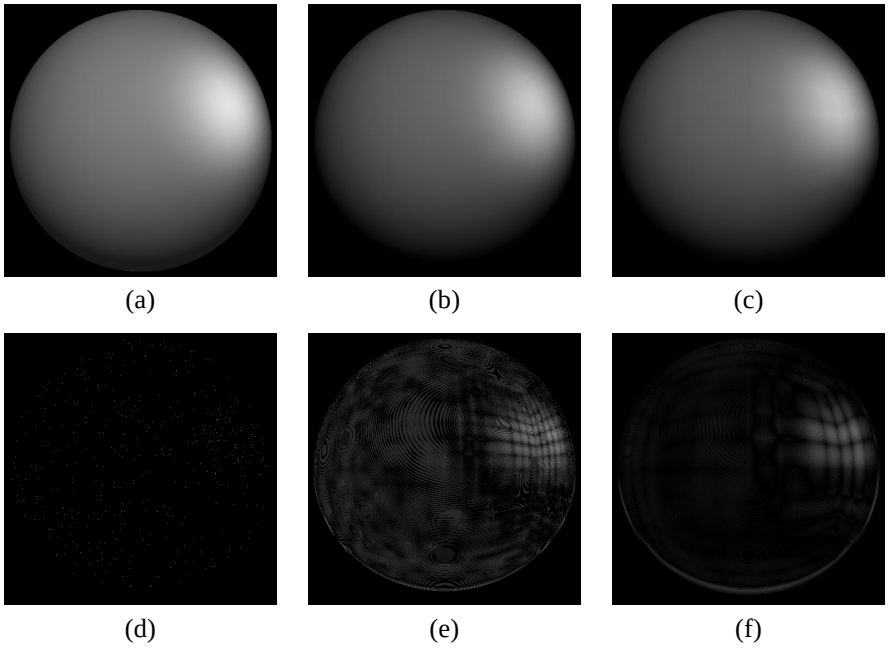


Figure 7.11: Shading quality of a Phong shaded sphere: (a-c) Shading using cube map: (a) Cube map of 256×256 entries. (b) Cube map of 16×16 entries. (c) Cube map of 8×8 entries. (d-f) Difference images to sphere shaded using floating point precision: (d) Error of (a), magnification 200. (e) Error of (b), magnification 50. (f) Error of (c), magnification 10.

easily be extended to colored light sources spending 24 bit color intensity information instead of 8 bit monochrome intensity in the specular shader table. The same applies for the ambient color which is also assumed to be white but can be multiplied with any user defined color. Three color components, the density value, and the opacity are forwarded to the compositing unit.

In contrast to the trilin unit where simple fixed-point arithmetic is applied[10], the combiner requires a more subtle type of fixed-point arithmetic. All parameters (color, material, and intensity) use unsigned n bit numbers ($[0 \ldots 2^n - 1]$) representing the full interval $[0.0, 1.0]$ including 1.0. Jim Blinn presented in his article "Three Wrongs make a Right" [Bli98] how to compute the correct result. For two unsigned n bit numbers a and b of interval $[0.0, 1.0]$, one has to compute the following.

$$\begin{aligned} tmp &= a * b + 2^{n-1} \\ result &= (tmp + (tmp >> n)) >> n \end{aligned} \tag{7.4}$$

Overall this requires one n bit multiplier, two $2 * n$ bit adders, and two shifters which are simply hardwired bit selections. This works since the *error* made is always below what can be represented with a number of n bit. When multiplying numbers with different bit width, the larger one can be discretized or the smaller one extended using repeated

[10] Generally, two values can be multiplied and the result shifted in case one of the two operands represents the interval $[0.0, 1.0]$ and the other one represents the interval $[0.0, 1.0[$.

fractions [Bli98]. Repeated fractions work by repeating a number as often as needed to expand it to a certain number of bits. This ensures an equal distribution of the original set of values onto the larger set even though not all values are used. One can look at this as perspectively projecting an interval of values onto a larger interval. The advantages of this fixed-point arithmetic are its accuracy and simplicity with respect to hardware implementation.

Instead of using the arithmetic described in Equation 7.4 and 7.5, it is also possible to spend one more bit such that k_a, k_d, and k_s represent $[0.0, 1.0]$ using $[0...2^n]$. However, memory space is wasted and the multipliers are enlarge by two bits because they only come in units of even sized operands[11].

Compos Unit

All colored samples of a ray need to be combined into a final pixel value. This is either done using MIP projection or a discretized version of the common volume rendering line integral, as described earlier by Equation 1.3 in Section 1.2.7. While MIP is a very simple operation and does not introduce a hard to solve bottle-neck, computing the discretized volume rendering line integral is the most difficult part of the entire architecture with respect to its hardware implementation. For each sample, the following computations are necessary.

$$color_{acc} = color_{acc} + (1 - \alpha_{acc}) * color_{sample} * \alpha_{sample} \quad (7.5)$$

$$\alpha_{acc} = \alpha_{acc} + (1 - \alpha_{acc}) * \alpha_{sample} \quad (7.6)$$

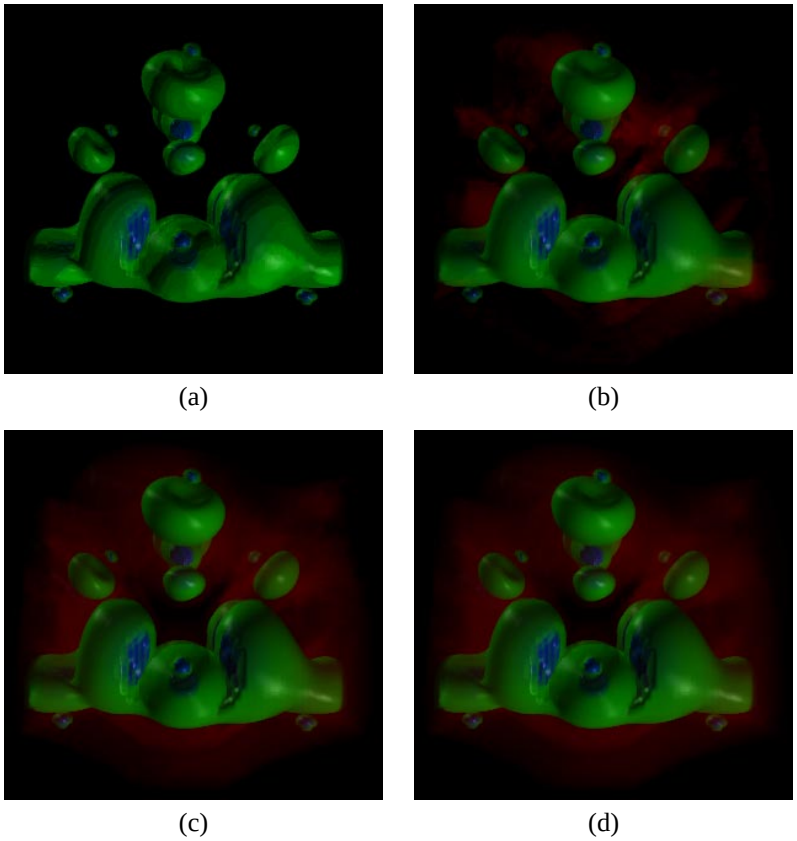
While the problem is not obvious in software, Looking at this from a hardware implementation point in a fully pipelined design, it requires several sequential operations (one 16 bit multiplication and one 16 bit add, see Equations 7.5 and 7.6) before the next sample value can be accepted. This is referred to as *compositing problem*. Due to the combinatorial delay of the multiply and accumulate operation, an estimated clock-rate of 44 MHz can be achieved on a XILINX Virtex XCV1000 FPGA (speedgrade 6)[12]. This simply determines the maximum clock-rate of the design which unfortunately is slower than the memory modules and therefore valuable bandwidth is wasted.

One limited solution to the *compositing problem* could be to reduce the precision using 8 bit instead of 16 bit. However, this would result in unacceptably low image quality for semi-transparent rendering and is not scalable. The precision of the accumulated color and α has a strong impact on the final image quality, especially for semi-transparent rendering. Figure 7.12 depicts four images of the neghip dataset rendered using 6, 8, 10, and 16 bits internal accumulation precision. Even though 10 bits seem to be sufficient, the maximum difference is up to 10% of the color channel's range. The 16 bit image has a maximum difference of 1% in pixel values compared to the image rendered using floating point arithmetic.

Thus, the only solution to the compositing problem is interleaving of rays similar to *multi-threading*. Such a concept has been introduced earlier but for a different purpose solving the high memory access latency when using space leaping based on an externally stored distance volume [BHM99]. However, the latency of the compositing problem is much smaller and by interleaving 3×3 rays only, a design running at 100

[11] This applies for the optimized XILINX components running at a high frequency (> 100 MHz).

[12] This is certainly faster on an ASIC but for FPGAs, logical operations need to be mapped on CLB slices which has longer signal paths.



(a) (b)

(c) (d)

Figure 7.12: Precision of the accumulated color and α: (a) 6 bits. (b) 8 bits. (c) 10 bits. (d) 16 bits.

MHz can be accomplished on a XILINX Virtex XCV1000. More rays could be interleaved to also reduce memory page crossings but comes at a certain price. It requires re-ordering logic to first process all rays within the same subcube. This re-ordering needs to be undone just before the compositing to again provide enough cycles for the compositing calculation before the subsequent sample of the same ray arrives. Furthermore, early ray termination can only be exploited on a group of rays since fewer rays would introduce idle cycles. Therefore, the less rays are interleaved, the higher the efficiency of early ray termination. However, early ray termination has an average performance gain of around 10-40 % for outside views. Higher efficiency factors are only exploitable in endoscopic applications where early ray termination can result in speedups in the order of one magnitude. Hence, the optimal granularity of number of interleaved rays depends on the dataset and chosen classification but needs to be at least 3×3.

The fixed-point arithmetic à la Jim Blinn is also used in the compositing unit. Again, values are represented as n bit where the full data range represents the interval $[0.0, 1.0]$ which can not be handled by a simple multiplication followed by a shift operation.

7.3 Analysis and Performance

By the end of this dissertation, no fully functional board was available due to delays in the production of the board. Several components of the board have not been available for weeks of even months. This has been due to the booming semi-conductor industry which made it difficult to get several basic board components. Furthermore, the first board built was too thick for the PCI slot making another revision unavoidable. However, the first running implementation is expected by the end of 2000.

7.3.1 Logic Consumption

All units have been designed in VHDL and all time critical components of the architecture have been synthesized using Exemplar Leonardo Spectrum synthesis tools and a Virtex XCV1000 FPGA (speedgrade 6) as target. For verification of the synthesis results, a VHDL model of each synthesized unit has been integrated into the original testbench and simulated. The resulting percentage of chip coverage as well as maximum clock-rate of these units are summarized in Table 7.1. The by far largest unit is

Unit	CLB [slices]	chip area [%]	clock-rate [MHz]
InDer	45	0.37	132
RayCaster	253	2.06	140
Address	405	3.29	100
Trilin	2069	16.84	85
Shader	383	8.12	59
Classify	20	0.02	306
Combiner	146	1.19	62
Compos	1210	9.85	44

Table 7.1: Space consumption and clock-rate per unit for a speedgrade -6 on a XILINX Virtex XCV1000 which has a total of 12400 CLB slices.

the Trilin which is due to the numerous linear interpolations. It consists of 28 linear interpolation units plus shift-registers. Thereafter follows the compositing unit which needs to accumulate three color channels at 16 bit precision using correct fixed-point arithmetic, as presented in Section 7.2.3. Almost the same amount of logic is needed for shader unit which needs to compute the reflection vector and implements two cube maps. All other units are neglectable with respect to their size.

7.3.2 On-chip Memory Allocation

Besides its 12400 CLB slices, a XILINX Virtex XCV1000 provides 32 4 Kbit BlockRAMs which are dual read and write. The BlockRAMs are used to implement the shader tables as well as the data and address FIFOs. For the current setting using a shader table of 16×16 entries, a total of four BlockRAMs is needed, The other BlockRAMs are used to implement the FIFOs. However, moving to the next larger chip of XILINX (Virtex XCV1000E, 1600E, 2000E, etc.) more BlockRAM is available allowing to possibly implement larger shader tables using 32×32 entries or higher

or possibly implementing two RPUs to enable oversampling without much additional cost.

Generally, the VIZARD II system is rather memory bound than logic bound since many tables and FIFOs need to be stored on-chip and the number of BlockRAMs is limited. Alternatively, CLB cells could be used as memory but this is less efficient.

7.3.3 Bandwidth Analysis

The PCI bus running at 33 MHz is only heavily used during downloading the volume data onto the card. While static datasets or fixed sequences of volumes can be stored initially on the card, real time volume updates of entire volumes is not possible since it would require more bandwidth on the front bus.

Generally, data is send to the board and stored in the SDRAM memory of the DSP. This is necessary because the FPGA is a memory mapped device which can only be controlled by the DSP but not by the PCI interface chip. Therefore, a dataset is first send over the PCI bus to the on-board SDRAM and then transfered from the SDRAM to the DIMMs. The transfer of a dataset of 256^3 voxels over the PCI bus takes roughly 0.15 seconds assuming that replication is generated on the FPGA and 0.30 seconds if replicated data is transfered. The transfer of the data from SDRAM to the DIMMs is in the same order. The PCI bus as well as the local board bus are 32 bit wide. While the PCI bus runs at 33 MHz, the local board bus runs at 50 MHz.

The classification tables are also sent over the PCI bus but are neglectable since they are only 2 KBytes in size. Even if the classification changes for every frame, 30 frames per second would require 60 KByte/s bandwidth which can easily be handled. The same applies for the two shader tables which are 3 KBytes in size. They are computed by the DSP and do not need to be transfered over the PCI bus. However, in case the light sources would be modified in every frame, the computations on the DSP become the bottle-neck since ray intersections and shader tables need to be computed several times per second. This bottle-neck can be reduced using the second DSP.

Transferring an entire image back to the host requires 65 KByte/s which results in almost 8 MByte/s for 30 frames per second. This can also be handled by the PCI bus and does not introduce any bottle-neck.

The overall performance limitation of the system is given by its memory interface. The used DIMMs run at 100 MHz and need 70 nsec for a pre-charge and row activate. In average, this results in 12.7 nsec for each sample. Thus, 80 million samples can be generated per second. For a one to one mapping of samples to voxels, this results in a maximum of 5 frames/s. However, for zoomed views it will be higher and early ray termination can further increase the frame-rate. Possible acceleration and optimization techniques are presented in the subsequent sections.

7.4 Further Implementation Enhancements

The VIZARD II system is capable of delivering interactive frame-rates for datasets of 256^3 voxels where each voxel has 32 bit (8 bit voxel and 8 bit per gradient component). Further improvements with respect to speed or how to store more information with the voxel but still being able to obtain shaded images are necessary. In the following, space leaping and indexed gradients are presented.

7.4.1 Space Leaping

One of the most severe problems for ray casting architectures is the waste of computation cycles and I/O bandwidth, due to redundant sampling of empty space. While several techniques exist for software implementations to skip these empty regions, few are suitable for hardware implementation. The few which have been presented either require a tremendous amount of logic or are not feasible for high frequency designs (e.g. running at 100 MHz), where latency is one of the biggest issues. The architectures mentioned in Section 7.1 either do not provide space leaping functionality [PHK+99, MKS98, DMK99, RS99] or require an entire distance volume to be precomputed [KS97] and stored [BHM99]. However, distance volumes come at a certain cost since an additional volume needs to be stored. This increases the memory requirements significantly and is impractical for larger volumes.

In the following, a much simpler algorithm for space leaping is presented without requiring an entire distance volume.

Basic Algorithm

The volume is subdivided into subcubes and for each subcube it is determined whether the subcube is empty or not. This information is stored in an occupancy map containing a single bit per subcube; a very space efficient representation of the volume. While casting a ray, the corresponding bit of the occupancy mask is checked for each sample position. If the entry in the occupancy map indicates a non-empty subcube, sampling along the ray is simply continued in uniform manner. Otherwise, all samples within this subcube can safely be skipped. Determining the distance value is done similar to the algorithm used to compute the ray/volume intersection (see Section 7.2.3). Basically, for each dimension the coordinate of the possible intersection point and the relative distance are computed. The desired result is the minimum value and is used to multiply the ray increment.

$$dist = \min\left\{ \left\lceil \frac{D_x}{V_x} \right\rceil, \left\lceil \frac{D_y}{V_y} \right\rceil, \left\lceil \frac{D_z}{V_z} \right\rceil \right\} \tag{7.7}$$

where D_i is the coordinate of the possible intersection and V_i the ray increment, both in dimension i.

The advantage of the occupancy map is its simple generation which can be done very fast by scanning once over the entire volume. VIZARD II provides an extremely high memory bandwidth capable of scanning a 256^3 dataset in 20 ms[13], allowing interactive control of classification to work with the space leaping approach.

Hardware Implications

The additional costs in terms of computational hardware to implement this space leaping algorithm are quite modest. To avoid calculating the exact distance in 3D space, only the relative distance along each axis is calculated, as described earlier. For subcubes of 16^3 voxels, three 4 bit subtractions are necessary to determine the potential intersection coordinates D_x, D_y, and D_z (see Equation 7.7). Secondly, D_x, D_y and D_z are divided by the ray increments V_x, V_y and V_z. To reduce the complexity of this division, the inverse of V_x, V_y and V_z can be pre-calculate and stored. For parallel projection, this is needed once per frame but once per ray for perspective projection. Alternatively, an

[13]The memory of the VIZARD II architecture is eight times interleaved. Hence, a volume dataset of 16 Msamples read from 100MHz SDRAM can be scanned 50 times per second.

adaptive division table for the possible range of values could be used. The resulting distance value is obtained by multiplying D_x, D_y and D_z by $1/V_x, 1/V_y$ and $1/V_z$ which requires three multipliers of 8×8 bits. Finally three compares are necessary to determine the minimum of all three values (*dist* of Equation 7.7). The resulting value is used to scale the ray increment which is then added to the current position. Parallel to the computation of the increment value, the occupancy map is checked and either the multiplied increment or the uniform increment is used based on the bit in the occupancy map.

The logic to implement the described skip calculator requires 124 CLB slices of a XILINX Virtex XCV1000 FPGA, utilizing 1% of the FPGA logic as well as one BlockRAM to store the occupancy map of 4 Kbit. The clock frequency of the resulting pipeline is well above 100 MHz and adds eight cycles of latency to the address computation. Therefore, the processing of eight or more rays will be sufficient to accommodate latency. As described earlier, rays need to be interleaved to enable the compositing stage to run at 100 MHz (see Section 7.2.3) To minimize memory stalling effects when using several rays, an *Overtaking FIFO* can be introduced, as presented by [Dog00]. The Overtaking FIFO reorders the memory addresses of different rays in order to minimize page changes in the memory.

Experiments

A set of five different real-world datasets is used to demonstrate the efficiency of the presented space leaping approach. Images of the datasets are shown in Figure 7.13 and the characteristics are summarized in Table 7.2. While fuel and neghip are both

Dataset	Size	Source	Occupied voxels
fuel	64^3	simulation	5,24 %
neghip	64^3	simulation	46,38 %
foot	256^3	CT-anio	28,94 %
skull	256^3	CT	88,42 %
vessel	256^3	CT-anio	1,01 %

Table 7.2: Set of datasets which have been used to evaluate the space leaping approach. Occupied voxels are voxels with value > 0.

results of physical simulations with different number of occupied voxels, the other three datasets origin from medical acquisition devices. The skull is a very compact block of occupied voxels and due to noise, only a few unoccupied voxels — further on referred to as empty voxels — exist. In contrast, the vessel dataset contains narrow structures which are present across the entire dataset but a large number of voxels is empty. Finally, the foot dataset is a relatively compact block of occupied voxels with few noise.

The percentage of empty voxels in each dataset is given in Table 7.2, but does not directly reveal an estimate of the potential gain that can be expected. The efficiency of space leaping depends only on the percentage of empty subcubes. Figure 7.14 shows the percentage of subcubes which can be skipped for different subsube sizes without and with exploiting the given classification[14](Figure 7.13(c,d,h-j) illustrate the applied

[14]All voxel values which are classified fully transparent are considered empty.

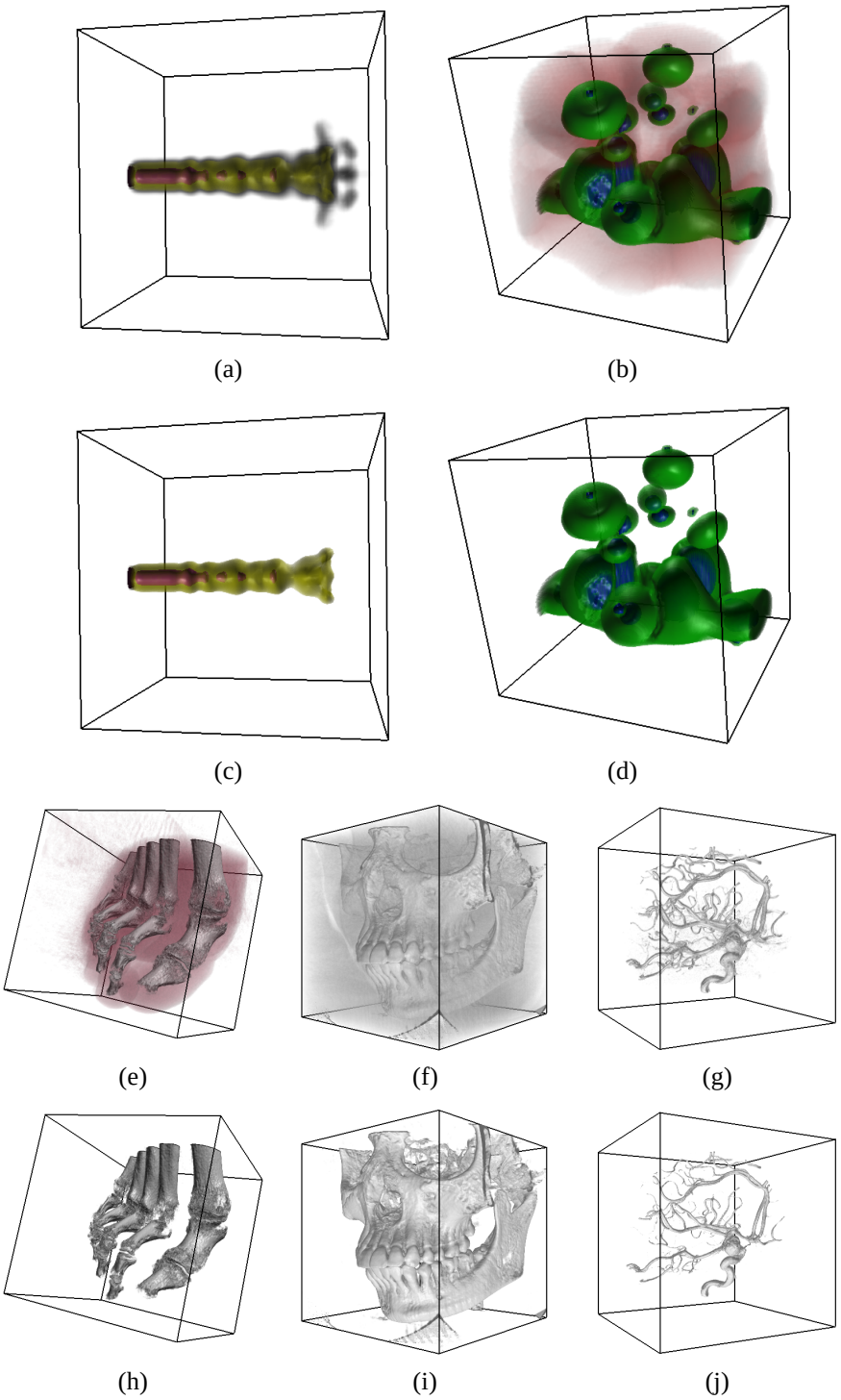
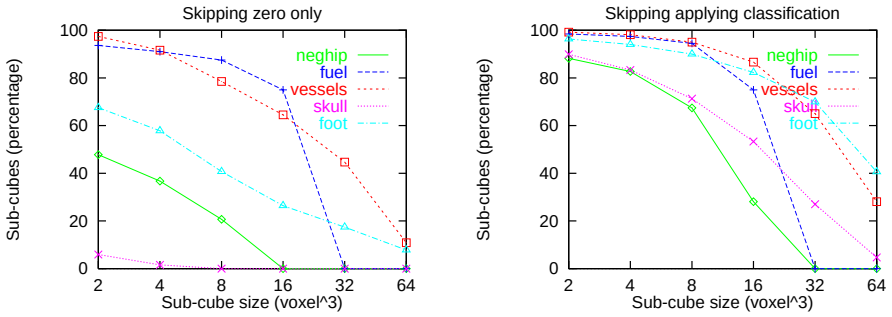


Figure 7.13: Test datasets: (a,b) and (e-g) have been rendered visualizing all occupied voxels. The other images were rendered applying a meaningful classification, as exploited in Figure 7.14(b).

classification). Obviously, the smaller the subcubes, the higher the percentage of sub-



(a) (b)

Figure 7.14: Percentage of subcubes which can be skipped. (a) Only empty voxels are exploited. (b) Classification is additionally exploited.

cubes which can be skipped. However, for each subcube at least one sample needs to be processed. Therefore, space leaping performance is not necessarily best for smallest subcube sizes. For the selected datasets, moderate improvements can be reported when skipping empty voxels only. However, when exploiting the given classification much higher improvements can be reported, especially for the noisy medical datasets.

Estimated Results and Discussion

Performance gains are generally limited to those parts of rays, which pass through empty subcubes. For a thorough analysis, an animation of 72 frames was generated for all five datasets, rotating around the center of the dataset starting with the views given in Figure 7.13. For each frame, the number of cycles needed to generate the image were measured using (i) no optimization, (ii) early ray termination, and (iii) space leaping with different subcube sizes. Table 7.3 shows the corresponding averaged frame-rates using a memory access time per voxel of 12.7ns, as described in [DMK99]. Generally, early ray termination is not a very efficient acceleration technique, unless the viewpoint is close to a highly opaque object covering large areas of the screen-space. For the presented views, performance gains due to early ray termination vary from almost zero for the fuel dataset to 25% for the neghip dataset. The only exception is the skull dataset, where a 90% performance gain is accomplished due to the screen filling opaque skull.

Space leaping based on skipping empty subcubes only, gives poor speed-ups for datasets with a high percentage of occupied voxels. This is illustrated with the skull dataset where 88% of all voxels are occupied (see Table 7.2). A similar observation can be made for the foot dataset. Only for the fuel dataset performance gains of 280% can be observed which are due to the large areas of non-occupied voxels surrounding the compact union of occupied voxels. Generally, much higher frame-rates can be accomplished exploiting the given classification. This results in performance gains ranging from 200% for the neghip dataset to 375% for the vessel dataset (additional to early ray termination). The performance gain for the neghip is only 200%, since a large number of samples still contributes to the final image. Overall, for the presented datasets of 256^3 voxels, frame-rates well above 15 frames can be accomplished.

Acceleration		none	ERT	4^3	8^3	16^3	32^3
fuel	'0'	16.1	16.4	27.3	41.9	45.8	16.4
	class	16.1	16.3	28.4	47.3	45.2	16.3
neghip	'0'	18.6	23.7	28.1	27.9	23.7	23.7
	class	18.6	23.3	36.2	46.2	31.5	23.3
foot	'0'	4.4	5.3	7.3	7.7	7.1	6.5
	class	4.4	5.3	9.0	14.4	19.1	16.8
skull	'0'	4.3	8.2	8.2	8.2	8.2	8.2
	class	4.3	7.8	12.9	17.3	16.0	10.8
vessel	'0'	4.3	4.7	7.8	10.4	10.7	8.3
	class	4.3	4.6	8.0	13.3	17.3	12.0

Table 7.3: Frame-rates for the five datasets skipping empty voxels only ('0') and exploiting the given classification (class). The frame-rates are averaged over 72 frames. Acceleration *none* stands for processing all samples along all rays and *ERT* stands for early ray termination.

While achieving good speed-ups additional to early ray termination, the selection of the appropriate subcube size is dataset and classification dependent. As a rule of thumb, a subcube size of 8^3 is suited for the smaller datasets (64^3) and subcubes of 16^3 for the larger datasets (256^3), even though for a few cases slightly higher frame-rates can be achieved for the next smaller or larger subcube size. Finding heuristics for the best suited subcube size is still subject of further research.

Overall, the presented space leaping approach achieves significant acceleration of the ray casting process without requiring an entire distance volume. With only $4Kbit$ of SRAM needed for the occupancy map and a simple skipping mechanism, a small latency is introduced computing the next sample position. This latency is significantly lower than pre-computing and storing a distance volume in the external volume memory [BHM99]. The latency due to the calculation of the new skipping value can be accommodated by interleaving the processing of eight rays, even in an FPGA design running at 100 MHz. Furthermore, the amount of extra logic required for the presented space leaping mechanism is less than 1% (130 CLBs) and one BlockRAM of a XILINX Virtex XCV1000 FPGA.

7.4.2 Indexed Gradients

Even though gradients should be a per voxel property as normals are a per vertex property in polygon graphics, storing the gradient comes at a certain price. Using 8 bit per gradient component leaves 8 bits for the voxel value in case 32 bit are available per voxel. In order to store 12 bit voxels, the gradient components need to be reduced to 6 bit and further reduction would be unavoidable to store additional segmentation information. However, shading based on gradient components with six or less bit is of low quality which prevents high image quality. One solution would be to provide more bits per voxel which is costly but could be accomplished using DDR SDRAM. However, it would reduce the overall cache space such that only $8 \times 16 \times 16$ voxels would fit in the caches of the DIMMs. Another approach could be to compute gradients on the fly but depending on the gradient estimation scheme, different bandwidth requirements would arise possibly reducing the overall frame-rate.

To circumvent the high bandwidth and connectivity requirements, a gradient look-up table is introduced. In contrast to storing the full gradient at voxel location, a smaller gradient index is stored. The gradient at sample location is hence calculated by performing eight gradient look-ups for the surrounding voxel locations and interpolating the resulting x,y, and z component of the returned gradients. A gradient table can be generated by uniformly subdividing a sphere in a grid of longitudes and latitudes or by starting with a good approximation of a sphere and recursively subdividing this. The granularity of the subdivision depends on the size of the gradient table. Figure 7.15 illustrates the partitioning of the sphere using longitudes and latitudes for a table containing 512 entries. The advantage of such a scheme is that it does not require any

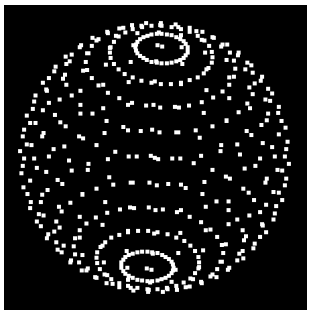


Figure 7.15: Quantization of the gradient space.

pre-processing. While downloading the volume data onto the VIZARD II board, all voxels are passed through the FPGA to the DIMMs. Thus, the corresponding index could be computed and stored on the fly.

With the described gradient look-up, it is feasible to reduce the storage requirements per voxel from 32 bits (voxel plus three gradient components) to 17 bits (voxel plus gradient index). Hence, up to 16 bit voxel values can be used leaving 7 bit for segmentation. The angular error due to the disretization of the gradients depends on the size of the used gradient table and the chosen quantization scheme. Table 7.4 shows the error made for different gradient table sizes using the subdivision scheme, as shown in Figure 7.15. Note that the maximum angular gradient error decreases continuously

Table entries	Average error [degree]	Maximum error [degree]
64	8.5	21.1
128	4.6	15.8
256	5.9	11.6
512	2.3	7.9
1024	2.9	5.5
2048	1.2	3.9

Table 7.4: Error of quantized gradients for the lobster dataset.

with increasing size of the gradient table whereas the average gradient error does not decrease continuously. This is due to the non uniform distribution of gradients within

the dataset as well as the gradients stored in the table. For a table size of 64, 256, 1024, etc. entries, an even number of latitudes is chosen excluding the gradients within the equatorial plane. A more uniform subdivision scheme than longitudes and latitudes would result in smaller errors. Nevertheless, the average angular error is already fairly small for a gradient table containing 512 entries.

Figure 7.16(a) shows the image of a lobster rendered with a gradient table containing 512 entries. Additionally, Figure 7.16(b) shows the magnified difference image

(a) (b)

Figure 7.16: Dataset of a lobster seen from below. (a) Rendered using a gradient look-up containing 512 entries. (b) Magnified difference image to correct gradient calculation.

compared with the original correct gradient calculation. The resulting difference is small and does not result in a noticeable loss of image quality.

Overall, the described pre-calculation and mapping of gradients onto a look-up table sacrifices gradient accuracy but still achieves high image quality for appropriate table sizes. However, there are also disadvantages of this method: First, there is no information about the length of the gradient since the index only points to a normalized gradient. This would additionally need to be stored together with the gradient index. Second, since trilinear interpolation is used, eight lookups would need to be performed every cycle. With respect to an implementation on a XILINX Virtex XCV1000 FPGA, this is not feasible due to the limited amount of BlockRAMs. However, when moving to the Virtex E-series (Virtex XCV1000E or higher), the number of available BlockRAMs increases and enables the integration of the presented indexed gradients scheme.

7.4.3 Other Potential Applications

The underlying VIZARD II board provides a high degree of flexibility, not only due to its high memory bandwidth but also by using DPSs and an FPGA chip with a large number of reconfigurable blocks. Even though hardware software co-design tools would be the ideal solution to remove the burden from the user to write and synthesize VHDL code, but a pipeline running at 100 MHz still requires strong hardware skills.

There is a number of potential applications for the VIZARD II systems within the

field of volume rendering and medical visualization as well as other fields. Compared to the presented ray casting implementation, the following designs are fairly small and can re-use quite a lot of the already designed components.

- On the fly gradient estimation
- Noise filtering
- Segmentation
- Integration of iso-surfaces

Real-time data acquisition devices demand on the fly rendering of volumetric data since data is generated within fractions of a second. Therefore, pre-computation of gradients is hardly feasible and the integration of on the fly gradient estimation mandatory. Since all data passes through the FPGA chip, the gradients could already be computed while downloading the volume data into the DIMMs. Alternatively, a gradient estimation scheme based on eight voxels could be implemented [Kni95].

Within the scope of volume rendering, pre-filtering of volume data is very important. Usually, a low-pass filter is moved across the volume data but the main bottle-neck of such algorithms is the memory access. On the VIZARD II board, this could be done several times per second due to the highly interleaved memory organization.

Segmentation algorithms are frequently used in medical applications. Usually, seed point based filling or flooding algorithms are used, possibly combined with knowledge about shape and/or size of the object to be segmented. Using one or more seed points, surrounding voxels are incrementally flooded as long as they fulfill a certain criteria based upon the density value. Again, the main bottle-neck of these algorithms is the memory bandwidth which could be provided on the VIZARD II system.

Finally, volume rendering using sampling based methods such as ray casting, splatting, or texture mapping, are not capable of finding a precise iso-surface. The combination of a rasterizer and polygon graphics is promising but due to the nature of modern graphics systems, accessing the content of the framebuffer several times per second (color and depth) is not feasible. Easier and simpler to realistic is the computation of precise iso-surfaces using ray casting which could be mixed with volumetric rendering.

7.5 Summary

VIZARD II is a special purpose hardware accelerator for true ray casting and high image quality. With its highly optimized memory interface, it is capable of generating up to 80 million trilinearly interpolated samples and gradients per second using one processing pipeline only. Each sample is Phong shaded using per sample material properties and composed at high precision to ensure highest image quality. By integrating the proposed space leaping approach into the design, 15-20 frames per second can be achieved for medical datasets of 256^3 voxels.

The strong advantage of using programmable and reconfigurable components, is the short design cycle which enables to change a design within hours. Furthermore, besides ray casting, other algorithms are feasible on the same board and hence, are superior to ASIC solutions. With respect to volume rendering, ASICs do not provide a clear advantage because volume rendering algorithms are memory bandwidth limited and FPGA designs running at full memory speed are feasible. However, this will

change once volume rendering becomes a larger market such that the costs of designing and fabricating an ASIC will amortize.

The future of special purpose hardware for volume rendering will mainly depend on the future developments in the field of the polygon graphics hardware. 3D texture mapping offers similar functionality (trilinear filtering) but with less quality. Furthermore, several important components such as shading and support for segmentation are missing. Generally, special purpose hardware such as VIZARD II will always be a step ahead since it is able to deliver functionality which is not supported in main stream (graphics) hardware. Special purpose hardware frequently serves as stimuli for what is put into the next generation main stream hardware.

Appendix A

A Cross-Platform Rendering Environment

For quite a number of years, working in the field of computer graphics has been a task that mainly involved expensive SGI machines, a low-level graphics API such as IrisGL, and maybe even a high level graphics API such as OpenInventor. During the last few years, PC-class machines invaded the market and as a result of an enlarged market as well as due to rapid advances in memory and processing technology, workstation graphics has almost been replaced by PC systems with PCI and AGP based graphics cards.

From a programmer's perspective, the wide availability of graphics cards has certainly a lot of advantages, but the disadvantage is that one has to deal with a larger number of common operating systems (Windows derivatives, IRIX, HP-UX, Solaris, Linux, etc.), as well as with different graphics APIs (OpenGL and Direct 3D). Once selecting one platform and API, one is limited to this avenue which might be a dead end possibly within a short period of time.

While GLUT is an OpenGL package available on most operating systems, it is limited with respect to modularity and GUI development. Furthermore, it does not offer navigation models. Within this chapter, a rendering environment providing different navigation models is presented. To obtain modularity and cross platform portability, the Qt graphics user interface toolkit — providing easily extendable GUI components — is used for the GUI development.

A.1 The Qt library

Qt[1] is a multi-platform graphics user interface toolkit based on C++. Due to its availability on all major operating systems, it enables programmers to efficiently build GUI applications that run on different platforms. As any other GUI library, it consists of a large variety of different GUI elements which can be used to build applications. One of the nice properties is that the GUI elements provide the typical look and feel for each platform. Furthermore, Qt comes with a superb documentation and open source packages — such as *doxygen* — allow to generate such documentation for own-developed classes.

The main difference to other GUI libraries is the handling of events within Qt. Generally, this process is called *signal/slot mechanism* which provides an excellent framework for component-based programming: Any object is able to *emit* any *signal*. On the other hand, objects can provide *slots* which are able to receive signals depending on whether they are *connected* to the emitting object. Hence, emitted signals will only have an impact in case they are connected to a slot of another object.

Despite all comfort and richness of Qt, the signal/slot-mechanism comes at a certain price. It requires a pre-compilation step performed by *moc* (meta object compiler) needed to compile signals, slots, emit()-calls, and connects into common class member methods which are executed sequentially. However, for the presented purposes, this has not been a limiting factor.

A.2 QGLViewer

Navigating through three dimensional scenes is not only a problem of the frequently missing collision detection to possibly prevent object penetration, it is also not trivial to handle six degrees of freedom with a simple mouse or the keyboard. Commonly, almost every programmer starts implementing its own camera model interpreting the mouse events as rotations, translations, and others. Besides the fact that this is a good exercise to understand the general principles of viewing in computer graphics, it is tedious to implement all kinds of navigation models.

The goal was set to develop a library consisting of a set of classes which provide an user interface, handling the input of mouse and keyboard as camera controls. By doing this, the viewing process controlled by mouse and keyboard events is strictly separated from the actual rendering. Hence, any renderer (render object) can be connected to such a GUI without the need of re-implementing the navigation model. Thus, one can start writing OpenGL code right away and simply use one of the navigation models. Another advantage is that the navigation models can simply be exchanged without the need of implementing them all. Therefore, the library provides a high degree of extensibility since more navigation models can be added and provided to other users.

The Qt library provides an OpenGL widget (QGLWidget) which is derived from the base class of all Qt widgets (QWidget). Additionally, it has two members which control the display format of the OpenGL context (QGLFormat) as well as the context itself (QGLFormat), as illustrated in Figure A.1.

Whenever the QGLWidget is resized or needs to be redrawn, Qt will call the corresponding methods of QGLWidget. To encapsulate this behavior and be able to connect this to other objects using signals, the class QSignalWidget is derived overloading these methods and emitting corresponding signals. To enable a set of viewers with different

[1] Qt can be downloaded from http://www.troll.no

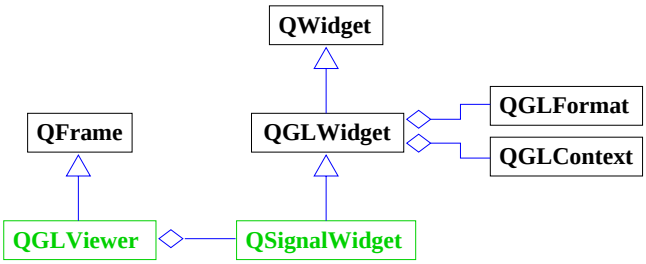


Figure A.1: Structural class diagram of the QGLViewer: QSignalWidget and QGLViewer (green) are needed in addition to the Qt classes (black).

navigation behavior, a general interface for all viewers has been designed: the abstract base-class QGLViewer. This class includes the common interface as well as one object of type QSignalWidget which is the OpenGL rendering area. Hence, any generalization of QGLViewer can provide its own individual GUI graphics user interface including different buttons as well as the actual drawing area (QSignalWidget). Signals emitted by the object QSignalWidget are connected to the QGLViewer and not emitted to the "outside" world. The QGLViewer first performs a few initializations steps and emits a few generic signals which are needed to toggle the actual rendering. Thus, the viewing control (navigation) is entirely separated from the rendering.

As mentioned earlier, different viewers can be implemented deriving them from QGLViewer. Figure A.2 shows an UML [BRJ99] structural diagram of the generalizations of the QGLViewer.

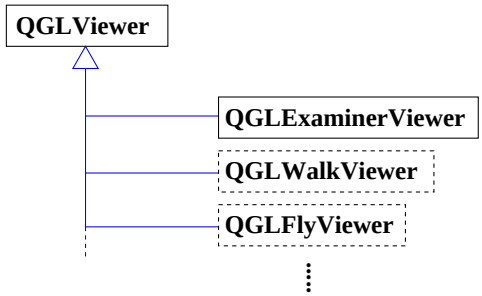


Figure A.2: Generalizations of QGLViewer: QGLViewer is the abstract base-class. Dashed boxes indicate viewers which are not yet implemented.

Generally, there are two types of signals emitted by the QGLViewer: one type is affecting the actual rendering and the other type allows the picking of objects (see Figure A.3).

Signals that affect the rendering are init(), resize(), and redraw(). The latter is emitted whenever a redraw of the scene is required, due to a change of the camera or due to moved windows. For picking there is one signal which emits the mouse event (pressed, moved, released) containing the coordinates within the rendering area. In case the connected renderer supports picking or manipulating the scene database, the signal can be used to perform certain actions to modify the elements of the scene.

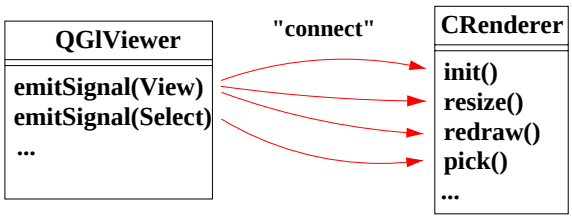


Figure A.3: Connecting a renderer to a viewer. There are signals for viewing as well as for picking.

One example application that provides different viewers is SGI's OpenInventor. It provides four different navigation models called *Examiner*, *Fly*, *Plane*, and *Walk*. This concept has been borrowed for the implementation of the QGLExaminerViewer, a generalization of the base-class (QGLViewer) providing examination functionality. The GUI of the QGLExaminerViewer is shown in Figure A.4.

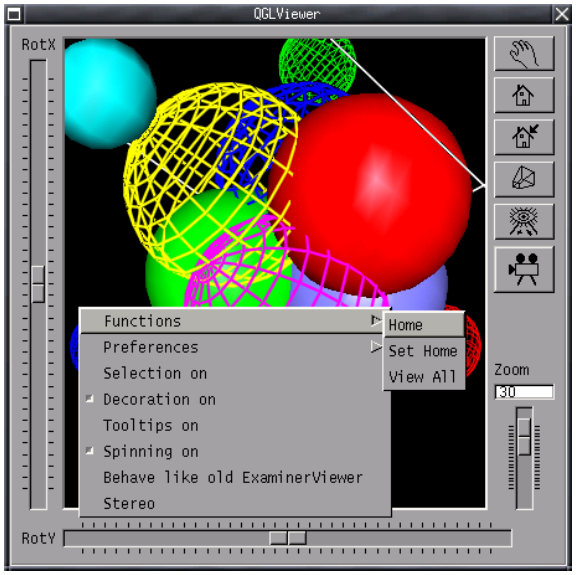


Figure A.4: The QGLViewer GUI provides six degree navigation, parallel and perspective viewing, and many other features such as drag and drop of cameras across different applications and picking mechanisms.

The QGLViewer library is available from *http://www.qglviewer.de* and is licensed under the terms of the GPL.

A.3 VolRen

VolRen is a volume rendering framework integrating different volume rendering techniques in one application. Rather than developing the fastest and most efficient implementations, the goal was to enable the integration of different volume rendering algorithms. Thus, the impact of various gradient estimation techniques, filters, discretization, etc. onto the image quality can be examined and compared. Most of the images presented throughout the chapters of this dissertation have been generated with VolRen or renderers being based on the QGLViewer package.

Figure A.5 shows a structural diagram of the available rendering classes. There is an abstract base-class *CVolRen*, as well as its currently available four derived implementations (generalizations). Furthermore, different ray casters (generalizations) are

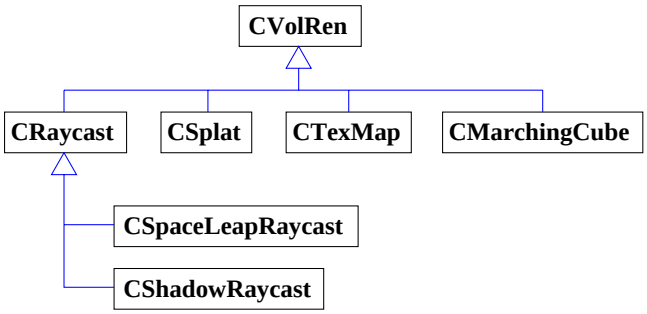


Figure A.5: Structural class diagram of VolRen: CVolRen is the abstract base-class of the volume rendering applications and currently four different generalizations exist as well as two further generalizations of the ray caster.

derived from the class *CRaycast* to enable space leaping or direct illumination in addition to local illumination. Each of these classes accepts a set of parameters to enable or disable certain rendering options.

The volume rendering application handles all the above described algorithms. Similar to the previously described QGLViewer, it is based on Qt. The application itself is split into the volume rendering application *QVolRenApp* and the GUI, handled in *QVolRenGUI*. This follows the paradigm of separating the document (data) from the view of the document. In addition to this central application and its two classes, there is a set of further classes and modules which allow further interaction. These are the volume class (*CVolume*), look-up tables for classification (*CLut*), further renderers to enable slicing planes (*QSlicer*), an editor for the transfer function (*QClassification*), a histogram (*QHistogram*), and a geometric library containing points, vectors, matrices, etc. Some of the user interface components are shown in Figure A.6, e.g. the QGLViewer is integrated as a component of the VolRen GUI. Furthermore, it shows the Marching Cube rendering of a human foot. Below are two renderings of the same foot using ray casting and two different classifications. One exhibits the bone only and the other one bone and tissue. In the lower left, the classification GUI is depicted. Finally, the display options for the OpenGL window are displayed on the top left.

The overall structural relationship diagram of the core classes of VolRen is illustrated in Figure A.7. The core element is the QVolRenApp which handles all the signals of QVolRenGUI. It administrates the different resources as volume data, classification

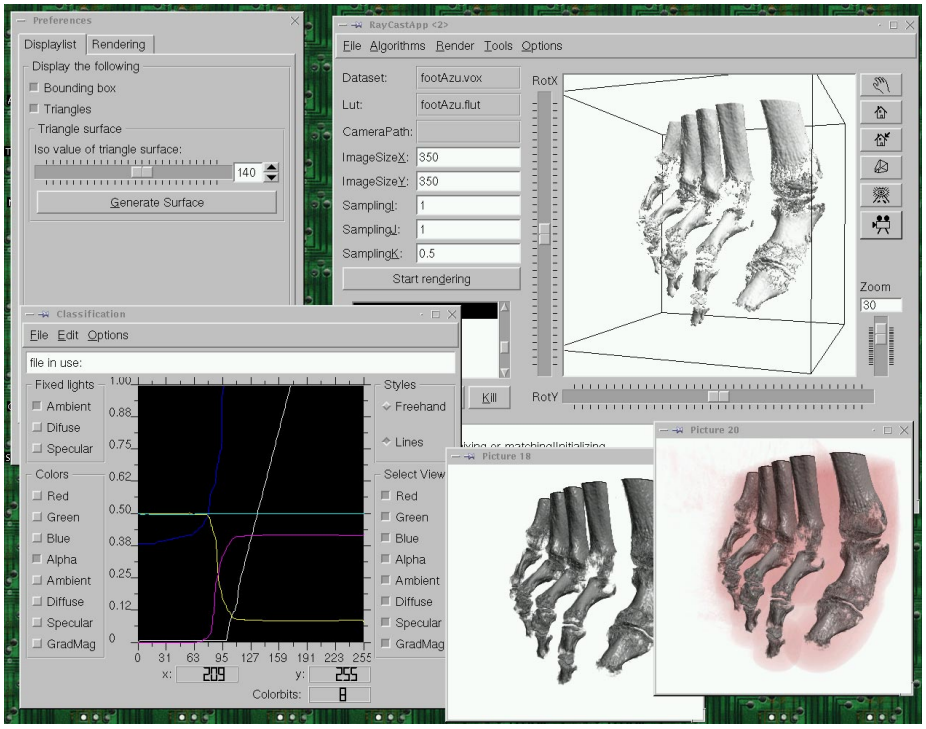


Figure A.6: Graphical user interface of VolRen: It consists of a main window which contains the QGLViewer in which the iso-surface of a CT scan of a human foot can be seen. Below are two views generated using ray casting. To the left is the GUI for Marching Cube preferences as well as the classification GUI.

table, list of light sources, and other GUI elements. For rendering, the required data is provided to the rendering object which can be any of the ones depicted in Figure A.5.

A.4 Summary

The early decision[2] of using OpenGL as well as the Qt library for the development environment turned out to be quite good. Within the last two years, the Qt library became more and more popular across all platforms and is frequently used within industry as well as academia. The KDE Linux desktop — fully based on the Qt library — might serve as an example. Also the selection of OpenGL graphics API and not Direct 3D was satisfactory since it is still around and very popular. Since the foundation of the OpenGL ARB — an organization of industrial partners to further extend and develop OpenGL —, OpenGL version 1.2 has been released and many extensions have been added.

QGLViewer provides a set of classes for camera control, as well as comfortable user interface that allows controlling the viewing parameters via mouse and buttons. QGLViewer is object oriented and programmed in C++ such that the programmer can

[2]Beginning 1998.

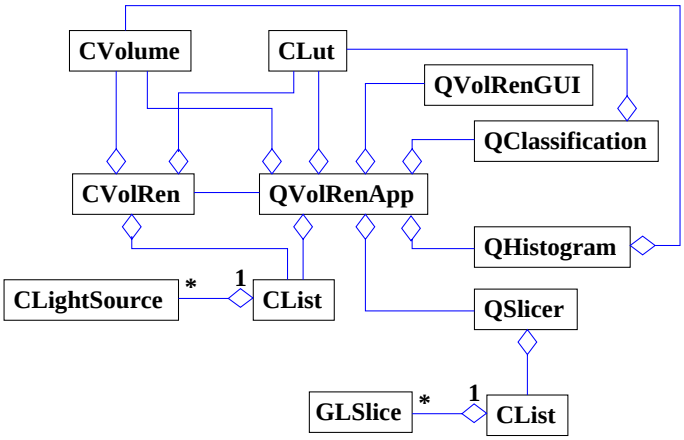


Figure A.7: Class diagram of the VolRen application: Aggregations are one to one if not depicted differently.

simply instantiate such a viewer and connect it to the own render routines. Thus, this saves time when implementing OpenGL based rendering algorithms and allows to exchange navigation models developed by other people.

Based on the QGLViewer, a modular component based volume rendering framework VolRen has been developed. The design of VolRen has rather been driven by modularity than efficiency (run-time), such that different rendering approaches and optimization techniques can easily be integrated and evaluated. Most of the images presented throughout this dissertation as well as comparisons of different filters, shading techniques, compositing schemes, etc. have been rendered using VolRen.

Releasing VolRen under the terms of GPL is currently in progress.

Bibliography

[ABJ+98] H. R. Arabnia, D. Bartz, A. Jacobsen, M. Meißner, M. Misra, H. Shen, and G. K. Thiruvathukal (eds). *Parallel and Distributed Processing Techniques and Applications (PDPTA98)*. Number ISBN 1-892512-6-8 in 3. CSREA Press, New York, 1998.

[AGS95] M. B. Amin, A. Grama, and V. Singh. Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In *Proc. of the IEEE Symposium on Parallel Rendering*, pages 7–14, Atlanta, GA, October 1995.

[AHH+94] 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 *Proc. of IEEE Visualization*, pages 31–38, Washington, DC, USA, October 1994. IEEE Computer Society Press.

[Ake93] K. Akeley. RealityEngine Graphics. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 109–116, August 1993.

[AMSW97] B. Anderson, R. MacAulay, A. Stewart, and T. Whitted. Accommodating Memory Latency In A Low-cost Rasterizer. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 97–101, Los Angeles, CA, USA, July 1997.

[ARB90] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[Ben95] M. Bentum. *Interactive Visualization of Volume Date*. PhD thesis, University of Twente, Enschede, The Netherlands, June 1995.

[BHM99] Vettermann B, J. Hesser, and R. Männer. Solving hte Hazard Problem for Algorithmically Optimized Real-Time Volume Rendering. In *Proc. of 1st Workshop on Volume Graphics*, pages 171–184, March 1999.

[BK97] I. Bitter and A. Kaufman. A Ray-Slice-Sweep Volume Rendering Engine. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 121–130, Los Angeles, CA, USA, July 1997.

[BKX90] R. Bakalash, A. Kaufman, and Z.-Y. Xu. Cube-3: Building a full scale VLSI-based volume visualization system. In *Proc. of the 5th EG Workshop on Graphics Hardware*, Lausanne, Switzerland, September 1990.

[BKX92] R. Bakalash, A. Kaufman, and Z.-Y. Xu. Cube-3: Building a full scale VLSI-based volume visualization system. In R. L. Grimsdale and A. Kaufman, editors, *Advances in Graphics Hardware V*, pages 109–116. Springer-Verlag, 1992.

[Bli82] J. F. Blinn. Light refelction functions for simulation of clouds and dusty surfaces. *Computer Graphics*, 16(3):21–29, July 1982.

[Bli98] J. Blinn. *Jim Blinn's Corner: Dirty Pixels*. Number ISBN 1-55860-455-3. Morgan Kaufman, San Francisco, 1998.

[BM99] D. Bartz and M. Meißner. Voxels versus Polygons: A Comparative Approach for Volume Graphics. In *Proc. of 1st Workshop on Volume Graphics*, pages 33–48, March 1999.

[BMH98] D. Bartz, M. Meißner, and T. Hüttner. Extending Graphics Hardware for Occlusion Queries in OpenGL. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 97–104, Lisboa, Portugal, August 1998.

[BMH99] D. Bartz, M. Meißner, and T. Hüttner. Opengl-assisted occlusion culling of large polygonal models. *Computers and Graphics - Special Issue on Visibility - Techniques and Applications*, 23(5):667–675, 1999.

[Bre96] R. Brechner. Interactive walkthroughs of large geometric databases. In *ACM SIGGRAPH course notes*, August 1996.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Number ISBN 0-201-57168-4. Addison-Wesley, Reading, MA, 1999.

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Workshop on Volume Visualization*, pages 91–98, Washington, DC, USA, October 1994.

[CK95] D. Cohen and A. Kaufman. A 3D skewing and de-skewing scheme for conflict-free access to rays in volume rendering. *IEEE Transactions on Computers*, 44(5):707–710, May 1995.

[CM93] B. Corrie and P. Mackerras. Parallel volume rendering and data coherence. In *Proc. of the Parallel Rendering Symposium*, pages 23–26, San Jose, October 1993.

[CN93] T. J. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Mapping Hardware. Technical Report TR93-027, Department of Computer Science at the University of North Carolina, Chapel Hill, 1993.

[dBGHM96] M. de Boer, A. Gröpl, J. Hesser, and R. Männer. Latency- and Hazard-Free Volume Memory Architecture for Direct Volume Rendering. In *Proc. of the 11th EG Workshop on Graphics Hardware*, pages 109–119, Poitiers, France, August 1996.

[dBHG+96] M. de Boer, J. Hesser, A. Gröpl, T. Guenther, C. Reinhart, and R. Männer. Evaluation of a real-time direct volume rendering system. In *Proc. of the 11th EG Workshop on Graphics Hardware*, pages 121–131, Poitiers, France, August 1996.

[DH92] 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, USA, October 1992.

[DKC+98] 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.

[DM99] M. C. Doggett and M. Meißner. A memory addressing and access design for real time volume rendering. In *Proc. of IEEE Circuits and Systems*, pages 7–14, April 1999.

[DM00] M. C. Doggett and M. Meißner. Ray casting: Antialiasing using multiresolution datasets. In *Informatik 2000: Graphiktag*, Berlin, Germany, September 2000.

[DMK99] 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.

[Dog95] M. C. Doggett. An Array based Design for Real-Time Volume Rendering. In *Proc. of the 10th EG Workshop on Graphics Hardware*, pages 93–101, August 1995.

[Dog00] Mc. C. Doggett. A ray queueing and sorting design for real time ray casting. In *International Symposium on Circuits and Systems*. IEEE, May 2000.

[EMPG97] J. Eyles, S. Molnar, J. Poulton, and T. Greer. PixelFlow: The Realization. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 23–30, Los Angeles, CA, USA, July 1997.

[FP81] H. Fuchs and J. Poulton. PIXEL-PLANES A VLSI-Oriented Design for a Raster Graphics Engine. *VLSI Design (formerly Lambda - The Magazine of VLSI Design)*, pages 20–28, 1981.

[GBW90] B. Garlick, D. Baum, and J. Winget. Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations. In *ACM SIGGRAPH course notes: Parallel Algorithms and Architectures for 3D Image Generation*, August 1990.

[Geo92] P. Georgiades. *Graphics Gems III*, chapter Signed Distance from a Point to a Plane, pages 223–224. Academic Press, Inc., New York, 1992.

[GK96] 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.

[GKM93] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 231–238, August 1993.

[Gla90] A. S. Glassner, editor. *Graphics Gems*. Number ISBN 0122861655. Academic Press, Inc., New York, 1990.

[GPR+94] T. Guenther, C. Poliwoda, C. Reinhard, J. Hesser, R. Männer, H.-P. Meinzer, and H.-J. Baur. VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine. In *Proc. of the 9th EG Workshop on Graphics Hardware*, pages 103–108, Oslo, Norway, September 1994.

[Gre95] N. Greene. *Hierarchical Rendering of Complex Environments*. PhD thesis, Computer and Information Science, University of California, Santa Cruz, 1995.

[Hei99] W. Heidrich. *High-quality Shading and Lighting for Hardware-Accelerated Rendering*. PhD thesis, University of Erlangen-Nürnberg, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford, CA 94305–4055, 1999. ISBN 3-89675-624-9.

[Hir99] J. Hirche. VHDL-Design und Synthese eines Phong Shaders für die Volumenvisualisierung. Master's thesis, University of Tübingen, 1999.

[HMB98] T. Hüttner, M. Meißner, and D. Bartz. Opengl-assisted visibility queries of large polygonal models. Technical Report ISSN 0946-3852, University of Tübingen, Tübingen, 1998.

[HP97] Hewlett-Packard. GL_HP_Occlusion_Test. *Specification document, available from http://www.opengl.org/*, 1997.

[HPP+96] 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.

[Hsu93] W. M. Hsu. Segmented ray-casting for data parallel volume rendering. In *Proc. of the Parallel Rendering Symposium*, pages 7–14, San Jose, CA, October 1993.

[IEP98] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a Texture Cache Architecture. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 133–142, Lisboa, Portugal, August 1998.

[Inc99] Silicon Graphics Inc. Silicon Graphics 320, Visual Workstation. *Specifiaction document, available from http://visual.sgi.com/products/320/index.html*, 1999.

[KH84] J. T. Kajiya and T. Von Herzen. Ray Tracing Volume Densities. *Computer Graphics*, 18(3):165–173, July 1984.

[Kir92] D. Kirk, editor. *Graphics Gems III*. Number ISBN 0124096700. Academic Press, Inc., New York, 1992.

[Kir98] D. B. Kirk. Unsolved Problems and Opportunities for High-quality, High-performance 3D Graphics on a PC Platform. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 11–13, Lisboa, Portugal, August 1998.

[KK98] K. Kreeger and A. Kaufman. PAVLOV: A Programmable Architecture for Volume Processing. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 77–86, Lisboa, Portugal, August 1998.

[KM96] U. Kanus and M. Meißner. Cube-4 – A Volume Rendering Architecture for Real-Time Visualization of High-Resolution Datasets. Master's thesis, State University of New York at Stony Brook, 1996.

[KMS+96] U. Kanus, M. Meißner, W. Straßer, H. Pfister, A. Kaufman, R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider. Cube-4 Implementations on the Teramac Custom Computing Machine. In *Proc. of the 11th EG Workshop on Graphics Hardware*, pages 133–143, Poitiers, France, August 1996.

[Kni93] G. Knittel. Verve - voxel engine for real-time visualization and examination. *Computer Graphcis Forum*, pages 37–48, sep 1993.

[Kni94] G. Knittel. A Scalable Architecture for Volume Rendering. In *Proc. of the 9th EG Workshop on Graphics Hardware*, Oslo, Norway, September 1994.

[Kni95] G. Knittel. A PCI-based Volume Rendering Accelerator. In *Proc. of the 10th EG Workshop on Graphics Hardware*, Maastricht, NL, August 1995.

[Kru91] W. Krueger. The application of transport theory to the visualization of 3d scalar fields. *Computers in Physics 5*, pages 397–406, 1991.

[KS97] G. Knittel and W. Straßer. Vizard - visualization accelerator for realtime display. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 139–146, Los Angeles, CA, USA, July 1997.

[Kug96] A. Kugler. The setup for triangle rasterization. In *Proc. of the 11th EG Workshop on Graphics Hardware*, Poitiers, France, August 1996.

[L. 97] L. Hong and S. Muraki and A. Kaufman and D. Bartz and T. He. Virtual Voyage: Interactive Navigation in the Human Colon. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 27–34, Los Angeles, USA, July 1997.

[Lac95] P. Lacroute. Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization. In *Proc. of the IEEE Symposium on Parallel Rendering*, pages 15–22, Atlanta, GA, October 1995.

[LC87] W. E. Lorensen and H. E. Cline. Marching–Cubes: A High Resolution 3D Surface Construction Algorithm. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 163–169, 1987.

[LCN98] B. Lichtenbelt, R. Crane, and S. Naqvi. *Introduction to volume rendering*. Hewlett-Packard Professional Books, Prentice-Hall, Los Angeles, USA, 1998.

[Lef93] W. Lefer. An efficient parallel ray tracing scheme for distributed memory parallel computers. In *Proc. of the Parallel Rendering Symposium*, pages 77–80, San Jose, CA, October 1993.

[Lev88] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(5):29–37, May 1988.

[Lev89] 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.

[Lev90] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Computer Graphics*, 9(3):245–261, July 1990.

[Lic95] J. Lichtermann. Design of a Fast Voxel Processor for Parallel Volume Visualization. In *Proc. of the 10th EG Workshop on Graphics Hardware*, pages 83–92, Maastricht, The Netherlands, August 1995.

[Lic97] 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.

[LL94] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp factorization of the Viewing Transform. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 451–457, July 1994.

[LWMT97a] P. P. Li, S. Whitman, R. Mendoza, and J. Tsiao. Parallel processing of the shear-warp factorization with the binary-swap method on a distributed-memory multiprocessor system. In *Proc. of the Parallel Rendering Symposium*, pages 87–94, Phoenix, AZ, October 1997.

[LWMT97b] P. P. Li, S. Whitman, R. Mendoza, and J. Tsiao. Parvox - a parallel splatting volume rendering system for distributed visualization. In *Proc. of the Parallel Rendering Symposium*, pages 7–14, Phoenix, AZ, October 1997.

[Max95] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[MBDM97] J. Montrym, D. Baum, D. Dignam, and C. Migdal. Infinite Reality: A Real-Time Graphics System. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 293–303, Los Angeles, CA, USA, July 1997.

[MBH+99] M. Meißner, D. Bartz, T. Hüttner, G. Müller, and J. Einighammer. Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models. Technical Report TR WSI-99-13, University of Tübingen, 1999.

[MC98] 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.

[ME98] M. Meißner and B. Eberhard. The Art of Knitted Fabrics, Realistic and Physically Based Modelling of Knitted Patterns. In *Proc. Eurographics*, pages 355–362, Lisboa, Portugal, August 1998.

[Mei98] M. Meißner. A Parallel Adaptive Volume Rendering System. In *Proc. PARA*, Umeå, Sweden, 1998.

[MEP92] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 231–240, Los Angeles, CA, USA, July 1992.

[MHB+00] 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.

[MHBW98] M. Meißner, T. Hüttner, W. Blochinger, and A. Weber. Parallel Direct Volume Rendering on PC networks. In *Proc. PDPTA*, volume 3, pages 1321–1328, Las Vegas, USA, 1998.

[MHS99] 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.

[MKS98] 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.

[ML94] S. Marschner and R. Lobb. An evaluation of reconstruction filters for volume rendering. In *Proc. of IEEE Visualization*, pages 100–107, Washington, D.C., USA, October 1994. IEEE Computer Society Press.

[MM00] J. McCormack and R. McNamara. Tiled Polygone Traversal Using Half-Plane Edge Functions. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 15–21, Interlaken, Switzerland, August 2000.

[MMC99] 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.

[MMG+98a] J. McCormack, R. McNamara, C. Gianos, L. Seiler, N.P. Jouppi, and K. Correll. Neon: A Single-Chip 3D Workstation Graphics Accelerator. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 123–132, Lisboa, Portugal, August 1998.

[MMG+98b] J. McCormack, R. McNamara, C. Gianos, L. Seiler, N.P. Jouppi, K. Correll, T. Dutton, and J. Zurawski. Neon: A (Big) (Fast) Single-Chip 3D Workstation Graphics Accelerator. Technical Report 98.1, Compaq Western Research Laboratories, 1998.

[MMMY97] 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.

[MMSE91] H.-P. Meinzer, K. Meetz, D. Scheppelmann, and U. Engelmann. The Heidelberg Ray Tracing Model. *IEEE Computer Graphics & Applications*, pages 34–43, November 1991.

[MPHK93] K. Ma, J. Painter, C. Hansen, and M. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In *Proc. of the Parallel Rendering Symposium*, pages 15–22, San Jose, CA, October 1993.

[MSHC99] 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.

[Neu93] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Proc. of the Parallel Rendering Symposium*, pages 97–104, San Jose, CA, October 1993.

[NL92] J. Nieh and M. Levoy. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In A. Kaufman and W. L. Lorensen, editors, *Workshop on Volume Visualization*, pages 17–24, Boston, MA, USA, October 1992.

[nVi99] nVidia. GeForce. *http://www.nvidia.com/Geforce256.nsf*, 1999.

[OPL+97] 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 *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 131–138, July 1997.

[PD84] T. Porter and T. Duff. Compositing digital images. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 253–259, 1984.

[PHK+99] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro Real-Time Ray-Casting System. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 251–260, Los Angeles, CA, USA, 1999.

[PK96] H. Pfister and A. Kaufman. CUBE-4 - A Scalable Architecture for Real-Time Volume Rendering. In *Symposium on Volume Visualization*, San Francisco, CA, USA, October 1996.

[PSL+98] 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.

[PTT97] M. E. Palmer, S. Taylor, and B. Totty. Exploiting deep parallel memory hierarchies for ray casting volume rendering. In *Proc. of the Parallel Rendering Symposium*, pages 15–22, Phoenix, AZ, October 1997.

[RS99] H. Ray and D. Silver. A Memory Efficient Architecture for Real-Time Parallel and Perspective Direct Volume Rendering. Technical Report CAIP-TR-237, Department of Computer Aids for Industrial Productivity, Rutgers University, 1999.

[Sab88] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. In *Computer Graphics*, pages 59–64, August 1988.

[SDWE98] O. Sommer, A. Dietz, R. Westermann, and T. Ertl. Tivor: An Interactive Visualization and Navigation Tool for Medical Volume Data. In *The Sixth International Conference in Central Europe on Computer Graphics amd Visualization*, February 1998.

[Sev99] K. Severson. VISUALIZE Workstation Graphics for NT. *Hewlett-Packard whitepaper*, March 1999.

[SGI99] SGI. Silicon Graphics Visual Workstation OpenGL Programming. Technical report, SGI, 1999.

[SOG98] N. Scott, D. Olsen, and E. Gannett. An Overview of the VISUALIZE fx Graphics Accelerator Hardware. *The Hewlett-Packard Journal*, pages 28–34, May 1998.

[SS92] P. Schröder and G. Stoll. Data parallel volume rendering as line drawing. In *Workshop on Volume Visualization*, pages 25–31, Boston, MA, USA, October 1992.

[THB+90] 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.

[TS91] S. Teller and C.H. Sequin. Visibility Pre-processing for Interactive Walkthroughs. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 61–69, August 1991.

[TT84] H. K. Tuy and L. T. Tuy. Direct 2-D display of 3D objects. *IEEE Computer Graphics & Applications*, 4(10):29–33, November 1984.

[UK88] C. Upson and M. Keeler. V-BUFFER: Visible Volume Rendering. *Computer Graphics*, 22(4):59–64, August 1988.

[VF94] D. Voorhies and J. Foran. State of the art in data visualization. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 163–166, July 1994.

[VFR92] G. Vézina, P. Fletcher, and P. Robertson. Volume rendering on the MasPar MP-1. In *Workshop on Volume Visualization*, pages 3–8, Boston, MA, USA, October 1992.

[vSSB95] 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.

[WE98] 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.

[Wes90] L. Westover. Footprint Evaluation for Volume Rendering. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 367–376, August 1990.

[WEWL99] M. D. Waller, J. P. Ewins, M. White, and P. F. Lister. Efficient primitive traversal using adaptive linear edge function algortithms. *Computers & Graphics*, 23:365–375, 1999.

[WGW94] O. Wilson, A. Van Gelder, and J. Wilhelms. Direct Volume Rendering via 3D Textures. Technical Report UCSC-CRL-9419, University of California, Santa Cruz, 1994.

[Wit98] C. M. Wittenbrink. Extensions to permutation warping for parallel volume rendering. Hpl-97-116(r.1), Hewlett-Packard Computer Systems Laboratory, Palo Alto, CA, April 1998.

[WMG98] 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.

[Woo90] A. Woo. *Graphics Gems*, chapter Fast Ray-Box Intersection, pages 395–396. Academic Press, Inc., New York, 1990.

[WS93] C. M. Wittenbrink and A. K. Somani. Permutation warping for data parallel volume renderin. In *Proc. of the Parallel Rendering Symposium*, pages 57–60, San Jose, CA, 1993.

[XS99] F. Xie and M. Shantz. Adaptive Hierarchical Visibility in a Tiled Architecture. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 75–84, Los Angeles, CA, USA, August 1999.

[YK92] R. Yagel and A. Kaufman. Template-based volume viewing. 11(3):153–167, September 1992.

[YS93] 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.

[ZMHH97] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 77–88, Los Angeles, CA, USA, July 1997.

Lebens- und Bildungsgang

11. Januar 1970	geboren in Tübingen
1976 - 1989	Gesamtschule WHO, Tübingen Abschluß Abitur
1986 - 1988	Entwicklung und Verkauf von Computerspielen für den 650x Microprozessor des Commodore C16 (in Assembler)
1989 - 1996	Studium der Informatik an der Eberhard-Karls-Universität Tübingen
1994	Werkstudent im IBM Forschungs- und Entwicklungslabor, Böblingen
1995/1996	Diplomarbeit an der SUNY SB, Stony Brook, NY, dabei auf Einladung zweimaliger Aufenthalt im Hewlett-Packard Research Laboratory, Palo Alto, CA
1996	Nebentätigkeit bei der Firma DD&T als Hardware Designer
1996 - 1997	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Automatisierungstechnik an der Fachhochschule Reutlingen
seit 1997	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Graphisch Interaktive Systeme am Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universität Tübingen (Prof. Straßer)