A Flexible Framework for Hardware-Accelerated High-Quality Volume Rendering

Christoph Berger* Markus Hadwiger† Helwig Hauser‡

VRVis Research Center for Virtual Reality and Visualization
Vienna / Austria
http://www.VRVis.at/vis/

Abstract

Because of an enormous development of graphics hardware and the invention of new rendering algorithms in the past it is now possible to perform interactive hardware-accelerated high quality volume rendering and iso-surface reconstruction on low cost standard PC platforms.
In this paper we introduce a framework that integrates several different rendering techniques which significantly improve both performance and image quality of standard texture based rendering approaches. Furthermore the most common graphics adapters are supported without additional need for setup as well as several vendor-dependent OpenGL-extensions like pixel-, texture- and fragment-shaders. Therefore it is easy to compare the varying results of different rendering algorithms on diverse graphics adapters with respect to quality and performance.

Keywords: volume rendering, volume visualization, graphics hardware, isosurface-reconstruction, OpenGL

1 Introduction

For visualization of volumetric data direct volume rendering [7, 8] is an important technique to get insight into data. The key advantage of direct volume rendering over surface rendering approaches is the potential to show the structure of the value distribution throughout the volume. Due to the fact that each volume sample contribution to the final image is included, it is a challenge to convey that value distribution simple and precisely.
Because of an enormous development of low-cost 3D hardware accelerators in the last few years (driven by the computer games industry) the features supported by consumer-oriented graphics boards like the NVIDIA GeForce family [17] or the ATI Radeon family [15] are also very interesting for professional graphics developers. Especially NVIDIA's pixel- and texture shader and ATI's fragment shader are powerful extensions to standard 2D and 3D texture mapping capabilities. Therefore high-performance and high-quality volume rendering at very low costs is now possible. Several approaches of hardware-accelerated direct volume rendering have been introduced to improve rendering speed and accuracy of visualization algorithms. Thus it is possible to provide interactive volume rendering on standard PC platforms and not only on special-purpose hardware.
In this paper we present an application that includes several different visualization algorithms for direct volume rendering as well as direct iso-surface rendering (no polygonal representation has to be extracted, instead special features of current rendering hardware are used). The major objective of the prototype is to provide comparison possibilities for several hardware accelerated volume visualizations with respect to performance and quality. On startup of the software, the installed graphics adapter is detected automatically and regarding to the supported OpenGL-features the user can switch between the available rendering modes supported by the current graphics hardware. The full functionality includes pre- and post classification modes as well as pre-integrated classification modes (more details on classification will follow in Section 3.2 and 3.3). All algorithms are implemented exploiting both 2D or 3D texture mapping as well as optional diffuse and specular lighting. Additionally we have adopted the high-quality reconstruction technique based on PC-hardware, introduced by Hadwiger et al. [5], to enhance the rendering quality through high-quality filtering.
The major challenge is combining diverse approaches in one simple understandable framework that supports several graphics adapters which have to be programmed completely different and still provide portability for implementation of new algorithms and support of new hardware-features.
The paper is structured as follows. Section 2 gives a short overview of work that has been done on volume rendering and especially on hardware-accelerated methods. Section 3 is then going to introduce the main topic, namely volume rendering in hardware (texture based), providing a brief overview of the major approaches and describing different classification techniques. In Section 4 we will then discuss the implementation in detail and problems that have to be overcome if supporting graphics adapters from different

*boerga@gmx.net
†hadwiger@vrvis.at
‡hauser@vrvis.at

vendors. This section will also cover some performance issues and other application specific problems we encountered during prototype implementation. Section 5 summarizes what we have presented and additionally some future work that we are planning at the moment will be briefly mentioned.

2 Related Work

For scalar volume data several visualization approaches have been introduced. Usually they can be classified into indirect volume rendering, such as iso-surface reconstruction, and direct volume rendering techniques that immediately display the voxel data.
In contrast to indirect volume rendering, where an intermediate representation through surface extraction methods (e.g. the Marching Cube algorithm [10]) is generated and then displayed, direct volume rendering uses the original data. Although the original implementation did not use texturing hardware, the basic idea of using object-aligned slices to substitute trilinear by bilinear interpolation was introduced by Lacroute and Levoy [6], the ShearWarp algorithm.
Cabral [2] presented a texture-based approach, exploiting the 3D texture mapping capabilities of high-end graphics workstations. This method has been expanded by Westermann und Ertl [19], who store density values and corresponding gradients in texture memory and exploit OpenGL extensions for unshaded volume rendering, shaded iso-surface rendering, and application of clipping geometry. Based on their implementation , Meißner et al. [12] have expanded the method to enable diffuse illumination for semi-transparent volume rendering. However, this approach requires multiple passes through the rasterization hardware, resulting in a significant loss in rendering performance.
Rezk-Salama et al. [13] presented a technique that significantly improves both performance and image quality of the 2D-texture based approach. But in contrast to the techniques presented previously (all based on high-end graphics workstations), they show how multi-texturing capabilities of modern consumer PC graphics boards are exploited to enable interactive volume visualization on low-cost hardware. Furthermore they introduced methods for using NVidia's register combiner OpenGL extension for fast shaded isosurfaces, interpolation and volume shading.
Engel at al. [3] expanded the usage of low-cost hardware and introduced a novel texture-based volume rendering approach based on pre-integration (presented by Röttger, Kraus and Ertl in [14]). This method provides high image quality even for low-resolution volume data and non-linear transfer functions with high frequencies by exploiting multi-texturing, advanced texture fetch and pixel-shading operations, available on current programmable consumer graphics hardware.

3 Hardware-Accelerated Volume Rendering

This section gives a brief overview of general direct volume rendering, especially the theoretical background. Then we focus on how to exploit graphics hardware for direct volume rendering purposes and afterwards we discuss the varying classification methods that we have implemented. Additionally we briefly mention the hardware-accelerated filtering method, that we use for quality enhancements.

3.1 Volume Rendering

Algorithms for direct volume rendering differ in the way the complex problem of image generation is split up into several subtasks. A common classification scheme differentiates between image-order and object-order approaches. An example for an image-order method is ray-casting, in contrast object-order methods are cell-projection, shear-warp, splatting, or texture-based algorithms.
In general all methods use an emission-absorption model for the light transport. The common theme is an (approximate) evaluation of the volume rendering integral for each pixel, in other words an integration of attenuated colors (light emission) and extinction coefficients (light absorption) along each viewing ray. The viewing ray $x(\lambda)$ is parametrized by the distance λ to the viewpoint. For any point x in space, color is emitted according to the function $c(x)$ and absorbed according to the function $e(x)$. Then the volume rendering integral is

$$I = \int_0^D c(x(\lambda)) \exp\left(-\int_0^\lambda e(x(t))dt\right) d\lambda \quad (1)$$

where D is the maximum distance, in other words no color is emitted for λ greater than D.
For visualization of a continuous scalar field this integral is not useful since calculation of emitted colors and absorption coefficients is not specified. Therefore in direct volume rendering, the scalar value given at a sample point is mapped to physical quantities that describe the emission and absorption of light at that point. This mapping is called *classification* (classification will be discussed in detail in Sections 3.2 and 3.3). This is usually performed by introducing transfer functions for color emission and opacity (absorption). For each scalar value $s = s(x)$ the transfer function maps data values to color $c(s)$ and opacity $\tau(s)$ values. Additionally other parameters can influence the color emission or opacity, e.g., ambient, diffuse and specular lighting conditions or the gradient of the scalar field (e.g. in [7]).
Calculating the color contribution of a point in space with respect to the color value (through transfer function) and all other parameters is called *shading*. Applying simple shading (color and opacity are defined simply

through classification) the volume rendering integral can be written as

$$I = \int_0^D c(s(x(\lambda))) \exp\left(-\int_0^\lambda \tau(s(x(t)))dt\right) d\lambda. \quad (2)$$

Usually an analytical evaluation of the volume integral is not possible. Therefore usually a numerical approximation of the integral is calculated using a Riemann sum for n equal ray segments of length $d = D/n$ (see Section IV.A in [11]). This technique results in the common approximation of the volume rendering integral

$$I \approx \sum_{i=0}^{n} \alpha_i C_i \prod_{j=0}^{i-1}(1-\alpha_j) \quad (3)$$

which can be adapted for back-to-front compositing resulting in the following equation

$$C'_i = \alpha_i C_i + (1-\alpha_i) C'_{i+1} \quad (4)$$

where now $\alpha_i C_i$ corresponds to $c(s(x))$ from the volume rendering integral. The pre-multiplied color αC is also called *associated color* ([1]).

Due to the fact that a discrete approximation of the volume rendering integral is performed, according to the sampling theorem, a correct reconstruction is only possible with sampling rates larger than the Nyquist frequency. Because of the non-linearity of transfer functions (increases Nyquist frequency for the sampling), it is not sufficient to sample a volume with the Nyquist frequency of the scalar field. This undersampling results in visual artifacts that can only be avoided by very smooth transfer functions. Section 3.3 gives a brief overview on a classification method realizing an improved approximation of the volume rendering.

3.2 Pre- and Post-Classification

As mentioned in the previous section classification has an important part in direct volume rendering. Thus there are different techniques to perform the computation of *c(s(x))* and *τ(s(x))*. In fact, volume data is presented by a 3D array of sample points. According to sampling theory, a continuous signal can be reconstructed from these sampling points by convolution with an appropriate filter kernel. The order of the reconstruction and the application of the transfer function defines the difference between pre- and post-classification, which leads to remarkable different visual results.

Pre-classification denotes the application of the transfer function to the discrete sample points before the data interpolation step. In other words the color and absorption are calculated in a pre-processing step for each sampling point and then used to interpolate *c(s(x))* and *τ(s(x))* for the computation of the volume rendering integral.

On the other side post-classification reverses the order of operations. This type of classification is characterized by the application of the transfer function after the interpolation of *s(x)* from the scalar values of the discrete sampling points. With respect to the graphics pipeline the advantage of pre-classification is, that an efficient implementation of this concept is possible on almost every graphics hardware. However the results achieved by this approach are not very convincing. Post-classification is usually more complex to implement but achieves superior image quality. The results of both pre- and post-classification can be compared in Figure 1.

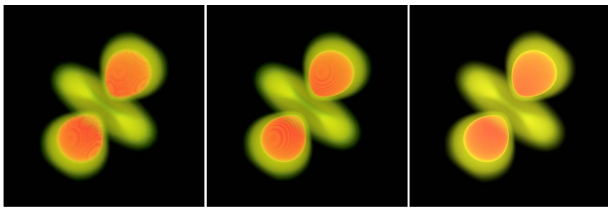

Figure 1: Direct volume rendering without illumination, pre-classified (*left*), post-classified (*middle*) and pre-integrated (*right*)

3.3 Pre-Integrated Classification

As discussed at the end of Section 3.1 to gather better visual results, the approximation of the volume rendering integral has to be improved. Röttger et al. [14] used a pre-integrated classification method to enhance cell-based volume reconstruction. This algorithm has been adapted for hardware accelerated direct volume rendering by Engel et al. [3]. The main idea of pre-integrated classification is to split the numerical integration process. Separate integration of the continuous scalar field and the transfer functions is performed to cope with the problematic of the Nyquist frequency.

In more detail, for each linear segment one table lookup is executed, where each segment is defined by the scalar value at the start of the segment s_f, the scalar value at the end of the segment s_b and the length of the segment d. The opacity α_i of the i-th line segment is approximated by

$$\begin{aligned} \alpha_i &= 1 - \exp\left(-\int_{id}^{(i+1)d} \tau\left(s\left(x(\lambda)\right)\right) d\lambda\right) \\ &\approx 1 - \exp\left(-\int_0^1 \tau((1-\omega)s_f + \omega s_b) d \, d\omega\right). \quad (5) \end{aligned}$$

Analogously the associated color $\tilde{C}_i^\tau$ (based on a non-associated color transfer function) is computed through

$$\begin{aligned} \tilde{C}_i^\tau &\approx \int_0^1 \tau((1-\omega)s_f + \omega s_b) c((1-\omega)s_f + \omega s_b) \quad (6) \\ &\quad \times \exp\left(-\int_0^\omega \tau((1-\omega')s_f + \omega' s_b) d \, d\omega'\right) d \, d\omega. \end{aligned}$$

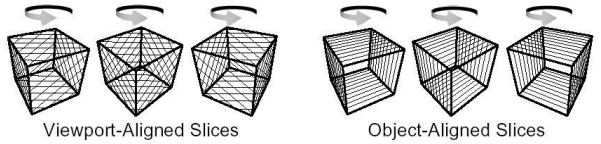

Figure 2: Alignment of texture slices for 3D texturing on the left, and 2D texturing on the right (image from Rezk-Salama et al. [13])

Both functions are dependent on s_f, s_b, and d (only if lengths of the segments are not equal). As usual the volume rendering integral is approximated by evaluation of Equation (3). Because pre-integrated classification always computes associated colors, $\alpha_i C_i$ in Equation (3) has to be substituted by $\tilde{C}_i^\tau$.
Through this principle the sampling rate does not depend anymore on the non-linearity of transfer functions, resulting in less undersampling artifacts. Therefore, pre-integrated classification has two advantages, first it improves the accuracy of the visual results, and second fewer samples are required to achieve equal results regarding to the other presented classification methods.
The major drawback of this approach is that the lookup tables must be recomputed every time the transfer function changes. That strongly limits the interactivity of applications employing this classification approach. Therefore, the pre-integration step should be very fast. Engel et al. [3] proposes to assume a constant length of the segments, thus the dimensionality of the lookup table is reduced to two. By employing integral functions for $\tau(s)$ and $\tau(s)c(s)$ the evaluation of the integrals in Equations (5) and (6) can be greatly accelerated. Adapting this idea results in the following approximation of the opacity and the associated color

$$\begin{aligned} \alpha(s_f, s_b, d) &\approx 1 - \exp\left(-\frac{d}{s_b - s_f}(T(s_b) - T(s_f))\right) \\ \tilde{C}^\tau(s_f, s_b, d) &\approx \frac{d}{s_b - s_f}(K^\tau(s_b) - K^\tau(s_f)) \end{aligned} \quad (7)$$

with the integral functions $T(s) = \int_0^s \tau(s)ds$ and $K^\tau(s) = \int_0^s \tau(s)c(s)ds$. This precalculation can easily performed since scalar values s are usually discrete. Thus, the numerical computing for producing the lookup tables can be minimized by only calculating the integral functions $T(s)$ and $K^\tau(s)$. Afterwards computing the colors and opacities according to Equations (7) can be done without any further integration. This pre-calculation can be done in very short time, so providing interactivity in transfer-functions changes. The quality enhancement of pre-integrated classification in comparison to pre- and post-classification can be seen in Figure 1.
How the presented classification methods can be adapted for hardware-based volume rendering will be discussed in Section 4.

3.4 Texture Based Volume Rendering

Basically there are two different approaches how hardware acceleration can be used to perform volume rendering.

3D texture-mapped volume rendering

If 3D-textures are supported by the hardware it is possible to download the whole volume data set as one single three-dimensional texture to hardware. Because hardware capable of 3D-texturing is able to perform trilinear interpolation within the volume, it is possible to render a stack of polygon slices parallel to the image plane with respect to the current viewing direction (see Figure 2, left).
This viewport-aligned slice stack has to be recomputed every time, the viewing position changes. Finally, in the compositing step, the textured polygons are blended onto the image-plane in a back-to-front order. This is done by using the alpha-blending capability of computer graphics hardware which usually results in a semitransparent view of the volume. Since slice polygons can be positioned arbitrarily, as many slices as required can be rendered, resulting in an image enhancement. However, in order to obtain equivalent representations while changing the number of slices, opacity values have to be adapted according to the slice distance. But rendering too many polygons results in even worse visualizations showing severe artifacts, because the frame buffer precision limits the number of slices, that can improve image quality further.
Since it is nearly a standard that 3D texture mapping capabilities are now available on consumer-oriented graphics adapters (like the ATI-Radeon family [15] or the NVIDIA GeForce 3 and 4 [17]) this approach is suitable for hardware accelerated volume rending on standard PC-platforms.

2D texture-mapped volume rendering

If hardware does not support 3D texturing, 2D texture mapping capabilities can be used for volume rendering. In this case, the polygon slices are set orthogonal to the principal viewing axes of the rectilinear data grid. Therefore if the the viewing direction changes by more than 90 degrees, the orientation of the slice normal has to be changed. This requires that the volume has to be represented through three stacks of slices, one for each slicing direction respectively, so the slice direction is object-aligned (see Figure 2, right).
2D texturing hardware does not have the ability to perform trilinear interpolation. Because of that the slice polygons cannot be positioned arbitrarily within the volume. So the alignment of the slices with respect to the viewport is not possible. The trilinear interpolation (as performed by 3D texturing hardware) is substituted by bilinear interpolation within each slice, which is although supported by hard-

ware. This results in strong visual artifacts due to the fact of the missing spatial interpolation. Another major drawback of this approach in contrast to the previous one is the high memory requirements, because 3 instances of the volume data set have to be hold in memory. As in the 3D texturing approach, to obtain equivalent representations, the opacity values have to be adopted. But now according to the slice distance between adjacent slices in direction of the viewing ray.
To enhance the image quality of 2D texture based volume rendering, multitexturing capabilities can be used. To avoid the artifacts caused by the lack of spatial interpolation Rezk-Salama at al. [13] has introduced an approach to produce intermediate slices on the fly. To enable real trilinear interpolation the missing third interpolation step is performed within the rasterization hardware. Two fixed adjacent textured slices are combined using a component-wise weighted sum, exploiting the register combiners OpenGL extension by NVIDIA (see Section 4). Thus linear interpolation between neighboring slices that are bilinearly filtered in themselves produces again trilinear interpolation. A closer look on the implementation of these approaches is covered by Section 4.1.

3.5 High-Quality Filtering

Commodity graphics hardware can also be exploited to achieve hardware-accelerated high-quality filtering with arbitrary filter kernels, as introduced by Hadwiger et al. [5]. In this approach filtering of input data is done by convolving it with an arbitrary filter kernel stored in multiple texture maps. As usual, the base is the evaluation of the well-known filter convolution sum

$$g(x) = (f * h)(x) = \sum_{i=\lfloor x \rfloor - m + 1}^{\lfloor x \rfloor + m} f[i]h(x - i) \quad (8)$$

this equation describes a convolution of the discrete input samples $f[i]$ with a reconstruction filer $h(x)$ of (finite) half-width m.
To be able to exploit standard graphics hardware to perform this computation, the standard evaluation order (as used in software-based filtering) has to be reordered. Instead of gathering all input sample contributions within the kernel width neighborhood of a single input sample, this method distributes all single input sample contributions to all relevant output samples. The input sample function is stored in a single texture and the filter kernel in multiple textures. Kernel textures are scaled to cover exactly the contributing samples. The number of contributing samples is equal to the kernel width. To be able to perform the same operation for all samples at one time, the kernel has to be divided into several parts, to cover always only one input sample width. Such parts are called filter tiles.
Instead of imagining the filter kernel being centered at the ”current” output sample location, an identical mapping of input samples to filter values can be achieved by replicating a single filter tile mirrored in all dimensions repeatedly over the output sample grid. The scale of this mapping is chosen, so that the size of a single tile corresponds to the width from one input sample to the next.
The calculation of the contribution of a single specific filter tile to all output samples is done in a single rendering pass. So the number of passes necessary is equal to the number of filter tiles the filter kernel used consists of. Due to the fact that only a single filter tile is needed during a single rendering pass, all tiles are stored and downloaded to the graphics hardware as separate textures. If a given hardware architecture is able to support $2n$ textures at the same time, the number of passes can be reduced by n.
This method can be applied for volume rendering purposes by switching between two rendering contexts. One for the filtering and one for the rendering algorithm, whereas first a textured slice is filtered according to the just described method, and afterwards the filtered output is then used in the standard volume rendering pipeline. This is not as easy as it sounds, thus implementation difficulties are described in more detail in section 4.2. For results see Figure 3.

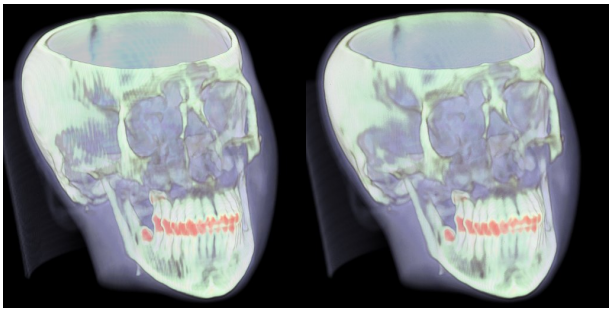

Figure 3: Pre-integrated classification without pre-filtered slices (*left*) and applying hardware-accelerated filtering (*right*).

4 Implementation

Our current implementation is based on a graphical user interface programmed in java, and a rendering library written in c++. For proper usage of the c++ library in java, e.g for parameter passing, we exploit the functionality of the java native interface [16], which describes how to integrate native code within programs written in java. Due to the fact that our implementation is based on the OpenGL API [18], we need a library that maps the whole functionality of the native OpenGL library of the underlying operating system to java. Therefore we use the GL4Java library [4]. The following detailed implementation description will only cover the structure of the c++ rendering library, because all rendering functionality is encapsulated there.
On startup of the framework, the graphics adapter cur-

rently installed in the system is detected automatically. Regarding to the OpenGL extensions that are supported by the actual hardware the rendering modes that are not possible are disabled. Through this procedure, the framework is able to support a lot of different types of graphics adapters without changing the implementation.
Anyway the framework is primarily based on graphics chips from NVidia and from ATI, because the OpenGL-extensions provided by these two vendors are very powerful features, which can be exploited very well for diverse direct volume rendering techniques. Minimum requirements for our application are multi-texturing capabilities. Full functionality includes the exploitation of the so called *texture shader* OpenGL extension and the *register combiners* provided by NVidia as well as the *fragment shader* extension, provided by ATI.
Basically the texture based volume rendering process can be split up into several principal subtasks. Each of these tasks is realized in one or more modules, to provide easy reuse possibilities. Therefore the implementation of new algorithms and the support of new hardware-features (OpenGL-extensions) is very simple by only extending these modules with additional functionality. The overall rendering implementation need to be changed to achieve support of new techniques or new graphics chips.

Texture definition

As described in section 3.4, in the beginning of the rendering process the scalar volume data must be down-loaded to the hardware. According to the selected rendering mode, this is either be done as one single three-dimensional texture or as three stacks of two-dimensional textures.
The selected rendering mode additionally specifies the texture format. In our context texture format means, what values are presented in a texture. Normally, RGBA (Red, green, blue and alpha component) color values are stored in a texture, but in volume rendering, other information as the volume gradient or the density value have to be accessed during the rasterization stage. For gradient vector reconstruction, we have implemented a central-difference filter and additionally a sobel-operator, which results in a great quality enhancement in contrast to the central-difference method, avoiding severe shading artifacts (see Figure 4).
When performing shading calculations, RGBA textures are usually employed, that contain the volume gradient in the RGB components and the volume scalar in the ALPHA component. As in pre-integrated rendering modes the scalar data has to be available in the first three components of the color vector, it is stored in the RED component. The first gradient component is stored in the ALPHA component in return. Another exception occurs for rendering modes, which are based on gradient-weighted opacity scaling, where the gradient magnitude is stored in the ALPHA component. Through the limitation of only four available color components, it is trivial that for the combination of some rendering modes it is not possible to store all the required values for a single slice in only one texture.

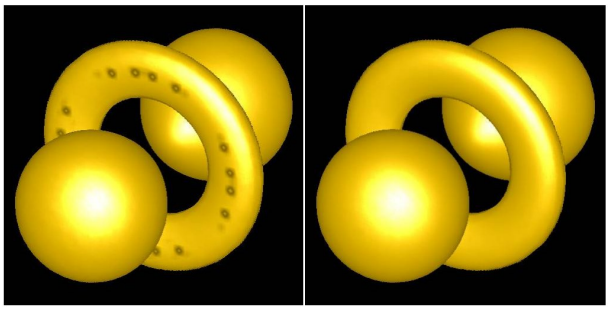

Figure 4: Gradient reconstruction using a central-difference filter (*left*) and avoiding the shading artifacts (black holes) by using a sobel-operator (*right*)

Projection

The geometry used for direct volume rendering, in contrast to other methods, e.g. iso-surface extraction, is usually very simple. Due to the fact that texture-based volume rendering algorithms usually perform slicing through a volume, the geometry only consists of a small number of primitives, one quadrilateral polygon for each slice, respectively. To obtain correct volume information for each slice, each polygon has to be bound to the corresponding textures that are required for the actual rendering mode. In addition, the texture coordinates have to be calculated accordingly. Usually this is a very simple task.
Just for 2D-texture based pre-integrated classification modes, it is a little bit more complex. Instead of the general slice-by-slice approach, this algorithm renders slab-by-slab (see Figure 5) from back to front into the frame buffer. A single polygon is rendered for each slab with the front and the back texture as texture maps. To have texels along all viewing rays projected upon each other for the texel fetch operation, the back slice must be projected onto the front slice. This projection is performed by adapting texture coordinates for the projected texture slice, which always depends on the actual viewing transformation.

Compositing

Usually in hardware accelerated direct volume rendering approaches, the approximation of the volume rendering integral is done by back-to-front compositing of the rendered quadriliteral polygon slices. This should be performed according to Equation (4). In general this is achieved by blending the slices into the frame

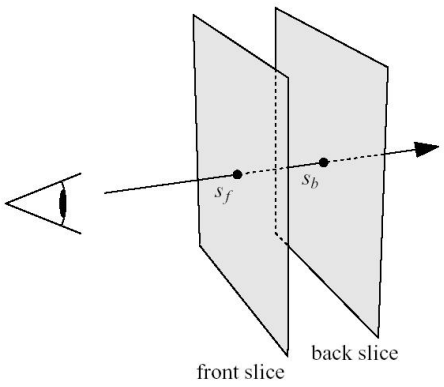

Figure 5: A slab of the volume between two slices. The scalar values on the front and on the back slice for a particular viewing ray are called s_f and s_b (image from Engel et al. [3])

buffer with the OpenGL blending function *glBlendFunc*(GL_ONE,GL_ONE_MINUS_SRC_ALPHA).
This is a correct evaluation only, if the color-values computed by the rasterization stage are associated colors. If they are not pre-multiplied (e.g. gradient-weighted opacity modes produce non-associated colors), then the blending function must be *glBlendFunc*(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA).
Iso-surface reconstruction in hardware is in general accomplished by storing the intensity value in the alpha-component of the fragment's color. The volume is then rendered into the frame buffer using the OpenGL alpha-test to display the specified isovalues only.
These two techniques can be combined for rendering semi-transparent iso-surfaces (see Figure 6, left), where the alpha-test is used for rejecting all fragments not belonging to an iso-surface, and afterwards the slices are blended into the frame buffer, as described above. All fragments not belonging to an iso-surface are assigned a special alpha-value of zero and the alpha test is then configured with the OpenGL function *glAlphaFunc*(GL_GREATER, 0.0), letting pass each fragment with an alpha value greater then zero.

Register settings

Depending on the selected rendering mode, during the rasterization process, the actual performed rendering technique often needs more input data than available through the slice textures (in general hold gradient and/or density information). For shading calculations the direction of the light source must be known. When modelling specular reflection the rasterization stage requires not only the light direction, but also the direction to the viewer's eye, because a halfway vector is used to approximate the intensity of specular reflection. Additionally some rendering modes need to access specific constant vectors, to perform dot-products for gradient reconstruction for example. This information has to be stored at a proper place. Therefore NVidia and ATI provide some special registers which can be accessed during rasterization process when using the *register combiners* extension or the *fragment shader* extension.
The *register combiners* extension, as described in [13], is able to access two constant color registers (in addition to the primary and secondary color), which is not sufficient for complex rendering algorithms. In the GeForce3 graphics chip, NVidia has extended the register handling by introducing the *register combiners2* extension, providing per-combiner constant color registers. This means that each combiner-stage has access to its own two constant registers, so the maximum number of additional information, provided by RGBA vectors, is the number of combiner stages multiplied by two, respectively sixteen on GeForce3. In contrast all ATI graphics chips (e.g. Radeon 8500, ...), that support the OpenGL *fragment shader* extension provide access to an equal number of constant registers, namely eight.
Due to the fact that miscellaneous rendering modes need different information contained in the constant registers, the process of packing the required data into the correct registers is more complex than it sounds. In addition these constant settings intensely influence the programming of the rasterization stage, where each different register setting requires a new implementation of the rasterization process.

4.1 NVIDIA vs. ATI

As mentioned above our current implementation supports several graphics chips from NVidia as well as several graphics chips from ATI. In this section we discuss the differences between realizations of several rendering algorithms according to the hardware-features supported by NVidia and ATI. The main focus is set on the programming of the flexible rasterization hardware, enabling advanced rendering techniques like per pixel-lighting or advanced texture-fetch methods. The differences will be discussed in detail by showing some implementation details for some concrete rendering modes after giving an short overview of rasterization hardware differences in OpenGL.
In general the flexible rasterization hardware consists of multi-texturing capabilities (allowing one polygon to be textured with image information obtained from multiple textures), multi-stage rasterization (allowing to explicitly control how color-, opacity- and texture-components are combined to form the resulting fragment, *per-pixel shading*) and dependent texture address modification (allowing to perform diverse mathematical operations on texture coordinates and to use these results for another texture lookup).

NVidia

On graphics hardware with an NVidia chip, this flexibility is provided through several OpenGL extensions, mainly GL_REGISTER_COMBINERS_NV and GL_TEXTURE_SHADER_NV. When the *register combiners* extension is enabled, the standard OpenGL texturing units are completely bypassed and substituted by a register-based rasterization unit. This unit consists of two (eight on GeForce3,4) general combiner stages and one final combiner stage.
Per-fragment information is stored in a set of input registers, and these can be combined, i.e. by dot product or component-wise weighted sum, the results are scaled and biased and finally written to arbitrary output registers. The output registers of the first combiner stage are then input registers for the next stage, and so on.
When the *per-stage-constants* extension is enabled (GL_PER_STAGE_CONSTANTS_NV), for each combiner stage two additional registers are available, that can hold arbitrary data, otherwise two additional registers are available too, but with equal contents for every stage.
The *texture shader* extension provides a superset of conventional OpenGL texture addressing. It provides a number of operations that can be used to compute texture coordinates per-fragment rather than using simple interpolated per-vertex coordinates. The shader operations include for example standard texture access modes, dependent texture lookup (using the result from a previous texture stage to affect the lookup of the current stages), dot product texture access (performing dot products from texture coordinates and a vector derived from a previous stage) and several special modes.
The implementation of these extensions results in a lot of code, because the stages have to be configured properly, and an assembler like programming is not provided.

ATI

On graphics hardware with an ATI Radeon chip, this flexibility is provided through one OpenGL extension, GL_FRAGMENT_SHADER_ATI. Generally this extension is very similar to the the extensions described before, but encapsulates the whole functionality in a single extension. The *fragment shader* extension inserts a flexible per-pixel programming model into the graphics pipeline in place of the traditional multi-texture pipeline. It provides a very general means of expressing fragment color blending and dependent texture address modification.
The programming model is a register-based model and the number of instructions, texture lookups, read/write registers and constants is queryable. E.g. on the ATI Radeon 8500 there are six texture fetch operations and eight instructions possible, both two times during one rendering pass, yielding maximum of sixteen instructions in total.

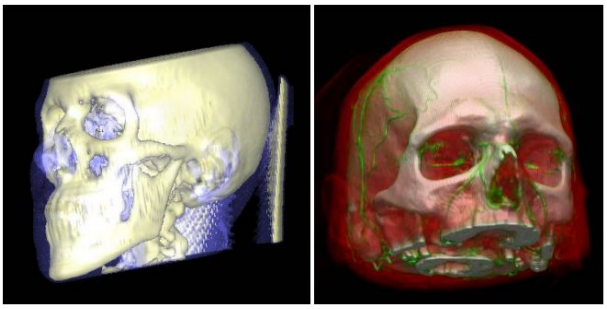

Figure 6: Semi-transparent iso-surface rendering (*left*) and pre-integrated volume rendering (*right*) of different human head data sets.

One advantageous property of the model is a unified instruction set used throughout the shader. That is, the same instructions are provided when operating on address or color data. Additionally, this unified approach simplifies programming (in contrast to the above presented NVidia extensions), because only a single instruction set has to be used and the *fragment shader* can be programmed comparable to an assembler language.
This tremendously reduces the amount of produced code and therefore accelerates and simplifies debugging. For these reasons and because up to six textures are supported by the multi-texturing environment, ATI graphics chips provide powerful hardware features to perform hardware-accelerated high-quality volume rendering.

Pre- and Post-classification

As described in detail in Section 3.2, pre- and post-classification differ in the order of the reconstruction step and the application of the transfer function.
Since most NVidia graphics chips support *paletted textures* (OpenGL extension GL_SHARED_TEXTURE_PALETTE_EXT), pre-classified volume rendering is easy to implement. *Paletted textures* means that instead of RGBA or luminance, the internal format of a texture is an index to a color-palette, representing the mapping of a scalar value to a color (defined by transfer-function). This lookup is performed before the texture fetch operation (before the interpolation), thus pre-classified volume rendering is performed. Since there is no similar OpenGL-extension supported by ATI graphics chips, rendering modes, based on pre-classification are not available on ATI hardware.
Post-classification is available on graphics-chips from both vendors, in case that advanced texture-fetch possibilities are available. As described in the beginning of this section when using the *texture*- and *fragment-shader*, dependent texture lookups can be performed. This feature is exploited for post-classification purposes. The transfer function is downloaded as a one-dimensional texture

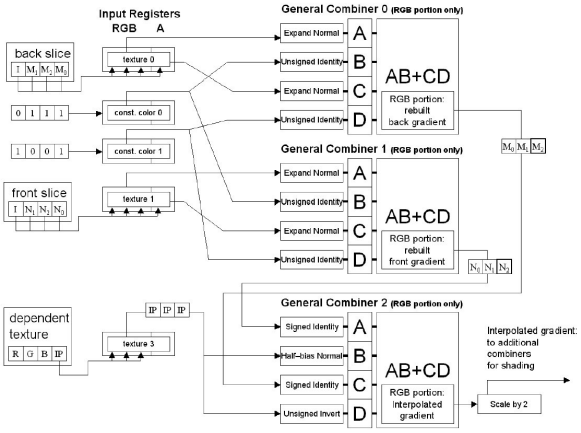

Figure 7: Register combiner setup for gradient reconstruction and interpolation with interpolation values stored in alpha (image from Engel et al. [3])

and for each texel, fetched by the given per-fragment texture coordinates, the scalar value is used as a lookup coordinate into the dependent 1D transfer-function texture. Thus post-classification is available, because the scalar value obtained from the first texture fetch has been bi- or trilinearly filtered, dependent on whether 2D or 3D volume-data textures are employed, and the transfer-function is applied afterwards.

Pre-integration

As post-classification, pre-integrated classification can also be performed on graphics-chips from both vendors if *texture shading* is available. The pre-integrated transfer-function, since dependent on two scalar values (s_f from the front and s_b from the back slice, see Figure 5 and Section 3.3 for details) is downloaded as a two-dimensional texture, containing pre-integrated values for each of the possible combinations of front and back scalar values.

For each fragment, texels of two adjacent slices along each ray through the volume are projected onto each other. Then the two fetched texels are used as texture coordinates for a dependent texture lookup into the 2D pre-integration texture. To extract the scalar values, usually stored in the red component of the texture, the dot product with a constant vector $v = (1, 0, 0)^T$ is applied. These values are then used for the lookup.

Pre-integrated volume rendering can also be employed to render multiple isosurfaces. The basic idea is to color each ray segment according to the first isosurface intersected by the ray segment. So the dependent texture contains color, transparency, and interpolation values ($IP = (s_{iso} - s_f)/(s_b - s_f)$) for each combination of back and front scalar. For lighting purposes the gradient of the front and back slice has to be rebuilt in the RGB components and the two gradients have to be interpolated depending on the given isovalue. The implementation of this reconstruction using *register combiners* is shown in Figure 7.

4.2 Problems

As mentioned in the previous sections, the integration of different rendering techniques in one framework supporting varying graphics chips requires a lot of care when implementing the varying methods. In addition to that implementation difficulties, we encountered other problems as well.

When applying the hardware accelerated high quality filtering method (see Section 3.5) in combination with an arbitrary rendering mode, we have to cope with different rendering contexts. One context for the rendering algorithm and one for the high quality filtering. A single slice is rendered into a buffer, this result is then used in the filtering context to apply the specified filtering method (e.g. bi-cubic), and this result is then moved back into the rendering context, to perform i.e. the compositing step. More difficult is the case of combining the filtering with preintegration, where two slices have to be switched between the rendering contexts. Through different contexts the geometry and the OpenGL state handling is varying depending on whether filtering is applied or not. It is a challenge to define and provide the correct data in the right context and not mixing up the complex state handling. Although the performance is not so high, the resulting visualizations are very convincing (see Figure 1).

Another problem that occurs when realizing such a large framework is that the performance that usually is achieved by the varying algorithms can not be guaranteed. Furthermore when performing shading, rendering datasets with dimensions over 256^3 results in a heavy performance loss, caused by the memory bottle neck. Which means that not the whole data set can be downloaded to the graphics adapter memory, instead of, the textures are transferred between the main and the graphics memory.

5 Conclusions and Future Work

On the basis of standard 2D- and 3D-texture based volume rendering and several high quality rendering techniques, we have presented a flexible framework, which integrates several different direct volume rendering and isosurface reconstruction techniques that exploit rasterization hardware of PC graphics boards in order to significantly improve both performance and image quality. Additionally the framework can easily be extended with respect to support of new OpenGL extensions and implementation of new rendering algorithms, by only expanding the proper modules. The framework supports most current low-cost graphics hardware and provides comparison pos-

sibilities for several hardware-accelerated volume visualizations with regard to performance and quality.
In the future we plan the integration of non-photorealistic rendering techniques to enhance actual volume visualizations as well as support of upcoming new graphics adapters. To overcome the problem that different graphics chips require different implementations, we will try the usage of a high-level shading language.

6 Acknowledgements

This work was carried out as part of the basic research on visualization (http://www.VRVis.at/vis) at the VRVis Research Center Vienna, Austria (http://www.VRVis.at/), which is funded by an Austrian governmental research project called Kplus.

References

[1] James F. Blinn. Jim Blinn's corner: Compositing. 1. Theory. *IEEE Computer Graphics and Applications*, 14(5):83–87, September 1994.

[2] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, October 1994.

[3] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on on Graphics hardware*, pages 9–16. ACM Press, 2001.

[4] Jausoft OpenGL for Java web page. http://www.jausoft.com/gl4java.html/.

[5] Markus Hadwiger, Thomas Theußl, Helwig Hauser, and Eduard Gröller. Hardware-accelerated high-quality reconstruction on PC hardware. In *Proceedings of the Vision Modeling and Visualization Conference 2001 (VMV-01)*, pages 105–112, Berlin, November 21–23 2001. Aka GmbH.

[6] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(Annual Conference Series):451–458, July 1994.

[7] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988. See corrigendum [9, 20].

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

[9] Marc Levoy. Letter to the editor: Error in volume rendering paper was in exposition only. *IEEE Computer Graphics and Applications*, 20(4):6–6, July/ August 2000.

[10] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[11] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995. ISSN 1077-2626.

[12] Michael Meißner, Ulrich Hoffmann, and Wolfgang Straßer. Enabling classification and shading for 3D texture mapping based volume rendering using openGL and extensions. In *IEEE Visualization '99*, pages 207–214, San Francisco, 1999. IEEE.

[13] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. pages 109–118.

[14] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. pages 109–116. IEEE Computer Society Technical Committee on Computer Graphics, 2000.

[15] ATI web page. http://www.ati.com/.

[16] JAVA web page. http://java.sun.com/.

[17] NVIDIA web page. http://www.nvidia.com/.

[18] OpenGL web page. http://www.opengl.org/.

[19] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 169–178. ACM SIGGRAPH, Addison Wesley, July 1998.

[20] Craig Wittenbrink, Tom Malzbender, and Mike Goss. Letter to the editor: Interpolation for volume rendering. *IEEE Computer Graphics and Applications*, 20(5):6–6, September/October 2000. See [7, 9].