A Generic Software Framework for the GPU Volume Rendering Pipeline

Joachim E. Vollrath[1], Daniel Weiskopf[1,2], Thomas Ertl[1]

[1]Visualization and Interactive Systems Group (VIS)
University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
Email: {vollrajm, weiskopf, ertl}@vismail.informatik.uni-stuttgart.de

[2]Graphics, Usability, and Visualization (GrUVi) Lab
School of Computing Science, Simon Fraser University
8888 University Drive, Burnaby, V5A 1S6, Canada

Abstract

This paper presents an extendable, simple, and efficient software framework that implements the GPU-based volume rendering pipeline. We use volume graphics for realistic image synthesis taking into account aspects of visual perception by means of real-time high dynamic range tone mapping. We propose a software architecture that embeds the volume rendering pipeline by using object-oriented design patterns, layers, and the concept of a shared application state. The pipeline is made flexible by representing stages as objects, loosely coupled via the shared state. We demonstrate the benefits of our architecture in a modern volume rendering application that incorporates state-of-the-art features such as pre-integration, pre-integrated lighting, and volumetric shadows. The full source code of our framework is made publicly available.

1 Introduction

Volume rendering is widely used in many fields of application, ranging from direct volume visualization of scalar fields for engineering and sciences to medical imaging and finally to the realistic rendering of clouds or other gaseous phenomena in visual simulations or for special effects. In recent years, texture-based volume rendering on consumer graphics hardware has become a popular approach for direct volume visualization. The performance and functionality of GPUs (graphics processing units) have been increasing rapidly while their prices have stayed attractive. Therefore, GPU-based volume graphics has become a standard approach for all classes of platforms.

Among different variations of GPU volume rendering, texture slicing is most widely used and well understood. From a software-engineering point of view, it is important to structure volume rendering in the form of a well defined pipeline. The goal of this paper is a flexible, extendable, and simple software framework that implements a pipeline for volume rendering on the GPU. An important objective is an efficient realization that may make use of the latest GPU features in order to achieve interactive rendering. The contributions of this paper are: (A) The construction of a comprehensive GPU volume rendering pipeline that combines traditional volume rendering techniques with concepts of realistic image synthesis, i.e., a physically based simulation of light transport in conjunction with aspects of visual perception. (B) An appropriate software architecture for the framework, based on established software design patterns. (C) A full implementation of state-of-the-art GPU volume rendering techniques, which includes recent developments such as pre-integrated classification, pre-integrated illumination, and volume shadowing. (D) High dynamic range (HDR) imaging with real-time tone mapping, which has not been used previously for volume visualization. (E) Source code for a fully functional implementation of the framework for Direct3D.

2 Previous Work

Volume rendering is a well-established and active field in visualization and computer graphics. A comprehensive overview is given by Kaufman and Mueller [11]. Pfister [23] presents a more specific survey of volume rendering on graphics hard-

ware. An early example of 3D texture-based rendering is viewport-aligned slicing according to Cullip and Neumann [4] and Cabral et al. [3]. More recent research on texture-based volume rendering has led to advanced volumetric illumination and shading techniques, most of which rely on advanced per-fragment processing in graphics hardware; see, e.g., the work by Westermann and Ertl [29], Rezk-Salama et al. [25], Engel et al. [7], Kniss et al. [13], Lum et al. [18], and Hadwiger et al. [10].

Besides texture slicing, other prominent approaches to volume rendering are ray casting [17], splatting [30], and shear-warp rendering [16]. Ray casting, in particular, has recently attracted increased attention because it has become feasible on GPUs [26, 15, 27].

GPU volume rendering is part of many commercial software products (e.g., in medical imaging) or numerous research prototypes. Typical examples for larger systems are OpenGL Volumizer [2], the OpenQVis project [20], or the volume node of the OpenSG [21] scene graph.

Realistic image synthesis, in general, has to take into account perceptual issues in addition to a simulation of light transport. The comprehensive framework by Greenberg et al. [9] is the basis for our approach to a pipeline for realistic volume rendering. In particular, visual perception is considered by applying tone mapping operators [5].

Software engineering is the field of computer science concerned with the technologies and methodologies involved in creating and maintaining software. Early works on modularization and information hiding in software development were carried out by Parnas [22]. A comprehensive collection of object-oriented software design patterns is presented in the seminal work of Gamma et al. [8].

3 Volume Rendering Pipeline

3.1 Mathematical Model

The standard optical model for volume rendering is based on the volume rendering integral

$$I(D) = I_0\,T(t_0) + \int_{t_0}^{D} g(t)T(t)\,\mathrm{d}t \quad , \qquad (1)$$

as described in the survey article [19]. The term I_0 represents the light entering the volume from the background at the position $t = t_0$; $I(D)$ is the radiance leaving the volume at $t = D$ and finally reaching the camera. Transparency T is defined

$$T(t) = e^{-\int_{t}^{D} \tau(t')\,\mathrm{d}t'} \quad .$$

The integral from Eq. (1) is typically approximated by a Riemann sum over n equidistant segments of length $\Delta x = (D - t_0)/n$. This approximation yields

$$I(D) \approx I_0 \prod_{i=1}^{n} t_i + \sum_{i=1}^{n} g_i \prod_{j=i+1}^{n} t_i \quad , \qquad (2)$$

where $t_i = \exp(-\tau(i\Delta x + t_0)\Delta x)$ and $g_i = g(i\Delta x + t_0)\Delta x$ are the transparency term and source term for the ith segment, respectively.

3.2 Generic Volume Rendering Pipeline

The computation of the volume rendering integral can be split into several subsequent stages of the volume rendering pipeline. According to the survey article by Pfister [23], the following stages are commonly found in volume rendering techniques: data traversal (pipeline stage p_1), interpolation (p_2), gradient computation (p_3), classification (p_4), shading (p_5), and compositing (p_6).

During data traversal, resampling positions are chosen along viewing rays. 3D texture slicing, for example, uses image-aligned slices [3, 4] as 2D proxy geometry to construct resampling positions. An interpolation scheme is applied at these positions to reconstruct the dataset at locations that differ from grid points. A typical reconstruction filter is based on built-in trilinear interpolation within 3D textures. Gradients of a discretized volumetric dataset are typically approximated by using discrete gradient filters, such as central differences or the Sobel operator. Gradients can be pre-computed and stored along with the scalar data (in RGB and alpha channels, respectively).

Classification maps properties of the dataset to optical properties for the volume rendering integral according to a transfer function. The classification typically assigns discretized optical properties that are combined as RGBA values. Volume shading can be incorporated into transfer functions by adding an illumination term (e.g. the Blinn-Phong model).

Compositing is the basis for the iterative computation of the discretized volume rendering integral (2). Both front-to-back and back-to-front strategies

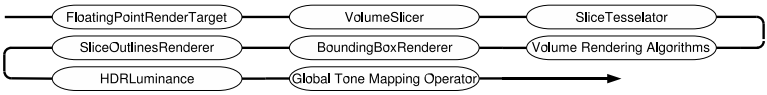

Figure 1: GPU-based volume rendering pipeline

with their respective compositing equations employ alpha blending. We support floating-point texture formats to avoid quantization problems and, at the same time, allow for a high dynamic range of radiance values. In addition to alpha blending, a number of alternative approaches may be used, such as maximum intensity projection (MIP) or weighted sum projections.

The volume rendering pipeline implements a physical simulation of light transport within participating media, neglecting multiple scattering. Such a simulation needs to be combined with a perception-oriented final display in order to achieve a realistic looking image [9]: Essentially, a final mapping from an image of the resulting physical simulation (i.e., radiance values per pixel) with its own (possibly high) dynamic range to the lower dynamic range of typical display devices has be to added to the core volume rendering pipeline by means of high dynamic range tone mapping.

3.3 GPU-Based Volume Rendering Pipeline

Performing volume rendering on a GPU requires an appropriate mapping of the volume rendering pipeline stages to the programmable rendering pipeline as implemented on modern GPUs. The architecture of the GPU pipeline leads to the following coarse mapping:

1. proxy geometry generation (p_1), as input to GPU geometry processing
2. actual rendering of the volume (p_2 to p_6), as part of GPU fragment processing
3. high dynamic range tone mapping, as an example of GPU-based image processing

Although the second stage comprises several elements of the generic volume rendering pipeline, p_2 to p_6 could still be separated into distinct shader components, for example by relying on function calls in high level shading languages.

In our framework the GPU volume rendering pipeline is segmented more finely into eight stages as shown in Figure 1. *FloatingPointRenderTarget* switches the GPU to a floating-point render target in the case of active high dynamic range tone mapping. The proxy geometry generation stage for volume sampling on the GPU is divided into *VolumeSlicer* and *SliceTesselator*. *VolumeRenderingAlgorithms* is the stage where actual rendering algorithms are executed. We have implemented classical back-to-front volume rendering [3, 4], volume rendering with pre-integrated classification [7], interpolated pre-integrated lighting [18], and volumetric self-shadowing [12]. The tone mapping stage consists of *HDRLuminance*, where the luminance of the rendered HDR image is computed, and the actual tone mapping operator using the result of the previous stage. Due to the flexibility of our software framework, additional stages can be easily added to the pipeline. For example, we have included the stages *BoundingBoxRenderer* and *SliceOutlinesRenderer* for convenience.

So far we have dealt with the processes involved in the rendering of a volume. What remains is to find appropriate representations for the data on which the volume rendering pipeline operates as well as efficient protocols for interoperation of pipeline stages. We propose solutions to these issues in the following two sections.

4 Software Layers

Since modern volume renderers incorporate a variety of components and rendering algorithms, an architecture has to be devised, that will effectively handle the resulting complexity of the software.

Moreover, the architecture should allow us to add new components with minimum effort, thus encouraging the implementation of advanced features and most importantly embracing what is inevitable in most software projects: *frequent changes*.

Following the principle of *separation of concerns* [6], our renderer is divided into three layers:

- *Application layer*, responsible for application logic and interaction through the graphical user interface.
- *Data layer*, storing scalar volumetric data and all derived data (e.g. histogram, transfer function).

- *Rendering algorithms layer*, grouping all components dealing with rendering a visual representation of the volume.

It is not mandatory, though, to have a direct representation of these layers in source code. Their purpose is more of a guiding nature as to which tasks are performed where in the software, improving code quality and facilitating maintenance.

5 Shared Application State

Encapsulation and organization in layers involves the issue of sharing data between software components. Since the task of rendering the volume is segmented into stages of the volume rendering pipeline, the ways in which subsequent stages depend on results of previous stages are manifold. This is even more so the case when the pipeline is intended to be flexible and extensible.

5.1 Drawbacks of Tight Coupling

We consider the exemplary case of a C++ class that manages a pre-integration lookup texture. Clearly, in order to allow other classes to bind the texture this class must provide some access to it:

```
class CLookupTexture {
public:
    PtrToTexture    GetPTex() {
        return m_pPreIntTex; };
private:
    PtrToTexture    m_pPreIntTex;};
```

Thus, classes in which the pre-integration texture is used will need knowledge of the existence of `CLookupTexture` (by statically including the appropriate header and storing a pointer to an instance) and its interface. This concept is referred to as *tight coupling*.

As the software evolves, the decision may be made to rename `CLookupTexture` and its source files into `CPreIntegration` and the access function into `GetPPreIntTex()`. For all dependent classes a different header must now be included, the pointer to the class instance must be retyped, and the call to the access function must be renamed.

Tight coupling produces one–to–many dependencies throughout the entire software. This increases complexity, exacerbates maintenance, and makes extending the software a tedious task.

5.2 Loose Coupling through the Shared State

To gain flexibility it is desirable for classes to be decoupled from each other. Picking up our example, classes binding the aforementioned pre-integration texture merely require a valid pointer to it. Information as to which class manages the texture and how to gain access to it is essentially of no concern.

Indeed, decoupling may be achieved by introducing a *shared application state*. This is a class that stores a copy of shared class variables, providing consistent access to them (hence making them state variables). The shared state serves as a connector between software layers and components, effectively reducing one–to–many dependencies that would otherwise occur throughout the framework to one–to–one dependencies. All information necessary for components to interoperate can be retrieved from the shared state, which thus decouples components and makes them re-usable.

The resulting software architecture including a shared application state is illustrated in Figure 2.

5.3 Broadcasting State Changes

By introducing the shared state, class dependencies may be effectively reduced. However, an indirect form of dependency remains, namely that of *state changes*. Often enough it is crucial that dependent classes are given the opportunity to react to changes of a certain state variable.

Since volume rendering strives to be a real-time application, actively polling all state variables for changes is not feasible. Fortunately, software engineering has a solution that solves this issue quite elegantly: Using the publisher-subscriber (a.k.a. observer) design pattern [8] allows dependent classes to be informed when state variables are changed. In this design pattern, a publisher holds a list of subscribers to be notified in the event of a state change.

In our context the shared state uniquely identifies each state variable and allows components to subscribe specifically to changes of certain variables. Figure 3 illustrates this notification scheme.

5.4 A Practical Example

Before going into implementation details we demonstrate an exemplary run through the volume rendering pipeline to illustrate how loosely coupled

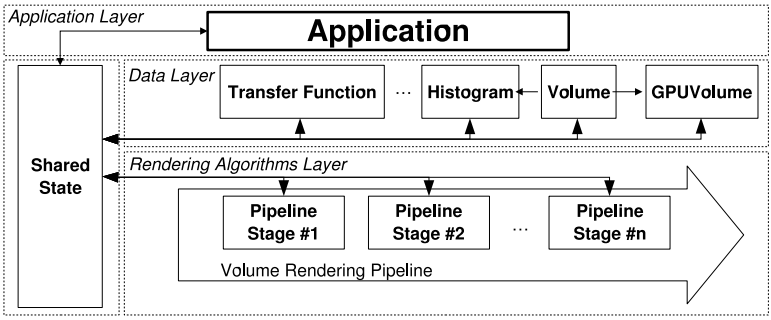

Figure 2: Architectural overview of our framework. Arrows indicate the flow of information between components.

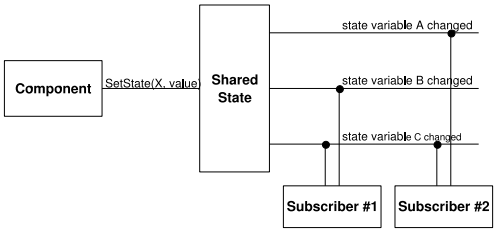

Figure 3: Publisher-subscriber notification scheme.

components may interact through the shared application state:

At first, if tone mapping is activated, *FloatingPointRenderTarget* switches the GPU to an offscreen floating-point render target and updates the pointer to the texture associated to it, stored in a state variable defined by the shared application state. All subscribing components are notified of this change and may react accordingly.

Then, with the current sampling distance and viewing direction provided by the shared state, *VolumeSlicer* creates a set of view-aligned proxy slice polygons that cut the volume. The pointer to a buffer holding slice polygons is also stored in the shared state.

SliceTesselator then tesselates the polygons and creates vertex and index buffers accordingly using the pointer to the slice polygon buffer. The pointers to both the vertex and index buffers are again stored in the shared state.

Now, a class implementing a volume rendering algorithm (say, pre-integrated volume rendering with lighting) renders the volume into the floating-point render target. It uses the vertex and index buffer for proxy slices together with the lighting direction which was previously stored in the shared state through user interaction with the application.

If activated, *BoundingBoxRenderer* renders a bounding box around the the volume. If activated, *SliceOutlinesRenderer* creates an own index buffer and renders the outlines of the slice polygons using the pointer to the slice polygon buffer set by *VolumeSlicer* and the pointer to the vertex buffer set by *SliceTesselator*.

HDRLuminance computes the luminance for each pixel of the rendered image and log-average luminance of the entire scene. Both are stored in textures to which the pointer is set in the shared state. Finally, the tone mapping operator performs its task, using the average and pixel luminance textures provided by the shared state.

6 Implementation Details

Our implementation is based on the C++ programming language and uses Direct3D as graphics API. The full source code is available from `http://www.vis.uni-stuttgart.de/texvolrend`

6.1 Shared State

The shared state defines an enumerator that uniquely identifies each state variable it stores. This enumerator is used as the key for an STL hash map that stores both the location in shared state memory and the size in bytes of each state variable. State variables are not stored as class variables in the shared state. Depending on its size, memory for each state variable is allocated in the constructor.

There are two reasons for handling state variables this way: Firstly, as the shared state merely acts as an interposed container for state variables, type information is of no concern to it. Secondly, by treating state variables as mere chunks of data, it is straightforward to provide access to them through a simple getter and setter interface function, independent of their type.

When setting a state variable, the caller provides the identifier of the variable and a pointer to the location in memory from where to fetch the new value. Together with the information stored in its hash map the shared state may then copy the new value to the correct position in shared state memory. Retrieving a state variable is carried out analogously: The shared state copies the value of a state variable to the location provided by the caller.

The interface functions establish an additional level of abstraction between classes. Provided that they are defined, there is no need to communicate with other objects in order to retrieve state variables. They may be accessed through the shared state, thus effectively decoupling objects from each other. A pleasant side effect of identifying state variables through an enumerator is that these are checked during compilation. Therefore, it is impossible to pass an identifier for a state variable that is not defined.

To allow classes to register for changes of specific state variables, vectors of pointers to subscribers are stored in a hash map. The shared state provides a subscription interface and additionally notifies subscribers in the event of a change of a state variable, which occurs when the setter function is called. Subscribers provide a consistent interface to receive notification of state changes. As an update protocol the pull method is employed, i.e., subscribers are merely informed of the event of a state change, the new value must then be retrieved actively. The pull method ensures that the shared state does not need further information about its subscribers. Otherwise dependencies would be concentrated in the shared state, destroying the benefits gained through this architecture.

6.2 Volume Rendering Pipeline

Stages of the volume rendering pipeline share the same base interface *PipelineStage*, which provides a common interface function *PerformAction*, by which they may be triggered independently of the task they perform. *PipelineStage* is derived from *GpuBase*, the base class offering a consistent interface for components using hardware-dependent resources.

Each pipeline stage manages its hardware resources independently and offers dependent stages access to them via the shared state if necessary. Resources are commonly allocated when the Direct3D device is created and deallocated in the class destructor. Stages using the GPU vertex or fragment shader units have an associated HLSL effect file that describes the shader programs.

The pipeline stages are managed in the class *VolumeRenderingPipeline*, which stores them in an STL vector and sequentially calls their *PerformAction* interface function to render the volume.

7 Results

Figure 5 shows high quality images produced with our renderer. Volumetric shadows are visually pleasing and add important visual cues to the rendering. Figure 6 shows how a rendering with an overly bright specular highlight is mapped to a displayable range by Reinhard's global tone mapping operator [24], as provided by the current DirectX sample framework.

A complete performance analysis of our framework would exceed the scope of this paper. To give an impression of the performance: Rendering the 256^3 head dataset to a 640×480 viewport with one slice per voxel, we achieve roughly 12.5 Hz without and 7.5 Hz with pre-integrated classification. With the same settings interpolated pre-integrated lighting performs with 3.2 Hz. Volumetric shadowing combined with interpolated pre-integrated lighting achieves 2.7 Hz. Tone mapping approximately costs an additional frame per second. All measurements where done on a 2.8 GHz Pentium IV machine with an NVidia GeForce 6800 graphics card.

The architecture of our framework makes it very easy to incorporate new components and features. From our experience, setting up a new stage in the volume rendering pipeline and providing it with all relevant data from the shared state becomes a trivial task that takes as little as a few minutes. We successfully used the framework during our research on realistic volume graphics, which required frequent changes to the rendering process.

8 Discussion

Our proposed shared application state is related to the mediator design pattern [8]. The mediator pattern also decouples components from each other by making them dependent only on the mediator and routing requests between components. The major difference between the two patterns is that the shared state does not concentrate interaction protocols and thus is not likely to become monolithic and complex itself. At the same time this makes the shared state appropriate only when interaction protocols are not overly complex, which is the case within the volume rendering pipeline where we do have many yet well defined dependencies.

Despite all advantages of the shared state, the achieved flexibility comes at a price. Certain drawbacks should not remain unnamed. The first issue is one of redundancy: For reasons of safety the shared state stores a copy of each state variable, which consumes additional memory. In our implementation the shared state allocates memory for state variables in the order of 1 kb, which we consider tolerable.

Furthermore, the issue of ownership of state variables remains unaddressed. In the presented setup any class may modify a state variable, regardless whether this makes sense conceptually or even is desirable. As a very simple solution, storing a pointer to the owner of each state variable appears imaginable. Classes would then register as owners and the setter function would be extended to demand the this-pointer of the calling class in order check for ownership before allowing to set a state variable. Of course, more sophisticated policies might also be exerted. For the sake of simplicity and due to the scientific nature of this work we have chosen to rely on programmer discipline.

Another pitfall is type safety and data validity. To make the getter and setter functions independent of the type of the state variable, a void pointer to the location of the value is passed. This ultimately makes type checking impossible. Moreover, if an invalid address is passed, runtime stability is put at risk when data is read from, or written to, this address. Templated call-by-value interface functions using typelists as shown by Alexandrescu [1] might be a safer but also more involved alternative.

Lastly, the set of state variables is static, which may be perceived as a disadvantage. On the other hand, allowing to dynamically register state variables entails classes having to check for the existence of any variable they depend on, thereby nullifying the advantages gained through decoupling.

In terms of software engineering one may certainly argue about the elegance of our implementation of the shared state. Since the envisioned field of application is scientific research where safety and stability is not the main focus we consider this as tolerable. We agree that for production software safer implementations should be found.

9 Conclusion and Future Work

We have presented a framework for modern volume rendering applications implementing the volume rendering pipeline on the GPU. While other frameworks implement a broad range of features, they are very complex to understand and not easily extended. Our proposed framework with its novel architecture is explicitly designed to be extensible with minimum effort, making it very easy to incorporate new features and rendering algorithms. To prove the applicability of our architecture we have implemented a volume renderer with state-of-the art features including high dynamic range tone mapping, which has previously not been applied to volume visualization.

We offer the source code of our framework as a starting point for future research, especially for researches that are new to volume rendering. We hope that with the simple architecture of our framework, researchers will feel encouraged to incorporate and exchange source code directly, making new approaches easily accessible by others.

Possible future work could include the implementation of advanced techniques within our framework, e.g., deferred filtering [14], deferred shading [10], multidimensional transfer functions [12], multiple scattering of light [13], and volume clipping [28].

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] P. Bhaniramka and Y. Demange. OpenGL Volumizer: A toolkit for high quality volume rendering of large data sets. In *2002 Symposium on Volume Visualization and Graphics*, pages 45–53, 2002.

[3] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Symposium on Volume Visualization*, pages 91–98, 1994.

[4] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture mapping hardware. Technical Report TR–93–027, Department of Computer Science at the University of North Carolina, Chapel Hill, 1993.

[5] P. Debevec, E. Reinhard, G. Ward, and S. Pattanaik. High dynamic range imaging. In *SIGGRAPH Course Program, Course 13*, 2004.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[7] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *2001 Eurographics / SIGGRAPH Workshop on Graphics Hardware*, pages 9–16, 2001.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] D. P. Greenberg, K. E. Torrance, P. Shirley, J. Arvo, J. Ferwerda, S. N. Pattanaik, E. Lafortune, B. Walter, and B. Trumbore. A framework for realistic image synthesis. In *Proceedings of SIGGRAPH 1997*, pages 477–494, 1997.

[10] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Eurographics 2005*, 2005.

[11] A. Kaufman and K. Mueller. Overview of volume rendering. In C. D. Hansen and C. R. Johnson, editors, *The Visualization Handbook*, pages 127–174. Elsevier, 2005.

[12] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[13] J. Kniss, S. Premoze, C. Hansen, P. Shirley, and A. McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, 2003.

[14] J. M. Kniss, A. Lefohn, and N. Fout. Deferred filtering: Rendering from difficult data formats. In M. Pharr, editor, *GPU Gems 2*, pages 669–677. Addison-Wesley, 3 2005.

[15] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *IEEE Visualization '03*, pages 287–292, 2003.

[16] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH 1994 Conference Proceedings*, pages 451–458, 1994.

[17] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 9(3):29–37, 1988.

[18] E. B. Lum, B. Wilson, and K.-L. Ma. High-quality lighting and efficient pre-integration for volume rendering. In *EG / IEEE TCVG Symposium on Visualization '04*, pages 25–34, 2004.

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

[20] The OpenQVis Project Web Page. `http://openqvis.sourceforge.net`, 2002.

[21] OpenSG Web Page. `http://www.opensg.org`, 2003.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[23] H. Pfister. Hardware-accelerated volume rendering. In C. D. Hansen and C. R. Johnson, editors, *The Visualization Handbook*, pages 229–258. Elsevier, 2005.

[24] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, 21(3):267–276, 2002.

[25] 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. In *2000 Eurographics / SIGGRAPH Workshop on Graphics Hardware*, pages 109–118, 2000.

[26] S. Röttger, S. Guthe, D. Weiskopf, T. Ertl, and W. Straßer. Smart hardware-accelerated volume rendering. In *EG / IEEE TCVG Symposium on Visualization '03*, pages 231–238, 2003.

[27] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Volume Graphics '05*, pages 187–195, 2005.

[28] D. Weiskopf, K. Engel, and T. Ertl. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.

[29] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH 1998 Conference Proceedings*, pages 169–179, 1998.

[30] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, 24(4):367–376, 1990.

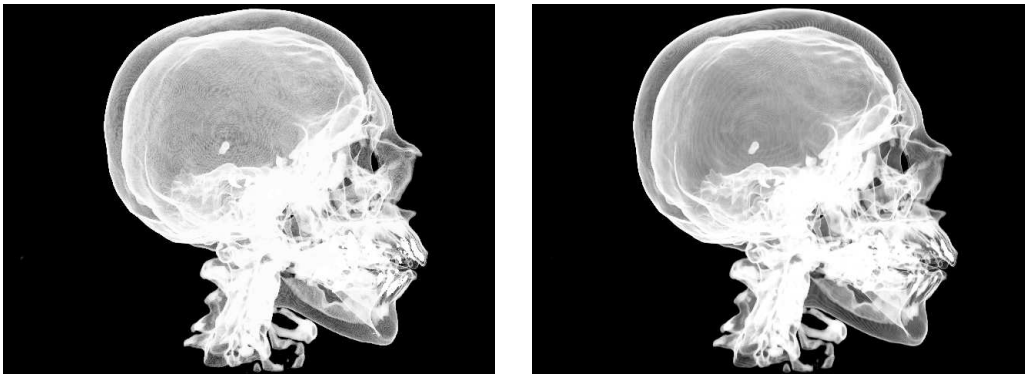

Figure 4: At the same sampling distance, pre-integrated classification (right) produces far fewer sampling artefacts than traditional slicing (left).

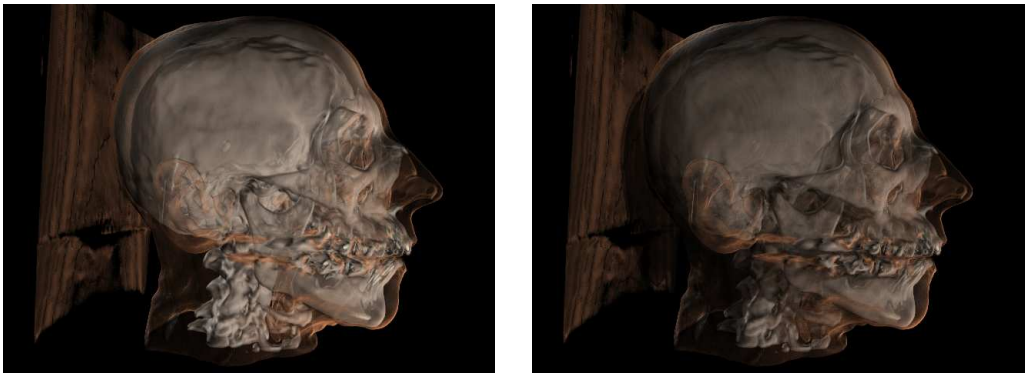

Figure 5: Volumetric shadows (right) add important visual cues and produce visually pleasing images. For comparison the same scene is depicted without shadowing on the left.

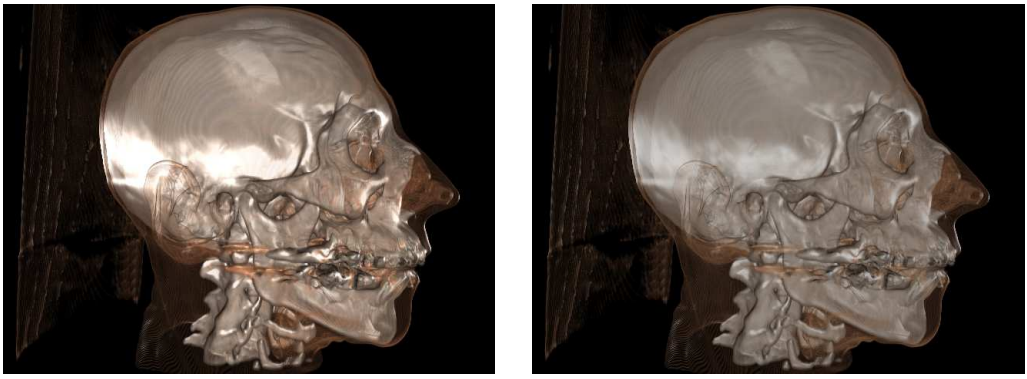

Figure 6: With Reinhard's global tone mapping operator, an image with overly bright regions (left) is mapped to a displayable range at interactive rates (right).