A Modular GPU-based Direct Volume Renderer for Visualising Scalar and Multi-dimensional Data

CompSci 780 project report by

Felix Manke

ID: 3892366, UPI: fman020

Under the supervision of Burkhard Wünsche

Graphics Group

Department of Computer Science

The University of Auckland

Auckland, 23rd September 2007

Direct Volume Rendering (DVR) is a technique for displaying volumetric data sets without generating intermediate representations. Because of the increasing capabilities of computer hardware and consumer graphics accelerators, DVR can now be performed interactively and is becoming more and more popular. Originally, DVR has only been applied to scalar data sets, such as Computed Tomography scans. However, the dramatic advances in biomedical imaging and other research areas like meteorology and thermodynamics have resulted in ever increasingly complex volumetric data sets. Novel DVR algorithms are therefore required to deal with non-scalar and multivariate data at interactive frame rates. The ability to combine several data sets is also demanded, since insights can often be gained by comparing the difference or correlation between data sets.

Current research in DVR mainly focuses on either improving the efficiency of rendering algorithms or the visual appearance of the renderings. Even though both aspects are of great importance, the efforts often lead to very specialised solutions that are only applicable in a certain domain. So far, no tool is available that works completely hardware-accelerated and, at the same time, supports the interactive exploration of arbitrary data as well as the easy integration of new data types and formats.

In this report, we present solutions to develop a flexible and modular direct volume rendering framework which enables the user to interactively explore high-dimensional and multiple data sets on programmable consumer graphics cards. We discuss how the framework can be used to incorporate the latest specialised direct volume rendering algorithms and how it can be adapted on the fly to change a visualisation or to visualise new variables, such as the difference between two scalar fields, in order to gain insight into the given data.

The capabilities of the framework are demonstrated by two simple case studies and the efficiency and effectiveness of the framework is evaluated.

Contents

List of Figures

List of Tables

1 Introduction

In many scientific research areas like medical imaging, geology, or thermodynamics, three-dimensional (3D) discrete data arrays are generated or acquired and need to be analysed. They usually sample some continuous field. The visualisation of the internal structures or objects of the data helps researchers to investigate and understand the data [Drebin et al., 1988].

Since there are many ways how the volumetric data is produced, there is a large variety of data formats and representations. Moreover, the sample points themselves can potentially be of arbitrary dimensionality and type. In medical imaging, the dimensionality of the individual sample points range from simple scalar radiodensity data acquired by Computed Tomography (CT) over Positron Emission Tomography (PET) and possibly Magnetic Resonance Imaging (MRI) to 3x3 diffusion tensor matrices in Diffusion Tensor Imaging (DTI). In meteorological weather simulations, different properties of air — like temperature, humidity, or air pressure — also result in multivariate data [Kniss et al., 2002b].

Especially when dealing with higher-dimensional data, new information has often to be derived before the volume is displayed. In DTI for example, diffusivity and anisotropy describe characteristic movements of water in different types of tissue. These entities have to be derived from the tensor matrix. Another point of interest is the visualisation of differences between two data sets acquired at different times, which can improve the understanding of how the data correlate and where it changes over time.

The field of volume rendering in general is not new, algorithms that are still in use have already been developed in the 1980s [Lorensen and Cline, 1987, Levoy, 1988]. With the upcoming of relatively inexpensive graphics accelerators in the end of the 1990s, volume visualisation in real-time became possible without the cost of highly specialised rendering hardware. With the introduction of programmable graphics processing units (GPUs), research was further intensified — as in the entire computer graphics area.

1.1 Objectives of the project

In this research project, we investigate which is the optimal design for a GPU-based volume renderer that is capable of both rendering at interactive frame rates as well as dealing with many different types of volume data and visualisation methods. The aim is to support an easy integration of new data formats into the rendering framework, regardless of their representation or their sample point dimensionality. Additionally, user-defined visualisations of the internal structures have to be possible, as the definition of objects of interest requires knowledge of the concrete domain. We also investigate mechanisms to easily allow the derivation of

new values out of existing data. Hereby, all required computation is to be performed on the GPU. The support for deriving new entities is essential, because especially multi-dimensional data can often not be displayed directly. Instead, meaningful information has to be extracted first (a good example is Diffusion Tensor Imaging data which can be visualised by computing the diffusion anisotropy and mean diffusivity). Another application of these mechanisms is the combination of several input volumes to derive information about how they differ.

Note that it is not the aim of this work to implement a fully featured volume renderer that can interpret many different data formats. Neither is the renderer supposed to implement a range of visualisation techniques. Instead, our main goal is to create a framework that enables the developer to easily extend the functionality as needed. Nevertheless, we have implemented several popular DVR techniques for our case studies in order to demonstrate the flexibility and power of our framework.

1.2 Outline of this report

In the next section, related work will be discussed. Afterwards, principles of the programmable graphics pipeline will be outlined in section 3. We also give an overview of high level programming languages that are currently used to write programs that are executed on the GPU. Chapter 4 discusses principles of direct volume rendering in general and GPU-based volume rendering algorithms in particular. In chapters 5 and 6, we explain how to realise the goals of the research project. First, the requirements of a modular framework are identified by showing differences in the DVR process for three scenarios. Then, the design of individual components is derived and our proposed implementation is presented. In section 7, we demonstrate the applicability of our framework with two case studies and discuss the results of performance tests. A conclusion and an outlook are drawn in the last chapter.

2 Related work

Many papers that relate to direct volume rendering discuss new rendering techniques and algorithms that aim to improve the visual quality of the rendering or increase the rendering speed. This is not surprising, as visual quality and efficiency is always an important issue. The papers might for example demonstrate new acceleration techniques [Rezk-Salama et al., 2000, Purcell et al., 2002, Krüger and Westermann, 2003], or advanced visualisation methods [Kniss et al., 2002c, Kindlmann et al., 2003]. Techniques for dealing with higher-dimensional data have also been proposed [Kindlmann et al., 2000, Kniss et al., 2002b, Kniss et al., 2002a]. For the purpose of demonstrating their achievements, the researchers usually implement renderers that are tailored to their needs. Depending on the domain, the demo applications can deal with scalar or multivariate data. Nevertheless, these implementations will likely not be flexible enough to support different data types and formats, as this is not their purpose.

Other publications discuss how to implement more general rendering frameworks. Stegmaier et al. present a framework that implements a volume ray casting algorithm [Stegmaier et al., 2005]. The framework allows to define visualisation techniques by implementing fragment shaders. Generally, the application is restricted to scalar volume data, for which a gradient can be pre-computed. Other data is not supported and other entities cannot be derived. Bruckner and Gröller discuss an interactive framework for non-photorealistic volume illustrations [Bruckner and Groller, 2005]. Their system, the "VolumeShop", allows to interactively explore and annotate scalar data. Hereby, more than one volume can be rendered simultaneously. Visualisation and shading effects are defined by adjusting a two-dimensional transfer function. The application uses hardware-accelerated rendering and *C for graphics (Cg)* as high level shading language.

Teem is a very flexible collection of open-source C libraries that offers a variety of functionality to process and visualise volumetric data [Kindlmann, 2003]. It supports data of arbitrary dimensionality and let the user specify which entities to derive and how to map the data to colours and opacities (by defining multi-dimensional transfer functions). However, Teem is not interactive, as it has to be run on command line.

The Silicon Graphics, Inc. developed the OpenGL Volumizer™, a powerful C++ toolkit for high-quality volume renderings of large data sets [Bhaniramka and Demange, 2002]. The API uses hardware-accelerated rendering techniques and implements many techniques for working with very large data (like volume roaming [Bhaniramka and Demange, 2002] and multi-resolution volume rendering [LaMar et al., 1999]). Shaders may be defined that specify the visualisation. As a toolkit, the OpenGL Volumizer™ is not an application on its own, but depends on other frameworks like the Visualization Toolkit (VTK) [Schroeder et al., 1998].

3 The programmable graphics pipeline

Before discussing GPU-based direct volume rendering algorithms in the next chapter, the *programmable graphics pipeline* is introduced to give a general understanding of how current graphics accelerators process data. In the second section of this chapter, concepts of high level shading languages are outlined. Hereby, differences to conventional general-purpose programming languages are emphasised.

A thorough discussion of principles of graphics hardware and its programming is considered to be very important, because the project aims to investigate GPU-based methods of volume rendering and the utilised concepts differ considerably from general CPU programming. In order to achieve the modularity and flexibility we demanded in the introduction, advanced GPU technologies will be utilised. To be able to discuss the design decisions of the developed rendering framework in section 5, a good understanding of the hardware architecture and programming concepts is necessary.

3.1 Introduction to the graphics pipeline

Figure 3.1 illustrates the data processing of a general graphics pipeline. This simplified model concentrates on the components relevant to our research. A more detailed description can be found in the literature (see for example [Foley et al., 1996, Watt, 1993, Engel et al., 2004, Fernando and Kilgard, 2003, Rost, 2006]).

A graphics application running on the CPU can access the graphics card by calling functions of a 3D API (like OpenGL® or Microsoft® Direct3D®). Apart from setting render states of the hardware, the application can also upload vertex data or texture maps that are to be displayed. Note that vertices can also be sent to the graphics card during the rendering, but since this requires the transmission via the system bus in every rendering loop, this method is generally slower. For objects with unchanging geometry it is therefore preferable to upload the vertex data once before entering the rendering loop [Akenine-Möller and Haines, 2002].

The rendering pipeline used in standard graphics APIs and graphics hardware was developed for drawing geometric primitives (like points, triangles, or polygons) which are defined by vertices. For each vertex, attributes like position, colour, the normal vector, or several texture coordinates can be assigned. In the vertex processing unit, the incoming vertices are manipulated. Traditionally, the transformations into the world or camera coordinate system as well as lighting operations are performed. Thereafter, the vertices are projected onto the viewing plane and the polygons are rasterised. During the rasterisation process a transition

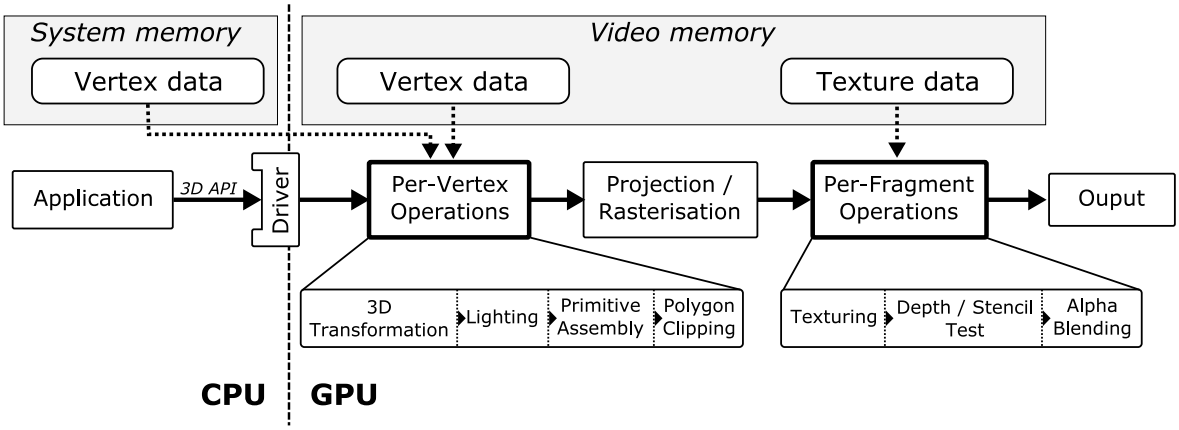

Figure 3.1: Simplified model of the general graphics pipeline. The per-vertex and per-fragment operations are the most important components for our research and are shown in more detail.

from a description in 3d space to a description in a pixel-raster is made. Hereby, the vertex attributes are linearly interpolated over the entire surface [Segal and Akeley, 2006].

The generated fragments (imaginable as pixels with a depth coordinate and the additional attributes of the vertices) are further manipulated by the fragment processor. At this stage, texture mapping operations are performed. If the fragment passes the z-buffer test (this usually means, if the fragment is closer to the viewer than every other fragment at the same position that has been processed before), the fragment is written to the target buffer and the rendering pass is finished. Depending on the render state settings, the new colour might be alpha-blended (i.e. modulated or mixed) with the existing value [Engel et al., 2004].

In order to optimise the fragment processing, newer graphics hardware usually implements an *early z-test*. In this case, a z-buffer test is also performed *before* the fragment is processed. If the test fails, the fragment is immediately discarded. Without this improvement, the fragment will be processed in any case [Sander et al., 2005].

3.2 Programmable shading units

In March 2001, the NVIDIA® Corporation released with the GeForce3™ the first consumer graphics hardware whose vertex and fragment processing units could execute custom micro programs, commonly called *shaders* [NVIDIA® Corporation, 2001]. Until then, graphics accelerators only allowed to configure pre-defined operations of the *fixed-function graphics pipeline*. These operations allow coordinate transformations and lighting calculations on the vertex stream (a per-vertex Phong lighting model[1] was implemented on graphics hardware). After rasterisation, the fixed-function graphics pipeline gives only relatively limited control over how the fragments are processed. However, by setting render states, it is possible to specify alpha-blending, texturing as well as stencil and depth testing.

[1] The well-known empirical local illumination model was introduced by Bui T. Phong in 1975 [Phong, 1975].

	1.1 (2001)		**2.0 (2002)**		**3.0 (2004)**		**4.0 (2006)**	
	VS	PS	VS	PS	VS	PS	VS	PS
Instructions	128	12	256	96	≥512	≥512	≥64000	
Input registers	16	6	16	10	16	10	16	32
Constant registers	96	8	256	32	256	224	16x4096	
Temporary registers	12	2	12	12	32	32	4096	
Flow control	n/a		static		static/dynamic		dynamic	
Textures	n/a	8	n/a	16	4	16	128	
Render targets	1		4		4		8	

Table 3.1: Overview of the main requirements and capabilities of vertex shaders (VS) and pixel shaders (PS) defined by different shader models. Adopted from [Blythe, 2006].

In contrast, the programmable graphics pipeline lets the programmer upload shader programs for either the vertex processing unit (called *vertex shader* or *vertex programs*) or the fragment processing unit (called *fragment shader* or *fragment programs*). In the pipeline the vertex processing unit replaces the 3D transformations and lighting calculations. The fragment processing unit provides a new way of manipulating each generated fragment. Even though fragment programs also have to perform texturing, the programmable fragment processing units offer a significantly greater flexibility than the conventional fixed-function pipeline. The ability to manipulate the fragment data not only allows conventional rendering, but also hardware accelerated image processing or general purpose computing [Akenine-Möller and Haines, 2002, Luebke et al., 2004].

The programmable shading arithmetic logic units (ALUs) are single instruction multiple data (SIMD) stream processors with highly specialised instruction sets. Operations can be performed on up to four floating-point components in parallel. The loaded programs are generally executed once per vertex or once per fragment. Since each element of the input stream is processed isolated, it is not possible to access data generated in a previous execution. This isolation allows for enormous parallelism and a superscalar architecture (today up to 320 parallel stream processing units are available on GPUs [Advanced Micro Devices, 2007]).

Over the past years the complexity of the shading units and the maximal length of shader programs have constantly increased (see table 3.1). To account for this, different *shader models* have been specified. Each model defines the instruction set of the ALU, the maximal length of the programs, the number of variables and constants, and other parameters. Together with the instruction set, mnemonics in an assembly language are defined [Microsoft® Corporation, 2007].

In early shader models, loops, branching, or conditional execution was not implemented. With shader model 2.0 *static flow control* was available, which only allowed to use variables in constant registers for conditions (in `if` statements and loops). Moreover, nested loops were not available. The shader model 3.0 introduced limited nesting for static loops and, more importantly, *dynamic flow control*. Since then, also temporary registers can be used in conditions [Engel, 2003].

Another feature that became available with shader model 2.0 is *off-screen rendering*. Instead of writing the result of the rendering pass into the frame buffer, a texture is used as render target. Today's graphics cards support to render into more than one off-screen render target [Engel, 2003].

3.3 High level shading languages

In order to avoid writing shader programs in an assembly language, several high level languages for real-time shader programming have been developed. The three most popular languages are currently: The *OpenGL® Shading Language* (GLSL) as part of the OpenGL® Standard 2.0, *C for graphics* (Cg), developed by the NVIDIA® Corporation, and the Microsoft® *High Level Shading Language* (HLSL). Syntactically, they are very similar to each other, as they are all based on ANSI C/C++ as well as on the RenderMan® Interface Specification [Pixar Animation Studios, 2005]. In fact, Cg and HLSL have been developed in close collaboration and have therefore almost the same syntax [Fernando and Kilgard, 2003]. To best suit the specialised graphics domain, the shading languages differ from C and C++ as general purpose programming languages.

In our research project, we will use Cg as high level shading language. HLSL is only available in conjunction with the Microsoft® Direct3D® API, and GLSL lacks some advanced features that will be used extensively for achieving modularity. Most importantly, GLSL does not support interfaces, include directives, semantics, and annotations as they will be described below. Because of these shortcomings of GLSL only the concepts of Cg will be discussed in the following. Unless otherwise noted the information is based on [Fernando and Kilgard, 2003] as well as the Cg language specification [NVIDIA® Corporation, 2005a].

Data types and variables

The most important data type in shader programming is the 32 bit floating-point type `float`. To potentially increase performance, Cg also defines a 16 bit equivalent `half`.

Since GPUs can process data in parallel, special vector data types are introduced (for example, `float3` represent a three component floating-point vector). The components of vector data types can be addressed using a dot operator as in C/C++. Since variables usually represent points, vectors, or colours, the components are labelled `.x`, `.y`, `.z`, `.w` or `.r`, `.g`, `.b`, `.a` respectively. To select multiple components, graphics hardware allows *swizzling* without performance loss and thus, swizzling is also part of the high level shading languages (for example `.rgg` or `.yyxw`).

In addition to vectors, matrix data types are defined. They can have up to four rows and columns. In Cg, matrices are defined by the base data type and the number of rows and columns (for example `float3x2`, `half4x3`, or `float4x4`).

To access texture maps, the `sampler` data type is used. A sampler is associated with exactly one texture object. The binding is done by the application. As with vectors and matrices,

different sampler types are available for textures with different dimensionality (for example `sampler2D` or `sampler3D`). In addition to the raw texture data and the texture coordinates, render states are associated with a sampler object to specify how to sample texels.

All shading languages allow to define structures (using the `struct` keyword) as a collection of variables, similar to the C equivalent. In contrast to GLSL, Cg additionally allows to define functions within structures as in C++. However, object-oriented concepts such as polymorphism and inheritance are currently not part of any shading language.

The different input registers in the shader ALUs have different meanings. Moreover, variables are initialised differently depending on the type of register they are stored in. To ensure a correct compilation of a program, different type modifiers are defined in the shading languages.

Constant variables within a shader are already defined at compile-time. To allow constant folding, the type modifier `const` can be used. The variables will be stored in the constant register [NVIDIA® Corporation, 2005b].

Uniform variables are those whose value is set by the application before the rendering pipeline is entered. They are named "uniform" since their value does not change during the whole execution of the shader program.

The term *varying variable* is used to identify variables that store the attributes of vertices and fragments (that is, it depicts variables stored in the input registers). Thus, they change with each processed element. Note that Cg does not define a type modifier keyword for varying variables.

To encode the meaning of varyings — in other words, to associate the data in the stream with shader variables — Cg uses *binding semantics* to associate element data with variables. Semantics are imaginable as annotations to a variable with a predefined name and meaning (for example `POSITION` or `TEXCOORD2`). To pass data from vertex to fragment shader both programs must associate variables with the same semantic. The name of the variable, however, is irrelevant. Semantics will be discussed in more detail later in this section (see page 9).

Functions

Functions are defined as in C or C++. The signature is given by the name and the parameter arguments. Parameters may be either input parameters (call-by-value), output parameters (call-by-result), or input-output parameters (call-by-value/result). The corresponding type qualifiers are `in`, `out`, and `inout`.

Every vertex and every fragment shader program needs an entry point. A vertex shader must at least write the position of a vertex. A fragment shader must at least output the colour of a fragment.

Predefined functions

The shading languages predefine commonly used functions. They are referred to as *built-in functions* in GLSL, *standard library functions* in Cg, and *intrinsic functions* in HLSL. Naturally, the provided functionality is highly adapted for graphics applications. Coarsely, the predefined functions can be categorised in the following classes:

General Math: like `clamp`, `pow`, `exp`, `log`, or `sqrt`.
Trigonometry: like `sin`, `cos`, `atan`, or `degrees`.
Geometry: like `normalize`, `dot`, `cross`, `length`, or `reflect`.
Texture access: like `tex2D`, `tex3D`, or `texCUBE`.
Others: like `lerp` or `noise`.

Other concepts of the Cg language

Because Cg is not part of any graphics API but a proprietary language, it must provide an external API and library that has to be included in the application. Additionally, Cg requires a runtime environment to allow the compilation and link of Cg shader programs during the execution of the graphics application. In the rendering pipeline (figure 3.1), the Cg runtime is located as an extra layer between the application and the graphics card driver. To make Cg available for both major graphics APIs (OpenGL® and Direct3D®) two different API-specific Cg runtimes are available. Note that the Cg compiler translates the source code in different ways depending on the API that is used. The final compilation into machine code is then done by the API specific driver [Fernando and Kilgard, 2003].

Cg allows to include additional files, using the `#include` pre-processor directive. This feature, which was omitted in GLSL for the sake of simplicity [Rost, 2006], will become important when discussing how to integrate use-defined Cg code into the rendering framework (see section 6.2).

As already mentioned, Cg uses *binding semantics* to identify variables as an alternative to the identification by name. There are predefined semantics for associating input and output variables with hardware registers. Additionally, user-defined semantics can be specified and used by the application to get handles to the variables. The syntax for semantics is as follows [NVIDIA® Corporation, 2005a]:

```
<data-type> <variable-name> : <binding-semantic>;
```

Another important feature of Cg is the concept of *interfaces*, which neither GLSL nor HLSL support. An interface is an abstract data type that only defines the function signatures which can then be implemented by structures. The syntax is much like in C#, but there is no real polymorphism in Cg: Shader programs can define and use variables of the abstract interface type, but *before* the shader is compiled, the data type of a structure that implements the interface must be specified. During the execution, the implementation of this specified data type is used. If another implementation (i.e. implementing structure) is to be

used, the shader must be recompiled. The NVIDIA® Corporation calls this "compile-time polymorphism" [NVIDIA® Corporation, 2005b]. Even though this expression bizarrely contradicts the definition of polymorphism in object-oriented paradigms, it depicts the concept nicely.

The CgFX file format

The NVIDIA® Corporation defined a file format (again in collaboration with Microsoft®) that according to the authors represents "complete effects and appearances" [Fernando and Kilgard, 2003]. Since shader programs are executed with a certain state of the graphics card, an effect describes the setup of the rendering pipeline including vertex and pixel shader programs as well as render states and potentially multiple rendering passes [Fernando and Kilgard, 2003].

The Cg runtime provides the API to work with CgFX files. To simplify the use, render states are automatically set and restored by the Cg runtime during the rendering.

An effect is identified by a named `technique` that contains one or more `pass`es, each of which specifies whether to use the fixed or programmable stages as well as render state settings such as blending. For each pass, the shader program and the compiler profile has to be specified if a programmable stage is to be used.

In addition, the CgFX file format introduces a new mechanism to give the application hints and further information about the elements defined in the file. Items like variables, functions or techniques may have *annotations* attached. Annotations have a data type, a name and a value. They are not interpreted by the Cg runtime itself, but can be read by the application [NVIDIA® Corporation, 2005b]. An example for annotations of a variable is:

```
texture volume
<
   string FilePath = "volume.raw";
   int    Width    = 128;
   int    Height   = 128;
   int    Depth    = 128;
   bool   Scalar   = true;
>;
```

4 Direct Volume Rendering

In this chapter, direct volume rendering (DVR) algorithms will be discussed. At first, a brief overview of volume rendering in general is given. Then, the fundamental model for direct volume rendering, the *emission-absorption model*, is introduced. Knowledge about this model is required because all DVR algorithms are based on an optical model which describes how light interacts with the material represented by the sampled data. The second half of the chapter focuses on GPU-based algorithms that are used today.

4.1 Overview of volume rendering

Volume visualisation techniques can be classified as being either *indirect* or *direct*. The former class avoids the computationally expensive rendering of the data set itself by representing features of the data set using graphical representations. An example is object boundaries defined by *iso-surfaces* which can be represented by polygonal surfaces and displayed using conventional rendering techniques. A well-known indirect volume rendering algorithm is called *marching cubes*, introduced by Lorensen and Cline in 1987 [Lorensen and Cline, 1987]. The iso-surface extraction — essentially a binary thresholding technique — is problematic. Once the polygonal iso-surface is generated, most of the information carried by the volume data is lost. In fact, it is just the aim of the indirect techniques not to integrate the data set during the rendering.

Direct Volume Rendering algorithms on the other hand directly use the volume data during the rendering. They do not rely on a polygonal approximation. A further distinction can be made by means of how the DVR algorithms perform the projection onto the final image plane. *Image-order* algorithms come from the pixels of the image and calculate the final colour at this location. Volume ray-casting, developed by M. Levoy, is an example of an image-order algorithm [Levoy, 1988]. For each pixel, a ray that originates at the viewer's position and travels through the pixel's centre is traced through space. Along the ray, the volume data set is resampled and the gathered light accumulated. In the end of this chapter, GPU-based ray casting is described (see section 4.3.2).

In contrast to image-order approaches, *object-order* algorithms project the volume data (i.e. the "object") onto the image plane. The shear-warp factorisation is one of the first algorithms of this category that could be implemented at almost interactive frame rates by projects the volume slice by slice [Lacroute and Levoy, 1994]. The algorithm uses three stacks of slices, each of which holds slices perpendicular to one of the volume's principal axes (see

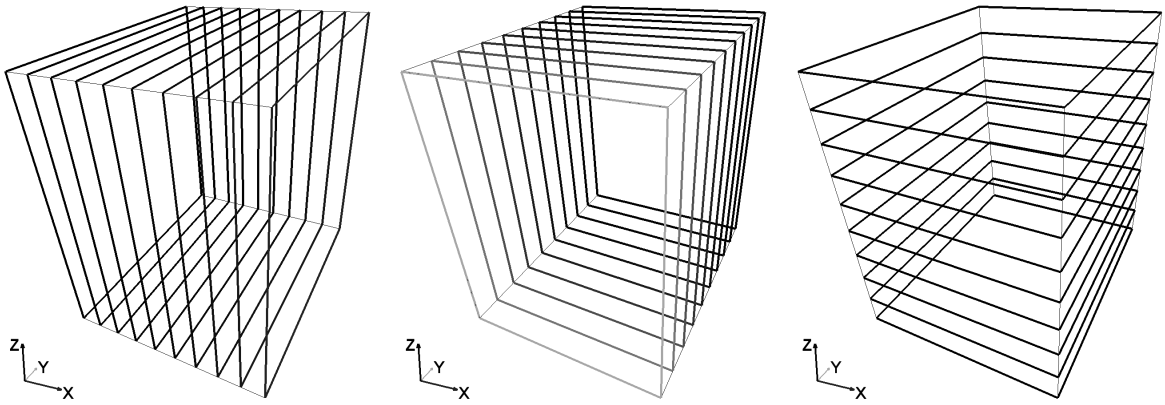

Figure 4.1: The three stacks of slices, each of which is perpendicular to one of the object's principal axes (left: x-axis, middle: y-axis, right: z-axis).

figure 4.1). During the rendering, the stack of the axis that is most parallel to the viewing direction is selected. To account for non-orthogonal viewing angles, the slices are sheared, scaled and warped appropriately. Even though no rays are cast, the algorithm also models viewing rays, as they directly determine the transformations of the slices.

4.2 The emission-absorption model

DVR algorithms directly operate on the volume and its voxels. During the rendering, an *optical model* is evaluated that describes the light interaction on a voxel basis in terms of absorption, emission, and possibly scattering and shadowing. Each voxel is considered to describe the interaction that happened at the corresponding point of the continuous space that was sampled. The volume can also be considered to represent a gaseous material, whose absorption, emission and other properties are modelled by the optical model. The mapping from the voxel's value to the optical properties is called *classification* and is usually done using a *transfer function* (see below) [Engel et al., 2004].

The goal of every DVR algorithm is to integrate the light interactions along the viewing rays that pass through the pixels of the final image. The interactions are modelled by the underlying optical model. For a parameterised viewing ray $r(t) = P_0 + t\vec{d}$, P. Sabella introduced the following integral in 1988, assuming an optical model that accounts for absorption and emission only [Sabella, 1988]:

$$C = \int_0^\infty c(t) \cdot e^{-\tau(0,t)} \, \mathrm{dt}, \tag{4.1}$$

where

$c(t)$ is the emissive colour and

$\tau(0,t)$ is the optical depth in the interval $[0,t]$.

This fundamental integral is usually called the *volume rendering integral*[2]. In [Max, 1995], N. Max gives a detailed discussion about several different optical models.

In equation 4.1, $c(t)$ describes the light that is emitted at position t. The optical depth originates from the fact that light that is travelling through the volume is attenuated by absorption or scattering due to the particles of the gaseous material. According to Sabella, the expected number of particles in an interval $I = [t_1, t_2]$ is given by:

$$\tau(t_1, t_2) = \int_{t_1}^{t_2} \kappa(u)\,\mathrm{du},$$

which is exactly the optical depth. $\kappa(u)$ is the particle density of the material at u (also called absorption coefficient [Engel et al., 2004]). Note that the term $e^{-\tau(0,t)}$ in the volume rendering integral is the approximation of the probability that the light is not scattered by other particles on its way. This probability actually gives the overall attenuation of the light along the viewing ray [Sabella, 1988].

To numerically solve the volume rendering integral defined in equation 4.1, each ray is sampled at discrete points. Let Δt be the sampling distance between two subsequent sample points. The emissive colour and the absorption coefficient of the i'th ray segment are then approximated by:

$$c(i) \approx c'_i = c(i \cdot \Delta t)\Delta t$$

and:

$$\kappa(i) \approx \kappa'_i = \kappa(i \cdot \Delta t)\Delta t,$$

The optical depth for the ray is likewise approximated by:

$$\prod_{i=0}^{t/\Delta t} e^{-\kappa'_i}.$$

Using these approximations, the discrete form of the volume rendering integral is finally given by [Sabella, 1988]:

$$C' = \sum_{i=1}^{n} c'_i \prod_{j=1}^{i-1} e^{-\kappa'_j} = \sum_{i=1}^{n} c'_i \prod_{j=1}^{i-1} (1 - \alpha_j). \tag{4.2}$$

Note that $e^{-\kappa'_i}$ is the *transparency* of the volume between i and $i + \Delta t$. Similarly, $\alpha_i = 1 - e^{-\kappa'_i}$ depicts the *opacity* in that interval [Engel et al., 2004].

[2] The work of P. Sabella is partly based on a more complex volume rendering equation that also accounts for scattering. Even though this was introduced four years earlier by J. Kajiya and B. Von Herzen [Kajiya and Herzen, 1984], Sabella's integral is considered to be the fundamental volume rendering equation.

Transfer functions

One of the most important tasks in DVR is to map the volume data to properties of the optical model, as this mapping can be used to emphasize regions of interest, such as different tissue types. Therewith, the objects are classified by different material properties in terms of the optical model.

A *transfer function* is usually used to specify this mapping. One of the simplest types of transfer functions for a scalar volume is a one-dimensional (1D) function $m : \mathbb{R} \mapsto \mathbb{R}^4$ that maps each scalar value to a colour tuple with four channels: red, green, blue, and opacity. In the later discussion, this type of function will be called *colour-opacity* transfer function.

Multi-dimensional transfer functions allow significantly more control for the classification, because a combination of several parameters can be used [Kniss et al., 2002b]. In the case of scalar volume data, other dimensions like the magnitude of the gradient or the principal curvature at a voxel can be derived. Note that the gradient gives the change in slope per voxel and can be approximated by computing the partial derivatives in x-, y- and z-direction. For a scalar field $f(x, y, z)$ it is defined as:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right).$$

For a discretised field, the partial derivatives can be approximated using the central differences [Levoy, 1988]:

$$\begin{aligned} \frac{\partial f}{\partial x} &\approx \frac{f(x+1,y,z) - f(x-1,y,z)}{2}, \\ \frac{\partial f}{\partial y} &\approx \frac{f(x,y+1,z) - f(x,y-1,z)}{2}, \\ \frac{\partial f}{\partial z} &\approx \frac{f(x,y,z+1) - f(x,y,z-1)}{2}, \end{aligned}$$

with $x, y, z \in \mathbb{N}$. In [Mihajlovic et al., 2003], more advanced approximations of the gradient are discussed. Since the gradient changes rapidly at boundaries of objects of interest (for example at the transition from bone to soft tissue in a CT scan), the combination of a colour-opacity transfer function with the gradient is particularly plausible for scalar fields.

In the case of higher dimensional volume data the voxels themselves are already defined in a multi-dimensional domain. Therefore, the combination of different values of the volume can be used directly to classify components or objects of interest without the need to derive other properties [Kniss et al., 2002b].

Volume shading

Using the gradient as additional entity, local illumination models like the previously mentioned Phong model can be implemented. The normalised gradient, $\nabla f / ||\nabla f||$, serves as estimation of the surface normal at the location of each voxel. This method was already discussed by Levoy in 1988 [Levoy, 1988].

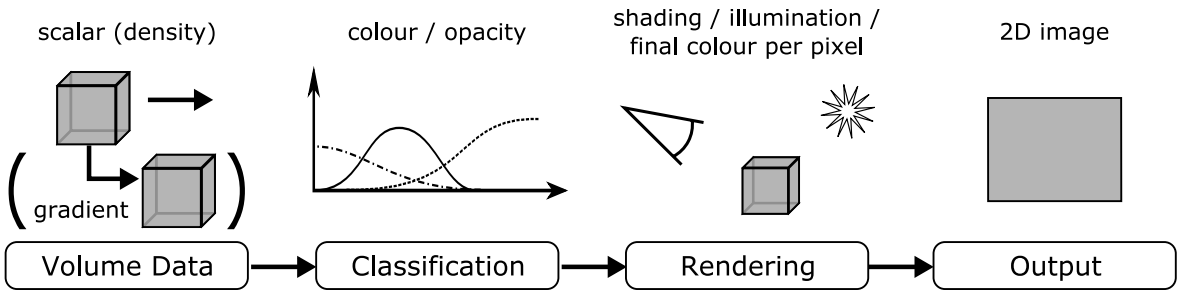

Figure 4.2: The steps of DVR for scalar volume data.

The general DVR process

To summarise the general DVR approach, figure 4.2 illustrates the rendering using a scalar volume. If shading is to be performed during the rendering, or if a 2D transfer function is to be used for classification, the gradient of the scalar volume has to be computed. During the classification, the transfer function (and indirectly the opacity model) is evaluated and therewith the colour and opacity determined. During the rendering, the coloured voxel might be illuminated according to a local illumination model and the volume is projected onto the image plane. The projection depends on the DVR algorithm that is used (image-order or object-order).

Figure 4.3 extends this pipeline by introducing more complex pre-processing steps, in which new entities (like in DTI) may be derived or different volume combined. Note that the transfer function might be higher-dimensional as well, depending on which entities are used for classification.

4.3 GPU-based DVR algorithms

In this section, algorithms for GPU-based direct volume rendering will be discussed. They can also be classified as either object-order or image-order approaches. In contrast to the history of the previous methods, hardware-accelerated object-order algorithms were developed first. A simple reason for this is that the graphics pipeline is also laid out in a strictly object-order

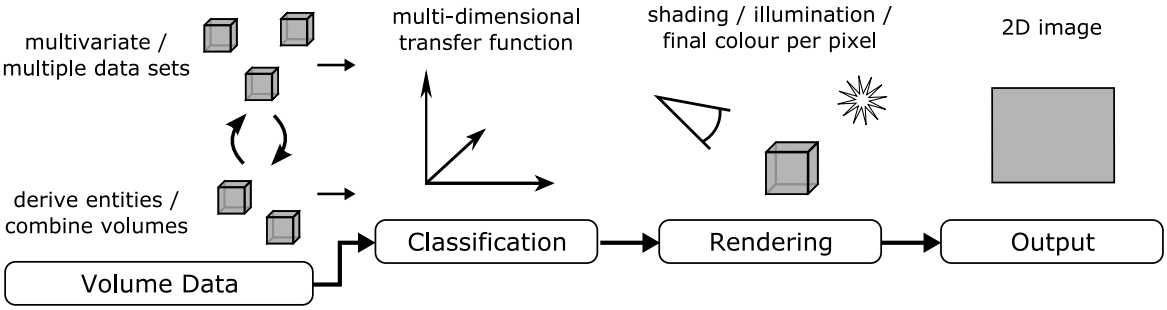

Figure 4.3: The steps of DVR for multivariate volume data.

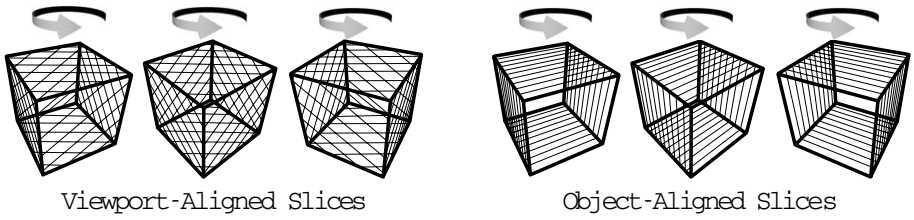

Figure 4.4: View-aligned (left) and object-aligned (right) slices used as proxy geometry for GPU-based object-order volume rendering. From [Rezk-Salama et al., 2000].

approach [Engel et al., 2004]. Additionally, the implementation of image-order algorithms requires more powerful programmable shading units that only became available in recent years.

4.3.1 Object-order rendering

Object-order DVR algorithms that run on the GPU make use of the texture mapping and blending capabilities of the graphics hardware. Since the graphics cards do not support volumetric rendering primitives, the volume needs to be rendered using a polygonal *proxy geometry*. Most often, these parallel planes are clipped against the bounding box of the volume. To avoid certain problems in case of perspective projection (see below), other proxy geometries — for example spherical shells — have been proposed [LaMar et al., 1999], but due to higher complexity these approaches are not as popular. In the following discussion, planes will be considered as proxy-geometry, but the basic principles also apply for other geometries.

Regardless on which proxy-geometry is used, the purpose is sampling the volume and therewith evaluating the discrete volume rendering equation (see equation 4.2). The sampling distance, Δt, for the reconstruction directly depends on how many slices are used. Whenever the transfer function is evaluated, the opacity obtained is defined with respect to the sample spacing of the volume, Δs, rather than the sampling distance of the reconstruction (note that Δs has been defined during data acquisition). To adjust the opacity, α, given by the transfer function, an *opacity correction* has to be applied [Lacroute, 1995]:

$$\alpha' = 1 - (1 - \alpha)^{\frac{\Delta t}{\Delta s}}. \tag{4.3}$$

As Rezk-Salama et al. point out [Rezk-Salama et al., 2000], two different types of planar proxy geometry can be considered: *view-aligned* slices, which are always perpendicular to the viewing direction, or *object-aligned* slices, which conceptually correspond to the slices defined by Lacroute and Levoy in the shear-warp factorisation. Figure 4.4 illustrates the two different types of planar proxy geometry.

The slices of the proxy geometry are rendered from back to front and blended together using the alpha blending capabilities of the graphics card colours [Engel et al., 2004].

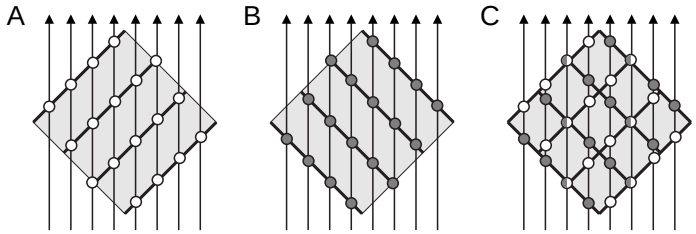

Figure 4.5: Switching the texture stack can cause visual artefacts, because the sampling locations change. Image (A) and (B) show the stacks between which is swapped. In image (C) they are blended together to illustrate the different sampling locations. From [Engel et al., 2004].

Rendering using object-aligned slices

Approaches that use object-aligned planar proxy geometry usually decompose the volume into 2D texture maps. Since GPUs are optimised for 2D texture fetches, the rendering is very fast. A major disadvantage is that three stacks of 2D textures have to be stored, one for each principal axis of the volume [LaMar et al., 1999]. Figure 4.1 shows the three stacks, each of which is perpendicular to one of the principal axes. Visual artefacts are introduced when switching stacks, because the sampling locations change slightly, as illustrated in figure 4.5.

If each slice of the proxy geometry addresses exactly one 2D texture, the rendering method directly corresponds to the shear-warp factorisation. Note that in this case the volume is sampled using bilinear interpolation. The image quality usually suffers from undersampling and the missing interpolation between slices. To increase the sampling rate and to implement a trilinear interpolation, it is possible to utilise the multi-texturing capabilities of graphics hardware. An arbitrary large number of planar slices can be used, each of which samples the two surrounding texture slices. Trilinear interpolation is achieved by linearly interpolating the two (bilinearly) sampled values [Rezk-Salama et al., 2000].

To avoid redundant storage of the volume in three texture stacks, the data can also be loaded into the video memory as a single 3D texture. Since today's hardware supports trilinear interpolation when sampling 3D texture maps, the result is comparable to the multi-texturing approach. A drawback of this method is that texture fetches into 3D textures are usually slower [Engel et al., 2004]. However, a comparison of the actual speed of a multi-texturing approach using 2D textures and a rendering approach with a single 3D texture did not show significant differences (see section 7.2, page 48). Even though the 3D texture fetch might be slower, the fragment shader neither needs to fetch two textures nor has to linearly interpolate between the two sample values.

Using object-aligned slices leads to a sampling rate that not only depends on the number of slices, but also on the viewing angle. If the viewing direction is not perfectly orthogonal to the current slice stack, the sampling distance is increased. This is illustrated in figure 4.6. In consequence, the assumed sampling distance Δt in the above mentioned opacity correction (equation 4.3) has to be corrected. Assuming that the viewing angle θ does not exceed $45°$(which is reasonable, because otherwise a different stack would have been chosen), the

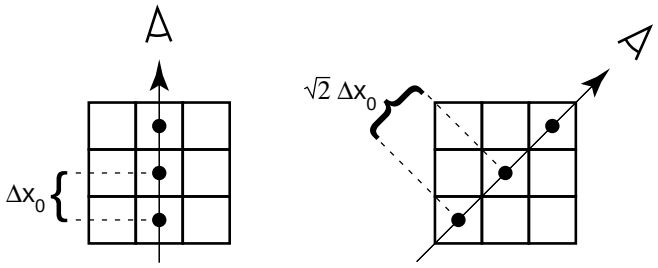

Figure 4.6: When using object-aligned slices the sampling rate also depends on the viewing angle. From [Lacroute, 1995].

sample spacing can be corrected by [Engel et al., 2004]:

$$\Delta t' = \frac{1}{\cos\theta} \cdot \Delta t. \tag{4.4}$$

Rendering using view-aligned slices

Volume rendering that utilises view-aligned planes as proxy geometry originates in algorithms that were implemented on specialised rendering hardware (see for example [Cabral et al., 1994, Cullip and Neumann, 1994, Gelder and Kim, 1996]). The major difference to object-aligned approaches is that the planes are always orthogonal to the viewing direction. In consequence, they need to be updated whenever the observer position changes. Additionally, it is not possible to use 2D textures for the rendering, since the slices are not aligned with the volume as it is the case with object-aligned methods.

The view-aligned slices of the proxy-geometry are not rectangles but polygons of up to six vertices. They are generated by clipping infinite planes against the bounding box of the volume. Generally, the clipping can be performed on the CPU (by implementing a clipping algorithm within the application) or on the graphics hardware (by utilising the ability to define additional clipping planes).

Since the slices of the proxy-geometry are always perpendicular to the viewer, the sampling distance, Δt, must not be corrected by equation 4.4.

Problems with perspective projection

All object-order approaches that use planar proxy-geometries lead to an inconsistent sampling rate if a perspective camera model is assumed. As figure 4.7 illustrates, the viewing rays are not parallel (as it is the case with orthographic projections). They all originate in a single point. As a consequence, different viewing rays hit the volume in different angles. This leads to visual artefacts that can be noticeable for extreme field-of-views [LaMar et al., 1999].

The problem with perspective projection directly corresponds to the problem of different viewing angles that approaches with object-aligned slices have to deal with (see above). The

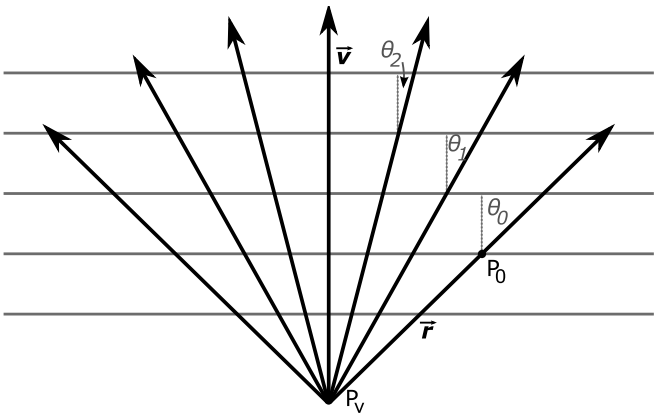

Figure 4.7: Under perspective projection, the sampling rate changes between different viewing angles, because all rays originate in a single centre of projection.

difference is that the viewing angle differs between all viewing rays and not only with different viewer positions.

As a possible solution, we propose to apply equation 4.4 per viewing ray. In a practical implementation, this can be done in a fragment shader. For this, the application has to provide the main viewing direction (i.e. the direction vector for the centre of the image plane), $\vec{v}$, and the position of the viewer, P_v, as uniform variables. Additionally, the position of the vertices, P_0, has to be passed to the fragment shader. This is done by a simple vertex shader program (because vertex shader outputs are interpolated during rasterisation, the fragment shader inputs correspond to the pixel positions).

Listing 4.1 shows an excerpt of a Cg effect file that performs a per-fragment correction of the sampling rate, Δt, as well as opacity correction. At first, all uniform variables are declared. By defining semantics, the application can initialise the variables easily. Then, a structure is defined that bundles the output variables of the vertex shader. Note that the output variable `P0` (line 13) will pass the vertex position to the fragment shader. The fragment shader first samples the volume (the exact source code is omitted here). Afterwards, the direction of the current viewing ray, $\vec{r}$, is computed in order to correct Δt (lines 35, 36, and 39). These lines of code actually apply equation 4.4. Finally, a full opacity correction according to equation 4.3 is performed and the resulting colour returned.

```
1  float4   Pv      : CAMERA_POSITION_VIEW_SPACE;
2  float3   v       : CAMERA_DIRECTION_VIEW_SPACE;
3  float4x4 wvMat   : WORLD_VIEW_MATRIX;
4  float4x4 wvpMat  : WORLD_VIEW_PROJECTION_MATRIX;
5  float    dt      : SLICE_SAMPLE_DISTANCE;
6  float    dsReci  : RECIPROCAL_VOLUME_SAMPLE_DISTANCE;
7
8  /// Output structure of the vertex shader
9  struct VsOutput
10 {
11     float4 Position : POSITION;
12     float3 TexCoord : TEXCOORD0;
13     float4 P0       : TEXCOORD1;
14 };
15
```

```
16 /// Vertex shader program.
17 VsOutput Vs_Perspective(float4 pos      : POSITION,
18                         float3 texCoord : TEXCOORD0)
19 {
20     VsOutput output;
21     output.Position = mul(wvpMat, pos);
22     output.P0       = pos;
23     output.TexCoord = texCoord;
24
25     return output;
26 }
27
28 /// Fragment shader program.
29 float4 Fp_Perspective(VsOutput input) : COLOR
30 {
31     // (1) Sample the volume / evaluate transfer function.
32     float4 color = /* ... */;
33
34     // (2) Compute direction of current viewing ray 'r'
35     float3 p0 = mul(wvMat, input.P0);
36     float3 r  = normalize(p0 - Pv).xyz;
37
38     // (3) Correct dt according to viewing angle.
39     float dt1 = dt / dot(r, v);
40
41     // (4) Perform per-pixel opacity correction.
42     float exponent = dt1 * dsReci;
43     color.a = 1.0 - pow(1 - color.a, exponent);
44
45     return color;
46 }
```

Listing 4.1: Cg effect for correcting different sampling rates caused by perspective projection.

4.3.2 Image-order ray casting on GPU

Implementing GPU-based ray casting is, as already pointed out, not as straightforward as the implementation of object-order algorithms. Ray casting in general is an image-order approach and therewith contradicts the graphics pipeline. The first algorithm for GPU-based general ray tracing was introduced in 2002 by Purcell et al. [Purcell et al., 2002]. One year later, GPU-based ray casting algorithms for direct volume rendering were described by Krüger and Westermann as well as Röttger et al. [Krüger and Westermann, 2003, Röttger et al., 2003]. Both papers criticise the object-order approaches, because a significant amount of voxels are processed even though they will not contribute to the final image. The algorithms discussed in the previous section neither recognise and skip transparent regions nor can they predict whether the currently processed fragment will be occluded by a later processed slice. Both problems originate from the back-to-front order rendering.

The methods described in the two papers are very similar in their algorithmic structure and will be outlined in the following. They differ slightly in the implementation, which will be disregarded at this point.

Since there is a one-to-one correspondence between pixels of the resulting image and the viewing rays that are traced, it is straightforward to encode the rays in 2D textures (one texture for each, the entry-point into the volume and the propagation direction). Additional

data that is required during the ray casting is stored in further texture buffers (like the current position of the rays in the volume or the accumulated colours and opacities). The volume itself is stored in a 3D texture as for rendering with view-aligned slices.

To compute the entry-points of the rays, those faces of the bounding box of the volume that point towards the camera are rendered (i.e. the *front faces*). Since the vertex positions are interpolated over the surfaces of the bounding box, the entry points are simply obtained by returning the interpolated positions in a fragment program.

To calculate the direction vectors of the rays the *back-faces*, which give the exit-points of the rays, are rendered. This time, the previously computed entry-points are subtracted from the interpolated exit-points. In summary, the direction $\vec{d}$ is computed as:

$$\vec{d} = P_{exit} - P_{entry}.$$

After the rays have been set up, the volume is sampled iteratively by reading and updating the texture buffers. In each iteration, the current position of a ray is computed and the volume is sampled at this position. The colour and opacity values are obtained by evaluating the transfer function.

If the accumulated opacity for a ray exceeds a threshold or if the ray exits the volume, the value in the depth buffer is set to zero. This causes the graphics card to skip the execution for this ray from there on due to the *early-z test*. With this trick, early ray termination is easily implemented.

Note that the two main parts of the algorithm, integration and early ray termination, are performed in separate rendering passes with different fragment shaders. The shader for sampling the volume is more complex and hence requires more computation time than the shader for early ray termination.

To achieve further optimisation, Krüger and Westermann included a simple *empty space skipping* in their algorithm [Krüger and Westermann, 2003]. In CPU-based algorithms, octree data structures are usually employed to speed-up the rendering. The space, i.e. the volume, is partitioned depending on the presence of dense material. Large empty regions are encoded as large blocks.

Since a recursive octree can not be represented and traversed in shader programs very well, the authors propose to use a simplified octree of only one level (and thus avoid the recursive traversal). In each dimension, that is in width, height and depth, the resulting 3D texture is 1/8 the size of the original volume. Thus, each voxel of this coarse volume represents an 8x8x8 block of the volume. The voxels encode whether the block in the original volume contains non-empty voxels. During the rendering pass that tests for early ray termination, the sampling interval is increased by a factor of eight and the coarse octree is read out. If the current block is empty, the depth buffer is set to zero, so that the skipping of all represented voxels of the original volume is forced. Therewith, it is avoided to execute the complex volume sampling for this block. If non-empty blocks are detected, the depth buffer is restored, and the volume will be sampled properly for the entire block.

5 Design of a modular framework

The goal of this project is to find methods to design an efficient modular GPU-based framework for direct volume rendering, which allows the user to explore multiple higher-dimensional data sets by rendering them simultaneously and by interactively deriving new data sets. When comparing our research objectives (section 1.1) with the DVR process illustrated in figures 4.2 and 4.3 it becomes clear that a flexible design is needed at each stage of the process:

Data initialisation stage: It has to be possible to load arbitrary volume data (multiple scalar and higher-order data sets). The framework must support the easy integration of arbitrary data types and file formats. It also must be possible to derive new entities at run-time. Examples are gradient and eigenvector fields, which subsequently can be used during rendering.

Classification stage: The classification stage must be flexible enough such that the user can use arbitrary components of the data or derived entities to classify the data to be rendered. For example, Diffusion Tensor Imaging data of the brain can be classified by using the mean diffusivity and diffusion anisotropy computed from the diffusion tensor data. Similarly, PET data can be used to determine regions of interest, such as a tumor, in an MRI data set.

Rendering stage: The rendering can be divided into three components: The reconstruction method used for sampling the data, the type of DVR algorithm (see section 4.3) and the desired rendering effect. The selected techniques directly affect the image quality as well as the rendering speed. Note that DVR algorithms that utilise texture mapping of graphics card usually implement a trilinear reconstruction filter. By controlling rendering effects, the user can emphasise different aspects of the data and improve the visual perception of features. Examples are gradient based shading, colour mapping of curvature, and emphasis of the silhouette boundary.

With the following three scenarios we illustrate the described stages and show how they differ (see also table 5.1). It is these differences that make a modular framework necessary.

Scenario 1 – Visualisation of a scalar CT data set

During data initialisation, a scalar volume data set must be loaded. Each voxel consists of a single value, typically encoded as an integer of 8–16 bit. Depending on the classification and the rendering effect, the gradient has to be derived and represented in a second 3D texture.

For the classification, a colour-opacity transfer function can be used. In a GPU-based approach, it is usually represented by a lookup table stored as 1D texture. A 2D transfer function that uses the magnitudes of the derived gradients as second dimension is also imaginable.

Different visualisation techniques can be suitable for the CT data set: A simple output of the density values, the evaluation of the transfer function, or additional illumination and shading effects.

Scenario 2 – Combined visualisation of CT and PET data

A PET scan records the concentration of radionuclides in the brain. By administering a patient glucose that is tagged with radioactive fluorine-18 this can be used to measure the brain activity. Such PET data is usually visualized using false colours, with lighter colours indicating regions of increased activity.

Typically, PET data is fuzzy due to low resolution and the fact that brain activity does not show clear boundaries. To set the acquired PET data into an anatomical context, it helps to simultaneously visualise the surrounding bone, which might be given by a CT data set.

In this scenario, two volume data sets as well as two transfer functions have to be used and loaded. Besides deriving the gradient for the CT data, it could be appropriate to encode both data sets in a single texture to speed-up the rendering by reducing texture lookups.

In the rendering stage, the visualisation technique must combine both data sets to give the anatomical context. However, it might also be desired to temporarily examine both data sets separately, which would require a change of the visualisation technique being used.

Scenario 3 – Visualisation of DTI data sets

In a DTI data set each voxel value is a diffusion tensor which can be mathematically represented by a symmetric 3x3 matrix. For the visualisation, it is common to derive several entities. For example, the mean diffusivity, λ_m, and mean anisotropy, λ_a, characterise different types of tissue in the brain. Both values can be derived from the eigenvalues of the tensors [Wünsche and Lobb, 2004].

During data initialisation, the DTI data set has to be loaded and the entities derived. Note that, for a GPU-based approach, the DTI data set will need to be distributed over several textures, because a single texture can only hold up to four components.

For the transfer function, λ_m and λ_a could serve as axes. Depending on which type of matter is to be emphasised, different colour and opacity values can be chosen. The data set could then be rendered by creating a Line Integral Convolution (LIC) texture in regions where mean diffusivity and anisotropy are high. Other tissue types are represented by colours. The different representations are then blended according to the function classifying the tissue type. A 2D example is given in [Wünsche and Lobb, 2004].

	Initialisation	Classification	Rendering
Scenario 1	Load CT data, derive gradient	1D or 2D transfer function	Different visualisations / shadings
Scenario 2	Load CT + PET data, derive gradient, possibly merge volume textures	1D or 2D transfer function for CT and 1D function for PET	Combined rendering, temporarily independent visualisation
Scenario 3	Load DTI data, derive eigenvalues and then λ_m and λ_a	2D transfer function for λ_m and λ_a	LIC texture where λ_m and λ_a are high, colour for other tissue types

Table 5.1: Overview of the differences of the three scenarios in the DVR process.

5.1 Requirements for the modular framework

After showing how the three scenarios differ and where the DVR process requires flexibility, we will now identify corresponding components in the framework, each of which is described by one of the following sections.

5.1.1 Support of arbitrary data

Different volume data sets may be stored in different file formats. To make the integration of arbitrary data simple and flexible, the application must allow to implement new "texture loaders". To make use of a specific implementation, it must be possible to identify the loader, to specify the actual data set (file path) as well as the variable or object in the fragment shader with which the loaded data is to be associated (that is, to which sampler object a loaded texture is to be bound).

One problem follows from scenario 3: If multivariate data has more than four components per voxel, a single volume data set must be associated with multiple texture sampler objects. In consequence, the framework has to support a splitting of the data into several textures and samplers.

5.1.2 Support for deriving new entities from existing data

A key functionality of our volume rendering framework is the derivation of new entities from existing data. One of the project objectives is to perform the required computation on the GPU using a fragment program. We call the modules or code blocks that implement the derivation of entities *operators*.

The output of an operator is a new texture object that holds the derived values. For a full specification of an operator the user has to be able to define the dimensions of the resulting texture as well as its format (number and data type of the channels), the input data sets, the fragment program that implements the operator and finally the sampler object to which the generated texture is to be bound.

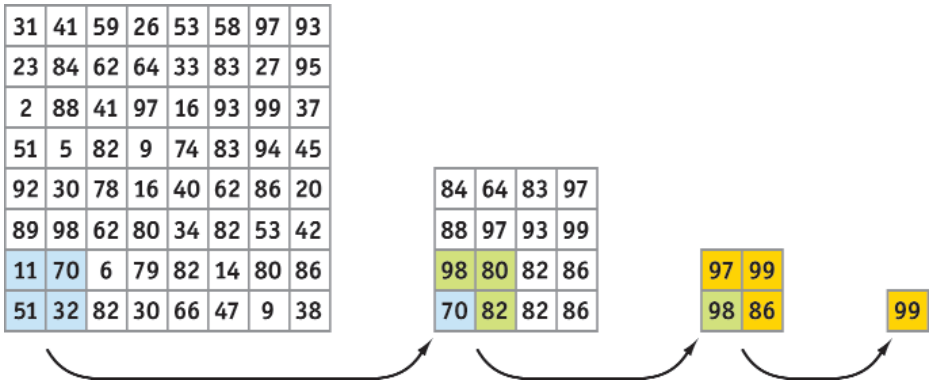

Figure 5.1: The concept of reduction operations illustrated for a 2D input texture of size 8×8. The shown reduction operation computes the maximal value. From [Luebke et al., 2004].

For operators, multiple render target capabilities of modern graphics hardware are a powerful concept that allows to output more than four values. Since each render target corresponds to a texture, and therewith to a sampler object in the fragment shader, it must be possible to define several output bindings.

Reduction operations

For some operators it might be necessary to compute a single value out of an entire input texture, for example the mean or the maximal / minimal value of the whole texture. This operation can be characterised by mapping some input data with n elements to a single value ($r : \mathbb{R}^n \mapsto \mathbb{R}$). Due to the streaming architecture of graphics hardware, GPU-based algorithms can generally not iterate through all items (texels of a texture), as it would be possible on CPU.

To compute a single value on the GPU, general purpose computing techniques describe so-called *reduction operations*. The idea is to iteratively (that is, in several rendering passes) reduce a given input texture by combining neighbouring texels. Thereby, each output texture serves as input for the next iteration. For a 2D texture of size $N \times N$ the reduction is done using four neighbours, as illustrated in figure 5.1 [Luebke et al., 2004]. In the 3D case, eight neighbours would have to be used.

To support the user with defining operators, reduction operations should be supported in our GPU-based framework as well.

5.1.3 Support of classification using transfer functions

In practice, transfer functions are represented by texture objects in GPU-based DVR. In our framework, we can therefore use the concepts described above to handle transfer functions.

As a first option, the user can specify a loader implementation to load a specific transfer function from hard disk. Note that the transfer function does not have to be saved as a

texture but can also be abstractly described, as long as the implemented loader generates a texture that can be bound to a sampler object.

In addition, if the mapping or the "shape" of a transfer function depends on actual data values, an operator can be used to compute the transfer function on-the-fly.

5.1.4 Support of different DVR algorithms and visualisation techniques

It must be possible to implement new DVR algorithms as well as selecting them for the rendering itself. The implementation of an algorithm must be flexible enough to integrate visualisation techniques defined by the user. Naturally, parts of a DVR algorithm will be executed on the GPU. The user's visualisation technique will likewise be implemented in a shader.

It is possible to separate the resulting fragment shader code into two classes: Code needed for the implementation of the algorithm (the basic structure) and code that implements a certain visualisation technique. Note that it is the visualisation technique that, in the end, determines the colour and opacity of a certain position in space. It is important to allow arbitrary combinations of algorithm code and user code.

5.2 Development of the framework design from the requirements

In this section, we will develop a design that meets the requirements described above. The following itemisation summarises requirements and user-actions we discussed.

- **Support of arbitrary data:**
  - Integration of new data formats by implementing a new texture loader.
  - Specification of the data set.
  - Specification of the appropriate loader implementation that is to be used.
  - Specification of the binding of the data to sampler objects (possibly with splitting to several samplers).
- **Support for deriving new entities from existing data:**
  - Specification of a GPU-based operator, for example as fragment program.
  - Specification of reduction operations if required for the computation.
  - Specification of the input data sets.
  - Specification of the dimensionality and size of the output texture.
  - Specification of the data format of the derived entities (texture format).
  - Specification of the binding of the result to sampler objects (possibly using multiple render targets).
- **Support of classification using transfer functions:**
  - Similar to support of volume data loading and operator definition.
- **Support of different DVR algorithms and visualisation techniques:**

- Integration of new DVR algorithms, with most of the computation on GPU.
- Straightforward integration of user-defined visualisation techniques into an existing algorithm implementation.
- Specification of the DVR algorithm that is to be used.
- Specification of the visualisation technique that is to be used.

When inspecting this list it becomes clear that some aspects extend the framework, whereas the others support the user to create visualisations. The integration of new data formats and the integration of new DVR algorithms are tasks for developers, since they have to be done *before* a user can use the framework. From this observation we derived a unified design for texture loaders and DVR algorithms that aims to make the integration of new implementations as simple as possible (discussed in the next section). The user-specific design concepts mostly relate to the loading and processing of data sets and the specification of visualisation techniques. They will be detailed afterwards. Finally, an overall view of the framework design will be given in section 5.2.3.

5.2.1 Extendible framework design

New texture loaders and DVR algorithms must be integratable into the framework with as little effort as possible. It is obvious that abstract classes (either for loading texture data or for implementing algorithms) are a good choice to allow different implementations by sub-classing. In this way, the application can use the abstract data type without the need to know the actual implementation.

However, during the initialisation of the application, an object of a specific sub-class must be instantiated. Moreover, state variables of this object might have to be initialised according to user settings (for example the number of slices for an object-order DVR algorithm). In a naive approach, the developer would need to write code for creating and initialising objects for each new sub-class. But clearly, this would lead to many conditional code branches and, more importantly, would not be very flexible and straightforward.

To overcome this problem, two design concepts have been applied: A generic factory for instantiating objects and a unified method for setting state variables, which we term *parameter design pattern*.

A generic factory

Our generic factory, designed using the *singleton* design pattern, makes it possible to create new objects of a common base-class using a unique identifier (for example a string). Sub-classes of the base-class can be registered by passing both a *prototype* object of this class and a unique identifier. Whenever an object of this sub-class has to be created, the factory clones the registered prototype object. For different base-classes, different factory objects exist (for example for texture loaders and DVR algorithms).

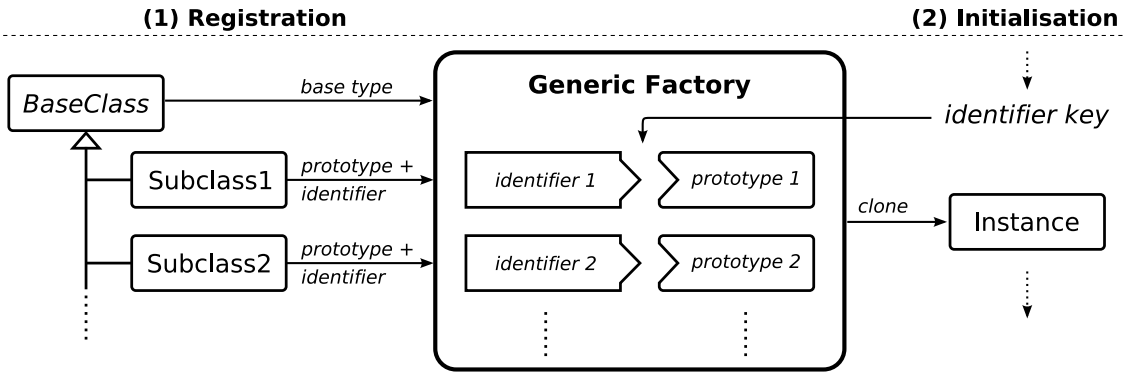

Figure 5.2: The generic factory. For registering a sub-class at a factory, the developer has to provide a prototype object and a unique identifier. During the application initialisation, a clone of this prototype can be obtained by providing the factory a corresponding identifier.

In consequence, the developer only has to register a prototype object of a new sub-class at the corresponding factory in order to make the new implementation available throughout the application. Later, a user can use the identifier to specify the concrete sub-class he wants to use (for example, a specific DVR algorithm or a texture loader for a specific data format).

Figure 5.2 illustrates the concept of the generic factory.

The parameter design pattern

In order to easily initialise objects at start-up, we developed a unified design for initialising all possible state variables. Two fundamental problems arise when dealing with state variables of unknown objects (as they may be present in the framework due to sub-classing by other developers): The state variables themselves are unknown (that is, their "name" or signature) and their data type may differ.

Before discussing our design, we make the following definitions: A *property* is a pair of a `Get...` and a `Set...` method for a certain state variable (note that the term *property* is also used in Java and C# for the same concept). Additionally, a *parameter* is defined as an object that encapsulates a property and maps the property's `Get...` and `Set...` pair to a string representation. This string representation is equal for all properties of the same data type.

To achieve an automated initialisation of state variables, every class (or base-class) that needs to be initialised allows to query its parameters and to use these parameters to set the states. This concept again allows developers to introduce new sub-classes with minimal effort. All that has to be done is specifying the available parameters. The application's initialisation module will then be able to initialise all registered parameters. This is possible because parameters have a unified string format. Implementing additional code for the initialisation for all possible properties for a new implementation is not necessary.

The concept of the parameter design pattern is shown in figure 5.3.

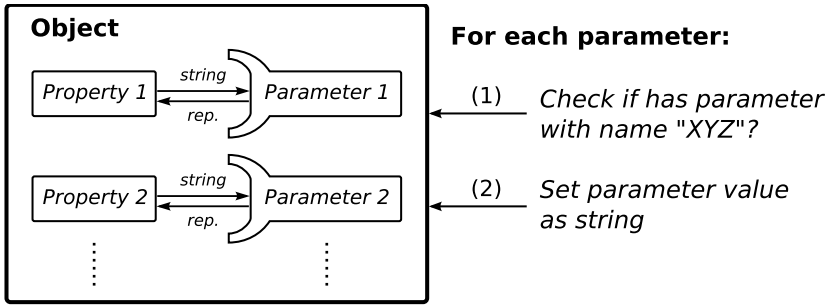

Figure 5.3: The parameter design pattern. For each property, the object provides a parameter object that abstracts from the property's data type using a unified string representation. The object can be queried for existence of parameters. The value can then be set using a unified string. Note that the property typically reads and writes a state variable of the object.

5.2.2 User-related design decisions

The most important user-specific aspects of the rendering framework are the specification of data to load (for example volume data sets or transfer functions), the processing of the data and derivation of entities using operators, and finally the specification of visualisation techniques. We will now describe how the shader code and the management of resources is organised in the framework.

At first, we can observe that loaded data is exclusively used by operators and visualisation effects, which both run on the GPU and therefore will be implemented by the user in shader programs. To keep the work with the framework practicable and clearly arranged, it is reasonable to have the specification of resources and the fragment shader definition at the same location. This is especially true when considering the fact that texture resources are bound to sampler objects of a shader.

Secondly, as already mentioned above, the executed fragment shader code can be separated into code that is specific to the DVR algorithm on the one hand and code that is specific to a user-defined visualisation technique on the other hand. The former is part of the application framework and is written whenever a new DVR algorithm is implemented. In contrast, the latter depends on what the user wants to display. It will likely be specific to a concrete domain or type of data. To make arbitrary combinations possible, the two types of code are physically and conceptually separated.

In order to meet these observations, each implemented DVR algorithm specifies a file that contains the algorithm-specific shader code. In addition, the user can specify a second file that contains the definition of resources as well as the implementations of operators and visualisation techniques. During the application start-up, both files are combined in order to obtain full functioning and valid shader code that will then be executed during rendering.

For organising and combining the shader code, we introduce a separate module in the framework. The functional concept is shown in figure 5.4. After combining the two shader source code files, the resulting shader is compiled and all resources that are needed during

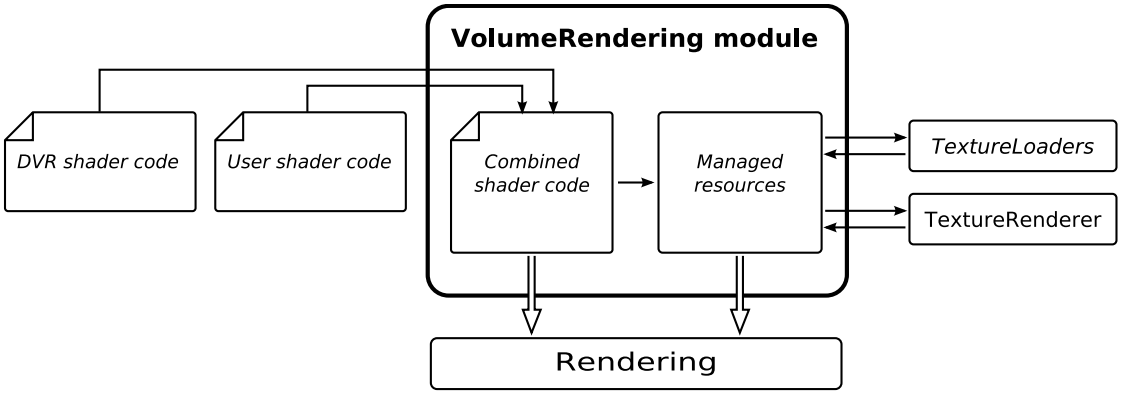

Figure 5.4: Organisation of the shader code and resources. The framework first combines the two shader source code files and then initialises the resources needed for the rendering. For loading textures, the specified texture loader is used. For executing operators, the texture renderer provides the required functionality.

rendering are initialised (that is, textures are loaded using the specified texture loader bound to sampler objects). If operators are defined, they are executed and the result stored in a newly created texture. For this purpose, the texture renderer provides functionality.

Note that one of the key functionalities of the framework is the switching between different visualisation techniques at runtime. The volume rendering module also performs the steps necessary for a switch (for example, releasing resources that are not needed anymore and loading new resources).

5.2.3 Overall view of the framework design

The main components of our rendering framework are shown in figure 5.5. Besides the concepts and modules discussed so far, the framework contains a controller object that controls the entire program execution (initialisation, rendering and termination). During the start, a configuration file is parsed. It contains settings that specify global states of the application. Further on, a renderer is introduced to render all graphical objects that are registered. It also updates the camera according to the user input.

Program flow

Figure 5.6 illustrates the basic states that the application is in during the execution. In the beginning, the factories are initialised by registering known implementations, as described above. Then, a configuration file is loaded and parsed. The configuration determines the settings in which the application will run (for example window size, the volume rendering implementation to use, the user-specified shader file, etc.). Before the actual rendering is started, the DVR and user's shader code is combined and parsed.

After all initialisation is finished, the main rendering loop is entered. In each loop, the user input is handled and a frame is rendered and displayed. If the user requests the termination of the rendering, the rendering loop is exited and the application terminates.

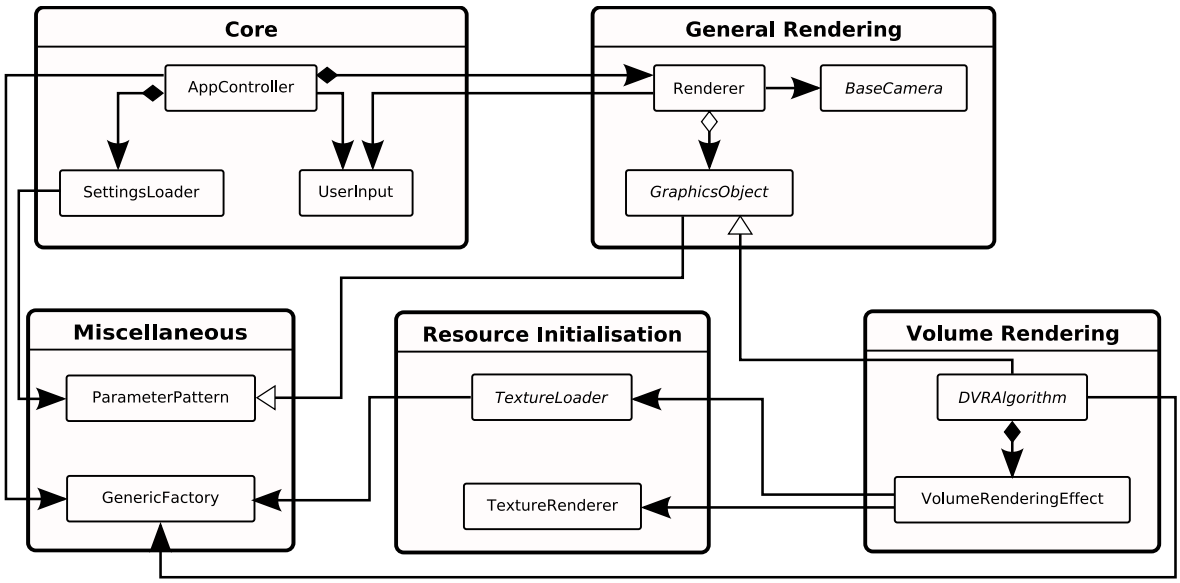

Figure 5.5: The main components of the volume rendering application. Note that for the volume rendering, the `DVRAlgorithm` is the base class for DVR algorithms. `VolumeRenderingEffect` combines shaders and manages the resources defined in the shader code.

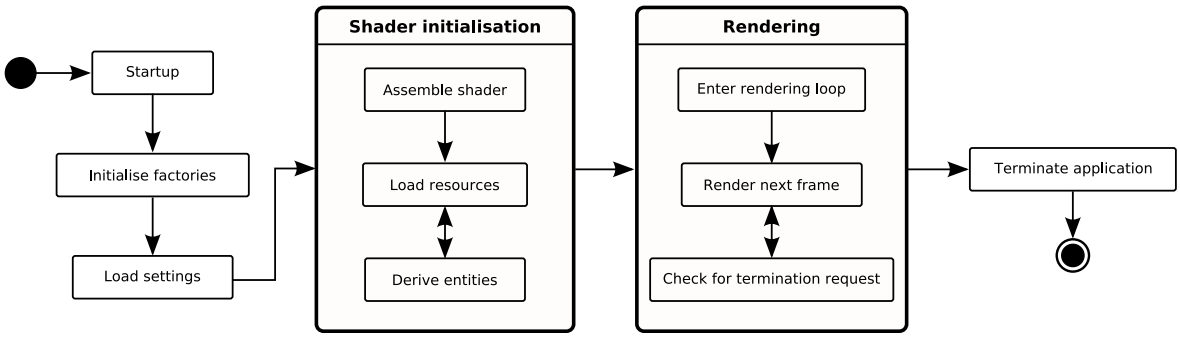

Figure 5.6: The program flow of the volume rendering application. The diagram shows the states that the application is in during the execution of the program.

6 Implementation details

Our prototypical implementation of the direct volume rendering framework is written in ANSI C++. We use OpenGL® as graphics API. As high level shading language we decided for Cg, because it has features which makes it superior to the other languages. Our implementation will heavily rely on CgFX files, semantics, annotations and interfaces — language concepts that GLSL does not offer. To use Cg, our application integrates the Cg API and runtime. Both are provided by the NVIDIA® Corporation as plain C libraries. To support the development, two additional open-source libraries are used. Graphics 3D [McGuire, 2007] is a cross-platform rendering engine. It provides basic mathematical functionality for 3D graphics (vector math for example). Additionally, Graphics 3D wraps the OpenGL® API and provides an object-oriented rendering framework. Further on, the Extensible Markup Language (XML) is used to specify the settings of the application, as will become clear below. A simple and minimalist open-source library, TinyXml [Thomason, 2007], is used to load and parse the XML files.

In the following, we outline our implementation. The discussion will follow the structure of the previous chapter and the program flow as illustrated in figure 5.6.

6.1 Application startup and initialisation

As shown in figure 5.6, the application at first initialises the factories and registers all subclasses that have been implemented (for details and an example, see below). Then, the application is initialised according to a configuration file. As already indicated, we use an XML file for this configuration. The advantage of XML over proprietary file formats is that it is standardised and human-readable as well as hierarchical and formal. Because of the tree structure of XML documents, parsing is straightforward. Another reason for using XML is that the documents can be validated against a *Document Type Definition* (DTD) or an *XML schema* [World Wide Web Consortium, 2004, World Wide Web Consortium, 2006], even though this is not been done in the current implementation and also not supported by the TinyXml library[3].

During the initialisation, we benefit from our design (generic factory and the parameter design pattern) which enables us to dynamically set the states of objects, without explicit knowledge of their implementation. The `SettingsLoader` can simply parse the configuration

[3] Note that other XML libraries offer such functionality (for example, the open-source project Xerces [The Apache Software Foundation, 2005]). They could easily be integrated and replace TinyXml.

file, instantiate new objects using the corresponding factory, and set parameter values using the string representation defined in the XML file.

Implementation of the generic factory and the parameter design patterns

The generic factory is implemented using C++ templates. The public interface of the template class looks as follows:

```
1  template<class TPrototypeBase, class TIdentifierType> class Factory {
2  public:
3     static Factory* Instance();
4     void RegisterPrototype(const TPrototypeBase*  prototype,
5                            const TIdentifierType& identifier);
6     TPrototypeBase* CreateInstance(const TIdentifierType& identifier) const;
7  };
8
9  // Example:
10 Factory<IVolumeRenderer, std::string> dvrFactory;
11 dvrFactory.RegisterPrototype(new XYZAlgorithm(), "XYZ_ALGORITHM");
```

Listing 6.1: Public interface of the template factory class.

Since the implementation of the `Factory` class follows the singleton design pattern, at most one object of a certain combination of `TPrototypeBase` and `TIdentifierType` exists. This makes it possible to easily access the same factory object in the entire application. As can be seen in the given example (lines 10 and 11), a developer only has to insert a single line of code in order to make a new implementation of a certain sub-class available throughout the application.

To realise the parameter design pattern, we make use of both templates and function pointers to member functions of classes. Recall that a *property* is a pair of a `Get...` and a `Set...` method for a certain state variable. A *parameter* encapsulates a property and maps the property's data type to a standardised string representation.

Properties are implemented using the concept of *functors* or *closures* [Haendel, 2007]: A pure virtual base-class defines one or more methods that will be used to call member functions of arbitrary objects. A generic sub-class then implements the method definitions by calling the member functions of the concrete template parameter class. In our application, the *`IProperty`* class is pure virtual and defines the methods for calling either a getter or setter method of the associated host object. A generic sub-class `Property` takes the data type of the host class and the data type of the property as template parameters. The function pointers to the getter and setter method are passed in the constructor.

To implement parameters, a pure virtual class *`IParameter`* defines methods for getting and setting the type and value of an arbitrary parameter of an object using a string representation. A generic sub-class implements this abstract parameter for the data type specified by the template parameter. In the constructor, it gets an *`IProperty`* object of the host class. The getter or setter function callbacks of this property will be invoked and string conversion will be performed according to the specified data type.

In the application, all classes that are configured by the `SettingsLoader` inherit from a base-class called *`IParameterizable`*. It holds a list of all parameters of the object. Additionally, it defines methods for checking whether a certain parameter exists and for invoking the getter or setter method using a string representation of the value.

This concept again allows other developers to introduce new sub-classes with minimal effort. All that has to be done is registering the available parameters in the list of the *`IParameterizable`* base-class. The `SettingsLoader` will then be able to initialise *all* registered parameters. Extending the loader for all possible properties of a new class is not necessary. This is possible, because parameters have a unified XML format, for example:

```
1 <VolumeRenderer type="ViewAlignedVolumeRenderer">
2   <Parameter name="Dimensions"     type="Vector3">0.78 0.9 1</Parameter>
3   <Parameter name="WireframeMode" type="Bool">false</Parameter>
4   <Parameter name="NumSlices"      type="UInt32">512</Parameter>
5 </VolumeRenderer>
```

In the shown example, the volume renderer with the identifier string "`ViewAlignedVolumeRenderer`" will be searched and a new object created. The specified parameters will be set by calling the renderer's properties after string conversion.

6.2 Initialisation of the Cg shaders

To integrate the Cg library (written in plain C) in our object-oriented framework, we encapsulate it in a `CgEffect` class. A specialised sub-class, `VolumeRenderingEffect`, is responsible for combining and parsing the CgFX code (remember that the code is conceptually separated into algorithm code and user-defined code).

Each volume renderer defines the code required to implements the underlying algorithm in a CgFX file. We use CgFX files because with the `techniques` it offers the greatest flexibility, because the full functionality of the graphics pipeline can be used. Each CgFX file of an algorithm contains an `#include` directive to input the user specified code that implements the sampling of the volume, the evaluation of a transfer function and possibly additional rendering effects. In this way, different volume renderer algorithms can be used with the same user-defined visualisations. Likewise, different visualisations can easily be used with the same algorithm. It is important to mention that the path to the user-specific code is not known beforehand. Thus, the `VolumeRenderingEffect` inserts the path into the source code at runtime.

To call the correct visualisation technique defined in the user's Cg code, a Cg interface type `IEvaluator` is defined. It contains a single function `Evaluate()` that takes a 3D texture coordinate as parameter and returns a `float4` colour value. In the algorithm code, a global instance of `IEvaluator` code is defined and used within the shader. The user is expected to define at least one structure that implements this interface. At runtime, the visualisation technique can then be changed by selecting the corresponding structure. Note that the Cg effect then needs to be recompiled.

The Cg source code is given in listing 6.2 (note that the line `#include ‘‘USER_SPECIFIED_INCLUDE_FILE.cgfx”` is replaced with a specified file path at runtime by the `VolumeRenderingEffect`, before the CgFX file is compiled):

```
1  ////////////////////////////////////////////////
2  // IEvaluator.cgfx
3  ////////////////////////////////////////////////
4  interface IEvaluator {
5      float4 Evaluate(float3 texCoord);
6  };
7
8
9  ////////////////////////////////////////////////
10 // DVR algorithm code
11 ////////////////////////////////////////////////
12 #include "IEvaluator.cgfx"
13 #include "USER_SPECIFIED_INCLUDE_FILE.cgfx"
14
15 IEvaluator evaluator  :  EVALUATOR_INSTANCE;
16
17 // Fragment shader of the algorithm
18 float4 Fp(VsOutput input) : COLOR {
19     // (1) Sample the volume / evaluate transfer function.
20     float4 color = evaluator.Evaluate(input.TexCoord);
21     // ...
22 }
23
24 ////////////////////////////////////////////////
25 // User-defined code (separate file)
26 ////////////////////////////////////////////////
27
28 // ... Definition of resources ...
29
30 struct DensityEvaluator : IEvaluator {
31     float4 Evaluate(float3 texCoord) {
32         float density = tex3D(scalarVolumeSampler, texCoord).x;
33         return float4(1, 1, 1, density);
34     }
35 };
36
37 struct GradientEvaluator : IEvaluator {
38     float4 Evaluate(float3 texCoord) {
39         float  density  = tex3D(scalarVolumeSampler, texCoord).x;
40         float  opacity  = tex1D(transferFunctionSampler, density).a;
41         float3 gradient = tex3D(gradientSampler, texCoord).rgb;
42
43         return float4(gradient, opacity);
44     }
45 };
```

Listing 6.2: Cg code for keeping algorithmic implementation and user-defined visualisation flexible.

6.3 Definition of resources

Resources (textures) must be specified by the user. The Cg mechanisms of semantics and annotations make a flexible and modular implementation possible. The user defines a Cg `texture` object as global variable. With annotations, the application can be told how to load the texture. In this way, the user specifies the identifier string of the texture loader implementation, the file path to the resource, specific hints about how to load the texture as

well as the name of the target Cg sampler object. The texture will be bound to this sampler by the application. The following simple example illustrates the concept:

```
texture transferFunction : LOAD_TEXTURE_FROM_FILE
<
  string TextureLoader   = "PIECEWISE_COLOR_TRANSFER_FUNCTION_LOADER";
  string FilePath        = "..\\volumes\\piecewiseTF_VisMale.xml";
  string SamplerBindings = "transferFunctionSampler";
  string LoaderHints     = "Resolution:512";
>;
sampler1D transferFunctionSampler = sampler_state {
  generateMipMap = false;
  minFilter      = Linear;
  magFilter      = Linear;
};
```

Note that there is no restriction on how the texture loader creates the texture objects. Data formats may range from traditional binary data to completely different representations. For example, the texture loader specified in the given example loads an XML file that contains an abstract description of a colour-opacity transfer function. The texture is then generated on the fly according to the XML specification.

As pointed out in section 5.1.1, higher-dimensional data might have to be split and distributed over more than one texture and sampler object. Thus, the specification of the target Cg sampler may contain more than one name. With each sampler name, the components of the texture that are to be associated with the sampler are defined. For a diffusion tensor imaging (DTI) data set that consists of six values per sample point, the texture definition could look like this:

```
texture dti : LOAD_TEXTURE_FROM_FILE
<
  string TextureLoader   = "TENSOR_FIELD_LOADER";
  string FilePath        = "..\\volumes\\braindata.tfd";
  string SamplerBindings = "channels_00_01_02>>dti1 \\
                            channels_11_12_22>>dti2";
>;
sampler3D dti1 = sampler_state { /* ... */ };
sampler3D dti2 = sampler_state { /* ... */ };
```

With the statements `channels_00_01_02>>dti1` and `channels_11_12_22>>dti2`, the `TextureLoader` object with the identifier "`TENSOR_FIELD_LOADER`" knows which component to associate with which Cg sampler.

6.4 Deriving new entities using operators

The implemented concept for operators is similar to the loading of textures from hard disk. The user again specifies a Cg texture and sampler object. To create the texture, additional information is needed: The dimensionality and size of each dimension, the desired texture format as well as the target sampler object. Since the operator is executed on the GPU, a full Cg technique with passes, render states and associated shader programs can be specified.

This guarantees maximal flexibility, because the full functionality of the graphics hardware and the CgFX framework is available for deriving the desired entities.

The execution of operator is managed by the `VolumeRenderingEffect` class, which makes use of the class `TextureRenderer`. It provides the functionality to render textures of one to three dimensions using a Cg effect technique. It has to be pointed out that graphics cards *always* render into 2D buffers. Consequently, it is not possible to directly render 3D textures (note that 1D textures can be considered as having two dimensions with a height of one pixel). To overcome this problem, the technique is executed slice-by-slice. Each slice is rendered into a 2D temporary buffer. The result is then copied into the 3D texture that is to be created. Copying data directly within the video memory is supported by an OpenGL® extension and no texture data needs to be copied via the system bus (`CopyTexSubImage3D`, see [Segal and Akeley, 2006]).

To render into multiple render targets the Cg annotations of the texture declaration are extended. All output colours (that is, Cg binding semantics) of the fragment shader are assigned to the desired sampler objects.

The following Cg code shows examples of operators for one and multiple render targets:

```
1  // CREATING THE GRADIENT FROM A SCALAR VOLUME
2  texture gradientTex : CREATE_TEXTURE_WITH_SHADER
3  <
4      int    Dimensions           = 3;
5      int    Width                = 128;
6      int    Height               = 256;
7      int    Depth                = 256;
8      string TextureFormat        = "RGBA8";
9      string RenderTechnique      = "Gradient";
10     string RenderTargetBindings = "gradientSampler";
11 >;
12
13 sampler3D gradientSampler = sampler_state { /* ... */ };
14
15
16 float4 FS_GRADIENT(float3 texCoord : TEXCOORD0) : COLOR0 {
17     // ...
18     return gradient;
19 }
20 technique Gradient {
21     pass {
22         VertexProgram   = NULL;
23         FragmentProgram = compile arbfp1 FS_GRADIENT();
24     }
25 }
26
27
28 // PERFORMING EIGEN-ANALYISES FOR A DTI DATA SET
29 texture eigenTex : CREATE_TEXTURE_WITH_SHADER
30 <
31     int    Dimensions           = 3;
32     int    Width                = 128;
33     int    Height               = 128;
34     int    Depth                = 33;
35     string TextureFormat        = "RGBA32F";
36     string RenderTechnique      = "EigenAnalysis";
37     string RenderTargetBindings = "COLOR0>>maxEigen COLOR1>>medEigen
38                                    COLOR2>>minEigen";
39 >;
40
```

```
41 sampler3D maxEigen = sampler_state { /* ... */ };
42 sampler3D medEigen = sampler_state { /* ... */ };
43 sampler3D minEigen = sampler_state { /* ... */ };
44
45 void FS_EIGENANALYSIS(float3 texCoord : TEXCOORD0,
46                       out float4 eMax : COLOR0,
47                       out float4 med : COLOR1,
48                       out float4 min : COLOR2) {
49     // ...
50     eMax.xyz = maxEV;  eMax.w = maxEVal;
51     eMed.xyz = medEV;  eMed.w = medEVal;
52     eMin.xyz = minEV;  eMin.w = minEVal;
53 }
54
55 technique EigenAnalysis {
56     pass {
57         VertexProgram   = NULL;
58         FragmentProgram = compile arbfp1 FS_EIGENANALYSIS();
59     }
60 }
```

Listing 6.3: Cg code for defining operators, either for a single output texture or multiple render targets.

Reduction operations

In our modular framework, reduction operations can be implemented with a single C++ class that realises the iteration process and by utilising Cg's concept of interfaces. In the case of 2D operations, an abstract Cg interface defines the signature of a function that combines four fragments. Different structures can then implement different reduction operations (min/max/mean/etc.).

In the user-defined Cg code, a variable can be defined that will hold the result of a reduction operation. Annotations provide information about which input sampler object is to be reduced and which reduction operation (that is, which structure that implements the interface) is to be used:

```
1  ////////////////////////////////////////
2  // Reduction operation interface
3  ////////////////////////////////////////
4
5  interface ReductionOperation {
6      float4 Reduce(sampler2D inputSampler,
7                    float2 texCoord1, float2 texCoord2,
8                    float2 texCoord3, float2 texCoord4);
9  };
10
11 ////////////////////////////////////////
12 // Example implementation
13 ////////////////////////////////////////
14
15 struct MaxReductionOperation : ReductionOperation
16 {
17     float4 Reduce(sampler2D inputSampler,
18                   float2 texCoord1, float2 texCoord2,
19                   float2 texCoord3, float2 texCoord4)
20     {
21         float4 v1 = tex2D(inputSampler, texCoord1);
22         float4 v2 = tex2D(inputSampler, texCoord2);
23         float4 v3 = tex2D(inputSampler, texCoord3);
```

```
24          float4 v4 = tex2D(inputSampler, texCoord4);
25
26          return max(v1, max(v2, max(v3, v4)));
27      }
28  };
29
30  ////////////////////////////////////////
31  // User-defined code
32  ////////////////////////////////////////
33
34  float4 maxMagnitude     :   PERFORM_REDUCTION_OPERATION
35  <
36      string InputSampler         = "some2DSampler";
37      string ReductionOperation = "MaxReductionOperation";
38  >;
```

Listing 6.4: Cg code for reduction operations (here, for 2D input textures). Note that the texture coordinates could also be computed on-the-fly in the fragment shader.

After the last iteration, the application can load the texture data into the system memory, read the resulting value, and initialise the uniform Cg variable. With the reduction operation technique, the amount of data that is to be transferred via the system bus has been minimised, as only a single texel has to be copied.

6.5 Definition of visualisation techniques

To implement visualisation techniques, at least one structure must be defined that implements the Cg interface `IEvaluator`. Besides this, additional structures with different implementations may be defined as well (see listing 6.2 for an example). The `VolumeRenderingEffect` lists all of these implementations during the parsing of the CgFX. The list of *evaluators* can be queried and a visualisation technique selected at runtime.

In order to make the definition of visualisation effects more practicable, the application offers commonly needed uniform variables. They can be used in the Cg shaders and are identified by predefined semantics. Examples can be seen in listing 4.1 (section 4.3, page 19). In the first 5 lines, vectors, points, and matrices are defined. They will be initialised by the application during rendering. Another useful entity that can be demanded is the texel-size for a texture associated with a defined sampler object:

```
1  float3 texelSizes    :    TEXEL_SIZES
2  <
3      string SamplerName = "scalarVolumeSampler";
4  >;
```

For example, the gradient operator shown in listing 6.3 uses this mechanism to sample the neighbouring texels of the source volume texture in order to approximate the partial derivatives in x-, y-, and z-direction.

7 Results and Tests

To demonstrate the applicability of the discussed concepts, our prototype implements two different DVR algorithms as well as different texture loaders. The first algorithm implementation realises a view-aligned object-order algorithm, the second one an object-aligned algorithm that samples 3D volume textures.

A restriction regarding the rendering algorithms has been introduced in the current implementation, as only 3D volume data is supported. When the user defines texture objects, they have to be associated with Cg sampler objects to make them accessible in shader programs. The decomposition of the volume into 2D texture stacks is not at all straightforward in a modular and flexible environment as described in the last chapter. If 2D texturing is to be supported, the user would have to specify both a 3D Cg sampler as well as a pair of 2D sampler objects. Otherwise, the user-defined CgFX file would not be compatible with different rendering algorithms. Moreover, the `IEvaluator` interface would have to be extended by an additional function for 2D texturing. In consequence, the user would have to implement both functions in any case.

Our tests show that an object-aligned direct volume rendering implementation using 2D textures is not faster (see below). Thus, the support for it was declined in order to make the definition of visualisation techniques as easy as possible. Additionally, since graphics accelerators are nowadays even capable of performing volume ray casting at interactive speed, we believe that 2D DVR texturing techniques are outdated.

Due to the limited time reduction operations used to derive scalar values from an input texture (see sections 5.1.2 and 6.4) have not been implemented yet. However, we discussed the concepts and gave detailed suggestions of a possible realisation within the rendering framework.

In an exemplary user CgFX file, several visualisation techniques have been implemented. To demonstrate the use of operators, the gradient of scalar volumes has been computed on-the-fly using a user-defined Cg technique. The created gradient texture has then been used by several visualisations (for example gradient shading or lighting calculations). Other implemented operators combine several volume data sets to reduce the number of texture fetches in the fragment shaders, which improves performance (see below for examples and tests).

7.1 Case studies

We made two case studies to show how our framework supports the user developing different visualisation techniques for different data sets. In fact, we realised scenario 1 and 2 of section 5. For the performance measurings of the renderings, the test computer **T2** has been used (specified below in section 7.2).

7.1.1 Visualisation techniques for conventional CT data

Example renderings with different visualisation techniques are shown in chapter A in the appendix (colour plates A.1 – A.3). The used data set is the head of the CT scan of the *Visible Male* (acquired by [National Library of Medicine, 2007] as part of *The Visible Human Project*®, downloaded from [Röttger, 2006]). The data set contains 128x256x256 scalar sample points. In the following, the different visualisation techniques will be explained (in the order as they are shown in the colour plates) and analysed.

Note that all required resources are specified in a single source file or computed on-the-fly using operators. The visualisation techniques, which are very different in style, can be implemented by only specifying new Cg structures that implement the `IEvaluator` interface. Because the user does not have to worry about the implementation of the DVR algorithm, the resulting Cg source code is very well structured and short (between two and 12 lines of code per evaluator in our examples).

Density: The density values of the CT scan are directly used as opacity for the output colour, without evaluating any transfer function. To improve the visibility of internal structures of the data, the density values have been scaled by 0.02.

Colour-Opacity: Implementation of a simple one-dimensional colour-opacity function, as discussed in section 4.2.

Diffuse Lighting: The colour component (red, green, and blue) obtained by evaluating the colour-opacity transfer function is shaded using an ambient-diffuse illumination model. It assumes that the displayed objects are perfect lambertian reflectors. To approximate the surface normal, an operator is implemented that computes the gradient vectors. Compared to the previous technique, the volume lighting clearly improves the perception of the structures of the object.

Gradient Shading: The normalised gradient is mapped into the range $[0, 1]$ (a scaling by 0.5, followed by a shift by 0.5) and visualised using the red, green and blue components of the computed colour: $x \mapsto r, y \mapsto g, z \mapsto b$. The returned opacity is obtained by evaluating the colour-opacity transfer function.

Artistic Shading: A shading effect that is not based on a conventional illumination model. The dot product between surface normal, $\vec{n}$, and light vector, $\vec{l}$, is computed as for

	Asm	tex1D	tex3D	Preparation	Rendering
Density	3	1	—	0.2773	0.0356
Colour-Opacity	2	1	1	0.2548	0.0401
Diffuse Lighting	14	1	1	0.5421	0.0883
Gradient Shading	8	1	1	0.5354	0.0818
Artistic Shading	12	1	1	0.5440	0.0880
Gradient Magnitude	6	1	1	0.5706	0.0655

Table 7.1: Complexity of different visualisation techniques. Shown are: The number of assembler instructions (Asm), the number of 1D and 3D texture fetches (tex1D and tex3D) as well as the preparation and the rendering speed per frame (in seconds)

diffuse lighting. However, a constant colour c is then modulated by the negated dot product and shifted by 0.5. In summary, the colour is computed as $(-(n \bullet l) \cdot c + 0.5)$. The opacity is again given by the colour-opacity function.
As can be seen in colour plate A.2 (bottom), the negation of the dot product enhances the silhouette of the skull where the angle to the light is large.

Gradient Magnitude: As discussed in section 4.2, the magnitude of the gradient, $||\nabla f||$, can be used as axis for a two-dimensional transfer function. Another possibility is to define a high-pass filter function with the magnitude as independent variable. This high-pass rejects all magnitudes below some threshold, $||\nabla f||_a$, and linearly decreases the attenuation up to some $||\nabla f||_b$ from which on all magnitudes pass unattenuated. The opacity of a 1D colour-opacity transfer function is then scaled according to that high-pass filter. Colour plate A.3 shows an example of this rendering effect. In comparison with the simple Colour-Opacity technique it becomes clear that object boundaries can be separated better when also taking the gradient into account.

Table 7.1 shows the complexity of the visualisation techniques in terms of the length of the fragment shader programs, the number of texture fetches, and the computation time. The latter is also illustrated by figure 7.1. The preparation time measures the initialisation (texture loading and execution of operators), whereas the rendering time gives the average time needed for rendering a frame. It is important to point out that the preparation is only performed once during the initialisation of the application.

As expected, the preparation time increases if an operator is executed (for Diffuse Lighting, Gradient Shading, Artistic Shading, and Gradient Magnitude). The rendering time also directly depends on the complexity of the shader. Note that the gradient operator also inserts the scalar value of the CT volume into the resulting colour. Thus, the fragment shaders do not need to sample two 3D textures.

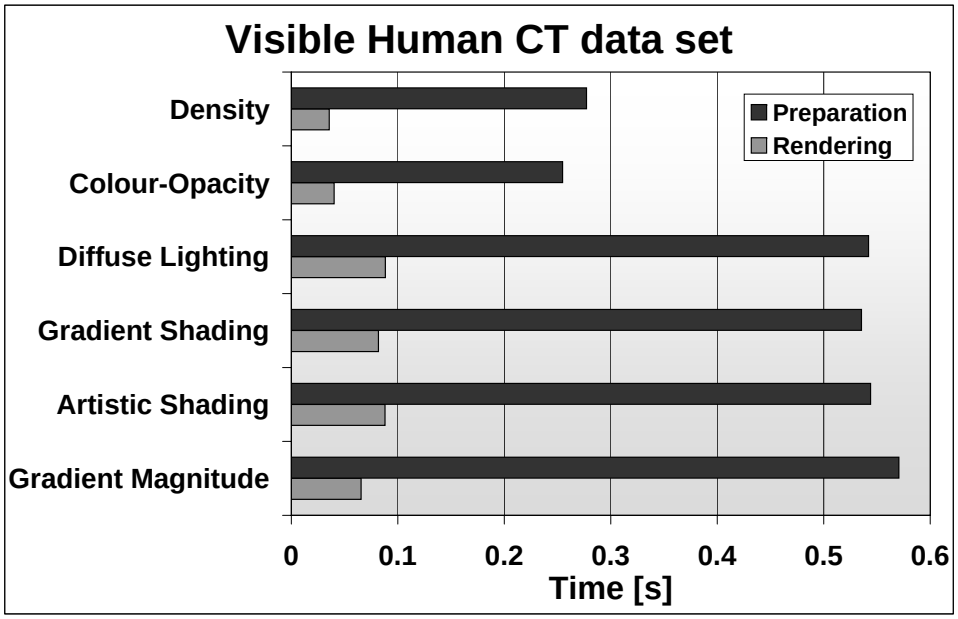

Figure 7.1: Performance of different visualisation techniques for a scalar CT data set. The preparation time depends on texture loaders and operators. The rendering time is given per frame.

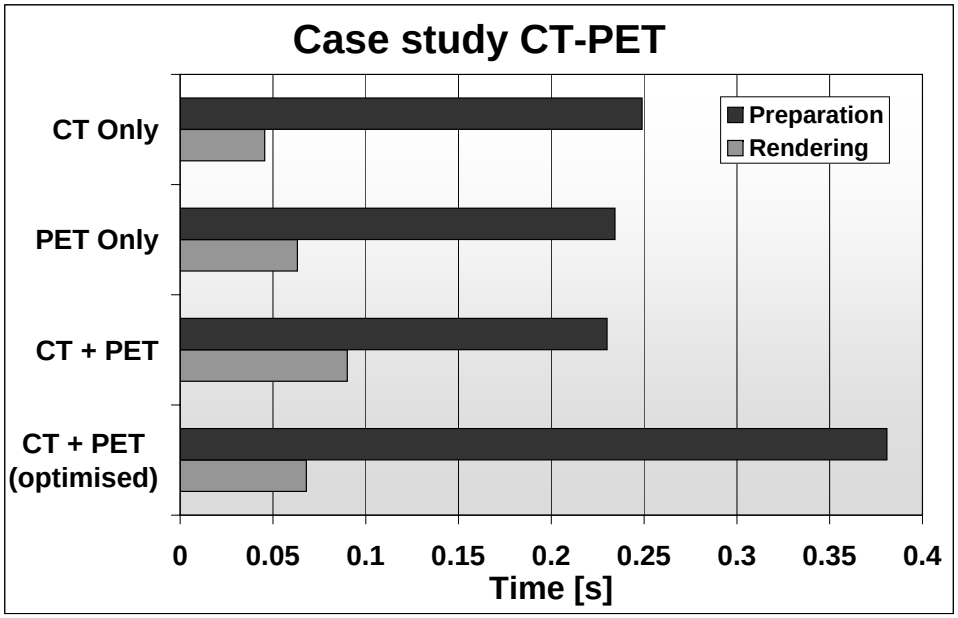

Figure 7.2: Performance of different renderings in the CT-PET case study. The preparation time mainly depends on texture loaders and operators that pre-process data. Note that the preparation is only performed once before the actual rendering starts. The rendering time is given per frame.

7.1.2 Combined rendering of CT and PET data

To demonstrate the combined visualisation of two data sets, we use a CT and a PET scan of a monkey's head, acquired by [Laboratory of Neuro Imaging, 2007] and downloaded from [Röttger, 2006]. The data sets have a resolution of 256x256x62 sample points. The PET data set has been pre-classified: The brain activity has been encoded in a false-colouring and each voxel therefore consists of an RGB triple. Without a pre-classification, we would have to define a simple 1D transfer function that assigns the same false colours to the scalar data values. We rendered onto an image plane of 800x800 pixels using our implementation of an object-order volume rendering algorithm with 256 planar view-aligned slices. Example renderings of this case study can be seen in colour plate A.4. Figure 7.2 shows the performance of four different visualisations.

CT Only: Rendering of the CT scan using a simple colour-opacity transfer function (see colour plate A.4, top-left). This is the same method as used in conventional DVR applications and needs one 3D and one 1D texture fetch (into the scalar volume and the transfer function lookup table). The Cg compiler generates a fragment shader with two instructions.

PET Only: Visualisation of the regions of increased activity (see colour plate A.4, top-right). Using the false-colouring of the pre-processed data, we only visualise data that do not contain any blue colour. The opacity is scaled depending on the green channel. Since no transfer function is used, the shader only performs one 3D texture fetch. Together with the scaling, eight instructions are needed. As the chart shows, the rendering is slightly slower than for the CT.

CT + PET: Combined visualisation of both data sets (see colour plate A.4, bottom). Each data set is exactly evaluated as described before. However, the colour resulting from the PET volume is only computed and returned if the colour of the CT transfer function is fully transparent.

It is not surprising that the rendering is slower. The shader code needs 14 instructions, one 1D texture fetch, and either one or two 3D texture fetches (because of dynamic flow control, the second 3D texture fetch is only performed if the PET volume has to be evaluated).

CT + PET (optimised): To increase the efficiency of the combined rendering, we implemented an operator that creates a new volume with four channels. It holds both the scalar CT values and the triplets of the PET data set. Hence, only a single 3D texture fetch has to be performed during rendering. The Cg compiler uses 13 instructions.

The result of this optimisation can easily be seen in the diagram. The fragment shader is about as fast as the **PET Only** technique. Since an operator has to be executed during the initialisation, the preparation time is increased. But note that the operator is executed only once before the actual rendering.

7.2 Performance tests

In the next sections, different test results will be shown and discussed. The following hardware setups were used for testing:

T1: Intel® Core™ 2 Duo 2.13 GHz; 2.0 GB main memory; NVIDIA® Quadro® FX 550, 128 MB video memory.

T2: Intel® Pentium® 4 3.4 GHz; 2.0 GB main memory; NVIDIA® Quadro® FX 3400, 256 MB video memory.

T3: Intel® Pentium® M 1.8 GHz; 1.25 GB main memory; ATI™ Mobility Radeon™ 9700, 128 MB video memory.

Comparison of operator execution

To analyse whether CPU-based or GPU-based operators are faster, the gradient was computed for an 8 bit scalar volume data set. The data set consists of $128 \times 256 \times 256$ voxels. The experiment was run for different sizes of the generated gradient texture (for $2^n \times 2^n \times 2^n$ voxels with $0 \leq n \leq 8$).

Figures 7.3, 7.4, and 7.5 show the time needed for the different test machines to perform the operator on CPU and GPU respectively. The sudden break in the rendering speed for the NVIDIA® Quadro® graphics cards of test machines T1 and T2 might be caused by some driver-internal issues, but this is an unverified assumption. However, since there is no discontinuity with the ATI™ card of machine T3, the application itself can likely be eliminated as a reason.

Figure 7.6 and table 7.2 show the relative speed of the operator implementations (the ratio CPU/GPU speed). As expected, the GPU-based operator is faster for reasonably large texture sizes (larger than about 32^3 – 64^3voxels). For larger volumes, the GPU-based operator performance is up to roughly 35–40 times faster. For extremely small sizes, the CPU-based approach out-performs the GPU rendering, which can be explained by less overhead required for setting up the operator.

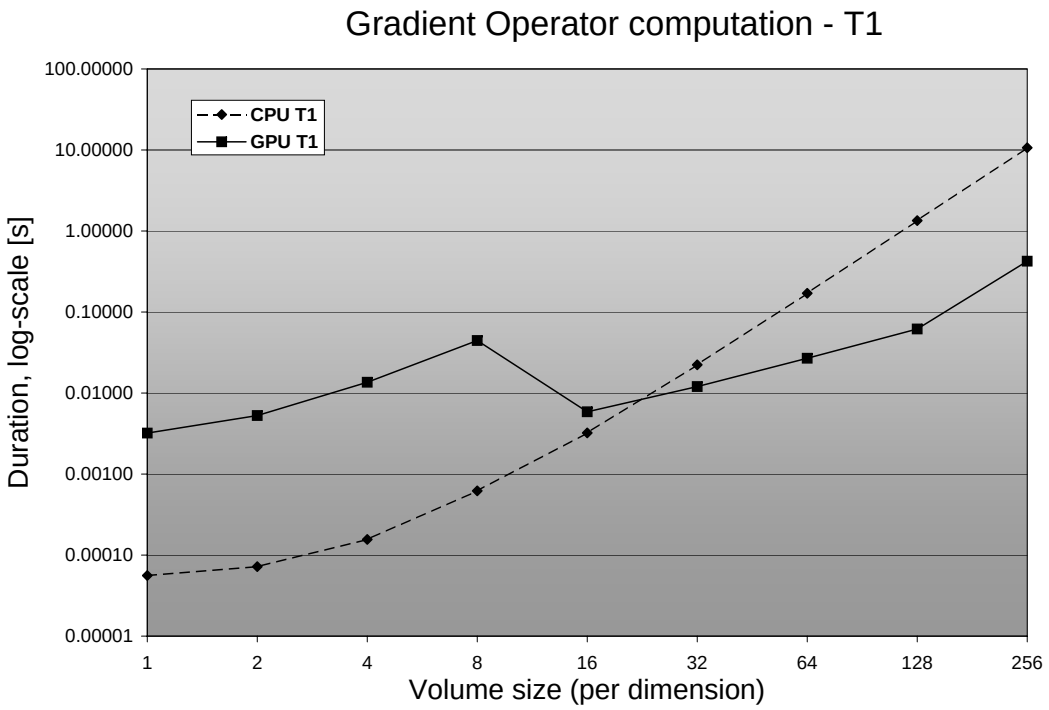

Figure 7.3: Computation time for gradient operator on test machine T1, both on CPU and GPU.

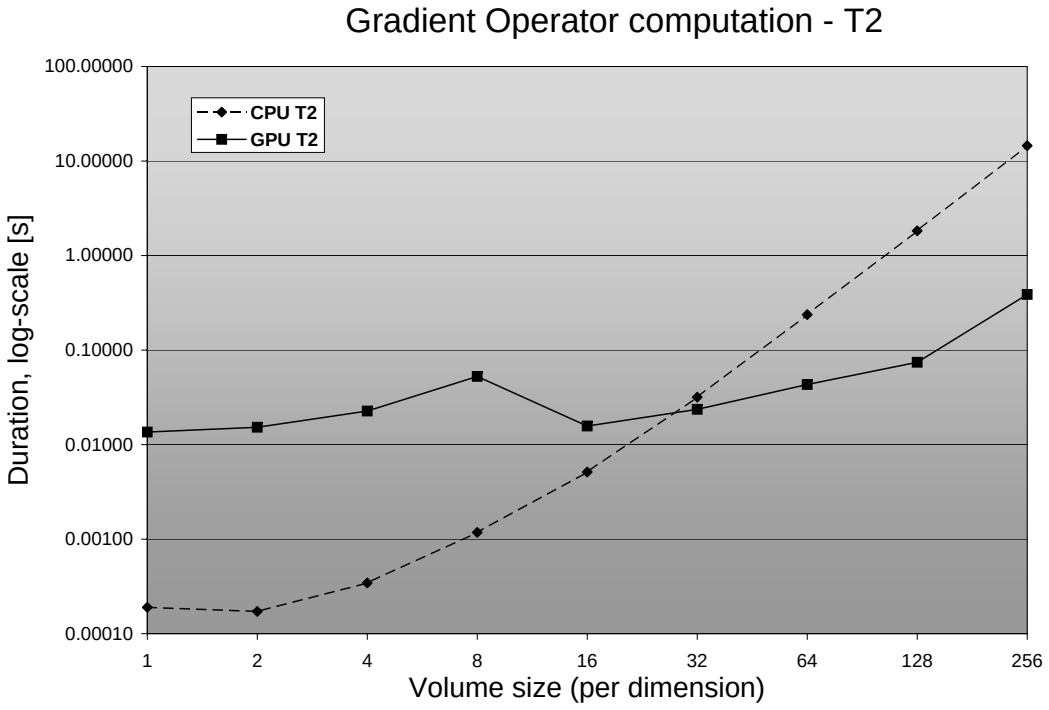

Figure 7.4: Computation time for gradient operator on test machine T2, both on CPU and GPU.

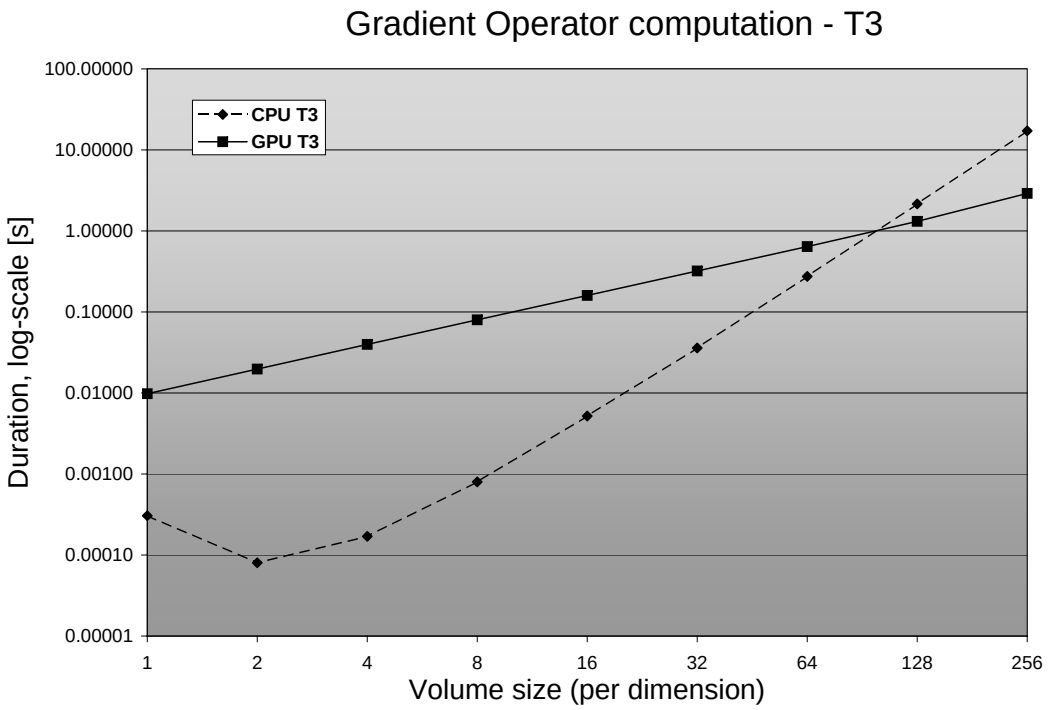

Figure 7.5: Computation time for gradient operator on test machine T3, both on CPU and GPU.

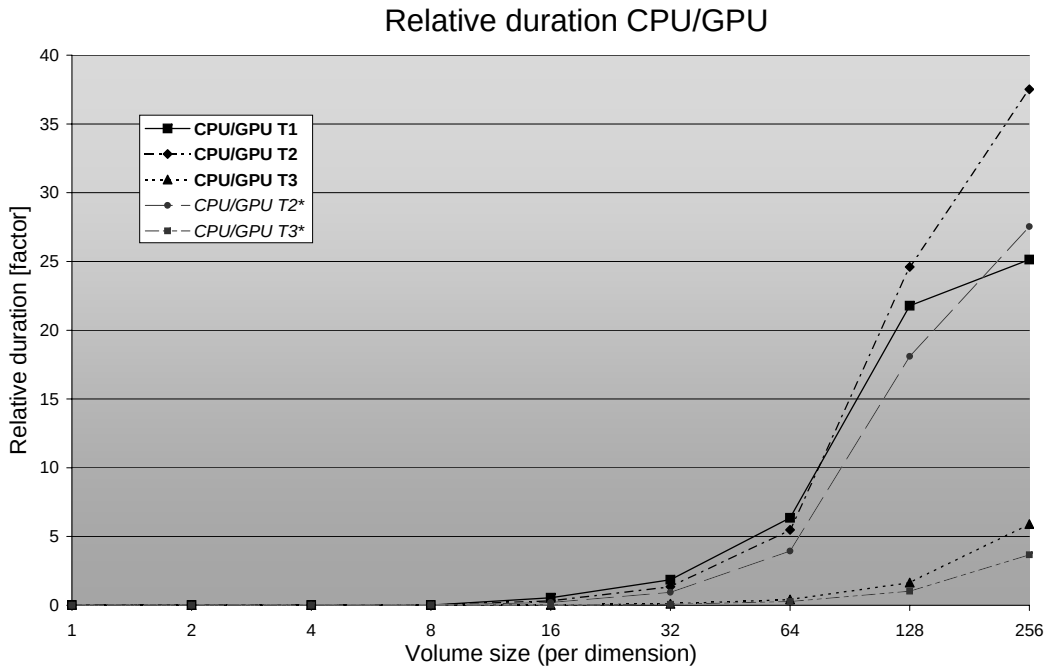

Figure 7.6: Relative computation time for the gradient operator of all test machines. In order to normalise the results for a single test setting, the GPU measures for T2 and T3 were also compared to the CPU speed of T1 (the fastest CPU). Lines *CPU/GPU T2** and *CPU/GPU T3** show this comparison. Note that the result is biased, because the GPU computation was performed with different CPUs).

Voxels	T1	T2	T3	*T2**	*T3**
1	0.0175	0.014	0.0312	0.0041	0.0057
2	0.0137	0.0113	0.0041	0.0047	0.0036
4	0.0115	0.0152	0.0043	0.0069	0.0039
8	0.014	0.0224	0.01	0.0118	0.0078
16	0.55	0.3263	0.0325	0.2052	0.0202
32	1.8549	1.3466	0.1121	0.9437	0.0696
64	6.3525	5.4781	0.4282	3.9414	0.2663
128	21.7854	24.6022	1.6444	18.1025	1.0251
256	25.135	37.5242	5.9051	27.5299	3.6702

Table 7.2: Relative computation time for the gradient operator of all test machines. In order to normalise the results for a single test setting, the GPU measures for T2 and T3 were also compared to the CPU speed of T1 (the fastest CPU). Lines *CPU/GPU T2** and *CPU/GPU T3** show this comparison. Note that these results are biased, because the GPU computation was of course performed with different CPUs).

Two-dimensional vs. three-dimensional texture fetches

It is of interest whether a slice-based volume rendering algorithm that uses object-aligned slices and 2D multi-texturing is faster than a similar algorithm that uses a single 3D texture (see section 4.3.1 for an algorithm description). Since the former algorithm cannot be realised easily with the current framework, a comparison is reasonable.

For the test, two simple fragment shaders were written. They are shown in listing 7.1. The first samples two 2D textures and linearly interpolates the obtained values. The second shader simply returns the value obtained by sampling a 3D texture. In this way, the shaders exactly simulate the differences in the algorithms. Note that with this test it is *not* tested which texture fetch is generally faster. The render target as well as the input textures had a size of 128 for all dimensions.

```
1 float4 FS_2D(float3 texCoord : TEXCOORD0) : COLOR0 {
2     float4 t1 = tex2D(s2Da, texCoord.xy);
3     float4 t2 = tex2D(s2Db, texCoord.xy);
4     return lerp(t1, t2, texCoord.z);
5 }
6 float4 FS_3D(float3 texCoord : TEXCOORD0) : COLOR0 {
7     return tex3D(s3D, texCoord);
8 }
```

Listing 7.1: Fragment shaders used to test texture fetch performance. Note that the definition of the sampler objects is omitted here. All samplers use linear interpolation for sampling the associated texture.

The execution of the shaders was performed $1.0E+7$ times. To calculate the computation time for a single fragment, the total duration was divided by the total number of processed fragments (i.e. $1.0E+7 \times 128^2 = 1.64E+11$). As can be seen in figure 7.7, the computation time per fragment is similar for the two approaches. In fact, the 3D texture fetch is slightly faster.

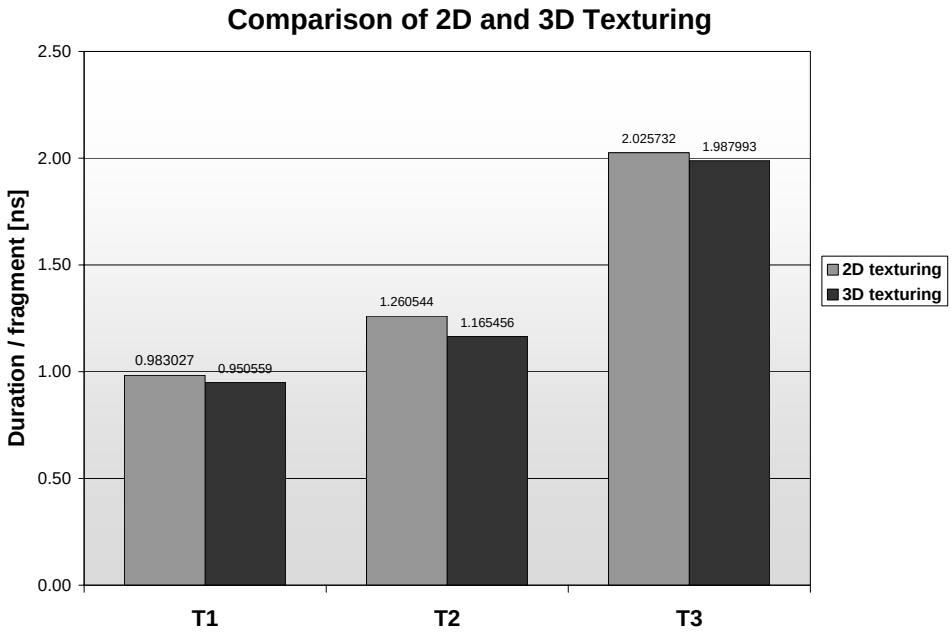

Figure 7.7: The average per-fragment computation time for texture fetches.

8 Conclusion and Future work

In our prototypical implementation, the different aspects of modularity, described in section 5, have successfully been implemented. The developed framework is flexible enough to support new types of data, rendering algorithms, and visualisation techniques using advanced mechanisms of the Cg high-level shading language and the CgFX framework, like binding semantics, annotations, techniques and interfaces.

With the abstract `TextureLoader` class, no restrictions on the data representation are made. As could be shown, even abstract descriptions of lookup tables, that are then generated on-the-fly, can be implemented (see section 5, page 36). A minor drawback is that data of higher dimensionality must be split into several texture objects, because of restrictions to at most four components of graphics hardware in general.

In order to derive new entities from existing data, the concept of GPU-based operators has been developed and implemented. It allows to perform the operator-computation on GPU, using the full functionality of the CgFX framework. The application ensures that all required resources are loaded. Complex operators may render into multiple render targets and therewith output up to four values per target.

By defining the abstract Cg interface `IEvaluator`, multiple visualisation techniques can easily be implemented and used. The user has the full flexibility of specifying how to sample which volume textures and how to evaluate some transfer function. The application supports this by providing commonly used vectors and matrices.

The application is currently in a prototypical state, and could therefore be extended in several ways in future. Very interesting would be the implementation of a GPU-based ray casting algorithm as described in section 4.3.2.

More functionality in general could be implemented as well. An important aspect of direct volume rendering is the use of clipping planes or geometries to cut out parts of the volume. This gives further insight into the internal structures. Defining clipping planes is a standard functionality of 3D APIs like OpenGL®, and the clipping is performed as part of the rendering pipeline. In [Weiskopf et al., 2002], Weiskopf et al. discuss in detail how to use fragment shaders to implement clipping against complex objects.

Usage of our framework could be simplified by creating a graphical user interface which offers menus and dialogues for loading data and for deriving entities using operators. CgFX files could be created on-the-fly according to the user's demands and control the rendering.

A Colour plates

Visible Male CT data set

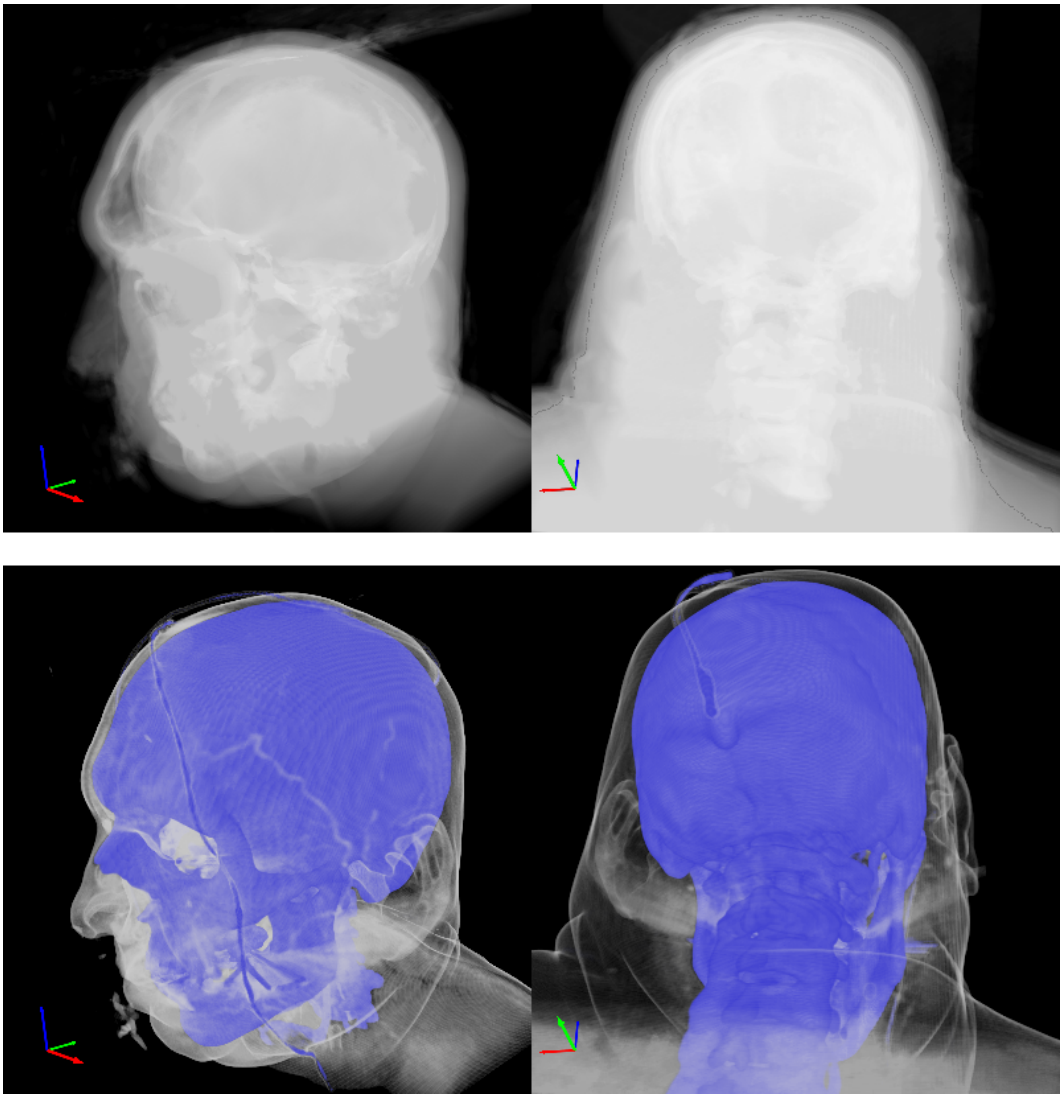

Figure A.1: Renderings of the head of the Visible Male CT data set. Top: CT density (scaled by factor 0.2); Bottom: Colour-opacity transfer function.

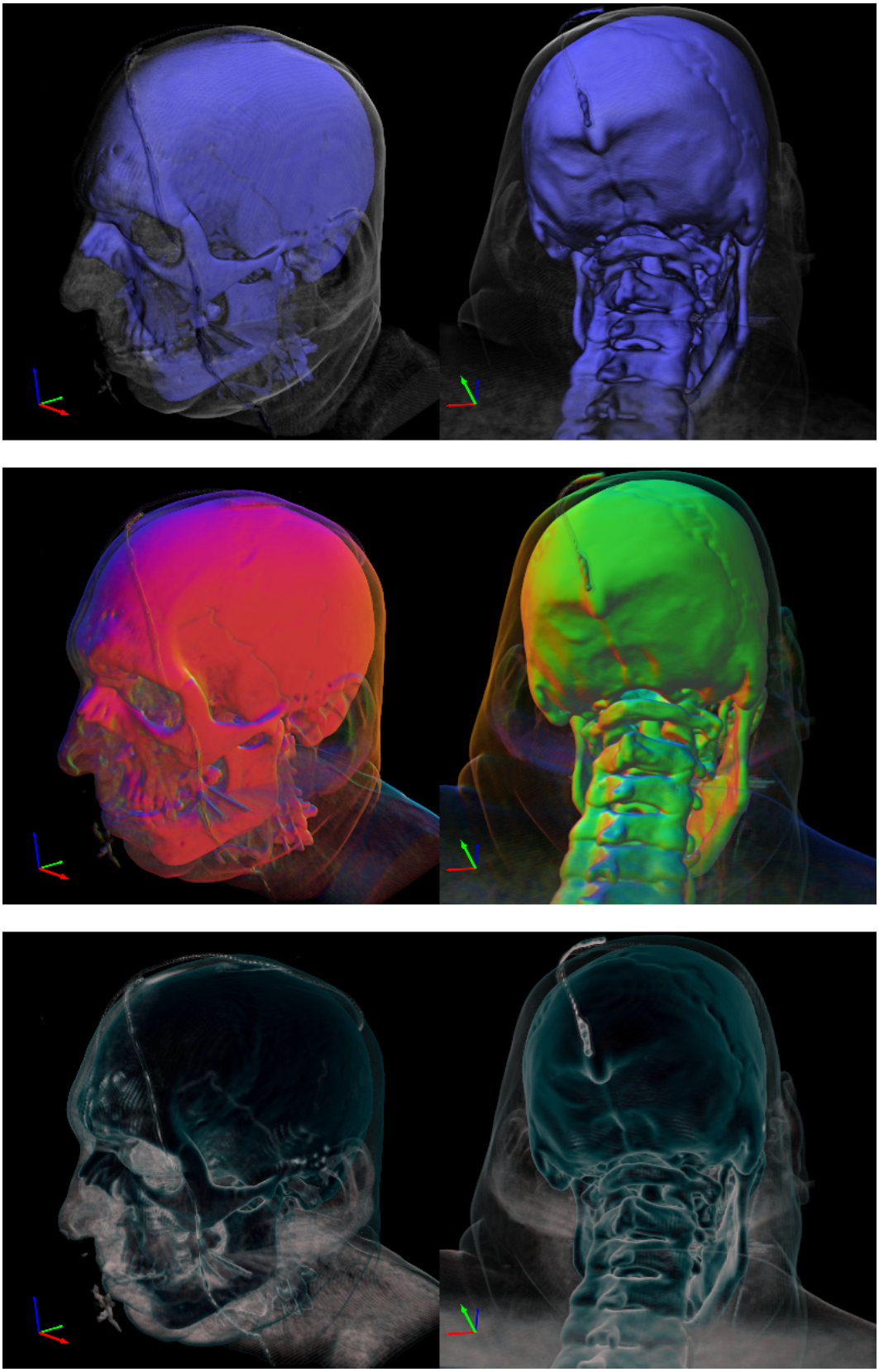

Figure A.2: Renderings of the head of the Visible Male CT data set. Top: Diffuse lighting; Middle: Gradient shading: Bottom: Artistic shading which enhances the silhouette of rendered structures.

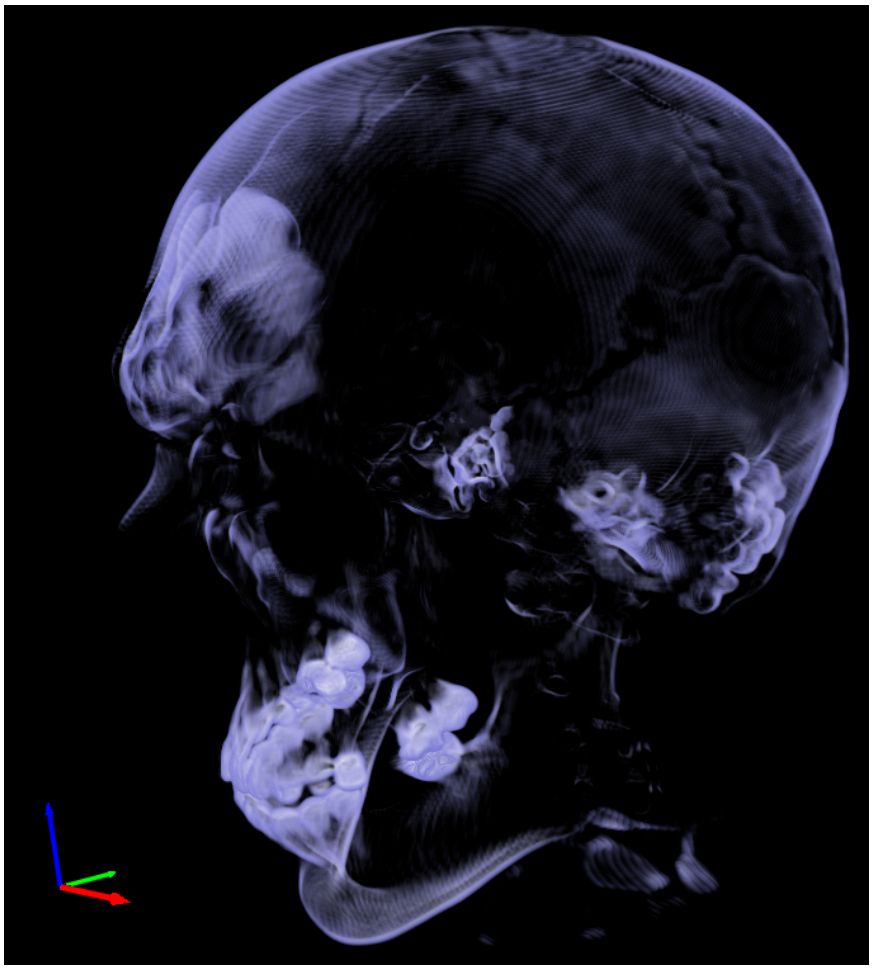

Figure A.3: Rendering of the head of the Visible Male CT data set. The alpha channel of a 1D colour-opacity transfer function was scaled according to a linear high-pass filter function that takes the magnitude of the gradient, $||\nabla f||$, as argument. This high-pass rejects all magnitudes below some $||\nabla f||_a$ and passes everything above another threshold, $||\nabla f||_b$ (here, $||\nabla f||_a = 0.25$ and $||\nabla f||_b = 0.6$ was used).

PET-CT case study

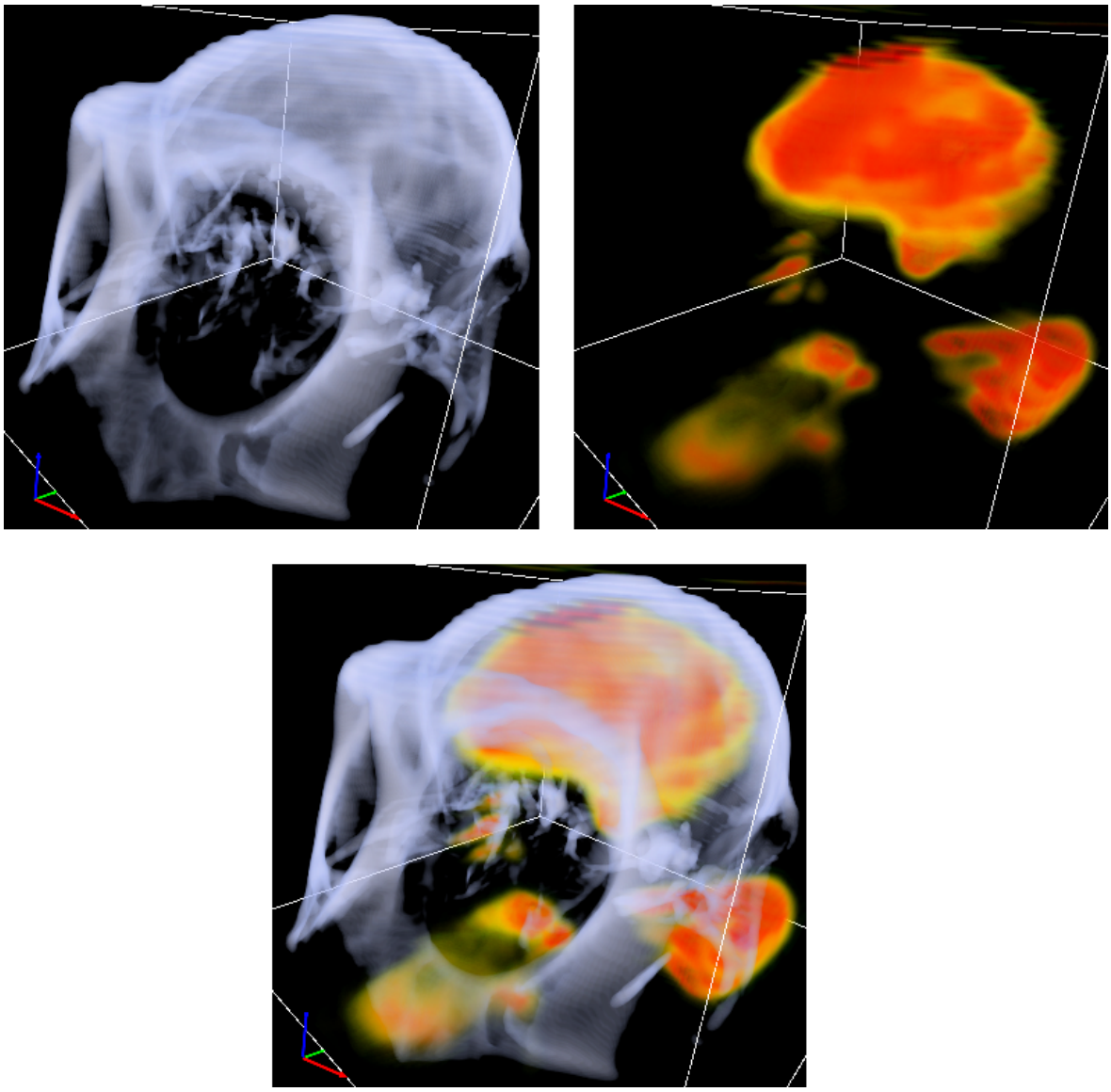

Figure A.4: Different renderings in the CT-PET case study. We used a CT and a PET scan of a monkey's head. Top-left: CT data set using a colour-opacity transfer function; Top-right: False-coloured PET data set showing the regions with increased activity; Bottom: Combined rendering. The PET data is set into an anatomical context, which is given by the CT data set showing the bones semi-transparent.

Bibliography

[Advanced Micro Devices, 2007] Advanced Micro Devices. ATI™ Radeon™ hd 2900 series - gpu specifications. (2007). URL: `http://ati.amd.com/products/Radeonhd2900/specs.html` [last checked: 24/08/2007].

[Akenine-Möller and Haines, 2002] Akenine-Möller, T. and Haines, E. (2002). *Real-Time Rendering*. A K Peters, Natick, MA, 2nd edition.

[Bhaniramka and Demange, 2002] Bhaniramka, P. and Demange, Y. (2002). OpenGL volumizer: a toolkit for high quality volume rendering of large data sets. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 45–54, Piscataway, NJ, USA. IEEE Press.

[Blythe, 2006] Blythe, D. (2006). The Direct3D 10 system. *ACM Transactions on Graphics*, 25(3):724–734.

[Bruckner and Groller, 2005] Bruckner, S. and Groller, M. E. (2005). Volumeshop: An interactive system for direct volume illustration. *IEEE Visualization 2005 (VIS 2005)*, page 85.

[Cabral et al., 1994] Cabral, B., Cam, N., and Foran, J. (1994). Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume Visualization*, pages 91–98, New York, NY, USA. ACM Press.

[Cullip and Neumann, 1994] Cullip, T. J. and Neumann, U. (1994). Accelerating volume reconstruction with 3d texture hardware. Technical report, University of North Carolina, Chapel Hill, NC, USA.

[Drebin et al., 1988] Drebin, R. A., Carpenter, L., and Hanrahan, P. (1988). Volume rendering. *ACM SIGGRAPH Computer Graphics*, 22(4):65–74.

[Engel et al., 2004] Engel, K., Hadwiger, M., Kniss, J. M., Lefohn, A. E., Salama, C. R., and Weiskopf, D. (2004). Real-time volume graphics. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 29, New York, NY, USA. ACM Press.

[Engel, 2003] Engel, W. F. (2003). *Shaderx 2: Introduction & Tutorials With Directx 9*. Wordware Publishing, Inc, Plano, TX.

[Fernando and Kilgard, 2003] Fernando, R. and Kilgard, M. J. (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, Boston, USA.

[Foley et al., 1996] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. (1996). *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 2nd edition.

[Gelder and Kim, 1996] Gelder, A. V. and Kim, K. (1996). Direct volume rendering with shading via three-dimensional textures. In *VVS '96: Proceedings of the 1996 symposium on Volume Visualization*, pages 23–30, Piscataway, NJ, USA. IEEE Press.

[Haendel, 2007] Haendel, L. The function pointer tutorials. (2007). URL: `http://www.newty.de/fpt/functor.html` [last checked: 24/08/2007].

[Kajiya and Herzen, 1984] Kajiya, J. T. and Herzen, B. P. V. (1984). Ray tracing volume densities. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA. ACM Press.

[Kindlmann, 2003] Kindlmann, G. Teem: Tools to process and visualize scientific data and images. (2003). URL: `http://teem.sourceforge.net/` [last checked: 24/08/2007].

[Kindlmann et al., 2000] Kindlmann, G., Weinstein, D., and Hart, D. (2000). Strategies for direct volume rendering of diffusion tensor fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):124–138.

[Kindlmann et al., 2003] Kindlmann, G., Whitaker, R., Tasdizen, T., and Moller, T. (2003). Curvature-based transfer functions for direct volume rendering: Methods and applications. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 67, Washington, DC, USA. IEEE Computer Society.

[Kniss et al., 2002a] Kniss, J., Hansen, C., Grenier, M., and Robinson, T. (2002a). Volume rendering multivariate data to visualize meteorological simulations: a case study. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 189–ff, Aire-la-Ville, Switzerland. Eurographics Association.

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

[Kniss et al., 2002c] Kniss, J., Premoze, S., Hansen, C., and Ebert, D. (2002c). Interactive translucent volume rendering and procedural modeling. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 109–116, Washington, DC, USA. IEEE Computer Society.

[Krüger and Westermann, 2003] Krüger, J. and Westermann, R. (2003). Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA. IEEE Computer Society.

[Laboratory of Neuro Imaging, 2007] Laboratory of Neuro Imaging, U. S. o. M. Monkey atlas. (2007). URL: `http://www.loni.ucla.edu/` [last checked: 24/08/2007].

[Lacroute, 1995] Lacroute, P. (1995). *Fast Volume Rendering using a Shear-Warp Factorization of the Viewing Transformation*. PhD thesis, Stanford University.

[Lacroute and Levoy, 1994] Lacroute, P. and Levoy, M. (1994). Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA. ACM Press.

[LaMar et al., 1999] LaMar, E., Hamann, B., and Joy, K. I. (1999). Multiresolution techniques for interactive texture-based volume visualization. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 355–361, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Levoy, 1988] Levoy, M. (1988). Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37.

[Lorensen and Cline, 1987] Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA. ACM Press.

[Luebke et al., 2004] Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., and Lefohn, A. (2004). GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA. ACM Press.

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

[McGuire, 2007] McGuire, M. G3d engine. (2007). URL: `http://g3d-cpp.sourceforge.net` [last checked: 24/08/2007].

[Microsoft® Corporation, 2007] Microsoft® Corporation. Microsoft developer network. (2007). URL: `http://msdn.microsoft.com/` [last checked: 24/08/2007].

[Mihajlovic et al., 2003] Mihajlovic, Z., Budin, L., and Quid, N. (2003). Reconstruction of gradient in volume rendering. *IEEE International Conference on Industrial Technology, 2003*, 1:282–286.

[National Library of Medicine, 2007] National Library of Medicine, N. I. o. H. The visible human project®. (2007). URL: `http://www.nlm.nih.gov/research/visible/visible_human.html` [last checked: 24/08/2007].

[NVIDIA® Corporation, 2001] NVIDIA® Corporation. GeForce3™ product overview. (2001). URL: `http://www.nvidia.com/object/LO_20010612_4376.html` [last checked: 24/08/2007].

[NVIDIA® Corporation, 2005a] NVIDIA® Corporation. Cg language specification. (2005). URL: `http://developer.download.nvidia.com/cg/Cg_1.5/1.5.0/0019/Cg_Specification.pdf` [last checked: 24/08/2007].

[NVIDIA® Corporation, 2005b] NVIDIA® Corporation. Cg toolkit user's manual. A developer's guide to programmable graphics. (2005). URL: `ftp://download.nvidia.com/developer/cg/Cg_1.4/Docs/CG_UserManual_1-4.pdf` [last checked: 24/08/2007].

[Phong, 1975] Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317.

[Pixar Animation Studios, 2005] Pixar Animation Studios. The RenderMan® interface specification. (2005). URL: `https://renderman.pixar.com/products/rispec/index.htm` [last checked: 24/08/2007].

[Purcell et al., 2002] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA. ACM Press.

[Rezk-Salama et al., 2000] Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G., and Ertl, T. (2000). Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118, New York, NY, USA. ACM Press.

[Rost, 2006] Rost, R. J. (2006). *OpenGL® Shading Language*. Addison-Wesley, Reading, MA, 2nd edition.

[Röttger, 2006] Röttger, S. The volume library. (2006). URL: `http://www9.informatik.uni-erlangen.de/External/vollib/` [last checked: 24/08/2007].

[Röttger et al., 2003] Röttger, S., Guthe, S., Weiskopf, D., Ertl, T., and Strasser, W. (2003). Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 231–238, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Sabella, 1988] Sabella, P. (1988). A rendering algorithm for visualizing 3d scalar fields. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 51–58, New York, NY, USA. ACM Press.

[Sander et al., 2005] Sander, P. V., Isidoro, J. R., and Mitchell, J. L. (2005). Computation culling with explicit early-z and dynamic flow control. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, pages 10–1 – 10–12, New York, NY, USA. ACM Press.

[Schroeder et al., 1998] Schroeder, W., Martin, K. M., and Lorensen, W. E. (1998). *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition.

[Segal and Akeley, 2006] Segal, M. and Akeley, K. The OpenGL® graphics system: A specification (version 2.1 - december 1, 2006). (2006). URL: `http://www.opengl.org/registry/doc/glspec21.20061201.pdf` [last checked: 24/08/2007].

[Stegmaier et al., 2005] Stegmaier, S., Strengert, M., Klein, T., and Ertl, T. (2005). A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195.

[The Apache Software Foundation, 2005] The Apache Software Foundation. Welcome to Xerces. (2005). URL: `http://xerces.apache.org/` [last checked: 07/09/2007].

[Thomason, 2007] Thomason, L. TinyXml. (2007). URL: `http://sourceforge.net/projects/tinyxml/` [last checked: 24/08/2007].

[Watt, 1993] Watt, A. (1993). *3D Computer Graphics*. Addison-Wesley, Reading, MA, 3rd edition.

[Weiskopf et al., 2002] Weiskopf, D., Engel, K., and Ertl, T. (2002). Volume clipping via per-fragment operations in texture-based volume visualization. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 93–100, Washington, DC, USA. IEEE Computer Society.

[World Wide Web Consortium, 2004] World Wide Web Consortium. XML schema part 1: Structures second edition. (2004). URL: `http://www.w3.org/TR/xmlschema-1/` [last checked: 24/08/2007].

[World Wide Web Consortium, 2006] World Wide Web Consortium. Extensible markup language (xml) 1.0 (fourth edition). (2006). URL: `http://www.w3.org/TR/REC-xml/#dt-doctype` [last checked: 24/08/2007].

[Wünsche and Lobb, 2004] Wünsche, B. C. and Lobb, R. (2004). The 3d visualization of brain anatomy from diffusion-weighted magnetic resonance imaging data. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 74–83, New York, NY, USA. ACM Press.