Design Concepts for Flexible Hardware-accelerated Direct Volume Rendering Frameworks

Felix Manke, Burkhard Wünsche

***Abstract*—Modern consumer graphics hardware develops at a rapid pace. At present time, graphics cards allow the visualisation of complex scenes and very large three-dimensional volumetric data sets at interactive frame rates. The video game industry certainly is the major driver of these developments, but research areas like medicine, science or engineering also play an important role, as data sets have to be explored interactively.**

However, designing flexible and interactive rendering applications that can easily be extended and customised is still a challenge, because of both the highly specialised graphics processing pipeline and the need of maximal performance. In this paper, we present solutions for the design of rendering frameworks for visualising volumetric data sets that overcome many of the restrictions and offer flexibility for developers and users while maintaining interactive rendering performance.

***Index Terms*—Hardware-accelerated rendering, direct volume rendering, GPU programming, rendering framework design.**

I. Introduction

In many scientific research areas three-dimensional (3D) discrete data arrays need to be analysed. The amount and complexity of data from simulations and measurements is increasing exponentially. Visualization is an essential tool to analyse and explore this overwhelming amount of data and to communicate findings to professionals and laymen.

Modern graphics processing units (GPUs) make interactive visualisations of volume data on consumer hardware possible without introducing intermediate polygonal representations. In recent years, the use of programmable graphics hardware became more and more popular in graphics applications, because it allows to implement custom "shader" programs that define how an object is rendered. Even though specialised high level programming languages simplify the development of shader programs, GPU programming is still very restricted and constrained and generally very different to conventional CPU programming. For the sake of rendering speed, the highly parallel SIMD stream processing pipeline of GPUs only allows limited flexibility [1].

In the field of visualisation, new direct volume rendering (DVR) algorithms have been developed that improve efficiency and perception [2], [3], [4], [5]. For the purpose of demonstrating their achievements, the researchers usually implement specialised renderers which are not flexible enough to interactively explore arbitrary volumetric data. Other publications discuss more general rendering frameworks, but they are either restricted to certain types of data [6], [7], not interactive [8] or rather difficult to use [9]. So far, no framework design has been developed that supports the interactive exploration of arbitrary volumetric data sets.

Graphics Group, Department of Computer Science, The University of Auckland, New Zealand. E-mail: fman020@ec.auckland.ac.nz, burkhard@cs.auckland.ac.nz.

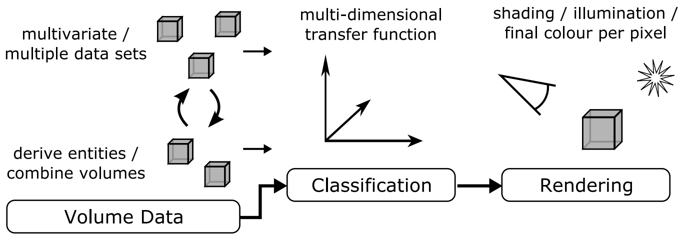
multivariate / multiple data sets
derive entities / combine volumes
multi-dimensional transfer function
shading / illumination / final colour per pixel
Volume Data
Classification
Rendering

Fig. 1. The stages of the DVR process. For multivariate data the preprocessing and classification are usually more complex than for scalar data.

In this paper, we discuss the design of an interactive and flexible framework for GPU-based direct volume rendering. The aim is to support an easy integration of arbitrary data into the rendering framework — regardless of its dimensionality or representation (for example scalar, vector, or tensor data). Moreover, user-defined visualisations of the internal structures have to be possible. Here, our main focus is to develop mechanisms that easily allow the derivation of new values out of existing data and the combination of data sets. With the proposed solutions, users can control the entire visualisation of the data. Conventional tools usually only allow to change predefined parameters. Generally, as much of the computation as possible is to be performed on the GPU to benefit from its enormous computational power.

II. The DVR Process

Figure 1 illustrates the steps that are performed in direct volume rendering. At first, input data sets are loaded. When dealing with multivariate data, meaningful entities often have to be derived or data sets have to be combined. Note that conventional visualisation tools usually only support to load scalar volume data and to derive the gradient vector from it. In the next step a "transfer function" is evaluated that maps data values to colours and opacities. Objects in a data set are differentiated by assigning different colours and opacities, which happens at exactly this stage. During the rendering, shading and illumination effects can be applied to achieve a better perception of the three-dimensional structures. Finally, the volume is projected onto the image plane and displayed.

III. Requirements and Design of a Flexible Rendering Framework

A. *Definition of Requirements*

When comparing our research objectives with the DVR process (figure 1) it becomes clear that flexibility is needed at each stage of the process:

Data initialisation stage: It has to be possible to integrate and load arbitrary volume data sets and file formats. Additionally, the user must be able to derive whatever entities are needed and to freely combine different data.

Classification stage: The framework must be flexible enough to support any classification that is suitable for the current domain and data set. The design of a transfer function largely depends on the data values and entities selected as input. Additionally, a reasonable classification of structures is only possible with knowledge of the data.

Rendering stage: New DVR algorithms must be integratable into the framework. More importantly, it must be possible to define and switch between visualisation techniques so that the user can emphasise different aspects of the data and improve the visual perception.

B. Framework Extensibility

We can observe that both the integration of new data formats and DVR algorithms are tasks for developers, since they have to be done *before* a user can work with the framework. Abstract classes are a good choice to allow different implementations by sub-classing, since thereby the application can use the abstract data type without the need to know the actual implementation.

However, during the initialisation of the application, objects of specific sub-classes must be instantiated. Moreover, state variables of these objects might have to be initialised according to user settings. 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.

We derived a unified scheme that makes extending the framework as easy as possible. The design is inspired by reflective programming paradigms that some programming languages support (for example Java and C# [10], [11]).

Figure 2 illustrates a generic template factory, designed using the *singleton* and *prototype* design patterns. The factory makes it possible to create new objects of a common base-class using a unique identifier. Instances of sub-classes of the base-class can be registered by passing both a prototype object of this class and an identifier. Whenever a new object of a sub-class has to be created, the factory clones the corresponding prototype object. Developers can register new implementations in a single line of code. Afterwards, a sub-class is instantiable throughout the application without the need of knowing the concrete data type.

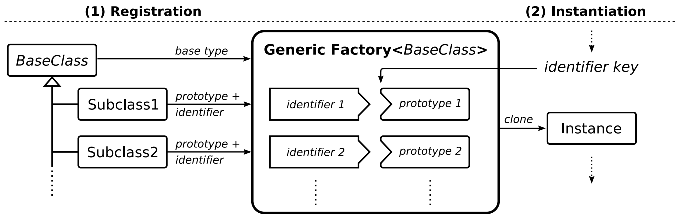

Fig. 2. Principle of the generic template factory. A clone of a registered prototype can be obtained by providing a corresponding identifier.

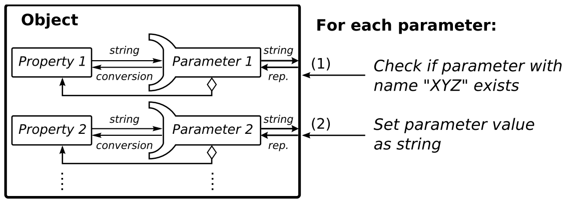

Fig. 3. Scheme of the parameter design pattern. For each property, the object provides a parameter object that abstracts from the property's data type using a unified string representation.

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

To be able to initialise the properties of unknown objects we introduce a design which we call *parameter design pattern*: Every property is encapsulated by an object we call *parameter*. It hides the data type using a standardised string representation. During initialisation, the application invokes the `Set...` method of the parameter objects and passes a string, which is then internally converted to the actual data type and the property is set. The concept is illustrated in figure 3.

With these two concepts, the generic factory and the parameter design pattern, we are able to automatically instantiate and initialise new implementations, even if the programming language does not support reflective programming. Writing additional code for the initialisation of all possible properties of a new sub-class is not necessary.

C. Modularisation of the DVR Process

The most important user-specific aspects of the rendering framework are the specification of data to load, the processing of the data, and finally the specification of visualisation techniques and shading effects. To achieve our goal of flexibility, we rely on advanced GPU programming techniques. To overcome the present restrictions we utilise features of the high level shading language *C for Graphics (Cg)* [12]. None of the other real-time shading languages is powerful enough to suffice our requirements.

The derivation of new entities is to be done on the GPU in order to achieve efficiency and to minimise data transfer time. We call the Cg shader code blocks or modules that implement this derivation *operators*. The main inputs of an operator are volume data sets and the output is a texture object that holds the derived values.

Note that loaded data is exclusively used by operators and visualization effects, which both run on the GPU and therefore will be implemented by the user as Cg shader programs. Hence, to facilitate a practicable work with the framework, we keep the specification of resources and the Cg shader definition at the same location.

Further on, after analysing existing DVR application we observed that the executed shader code can be separated into

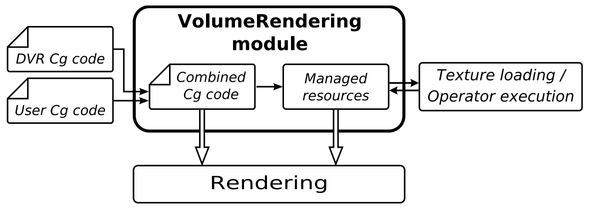

Fig. 4. The volume rendering module that manages the shader code and resources.

code that is specific for a particular rendering algorithm (written by developers) and code that is specific to a user-defined rendering effect (written by the user and which might have to be changed at runtime). To make arbitrary combinations possible, we separate the two types of code physically and conceptually.

Figure 4 shows that our framework contains a module that assembles the Cg code of the DVR algorithm and the user's Cg shader code and manages the specified resources (volume data sets or textures). During the initialisation, the compiled Cg code is analysed using the Cg Core Runtime API, data sets are loaded and operators executed.

By making use of *interfaces* — a well-known concept of the object-oriented programming paradigm that actually only Cg provides for GPU programming — the implementation of visualisation techniques (which we call *evaluators*) is held abstract and the user can define as many different evaluators as desired using several implementations of the abstract interface. The volume rendering module makes interactive switching of evaluators possible. Without using interfaces, the *entire* shader code would have to be duplicated for every custom visualisation effect.

D. Components of the Framework

The main components of our rendering framework are shown in figure 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 that renders all graphical objects and updates the camera according to the user input.

IV. Implementation

To demonstrate the effectiveness and flexibility of our framework and the discussed concepts, we developed a prototype, implemented in C++ for maximal performance. As already mentioned, we used Cg as shading language to benefit from its advanced features. Basic mathematics for 3D graphics is provided by Graphics 3D [13] which has the additional advantage that it wraps the OpenGL 3D-API and provides an object-oriented rendering framework. Further on, the Extensible Markup Language (XML) is used to specify the settings of the application. A simple and minimalist open-source library, TinyXml [14], is used to load and parse the XML files.

V. Results

In three case studies we show how to define resources, operators and evaluators. At first, we used a scalar Computed Tomography (CT) scan of the head of the *Visible Male* (acquired by [15], downloaded from [16]). Renderings of different evaluators are shown in figure 6. Then, we used a CT and a Positron Emission Tomography (PET) data set of a monkey (acquired by [17], downloaded from [16]) to demonstrate how to combine several volumes (see figure 7). In the third case study, we procedurally generated a 3D vector field with an operator and visualised it using an interactive 3D Line Integral Convolution (LIC) technique (see figure 8).

Note that, besides the multitude of implemented evaluators and besides implementing several different operators, the Cg source code is very well structured and short (between two and 16 lines per evaluator), because the Cg code for the DVR algorithm is separated by making use of Cg's interfaces.

For a detailed discussion of the design of our framework and the results we outlined in this paper, please see [18].

VI. Conclusion and Future Work

We have presented a modular and flexible framework for interactive direct volume rendering of complex data sets. The framework is entirely GPU-based and can be easily extended by developers.

In contrast to existing applications and toolkits it is completely modular and the user can interactively derive new entities and modify rendering and shading effects to explore complex data sets. We use advanced mechanisms of the Cg language to provide a flexibility that is usually difficult to achieve on the specialised and restrictive graphics hardware.

In the future, our implementation could be further extended to provide more functionality (for example, volume clipping). Additionally, we want to make the framework more user-friendly and provide a graphical user interface which offers menus and dialogs for loading data, deriving entities with operators, selecting shaders and effects or developing new ones, for example by providing formula editors with GUI for deriving new data sets.

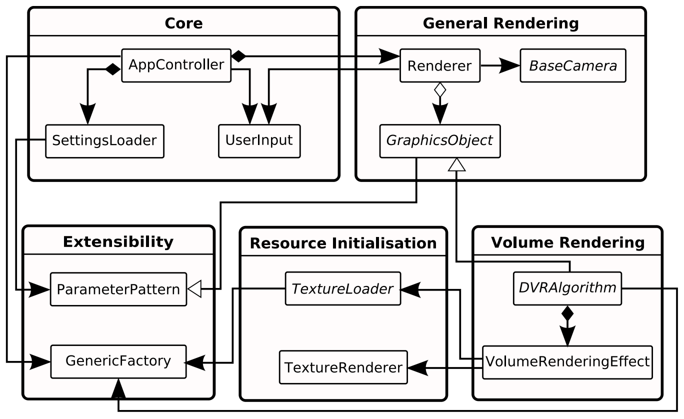

Fig. 5. The main components of the volume rendering application. Note that `VolumeRenderingEffect` combines Cg shaders and manages the specified resources.

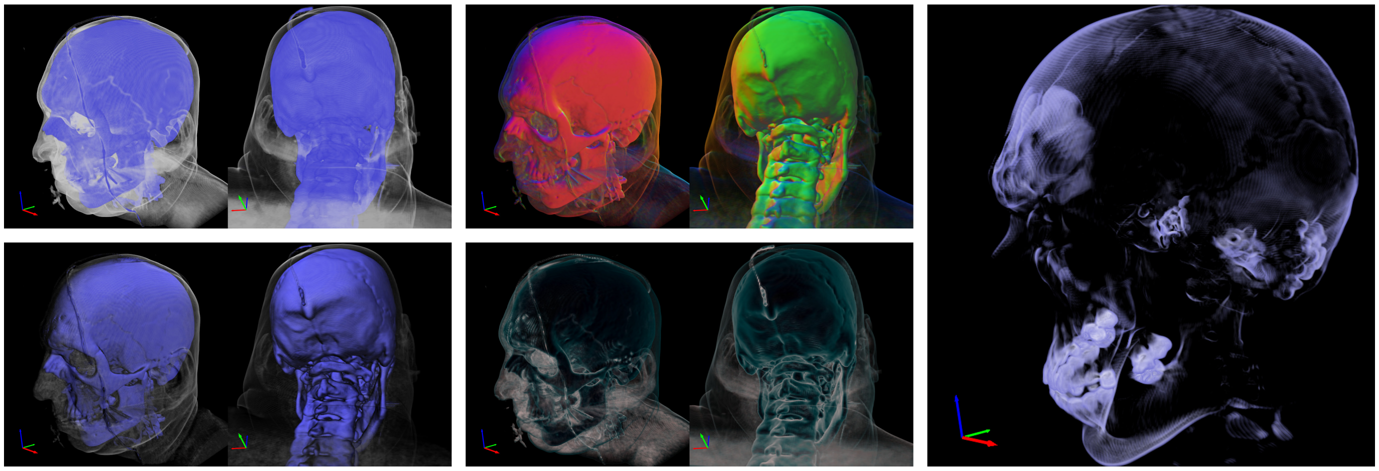

Fig. 6. Different renderings of a CT data set. Top-left: Use of a basic 1D transfer function. Bottom-left: Additional diffuse lighting. Top-centre: Gradient shading that shows the direction of the gradient vectors. Bottom-centre: Artistic shading that enhances the silhouette of rendered structures. Right: 2D transfer function using the scalar data value and gradient magnitude.

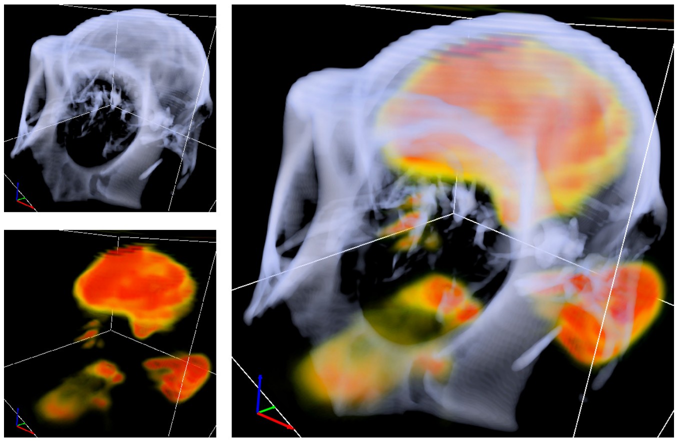

Fig. 7. Combined rendering of a CT (top-left) and a PET (bottom-left) data set of a monkey.

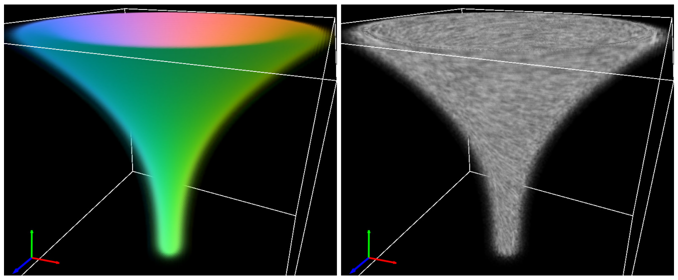

Fig. 8. Renderings of a 3D vector field. Left: Colour-encoded procedural vector field (normalized vectors are mapped into RGB range $[0,1]^3$). Right: Interactive LIC rendering. The opacity is proportional to the vector length.

REFERENCES

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

[2] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "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*. New York, NY, USA: ACM Press, 2000, pp. 109–118.

[3] J. Kniss, S. Premoze, C. Hansen, and D. Ebert, "Interactive translucent volume rendering and procedural modeling," in *VIS '02: Proceedings of the conference on Visualization '02*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 109–116.

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

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

[6] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, "A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting," in *Proceedings of the International Workshop on Volume Graphics '05*, 2005, pp. 187–195.

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

[8] G. Kindlmann. (2003) Teem: Tools to process and visualize scientific data and images. Website. [Online]. URL: http://teem.sourceforge.net/

[9] Kitware, Inc. (2007) The visualization toolkit. [Online]. URL: http://public.kitware.com/VTK/

[10] Sun Microsystems, Inc. (2007) Reflection. [Online]. URL: http://java.sun.com/javase/6/docs/technotes/guides/reflection/index.html

[11] Microsoft Corporation. (2007) Reflection (c# programming guide). [Online]. URL: http://msdn2.microsoft.com/en-us/library/ms173183(VS.80).aspx

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

[13] M. McGuire. (2007) G3d engine. Website. [Online]. URL: http://g3d-cpp.sourceforge.net

[14] L. Thomason. (2007) TinyXml. Website. [Online]. URL: http://sourceforge.net/projects/tinyxml/

[15] National Library of Medicine, National Institutes of Health. (2007) The visible human project®. [Online]. URL: http://www.nlm.nih.gov/research/visible/visible_human.html

[16] S. Röttger. (2006) The volume library. [Online]. URL: http://www9.informatik.uni-erlangen.de/External/vollib/

[17] Laboratory of Neuro Imaging, UCLA School of Medicine. (2007) Monkey atlas. [Online]. URL: http://www.loni.ucla.edu/

[18] F. Manke, "A Modular GPU-based Direct Volume Renderer for Visualising Scalar and Multi-dimensional Data," Oct. 2007, CompSci 780 project report. [Online]. URL: http://www.cs.auckland.ac.nz/~burkhard/Reports/2007_S1_FelixManke.pdf