The VRE Volume rendering engine

Michal Hučko*
Comenius University, Bratislava
Slovakia

Michal Vanek†
Comenius University, Bratislava
Slovakia

Miloš Šrámek‡
Austrian Academy of Sciences, Vienna
Austria

Abstract

We present the extendable volume rendering engine VRE which provides an open and flexible environment for both experimental and production level implementation of a wide range of volume visualisation algorithms, including various CPU and GPU based ones. We identify parts of renderer functionality suitable for isolation in logical units and propose various types of plugins. As the support for various volume data file formats, internal data representation and rendering algorithms is realised by the plugins, the engine can be easily extended by new functionality. We define a general application interface which enables to develop arbitrary visualisation applications, being it command line, batch ones or applications with a graphical user interface. The proposed architecture provides for multiple concurrent renderings which can be with advantage utilised in the client/server version of the engine. In this setup the server side component of the engine allows access of multiple peers to a single instance of the engine, which makes sharing of the visualised data by multiple clients possible. The VRE software is released under the GPL licence, opening the potential of the environment to all interested parties.

Keywords: volume visualisation, rendering, client/server architecture, cross-platform

1 Introduction

Volumetric data is commonly produced in modern medicine, biological research, industry, etc. The source devices for this type of data include computer tomographs, magnetic resonance scanners, ultrasound scanners, confocal microscopes and others. Visualisation of volumetric data can be performed on a variety of hardware devices by a variety of methods. With new scanner devices there arise new data types, often with a higher spatial and spectral resolution and thus larger in size. New visualisation hardware (e.g. graphics accelerators and multicore processors) permit processing of larger volumes, achieving speed-up either by utilising parallelism or just by reimplementing algorithms, which required special hardware in the past, on a consumer hardware. For all the mentioned reasons, processing and visualisation of volume data is still a subject of active research with development of new algorithms and reimplementations and modifications of the known ones [Engel et al. 2006].

*e-mail: michal.hucko@fmph.uniba.sk
†e-mail: michal.vanek@st.fmph.uniba.sk
‡e-mail: milos.sramek@oeaw.ac.at

The existing research teams use various software tools for processing and visualisation of the volume data. These are often in-house applications which might be even designed for some specific job and not suitable for the general use. Not only is it often difficult to transfer code between different institutions, but because of the high specialisation sometimes it might become a problem to merge code from various groups of the same institution. Therefore, our goal in the design of VRE was to develop (i.e. to design and implement) a general, flexible and extendable visualisation framework for experimentation, education and production, which would enable researchers from multiple institutions to cooperate, while minimising the amount of repeatedly written code. A similar problem arises also in the case of the necessity to create platform independent applications, being it hardware or software cases.

Identification and isolation of suitable functional modules and their implementation in the form of plugins allow VRE to achieve high flexibility. Extension of the engine by new data file format readers, internal storage representations or rendering algorithms can be done independently, without the need to modify other parts of the system. Support for multiple concurrent renderings can be further utilised either in a local or a remote client/server version of the engine. In the remote version the server hosts multiple peers, effectively managing their demands for resources by sharing the loaded volume. General application interface of VRE allows that both variants can be used from either interactive or non-interactive, graphical or non-graphical applications.

This paper is structured as follows. In Section 2 we cover other visualisation applications and tools describe their properties, advantages and disadvantages. These are then addressed in Section 3 where we present our architecture with respect to the mentioned goals of the visualisation engine. This section starts with the description of the scene, the base object representing a visualisation instance, continues with the introduction of all plugin types and of the client/server version of the engine. Section 4 describes implementation details, covering manager classes responsible for data and scene management and specifics of the client/server variant. Here we also cover several implemented plugins with more detailed description of a specialised rendering algorithm using run-length compressed data. This algorithm is useful for visualisation of huge voxelised objects defined by implicit functions.

2 Visualisation Solutions and Related Work

The existing volume visualisation applications and tools predominantly differ in whether they deal with just visualisation or can be used also for other tasks as, for example, data processing. In this section we will cover the visualisation part first and then have a look at the applications with data processing, which are based on the data-flow model.

2.1 The Scene

Visualisation is in principle done by rendering a virtual scene. This scene usually consists of the data volume or volumes and other el-

ements modifying the process of visualisation. These usually include a virtual camera, lights, clipping geometry or a transfer function. All these elements have to be stored internally in appropriate data structures. For applications which provide a static set of visualisation algorithms, it is convenient to store these elements in structures native for these algorithms. For example, if rendering is done solely on a GPU, data will be most probably stored in a 3D texture, clipping will be done through hardware supported clip planes and transfer functions will be represented by 1D or 2D textures. While this is suitable for GPU rendering, it is totally useless if the computation is to be done on a CPU.

Having defined the scene, rendering by a visualisation algorithm can be performed. The tools usually offer a set of visualisation algorithms which eventually may be extendable by new ones. When rendering is done on a GPU, a common way of allowing user to adjust the visualisation process is to allow modification of the GPU shaders [Meyer-Spradow et al. 2009]. These allow to change the way how the optical properties are mapped to data and the way how samples are blended together. To achieve extendability beyond this it is required that new algorithms can be added to the application. The possibility to add arbitrary algorithms implies that the input data entering the algorithm have a flexible internal representation. For GPU renderers it is natural to use 3D textures. A specialised algorithm might, however, need to receive data in some other representation and so the application has to be flexible enough to enable this.

2.2 Application Types

One type of commonly seen applications are GUI tools which implement all necessary tasks. If they are implemented in a monolithic fashion they combine and integrate all processes together. As dependencies between various parts of such application are usually strong, it is difficult to use only parts of the application or to add new functionality to it. Usually the only possibility for modification of visualisation algorithm lies in the mentioned replacement of the shaders. Another weak spot of this type of application is volume file format support. Each tool usually uses its own file format [Landis 2002; Simian 2002] with optional support of some more general file formats like uncompressed raw data. Depending on the provided functionality it may be possible to do certain tasks, as, for example, exact camera setting by specification of numeric parameters, batch processing, creation of animations, etc. One reason for this is that many of these tools are developed as research ones or demonstration applications for a new visualisation method, where the user interaction plays only a negligible role.

This class of applications is exemplified by the OpenQVis[1] project which was presented in [Landis 2002] (Figure 1). The related GPU-based ray caster was presented in [Stegmaier et al. 2005].

Another common approach used in visualisation is based on the data-flow model [MeVisLab 2009; SCIRun 2009; Meyer-Spradow et al. 2009; Šrámek et al. 2004]. Applications using this approach offer isolated objects (we call them filters) which take various number and types of input, perform some processing and produce one or more outputs of various types. The number and type of each input/output is defined by the filter. The application provides connectors which can be used to connect output of one operation to input of another operation. These applications can be used to create complex operations on the data consisting of an acyclic oriented graph of filters connected by connectors. This approach enables to create complex sequences of operations in a simple way. Solutions based

[1] `http://openqvis.sourceforge.net/`

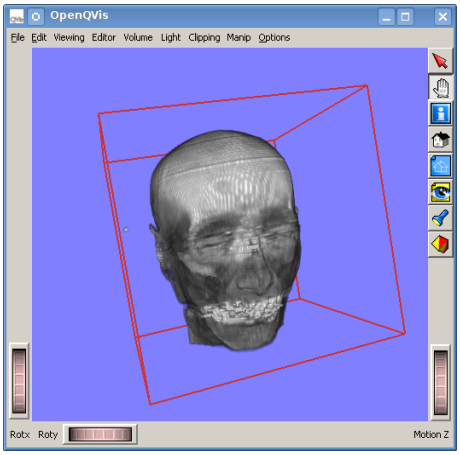

Figure 1: OpenQVis application.

on the data-flow model are also highly extensible because of isolation of various basic operations into separate objects. Visualisation is usually realised as a single network object with similar properties as it was described in the section 2.1. Loading of a volume file is no longer a problem, as this is realised by a separate object. Support of additional data file formats can be added by implementing new file reader objects. Ability to extend the application by new internal data representations and visualisation algorithms depends on the way how filters and connectors are designed. Usage of some specialised internal data representation might force the user to adapt core application objects resulting in rebuilding of the whole application. This might not be possible in case of closed source solutions or might be too laborious and require deeper knowledge about the system otherwise.

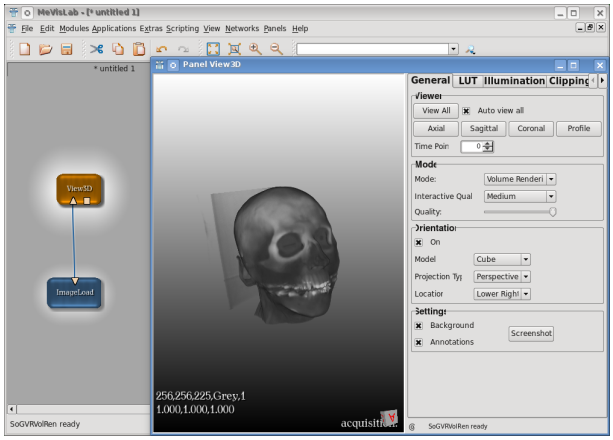

Figure 2: MeVisLab application with file reader and 3D volume viewer objects.

Examples of applications based on the data-flow model include MeVisLab [MeVisLab 2009] (Fig. 2) or SCIRun [SCIRun 2009] of University of Utah. Voreen [Meyer-Spradow et al. 2009], although targeted mainly on visualisation, is also based on the data-flow model (Fig. 3). The f3d tools use process-based filters for data processing and separate visualisation application for viewing (processed) data files [Šrámek et al. 2004]. Here, however, the networks are not set up by graphics tools but rather by shell scripting [Varchola et al. 2007].

To sum up, existing applications have various disadvantages at certain situations.

- Closed-source and proprietary applications provide limited possibilities for extending the application.
- Monolithic applications can be extended only at the cost of rewriting the whole application.
- Applications implementing a set of visualisation algorithms usually use a certain internal data representation and either do not allow to add arbitrary rendering algorithms or it is possible only at the cost of rewriting large portions of the code.
- Definition of the scene may permit only parameters meaningful for the set of available rendering algorithms or these may allow only certain supported values. This can be exemplified by the clipping geometry where only 6 clipping planes might be supported (limitation of natively supported clip geometry by a GPU).
- It is not possible to use the application in text-mode or for batch processing.
- Development of specialised user interfaces for certain groups of users might not be possible, because of inability to arbitrarily connect the control elements to visualisation parameters.

3 The VRE Architecture

In this section we introduce architecture of the VRE engine, cover its key characteristics and architectural decisions and explain reasons why we chose them. Implementation details are mainly covered in the next section, however, some have to be mentioned alongside the description of the architecture decisions.

As we were mainly interested in visualisation of already preprocessed data, we based the engine on the scene model. Thus, an application based on VRE can be a standalone one, or VRE can become basis of a visualisation object in a data-flow application.

The primary purpose of the engine is to provide the user (developer) with a flexible environment, which would fulfil his/her requirements in experimentation with visualisation algorithms and user interfaces as well as in development of real visualisation applications. From this point of view the crucial features of the engine design are flexibility with respect to various input file formats and technical characteristics of the input data (spatial resolution and discretisation, variable sampling patterns), different algorithm-specific possibilities for data memory representation (compression, blocking, storage on a GPU, etc.) and finally implementation of various visualisation algorithms. Therefore, we designed the engine as a library with a flexible application interface, with a scene object in its core. The scene object is extendable by a multitude of plugin types to achieve the mentioned goals. Individual components of the engine are covered in the following subsections.

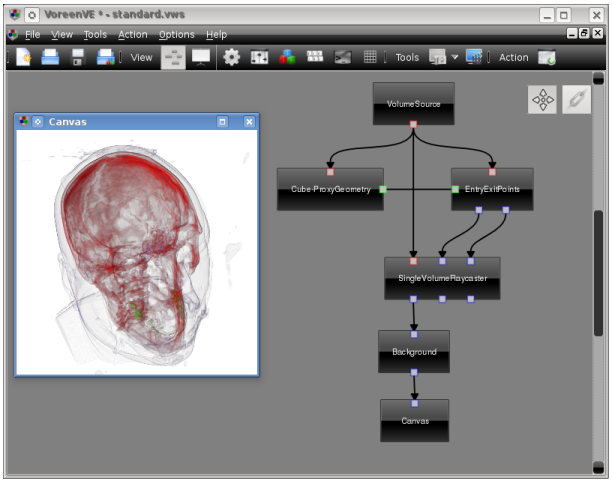

Figure 3: Voreen with default visualisation network.

3.1 The Scene Object

A virtual scene in volume visualisation applications consists of a data volume or volumes and other elements like virtual camera, clip planes, etc. Visualisation algorithms often require also other parameters which adjust the rendering. Although these parameters are closely related to the visualisation algorithms, they can be put next to other scene elements as they modify the way the scene is rendered. Various algorithms may require various parameters, but some of them are more common than others. We identified these and made them a part of the scene object (camera, light sources, clip planes, 1D and 2D transfer functions and settings for displaying auxiliary information as, e.g. rendering speed in fps, scene bounding box and others). As not all renderers need to use every scene element, unsupported ones are silently ignored.

In order to provide support for additional (custom) parameters the scene object should be extendable by an arbitrary number of supplementary parameters alongside the predefined ones. As they are unknown to the engine, they should contain name and description informing the user what does each parameter control and which values are allowed. This information should be set by the visualisation algorithm. These parameters can be of various types; we believe, however, that numeric, Boolean and string types, with addition of records (structures in the C++ terminology) and numeric arrays should cover most of the possible requirements.

3.2 Extendability

One of the most important requirements put on the engine is to enable simple extendability by adding new functionality, mainly various visualisation algorithms, their various implementations, data representations etc. As modifying the whole engine code is not acceptable, functionality which is prone to be extended was isolated from the rest of the code. A stable but flexible interface is a requirement which allows implementation of various algorithms in an easy way. When such isolation is achieved, these objects can be placed in separately built modules or plugins which can be easily installed. It is then the end-user who selects which algorithm should be used, which data file should be opened and so he/she either directly or indirectly selects plugins. The user should be therefore able to obtain information about every plugin.

3.3 Multiple Scenes

In our design we explicitly support existence of multiple scene objects. These can refer to different data volumes or to the same data volume. Each scene has its own parameters and its own instance of the rendering algorithm. We adopted a centralised approach where the scenes are owned by a single object – the scene manager. This

allows us to utilise resources more efficiently, as loaded data volumes can be shared among the scenes. Sharing could have been used also in a decentralised architecture, however it would be user's responsibility to take care of all the necessary tasks to be able to achieve it.

Motivation for existence of multiple scenes can be found in graphical user interface applications. When graphical user interface is used, repeated user interaction is expected which results in rendering of lots of images in succession. In a single scene setup, if more views of the same data set with different rendering parameters are simultaneously displayed, changes in certain scene settings can cause unacceptable overhead. If the only difference in the views is the position of the camera, there is no overhead in rendering the first view, followed by changing the camera setting and finally rendering the second view. A significantly different situation is, however, if views use different rendering algorithms with different internal representation of the data. Switching of the renderers would require freeing internally loaded data and then reloading of the data in the other internal format. As this type of operation is rather lengthy the overhead would be too high to achieve decent rendering times.

The described problem can be solved by using a separate scene for each view. Each scene will hold its own parameters and these would be only changed if the corresponding view is modified. Although independent in the means of parameters, scenes should be able to share the loaded data volumes. If multiple views of the same data file by the same rendering algorithm is used, the data volume stored internally can be shared by all scenes. Sharing could occur anytime when two rendering algorithms use the same internal data representation and the same volume file.

3.4 Plugin Types

We already mentioned the need for plugins. In this section we cover their types and explain their purpose.

3.4.1 The Loader Plugin

Loaders are the most obvious type of plugins as they take care of data files of various formats. A loader acts as a translator of the input data file format to a plain uncompressed one, thus providing a general interface for reading of files. File is not loaded as a whole, but instead the loader acts as a random-access reader, similar to the I/O stream objects of common programming languages. The `VRE` loaders also hide details about the way how is the input data stored on the storage medium. For example, in the case of the DICOM format [ACR-NEMA 1993] each slice is stored in a separate file. The DICOM loader would hide this specificity and acts as if there was only one data item available. Although this refers to loading of the data volume internally it is convenient to use this functionality also for browsing the file system. The engine provides interface for enumerating volume data files in a directory, listing only files of supported formats and substituting multiple files of the same data volume with a single virtual one.

Various rendering algorithms may require different memory representations of the rendered data. Therefore, a loader should provide as flexible interface as possible and implement appropriate internal optimisations. The decision whether to load the whole file initially or to allow access to the file at any time has to take all possible situations into account. As we try not to impose any limits on the possible rendering algorithms, we have to expect that it even may be necessary to reload the data from the file after a specific parameter change. This favours the solution with the loader available at any time and not only in the moment of initial reading of the data. Another example, when this approach might be preferred, is when a really huge data volume is rendered, which exceeds the memory storage possibilities. In this case a strategy to bypass the memory limitations have to be adopted, with bricking or sequential reading being the most common ones in use. It may be, however, inefficient to buffer the bricks to disk during successive renderings and recreation of bricks for each frame may be the right solution.

3.4.2 The Builder Plugin

Naturally, volume data has to be stored internally in memory in an appropriate way as rendering algorithms rarely access data in the sequential fashion of data files. The most obvious example of such special representation is the case of rendering on a GPU when data is stored as 2D or 3D textures in its graphical memory. One possibility at hand is to let the visualisation plugin to implement its own memory representation. However, there exist common ways how to store data internally, which can be shared among different types of renderers. It would also be inefficient to ask the programmer of the rendering plugin repeatedly to take care of storing the loaded volume data. Therefore we introduced builder plugins which use a loader to access data files and store data in appropriate internal structures. One builder can be then used by multiple renderer plugins provided that they use the same internal data representation. As the memory capacity is the common limitation prohibiting storing the whole data in memory at once, bricking is often used. It is therefore convenient to include this functionality into the base builder design. As the concept of bricking is rather general and does not depend on the actual internal data representation, it can be implemented above any particular builder. Between subsequent renderings, bricks can be cached on the hard disk and loaded only when required. The exact bricking strategy, however, can be controlled by means of global engine parameters. For example, bricking can be switched off by specification of only one brick.

3.4.3 The Renderer Plugin

The renderer plugins in `VRE` implement various rendering algorithms. Each renderer uses a particular internal data representation which is realised by a builder plugin. Therefore, the renderer should keep information about the required builder and a connection should be established between these plugins. To be able to render the data the renderer requires also access to other scene elements in addition to the volume data. The plugin may copy information from scene elements to its own internal structures. For example, a transfer function may be stored in a texture in the case of GPU-based rendering.

The scene object is positioned in the hierarchy of the `VRE` objects higher than the renderer. This is because we want the scene to preserve the state even when renderers are changed. Further, because we do not want to limit rendering to some architecture or device, we have to provide ways how to represent the rendered result in a general way. For example, when rendering is done on the CPU, the output is stored in a memory buffer, which can be then saved to a file as an image or copied to a graphical window or context. Thus renderers do not need to support all possible image formats and alignments of data as the necessary conversion is performed by the engine.

When rendering is done in a GUI application, we naturally want the rendered image to appear in a window. Different operating systems and desktop environments use different objects and resources to access window's content. Selection of an universal API to access

the screen is therefore necessary. OpenGL is such API as it is platform independent and available on different operating systems [Kilgard 1996]. Usage of various GPGPU environments is also possible as they provide means for OpenGL interconnection [CUDA 2010; OpenCL 2009].

3.5 Client/Server Rendering

The proposed architecture strictly separates to individual objects user interaction, scene, data management and rendering. Extension of this architecture to remote rendering is therefore straightforward. The engine should be just divided in server and client parts and both should be extended by the necessary communication capabilities. The server receives commands sent over the network from the client, interprets them and executes the corresponding calls. These can be reading or modification of scene elements, assigning data files or renderers to the scene or requests for rendering. Results of the calls can be serialised and sent back over the network to the client. In the case of rendering the result is stored in a buffer and is subsequently sent as binary data to the client.

The client part of the engine should provide exactly the same application programming interface as the local version and additionally should transparently encode function calls and send them over the network. After receiving the results, these are deserialised and used as if all the computing was done locally. In the case of rendering, either the received binary data is provided to user directly or it is rendered to the OpenGL context in the case of GUI applications.

The proposed solution does not utilise local resources or hardware which can be an advantage in various cases. One model of usage is when users work on inexpensive workstations and rendering is done on special dedicated server with high-end components. Other case can be rendering of huge data sets, originating, for example, from the Visible Human project [Spitzer et al. 1996]. The data can be stored centrally at one place without the necessity to have a copy of it on each workstation. The introduced network overhead should not present a problem as the proposed cases of application expect either low performance workstations or very large data volumes. Image compression, which would be necessary in the case of slow interconnection, is nowadays a well explored area [Myojoyama and Saitoh 2007]. The advantage of the proposed client/server solution resides in that the application can be recompiled easily to work either locally or remotely owing to just minor differences in the code, as interface of both versions are identical and network communication is transparent to the user.

4 Implementation

The priorities in implementation of the proposed rendering engine resided in its platform independence and coverage of as wide range of applications as possible. Further, rendering speed was another important requirement. Therefore, we chose the standard C++ language and the already mentioned OpenGL for its implementation. We used the POSIX threads as they are native on Linux and a Windows implementation is also available. The internal components use XML for which the TinyXML library was used. The engine is released as open-source software under the GPL licence.

We used an object-oriented approach to implement the engine. The main interface is provided by a couple of classes and polymorphism is used in plugin implementation. The engine defines interface classes for all plugin types. A plugin is then realised by deriving class and implementation of the interface. It is then compiled into a dynamic library, containing functions which create and return instances of the plugin classes.

4.1 Multi-threading

As the engine also allows usage in the client/server setup, we needed to support concurrent execution of operations by multiple users. Because we wanted the volume data to be shared between scenes, all users should operate on the same engine instance. In order to not to block other users when some operation is performed, the engine or at least its server side has to support multiple threads of execution. We finally decided to include threads in the implementation of the engine even when used locally as it brings certain advantages. For example, if the user works with two renderers at the same time and one uses GPU for rendering and the other runs on the CPU, they can work simultaneously without fighting for resources. Other example when multi-threading might be useful is when operating on a Windows GUI application. When using the Windows API, each application has an event queue which is processed just by one thread. When doing lengthy operations (which volume rendering might be) either the user interface becomes unresponsive or the programmer has to use multiple threads internally. By adopting threads in the engine, we can relieve the programmer from taking care of these details. The solution resides in providing asynchronous operations which immediately return without waiting for the computation to be finished. The programmer has, however, to provide a callback or an event handler which is called on operation completion.

Inclusion of threads introduced certain complications which had to be taken care of. When multiple threads run concurrently and operate on shared variables, they might break consistency of the data exactly at the moment when data is accessed from two or more threads. On a multicore system threads might be really executed concurrently and even atomic operations may cause such problems. Therefore, we had to adopt a synchronisation strategy. From the programmer's point of view, most of the engine parts are thread-safe, however some procedures have to be observed. A user most of the time access only the scenes and these have to be locked prior to use. The lock can be read-only or for reading and writing. There can be multiple reading threads on a scene, but writing is exclusive. As user interaction with a scene is most likely to be sequential, this introduces almost no programming complexity for the user. Internal protection against access from multiple threads has to be enforced when loading and freeing data (which is potentially shared among scenes) and operations on scenes like rendering or setting of scene elements.

4.2 Managers

The engine is implemented as a static library. As we mentioned in the architecture part of the paper, the engine has centralised design and the user does not need to care about communication between various components of the engine. Implementation therefore provides one main class through which the user works with the engine. We call this class the main manager as it manages all engine resources and processes. It contains two more manager classes – the scene and the data manager.

4.2.1 The Data Manager

The data manager is responsible for loading and storing of the data. It contains a data repository for each internal data representation

which is used by an active renderer. Repositories contain instances of builder plugins which themselves contain instances of loader plugins with opened volume files. Figure 4 illustrates configuration of the data manager when three files in two different internal representations are opened. In order to enable sharing data among scenes the data manager keeps the counter of scenes which use each opened data file. If the user assigns a data file which is not yet loaded in the required internal representation to a scene, the data manager loads this file from a disk. If the data file is already loaded, only the reference is returned and the counter is incremented. Freeing of the data works similarly. When the data file is removed from a scene, the counter of active users of the data volume is decremented. The data is actually freed only when the counter reaches zero.

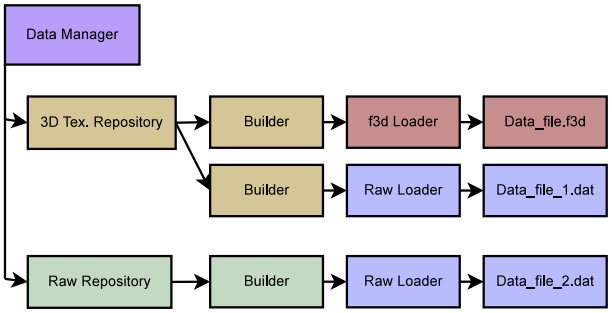

Figure 4: The data manager. Repository for 3D textures and data stored in plain uncompressed representation in main memory are presented. Two files are loaded as 3D textures, one in the f3d format and one in the raw format. The other repository has one raw file loaded.

4.2.2 The Scene Manager

Because the engine allows existence of multiple scenes, they need to be managed. This is the responsibility of the scene manager, which has a public interface so that the user can access the scenes. The user can add new or remove existing scenes, modify scene elements and assign data files and renderer plugins to a scene. Configuration of a scene manager with a couple of scenes is illustrated in Figure 5. Rendering of a scene can be done either to an OpenGL context or to a memory buffer. Synchronous and asynchronous methods are provided. In the case of synchronous variants, execution is done from the calling thread. As it was already mentioned, the asynchronous variants return immediately and execution is done from a separate thread. In case of an OpenGL rendering it is required that an OpenGL context is bound to the thread. If a new thread for each asynchronous operation was created, lot of OpenGL context binding would be done. Because this operation is rather costly, it is imperative that one minimises their number. Therefore each scene has two persistent threads which upon completion of an operation are not discarded. They fall asleep until the next operation request arrives instead. This way the OpenGL context binding can be preserved between subsequent asynchronous operations. One thread operates on user–provided OpenGL context, for example belonging to a Window created by the user, and the the other thread operates on an internal invisible context. The second thread is necessary for situations when an OpenGL context is required, but user did not provide any. Example of such situation is rendering by OpenGL when using command line application.

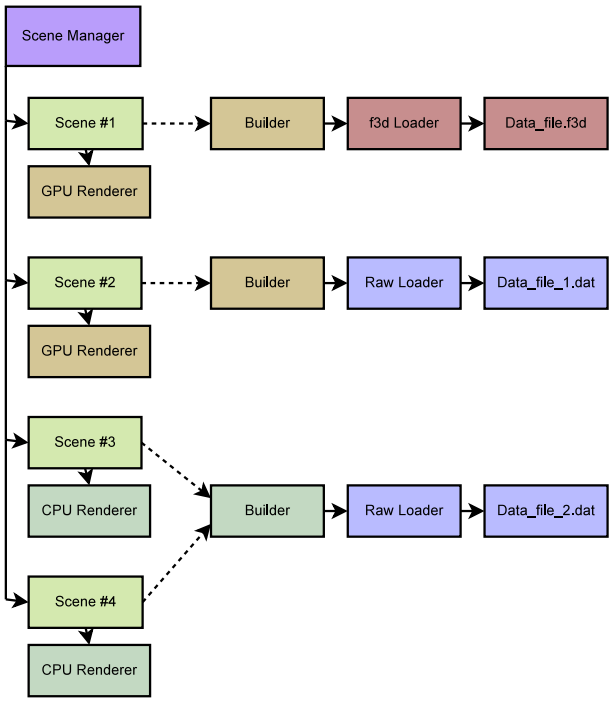

Figure 5: The scene manager. Four scenes were added and data files were assigned. Scenes #1 and #2 use a GPU renderer which requires a 3D texture builder plugin. Scenes #3 and #4 use a CPU renderer and as both were assigned the same data file, they share the same builder instance. Builders and loaders are actually stored in the data manager as was illustrated in Figure 4.

4.3 Server

Implementation of the engine supports multiple scenes with multiple data files opened and visualised by different renderers. As the local engine implementation is thread-safe, the server operates on single local engine instance and only takes care of communication with clients and ensures that clients access just the scenes owned by them. All communication is done by sending XML-encoded information. This includes target manager name, method name and all parameters of the call. All requests are put in a queue which is served by number of threads. Each operation is executed and the result is again encoded using XML. The client waits until it receives result from the server. Asynchronous operations return just acknowledgement that operation request was received. After the asynchronous execution finishes, the client is contacted and invokes a callback call.

Comparison of rendering times for local and remote rendering is shown in Table 1. First, we measured overhead which comes with use of the engine, when only a simple renderer filling the whole frame buffer with homogeneous colour was used. Then we rendered images using a 3D texture based GPU renderer with object-aligned slices using OpenGL. Please note that the used renderer is suitable only for comparison of times between various rendering modes, as it was not optimised. We rendered the data directly to an OpenGL window and to a memory buffer when image was extracted from an invisible OpenGL context. Next we used the client/server version in a configuration when both client and server were located on the same machine. On the server side the rendered image was extracted from the OpenGL context and sent over the network. Client either provided the received image buffer to user or

Table 1: Rendering times for various configurations of the engine. A 128x256x256 volume was rendered to an 512x512 image using a 3D texture based GPU renderer with object-aligned slices using compositing. From top to bottom: overhead of rendering to an OpenGL window, rendering time for GPU rendering to an OpenGL window, rendering to an image file including additional time needed to extract the image from the OpenGL context. Times for client/server configuration follow, starting with times for rendering when both client and server are on the same computer and when rendering is remote. Images sent over network have to be rendered to client OpenGL window in the last two configurations.

Configuration	Time [ms]
Empty to OpenGL	1.17
GPU to OpenGL	183.96
GPU to Memory	190.48
GPU to Memory (local)	348.44
GPU to OpenGL (local)	372.68
GPU to OpenGL (remote)	557.12

displayed it in an OpenGL window. Finally, we measured rendering to OpenGL window in configuration when client and server were on different computers on a 100Mbit/s local network. By comparing the neighbouring rows in the table, overhead of various operations can be estimated.

4.4 The Implemented Plugins

Apart from the main engine and its server version we implemented several plugins. The loaders for raw data files, f3d [Šrámek et al. 2004] and nrrd [Nrrd 2008] data format were implemented. Builders storing the loaded data as uncompressed bricks in the main memory and as a 3D textures in the graphics memory are available. The implemented renderers are CPU and GPU ray casters and a 3D texture based GPU renderer using view-aligned slices. We are currently using a simple command line interface and simple GUI applications for testing. Implementation of a full-featured GUI is underway. Apart from the simple testing builders and renderers we implemented a renderer using run-length encoded data on GPU, which can be used to illustrate how to develop new plugins requiring special memory representation.

4.4.1 GPU Renderer for RL-compressed data

Motivation for this type of renderer stems in the domain of volumetric data obtained by voxelisation of complex models defined by implicit functions [Parulek et al. 2009]. If such models have sharp or small details, they need to be densely sampled and therefore result in huge grids of 1000^3 and more voxels. However, in such data we are interested only in surfaces of the objects and thus it is sufficient to fully represent only voxels close to the surfaces, i. e. only a marginal part of the volume. The rest, voxels inside or outside the object, can be compressed by the run-length (RL) compression technique. This technique results for noise-free voxelisation data in very high compression ratios up to $1:20$ [Novotný 2007].

Upon creation, a builder reads the volume file and compresses the data. Such a compressed volume is represented by a 2D array of compressed rows, where off-surface areas are encoded by the RL technique and the surface areas are represented by uncompressed runs of voxels in order to achieve fast voxel access [Novotný 2007]. The uncompressed runs of surface voxels are stored one by one in a 2D texture (we refer to it as the surface texture). Another 2D texture contains encoded lines (we call it the line texture). In this texture pairs of values – count of voxels and the offset to the surface texture – are stored consecutively for each row. The compressed inside and outside voxel runs are fictively stored at offsets 0 and 1 so one can distinguish between compressed and uncompressed areas easily. By this technique we in fact emulate pointers. As we store lines of the volume one by one, it is inefficient to search the whole line texture each time a voxel is requested, so we use one more additional texture – the index texture – storing offsets to beginnings of each line. All these textures are created by the builder and stored in the graphics memory.

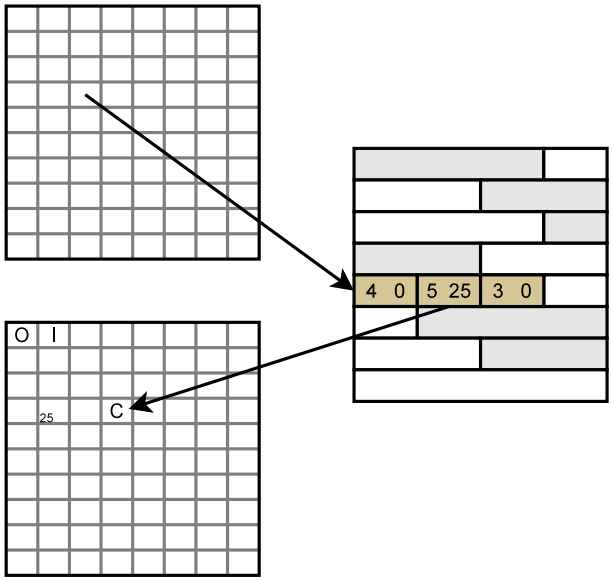

Figure 6: The process of retrieving a voxel value. The index texture contains offsets to the line texture for the requested y, z coordinates. For illustration let the x coordinate be 6. Shader counts, until it finds the pair for the requested voxel – in our case the pair $5, 25$ ($4 \leq 6$ and $6 < 4+5$). At offset 25 in the surface texture voxels of uncompressed area start, but as we need value of the third voxel of the uncompressed area, we use offset 27. C denotes value of the requested voxel. O and I are two special values – the outside and inside voxels.

Visualisation is performed on a GPU by a fragment shader written in the GLSL language. The ray casting method is used. At the sample points values of volume data need to be computed. As only outside/inside voxels are compressed, we do not need to actually fully decompress it, since it is sufficient only to distinguish between the uncompressed surface voxels and voxels belonging to outside/inside of the object. When the voxel at position $[x,y,z]$ is requested, the shader program looks up in the index texture the offset to the line texture containing the compressed yz-line. After the line is located, the shader reads the count-offset pairs until the appropriate one for the requested x position is found. Then, based on the offset value the shader decides whether the requested voxel is inside/outside one or is located in the surface area. In the first case the value from the surface texture directly at the offset is returned as the offset points to value common for all inside and outside voxels respectively. The surface area voxels with offset greater than 1 are located by adding position of the voxel in the surface area to the offset. The process is illustrated in Figure 6. Image 7 was rendered using the described renderer.

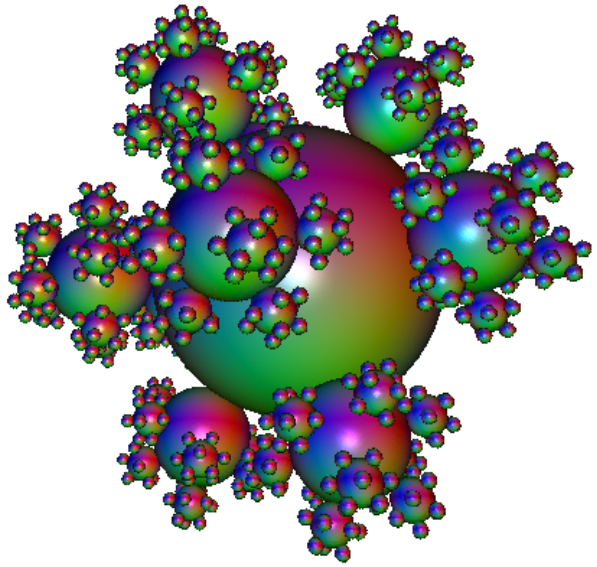

Figure 7: Image rendered using the GPU renderer for RL-compressed data.

5 Conclusions and Future Work

We presented architecture of a volume rendering engine which is aimed at providing high extendability and support of various operating systems, various hardware architectures and various rendering algorithms. The proposed engine was implemented together with a client/server version enabling remote visualisation and couple of demonstration plugins. We described details of a specialised renderer for RL compressed data. In the future we plan to incorporate support for GPGPU frameworks like CUDA and OpenCL into the engine. Development of a full-featured GUI utilising the engine is already underway.

Acknowledgements

This work was supported by the Slovak Research and Development Agency under contract No. APVV-20-056105 and by Comenius University under contract No. UK/515/2010.

References

ACR-NEMA. 1993. *Digital Imaging and Communications in Medicine (DICOM): Version 3.0*. ACR-NEMA Committee, Working Group VI, Washington, DC.

CUDA. 2010. *NVIDIA CUDA - Programming Guide, Version 3.0*. NVidia Corporation.

ENGEL, K., HADWIGER, M., KNISS, J. M., REZK-SALAMA, C., AND WEISKOPF, D. 2006. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA.

KILGARD, M. 1996. *OpenGL Programming for the X Window System*. Addison-Wesley, Reading, MA, USA.

LANDIS, H., 2002. High-quality volume graphics on consumer pc hardware. ACM SIGGRAPH 2002 Course #42 Notes.

MEVISLAB, 2009. MeVisLab: Medical image processing and visualisation [Internet]. `http://www.mevislab.de`. [cited Feb. 28, 2010].

MEYER-SPRADOW, J., ROPINSKI, T., MENSMANN, J., AND HINRICHS, K. 2009. Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations. *IEEE Computer Graphics and Applications 29*, 6–13.

MYOJOYAMA, A., AND SAITOH, H. 2007. Real-time volume rendering by network image transmission. *IFMBE Proceedings 14*, 4, 2427–2430.

NOVOTNÝ, P. 2007. *Voxelization of Solids With Sharp Details*. PhD thesis, Comenius University.

NRRD, 2008. Nearly raw raster data [Internet]. `http://teem.sourceforge.net/nrrd/index.html`. [cited Feb. 28, 2010].

OPENCL. 2009. *The OpenCL Specification*. Khronos OpenCL Working Group.

PARULEK, J., ŠRÁMEK, M., AND ZAHRADNÍK, I. 2009. Geomcell, design of cell geometry. In *Recent Advances in the 3D Physiological Human*, N. Magnenat-Thalmann, J. J. Zhang, and D. D. Feng, Eds. Springer, 21–36.

SCIRUN, 2009. SCIRun: A scientific computing problem solving environment [Internet]. `http://www.scirun.org`. [cited Feb. 28, 2010].

SIMIAN, 2002. Volume rendering [Internet]. `http://www.cs.utah.edu/~jmk/simian/index.htm`. [cited Feb. 28, 2010].

SPITZER, V., ACKERMAN, M. J., SCHERZINGER, A. L., AND WHITLOCK, D. 1996. The visible Human Male: A technical report. *Journal of the American Medical Informatics Association 3*, 2, 118–130.

ŠRÁMEK, M., DIMITROV, L. I., STRAKA, M., AND ČERVEŇANSKÝ, M. 2004. The f3d tools for processing and visualization of volumetric data. *Journal of Medical Informatics and Technologies*, MIP–71–MIP–79.

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 Volume Graphics 2005*, 187–195.

VARCHOLA, A., VAŠKO, A., SOLČÁNY, V., DIMITROV, L. I., AND ŠRÁMEK, M. 2007. Processing of volumetric data by slice- and process-based streaming. In *AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, ACM, New York, NY, USA, 101–110.