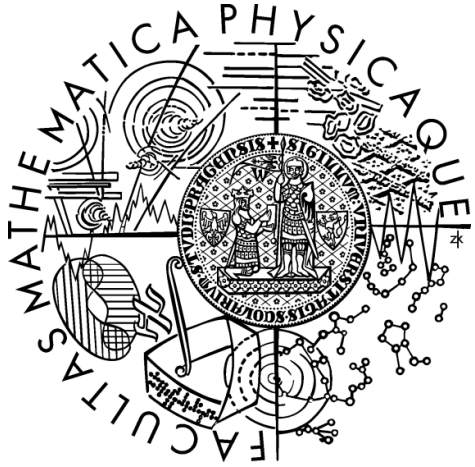
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jakub Hlaváček

Zpracování medicínských dat na GPU

GPU-accelerated processing of medical data

Department of Software and Computer Science Education
Supervisor: Mgr. Lukáš Maršálek
Study program: Informatics, Software Systems,
Computer Graphics

Chtěl bych poděkovat vedoucímu práce Mgr. Lukášovi Maršálkovi za rady a všechny připomínky, provázející tvorbu této práce.

Také bych zde chtěl poděkovat celé mé rodině za podporu během psaní této práce a během celého studia.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 18. 4. 2008 Jakub Hlaváček

Contents

List of Figures

List of Tables

List of Diagrams

List of Code Snippets

Název práce: Zpracování medicínských dat na GPU
Autor: Jakub Hlaváček
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí diplomové práce: Mgr. Lukáš Maršálek
e-mail vedoucího: Lukas.Marsalek@mff.cuni.cz
Abstrakt:

Cílem této práce je navrhnout a experimentálně implementovat ucelený systém zaměřený na urychlení a zjednodušení vývoje systémů pro zpracování a zobrazování medicínských dat v prostředí C#. V dnešní době existují jak systémy orientované na postupy vycházející z vědeckého odvětví zpracování obrazu, jako jsou filtrace, registrace, segmentace a klasifikace, tak systémy zaměřené na zobrazování 3D dat. Neexistuje však konzistentní systém pro obě odvětví, který by navíc využíval možností současných grafických a vícejádrových procesorů a zároveň využíval výhod platformy .NET a jazyka C#.

V této práci uvádíme přehled současného volně dostupného software, návrh programového rozhraní a implementaci hlavních částí tohoto rozhraní. Důležitým rozdílem oproti ostatním systémům je, že naše implementace je od začátku psána v prostředí platformy .NET Framework, který zaručuje dobrý komfort pro koncového programátora a přesto výkon celého systému je díky využití všech zdrojů srovnatelný s nativně kompilovaným prostředím.

Klíčová slova: zpracování obrazu, segmentace, objemové zobrazování, vrhání paprsku, cg, .NET, C#

Title: GPU-accelerated processing of medical data
Author: Jakub Hlaváček
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Lukáš Maršálek
Supervisor's e-mail address: Lukas.Marsalek@mff.cuni.cz
Abstract:

The aim of this thesis is to design and experimentally implement a complex framework dealing with accelerating and simplifying the development of systems for processing and visualization of medical volume data in C#. Currently, there are application interfaces and their implementations for both, techniques based on image processing, like filtering, registration, segmentation and classification, and also for techniques based on 3D image visualization. But there is no consistent framework for both tasks, which would take advantage of features of modern graphics processing units and multi-core processing units along with features of .NET Framework and of language C#.

The thesis presents overview of current free and open source software, design of application interface, and implementation of main API features. One of important differences to other software is that the implementation has been developed natively in the managed environment of .NET Framework, offering a good level of comfort for an end application programmer, but system performance is comparable with natively compiled environments thanks to utilization of all resources.

Keywords: image processing, segmentation, volume visualization, ray casting, cg, .NET, C#

Chapter 1

Introduction

With computer hardware development, interactive visualization of the 3D data sets is more and more available in the last few years. These 3D data sets bring more possibilities and advantages to users from medical environment.

Computer tomography and magnetic resonance are common techniques applied in today medicine, producing 3D data. This thesis aims to simplify development of programs that deal with loading, processing and visualization of this type of data.

Difficulties with development of software with such a pipeline are mainly connected with the necessity of knowledge from both disciplines - image processing and data visualization - and with design of data structures, which must be efficient in algorithms from both tasks.

Basic review of computer science branches related to the main topic is presented in the next chapters.

1.1 Medical Data Image Processing

Digital image processing is a discipline where various algorithms are used to modify digital image data in a computer. A typical image processing cycle consists of the following steps:

- Digitization (acquisition of data)
- Preprocessing (image enhancement, image restoration)
- Classification (pattern recognition, segmentation)
- Encoding and compression

In our case the **digitization** is performed by CT[1], MRI[2], PET[3], SPECT[4] and others techniques, which produce 3D data sets ([1], [2]). A typical output of 3D

[1] Computer tomography
[2] Magnetic resonance imaging
[3] Positron emission tomography
[4] Single photon emission computed tomography

digitization process in medicine is a set of 2D slices, which are stored in the DICOM format (described in chapter 1.5). Next, we will assume, that we have a 3D matrix of values from the regular rectangular grid. This matrix is called a "volume" and each single value in the volume is called a "voxel" (from volume element, Figure 1). Each voxel can be addressed by an index (i, j, k). Number of voxels in the volume in each direction is called dimensions (dim_x, dim_y, dim_z). Dimensions of voxel are named vox_x, vox_y, and vox_z. A set of voxels, where one coordinate is constant, is called slice.

Real value of each voxel depends on the process of acquisition in a single scanner, but we will consider it as discretization of original continuous image function.

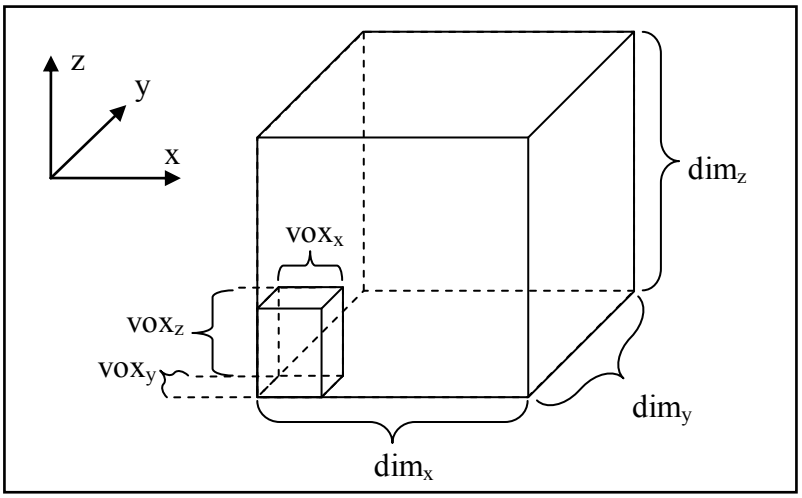


Figure 1: Volume and voxels relation

Preprocessing is a phase, which deals with correction of the image. Most important algorithms resolve tasks on modification of contrast and brightness, suppression of noise and edge detection. There is a range of algorithms applying a similar procedure. It consists of a loop through all output voxels, setting a new value for each of them from a small neighborhood of voxels from the input volume.

Another part of preprocessing is registration. Its goal is to find corresponding parts of two images and find transformation from one to the other. It can be widely used in medicine, when a patient is scanned in several time intervals and these scans must be compared for changes, or when there are several scans with different modalities.

Classification is the next task of image processing and its goal is to recognize certain parts of image, for example to divide the volume of the body into organs. This provides the possibility to overlook only these parts, which we are interested in, and don't be bothered by surroundings. The classification can be also used for measuring the organ's size and capacity, which is one of the possibilities, how to predict some diseases.

Image processing concerns mainly with 2D images, but most of algorithms can be generalized into the third dimension, or they can be applied to parallel 2D slices separately without spatial information (stored in third dimension).

Due to time efficiency concerns, the above mentioned algorithms use data, which are not compressed and thus they occupy unnecessary space in the memory. For long-time storage of data there is the **encoding and compression** task.

Book [3] from Gonzalez and Woods is recommended for further reading about image processing.

1.2 Volume Rendering

Volume rendering is a technique used to display a 2D projection of a 3D sampled data set. For all techniques we need the following entities:

- Data set. It is usually stored in regular grid.
- Projection matrix. It defines way of transforming a point in 3D space to 2D screen.
- Model-view matrix. It is generally called a "camera" and defines position and the viewing direction of the observer relatively to volume.

There are two main approaches how to display the volumes: surface fitting and direct volume rendering.

Surface fitting is based on transformation of the volume representation with regular grid to representation with a set of polygons. We will focus on direct volume rendering in this thesis, for more information about surface fitting see [4] and [5].

1.2.1 Direct Volume Rendering

Direct volume rendering (DVR) is more dependent on processor speed than surface fitting, but there are more possibilities of rendering and generally more accurate images can be generated.

DVR methods require every sampled value to be mapped to the opacity and the color, which is done with a "transfer function". Resulting sample color is then applied to the corresponding pixel of the frame buffer. This general process is the same for all rendering techniques described later.

Splatting [6] is a technique, where every voxel is splatted on the projection plane in back to front order. These splats are rendered with various profiles depending on the volume density and the transfer function.

Shear warp factorization [7] is a technique, where the viewing transformation is transformed in such a way that the faces of the volume become axis aligned with image plane and the voxels to pixels scale is fixed. Once all slices have been rendered, the buffer is warped into the desired orientation.

Texture mapping [8] approach is based on blending of textured slices. Volume is stored on the GPU in three sets of 2D textures (one set for each axis) and instead of stepping through voxels or rays, these textures are rendered using alpha blending. Second possibility is to store volume in one 3D texture and then render textured polygons using alpha blending. These polygons are defined as intersection of volume boundaries and planes parallel with projection plane.

Volume ray casting is a basic technique for volume visualization. We will describe this technique in more detail, because it is base for this work.

1.2.2 Volume Ray Casting

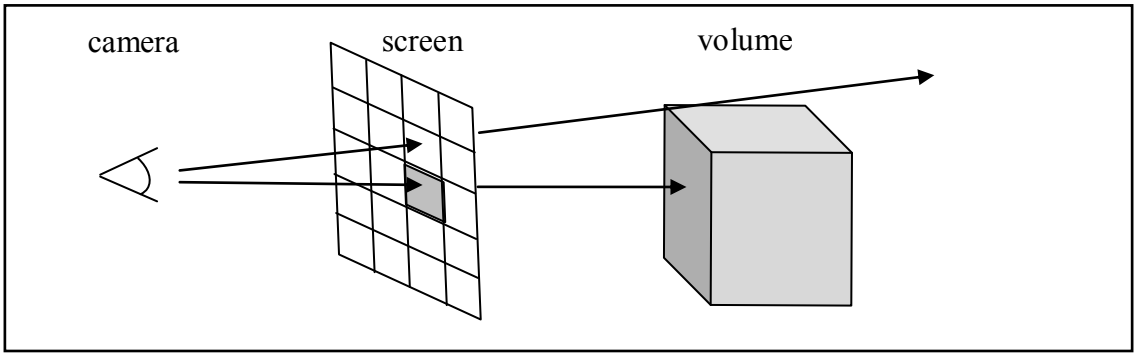


Figure 2: Rays are casted through each pixel of the projection plane

The idea is same as in the common ray casting. For each desired image pixel a ray (defined by origin and direction, Figure 2) is generated. The ray is clipped by the boundaries of the volume. Then the ray is sampled at regular intervals throughout the volume. The data are interpolated for each sample point from eight surrounding voxels values by trilinear interpolation (Figure 3). Than the transfer function is applied to form an RGBA[5] of the sample, which is composed onto the accumulated color of the ray, and the process is repeated until the ray exits the volume. The process is executed for every pixel on the screen to form the complete image.

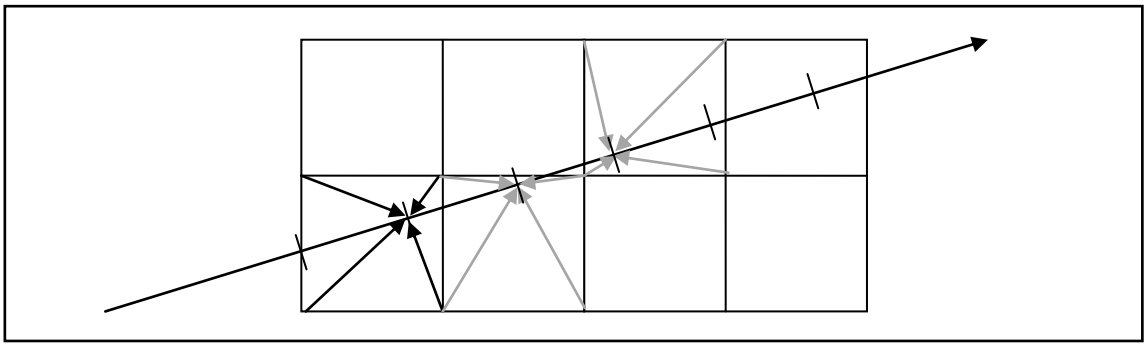


Figure 3: Sampling of points along the ray and linear interpolation (2D)

Composition of sample color depends on the type of the application. Common techniques are:

- Maximum intensity projection (MIP) – sample with maximum value on the ray is taken as a result.
- Isosurface ray casting [9] – it is a form of thresholding, where points with a specific value are displayed. Surface color is usually defined explicitly.

[5] three color channels (red, green and blue) and one opacity channel

- Compositing – final pixel color is determined from the mix of all sample colors and opacities encountered on the ray.

Isosurface ray casting is similar to the surface fitting, but data are not transformed to polygonal structure. User input to this method is a threshold value, which defines desired isosurface[6] in the volume. Because we step along the ray from start to end, we know exactly, between which two samples (Figure 4, green dots) intersection of the ray with surface is. Several possibilities are at this point. If the step size is enough small, we can simply chose nearest neighbor sample by intensity (red line). More accurate methods can interpolate the depth of surface from samples' intensities, or also iterative method can be used (blue lines). If the isosurface is not transparent, the computation of actual ray is finished. Otherwise, the loop can continue in computations along the ray and try to find another surface on the rest of the ray.

Once the intersection point is known, the isosurface color is assigned and optionally one of shading models can be used for further image improvement. The most popular model is Phong's illumination [33].

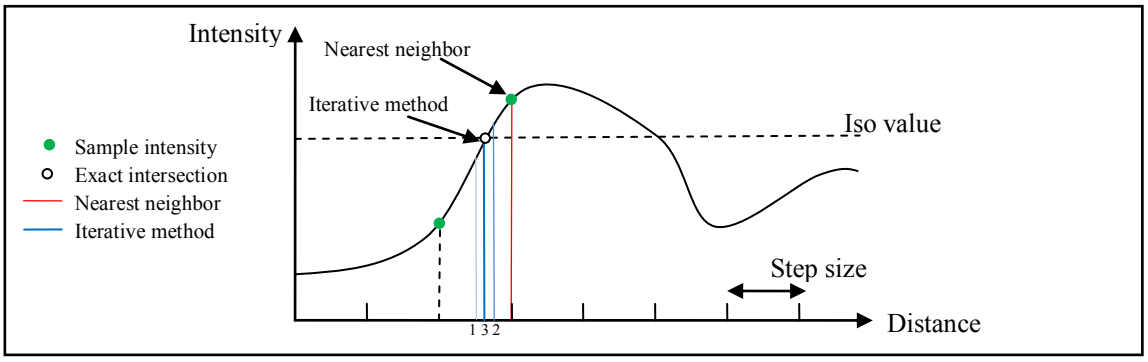


Figure 4: Isosurface ray casting

Color transfer function is a fundamental point of the compositing technique. There are typically many tissues in the medical image data, where each tissue has assigned its own interval of densities. The transfer function is an ideal way, how to simply separate these intervals during visualization (essentially, it's a form of classification of the data set, Figure 5).

The color transfer function defines mapping of voxel's value to the color. Usually, it is a function stored in a form of the look-up table, i.e. a 2D table, where along x-axis there are all possible values of densities and along y-axis RGBA values are assigned.

[6] Isosurface is surface, which represents points of constant value (e.g. density) within a volume.

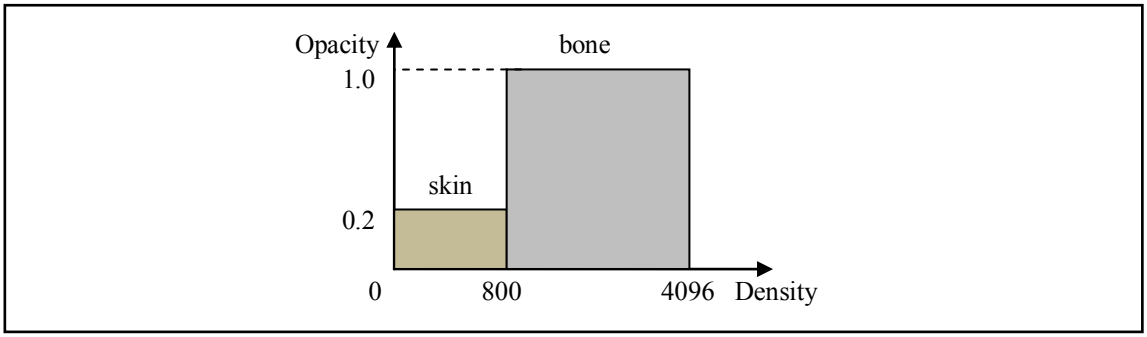


Figure 5: Simple classification with transfer function

To complete the idea of transfer functions, there is also possibility to map multi-component data on RGBA values with transfer functions, where one dimension must be added to transfer function table with each component stored in the data. This is useful, when e.g. volume contains more organs and the id of the organ is stored in the second component of the volume.

Compositing technique (emission-absorption case) idea can be derived from mathematical description of situation, when viewer looks into the volume along the ray and each point emits some intensity toward a viewer and all points on the way absorbs some of that intensity. Numerical approximation of that situation can be implemented in software using following formula for each step along the ray. The derivation is in [34].

$$\begin{aligned} C_{dst} &= C_{dst} + (1 - a_{dst}) \cdot a_{src} \cdot C_{src} \\ a_{dst} &= a_{dst} + (1 - a_{dst}) \cdot a_{src} \end{aligned} \tag{1}$$

Where

- C_{dst} is destination color
- C_{src} is sample color
- a_{dst} is destination opacity
- a_{src} is sample opacity

Main disadvantage of ray casting is its time consumption. Fortunately, in last few years the graphics hardware allows to implement the ray casting on a common consumer GPU[7], so nowadays it is possible to view images from ray casting in accurate quality in real-time [11]. On the other hand, ray casting has advantages against other methods. One of them is that transfer function can be edited in real-time. Other is that isosurfaces can be computed together with compositing technique.

[7] Graphics processing unit

1.3 Shader-accelerated Visualization

This chapter describes some basic facts about OpenGL[8] [12] and its features applied in our implementation. It is a software interface for graphics hardware specialized for developing applications that produce 2D and 3D graphics. OpenGL was started by Silicon Graphics Inc. (SGI) in 1992 and today it is considered as a fundamental interface for computer graphics.

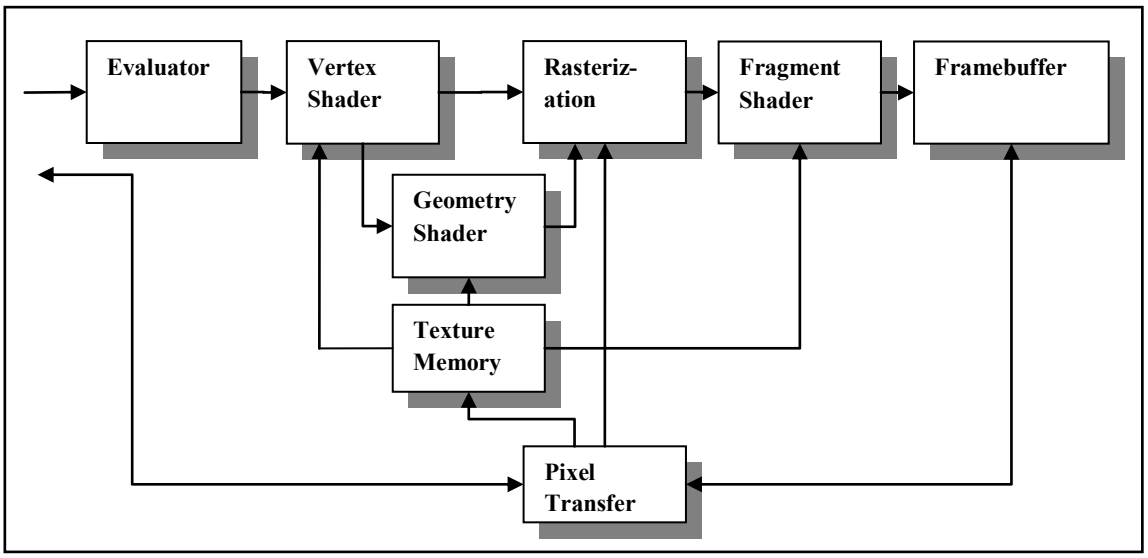


Diagram 1: A block diagram of programmable graphics pipeline

Diagram 1 shows main parts of the OpenGL. On the left two types of commands enter, some specify geometric primitives (like points, lines and polygons) to be drawn and other operations specify how the objects are handled at various stages.

Shaders are available for programming the GPU since OpenGL version 1.5. Shaders are small programs that are written in specialized languages (similar to C language), allowing modification of standard computation from the fixed-function pipeline (FFP). There are three types of them.

- Vertex shader replaces *per-vertex operations* from the FFP. In this program color, position and texture coordinates of each vertex can be modified.
- Geometry shader is a new type of shader that is placed immediately after the vertex shader. The difference from the vertex shader is that new graphics primitives can be generated, such as points, lines and triangles. Geometry shader is supported only in newest OpenGL extensions.
- Fragment shader replaces *per-fragment operations* from FFP, computing resulting color and possibly depth for each fragment of target image.

The advantage of GPU against CPU lies in parallelisms. Modern GPUs are SIMD[9] processors, having a number of programmable parts and thus are faster for

[8] Open Graphics Library
[9] Single instruction, multiple data

algorithms, where same instructions are used for multiple input data. This is typical for algorithms done in both *per-vertex operations* and *per-fragment operations*.

Overview of NVIDIA and AMD/ATI graphics cards support for shader model 3.0 and 4.0 is shown in Table 1. The fragment shader model 3.0 is the first fragment shader, where loops can be used in source code of program.

Shader technology opens a number of possibilities for GPU utilization in other tasks than graphics. For example there are implementations of Fast Fourier Transform, digital signal processing, neural networks, database operations and many others applications on GPU. Collectively these algorithms are called GPGPU[10] [13].

Shader Model	OpenGL Extension	Cg profile	DirectX version	NVIDIA GeForce Series	ATI Radeon Series
3.0	NV_vertex_program3	Vp40	9.0c	6	R520
	NV_fragment_program2	Fp40			
4.0	NV_gpu_program4 + NV_vertex_program4	gp4vp	10.0	8	R600
	NV_gpu_program4 + NV_geometry_program4	gp4gp			
	NV_gpu_program4 + NV_fragment_program4	gp4fp			

Table 1: OpenGL extensions and DirectX versions on current GPUs

1.4 .NET

Microsoft .NET Framework 2.0 [27] is a software component that is a part of Microsoft Windows operating systems. The .NET Framework includes a broad set of supporting class libraries for common programming problems like user interface, database access, web application, network communication, and others. Programs written for the .NET Framework are executed in a software environment (CLR[11]) that manages the program's requirements. The CLR provides services such as security, memory management, and exceptions handling.

The result of compilation of program's source code is MSIL[12]. When executing, the code is not interpreted, but compiled (in meaning of JIT[13]) into native code. The advantage of this concept is that there can be various languages and corresponding compilers only to the MSIL. On the other hand, there is little delay before a program is executed. Most widely known programming languages are C# and Visual Basic .NET.

[10] General-purpose computing on graphics processing units
[11] Common language runtime
[12] Microsoft intermediate language
[13] Just-in-time compilation

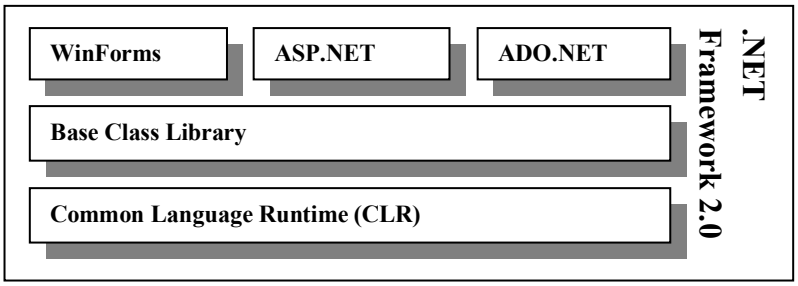


Diagram 2: Parts of .NET Framework 2.0

CLR frees the developer from the memory management (allocating and freeing of memory). This feature is based on reference counting and garbage collection. Each object stores a number of references on it. GC[14] then periodically (but not deterministically) checks these objects, whether the reference number is zero. If so, GC releases the used memory.

Both MSIL compiler and memory management are parts of CLR. An overview is shown in Diagram 2.

1.4.1 C#

C# [28] is a modern and high level programming language with many similarities to Java. Syntax of C# comes from C++ language. There are many common programming features supported, like object-oriented programming, templates and collections, delegates, partial classes, preprocessor macros, XML documentation system, and many others.

The language specification was approved as a standard by ECMA[15] and ISO[16], so that there can be other implementations than that from Microsoft. One of them is an open source project Mono [15], which now implements about 95% of .NET Framework 2.0 specification.

1.5 Format for Storing Medical Data

Digital Imaging and Communications in Medicine (DICOM) is a standard [14] developed by the American College of Radiology (ACR) and National Electrical Manufacturers Association (NEMA). Its main parts are file format definition and a network communications protocol.

The off-line media files correspond to part 10 of the DICOM standard. A single file contains both the header, as well as the image data. The header contains text information about the patient's name, dimensions of image, etc. The size of the header depends on how much information is stored.

[14] Garbage collector
[15] Ecma International, an international private standard organization
[16] International Organization for Standardization

Tag	Description	Example value
(0018,0050)	Slice Thickness	[5]
(0028,0010)	Rows	[512]
(0028,0011)	Columns	[512]
(0028,0030)	Pixel Spacing	[0.34765625\0.34765625]
(0028,0100)	Bits Allocated	[16]
(0028,0101)	Bits Stored	[12]
(0028,0102)	High Bit	[11]

Table 2: Most important elements stored in DICOM file header

Three-dimensional data sets can be stored in a single file image data or in multiple files, where each slice is stored in one file. A media directory file, the DICOMDIR file, must be present, providing index and summary information for all the DICOM files on the media. No information is allowed to be stored in a filename.

Many other rules are described in the specification, but the last we want to mention is that each file has a unique identifier, so that the software that wants to create DICOM files must be registered and must have a set of these identifiers assigned.

1.6 Goals of the Thesis

The following goals were defined at the beginning of the project. Each point represents one feature of the desired framework.

(g1) Compactness

Framework can read, process, and display volume data without unnecessary transformations between different representations of data, which could lead to both the system memory and the system speed performance losses.

(g2) Data abstraction

Volume data handlers should be independent of types stored in each voxel. We need the possibility to store different types of data in volume, but the processing and visualization tasks should be independent on the incoming data types.

(g3) Data processing

The goal of framework is not to define all possible image processing algorithms, but to provide a good interface for writing algorithms by "user programmer" (programmer, who uses our framework) aimed on volume data sets. This includes the possibility of parallel computing without any programmer's effort.

(g4) Data visualization

Framework should be capable of visualization of data in common ways from medical environment. That is *direct volume rendering*, *isosurface rendering*, definition of final color through color *transfer function*, displaying of output from segmented data.

(g5) Interactivity

We want the frame rate of the visualization to be as high as possible. Ideal is 20 frames per second or higher. Frame rate is important when end user moves or interacts with observed data. If user interface does not response in 50ms, it is very annoying for him.

(g6) Minimize time between starting the segmentation and displaying some results

When some complicated segmentation algorithm takes place in the processing pipeline, it is useful, when the end user can see temporary results before the algorithm finishes its activity. It is important in cases when user repeats his input, because he can stop the processing if the result is not what he expected. This should be performed by parallelization of tasks from the pipeline.

(g7) Written in C#

C# is a modern and simple object oriented programming language with good support for writing user interfaces. The framework should be natively written in it.

(g8) Visualization on GPU

Visualization should be accelerated on Graphics Processing Unit, which in addition frees CPU for other computations.

1.7 Structure of this Document

Chapter one introduces basic knowledge from the science branches related to this topic and lists the required features of our framework. Chapter two is about current software and their comparison. Design of application interface, problems and features are described in chapter three. Problems and specific implementation issues are described in the fourth chapter. Chapter five is devoted to the comparison between our new framework and Visualization Toolkit, leading free software in 3D computer graphics. The last chapter presents a summary of the work and evaluation of the targets. Suggestions for future work are also mentioned there.

Chapter 2

Currently Available Software

In this chapter we want to mention some of the existing software related to image processing or 3D visualization. It is a summary overview of the systems, their history, targets and their common usage.

2.1 ITK

The Insight Segmentation and Registration Toolkit (ITK, [16]), an open source image segmentation and registration software library, was developed for analyzing the images of The Visible Human Project[17]. The ITK development was funded from the National Library of Medicine (U.S.) and one of the remarkable contributors was Kitware Inc. [17].

The ITK does not solve visualization or graphical user interface, which is left to other toolkits, such as VTK. Similarly, this toolkit provides minimal functionality for file interface.

The toolkit provides important segmentation and registration algorithms in two, three, and more dimensions. There is also support for multi-threaded parallel processing.

The ITK is based on data-flow architecture. That means, that there are data objects, which are processed by process objects (filters) and they both are connected together into the pipeline.

This framework is natively written in C++. To get ITK to work with a managed .NET application there are three options. First approach is PInvoke[18], i.e. build a project as an unmanaged library, that uses the ITK toolkit, and in .NET application use PInvoke for entry points to library. Second approach is to write managed class wrappers around ITK classes which take care of converting data between managed and unmanaged code.

[17] http://www.nlm.nih.gov/research/visible/
[18] Platform Invocation Services

The third approach is to write ITK part as a COM[19] component and expose interfaces to be used in a .NET application.

The ITK covers two of our goals, data abstraction (g2) and data processing (g3).

2.2 VTK

The Visualization Toolkit (VTK, [18]) is an open source software system for 3D computer graphics, image processing, and visualization used by thousands of researchers and developers around the world.

VTK was initially created in 1993 as companion software to the book "*The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*" published by Prentice-Hall. This book was written by three people (W. Schroeder, K. Martin, and B. Lorensen) who later founded Kitware Inc. [17]. Kitware now provides professional support and products for VTK.

The conceptual overview of the VTK pipeline (similar to ITK pipeline) is shown in Diagram 3. Data are read in the source module and then filtered by one or more filters. A mapper is then used to create a visual representation that can be interacted with and transformed by the actor.

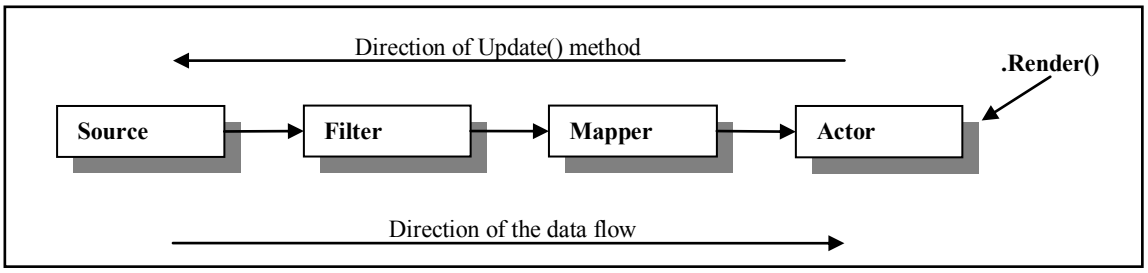


Diagram 3: The schematic of VTK pipeline

Also other system and development processes are the same as in ITK. Namely:

- Source code is heavily templated.
- The toolkit is cross-platform.
- There are wrappers to interpreted languages (Tcl, Python, and Java)
- For building of a project CMake (Cross-Platform Make, also developed by Kitware) is used.
- The toolkit uses its own system of reference counting and garbage collecting (called *smart pointers*).

There is now available fourth edition of the mentioned book [19] for further reading.

VTK can be used through wrappers [29] in .NET Framework, or possibilities mentioned in last paragraph from the ITK chapter can be used.

[19] Component Object Model

This framework covers these of our goals: data abstraction (g2), data processing (g3), data visualization (g4), and partially interactivity (g5) and visualization on GPU (g8).

2.3 VTK and ITK Extensions

2.3.1 MITK

The Medical Imaging Interaction Toolkit (MITK[20]) is a C++ library for the development of medical imaging applications. It is an extension to ITK and VTK, so all functionalities are available in MITK. MITK adds support for synchronized multi-viewer layouts and allows construction and modification of data objects. MITK can be added to existing applications and allows the construction of applications with specialized task without unnecessary other features.

Goals covered by MITK are data abstraction (g2), data processing (g3), data visualization (g4), and partially interactivity (g5) and visualization on GPU (g8).

2.3.2 KWWidgets

KWWidgets[21] is graphical user interface (GUI) toolkit, which provides common low-level widgets like buttons, textboxes, menus, and so on. On top of these it provides set of widgets for visualization libraries like VTK. For example there is surface material editor, transfer function editor, etc.

This framework covers these of our goals: data abstraction (g2), data processing (g3), data visualization (g4), and partially interactivity (g5) and visualization on GPU (g8).

2.4 Volumizer

OpenGL Volumizer[22] [20] is the commercially available, cross platform, high-level volume rendering application programming interface (API) for the energy, manufacturing, medical, and sciences markets. It is a graphics API designed for interactive visualization of large volumetric data sets.

The Volumizer covers four goals, data abstraction (g2), data visualization (g4), interactivity (g5), and visualization on GPU (g8).

[20] http://www.mitk.org/
[21] http://www.kwwidgets.org
[22] http://www.sgi.com/products/software/volumizer

2.5 VGL

VGL[23] is a commercial graphics library aimed at volume rendering. It supports multiple volumes rendering through the software ray-tracing. The hardware rendering techniques used in VGL are 2D and 3D texture based volume rendering.

Goals covered by VGL are data abstraction (g2), data visualization (g4), and partially interactivity (g5) and visualization on GPU (g8).

2.6 Medical Imaging Programs

VolView[24] is a graphical interface for volume rendering and data visualization. VolView was developed by Kitware and designed to enable easy exploration of volumetric data. No programming skills are required to use VolView, but there is a possibility to extend the framework through plug-in interface. Currently some of ITK and VTK filters in VolView are supported.

ParaView[25] is application built on top of VTK and ITK libraries. ParaView adds features, such as visualization using parallel processing and large data handling.

3D Slicer[26] is third application from Kitware and it is intended for interactive visualization of images, manual editing, and automatic segmentation. It was developed with KWWidgets, TCL, VTK, and ITK.

MeVisLab[27] is graphical interface that uses visual dataflow programming to create custom applications and visualization tools. MeVisLab support 2D/3D visualization with Open Inventor, OpenGL fragment shader, or VTK.

SCIRun[28] is program for wide variety of applications including image processing and 3D volume rendering. Its advantages are ITK and MATLAB integrations.

2.7 Summary

Name	Develop. language	License	Purpose	Origin	Develop. by
ITK	C++	open source	Registration and segmentation	1999	Kitware
VTK	C++	open source	Visualization	1993	Kitware
Volumizer	C++	commercial	Visualization of large data sets	2002	SGI
VGL	C++	commercial	Visualization of large data sets	1997	Volume Graphics

[23] http://www.volumegraphics.com
[24] http://www.volview.org/
[25] http://www.paraview.org/
[26] http://www.slicer.org/
[27] http://www.mevislab.de/
[28] http://www.software.sci.utah.edu/scirun.html

Name	Develop. language	License	Purpose	Origin	Develop. by
MITK	C++	open source	ITK and VTK extension	2004	Kitware

Table 3: Overview of existing frameworks

There are many[29] frameworks oriented on 3D data set processing and visualization. Most important of them are listed in Table 3.

The most usable framework according to our goals is one of those that provide interoperability with ITK and VTK classes. Their disadvantage is in necessity of learning three frameworks altogether.

Major part of frameworks was developed for many years and contains many classes and processes. This fact obviously extend the time of learning and developing of application. On the other hand it is better to use any of presented frameworks than develop a project from the scratch.

For further comparison see e.g. [21] and [22].

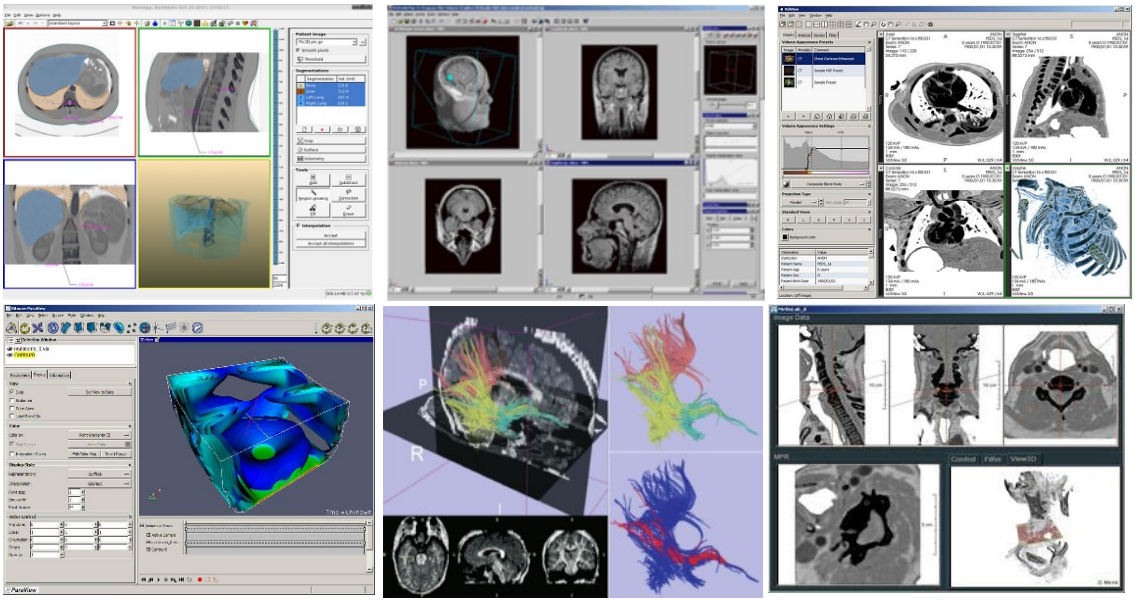


Figure 6: Screenshots of some visualization programs. MITK (top left), VGStudio (top middle) VolView (top right), ParaView (bottom left), 3D Slicer (bottom middle), MeVisLab (bottom right)

[29] http://www.virtual-anthropology.com/support/software/

Chapter 3

Design of Application Interface

In this chapter we will describe design of application interface (API). Within that the problems and required features from chapter 1.6 Goals of th are included.

Source code of interface is a part of project VL (stands for Volume Library) on the CD as a separate namespace `VL.Api`. Only `interfaces`, `abstract classes`, and `enums` are defined in this namespace. General design features of `VL.Api` (except those mentioned in chapter 1.6) are following:

- Independence on Windows Forms. All features making use of WinForms is left on implementation. Note that our implementation is using WinForms, but generally, the choice of user interface (UI) is left on other possible implementations of VL.Api.
- Like state machine. The behavior of VL.Api for programmer is based on the state machine that controls building or modification of required pipeline and its execution.

Note that VL Reference Manual is available on the attached CD and feel free to look there for exact features and behavior of each class and interface.

3.1 Pipeline

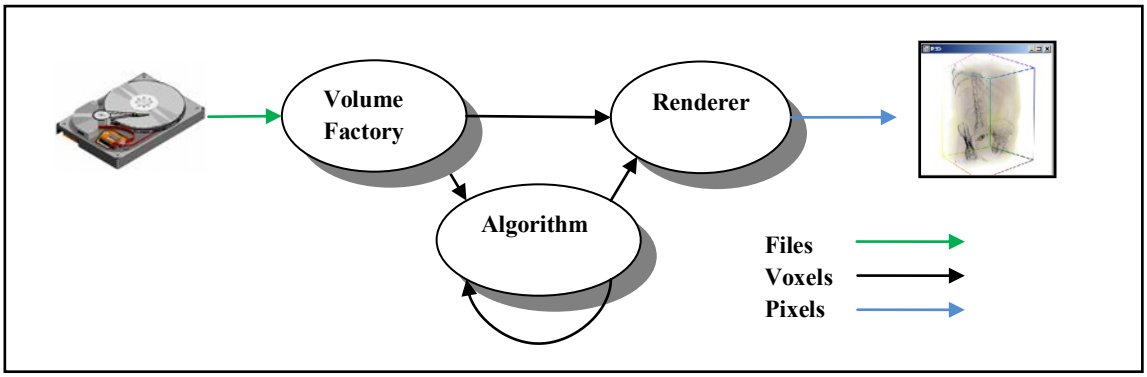


Diagram 4: Data flow diagram

Diagram 4 shows general data flow diagram of the VL application. The diagram shows reading of the 3D data sets from the hard disk drive through the use of VolumeFactory. Next data can be processed by Algorithm or directly send to Renderer. Renderer then visualizes the data into the application window.

The module connection in VL application is performed through the use of data structure IVolume. This adjustment is shown on Diagram 5 with a simplified example of code.

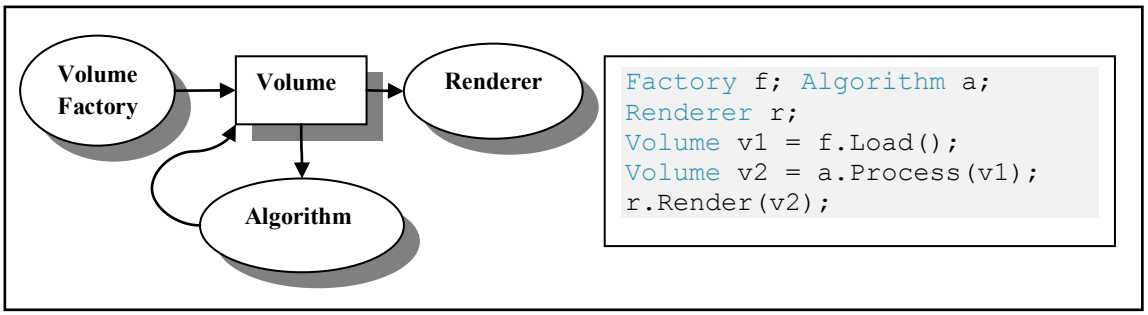


Diagram 5: Creation of processing pipeline through the use of `IVolume` data structure

Next we will analyze this situation in detail.

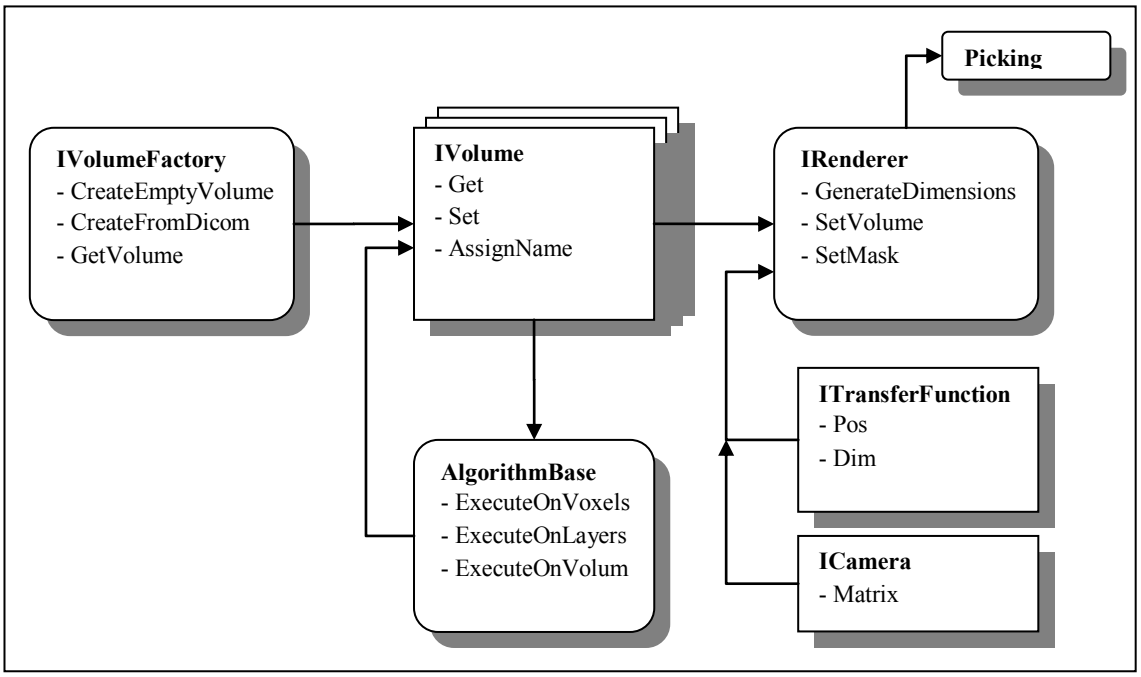


Diagram 6: Block diagram of general VL pipeline

Diagram 6 shows a schematic structure of data processing pipeline in VL. A typical data cycle in medical program, which makes use of VL, starts by loading volume data stored on a hard disk drive in the DICOM format. That functionality provides `interface IVolumeFactory`, which creates instances of `classes` derived from `interface IVolume` (chapter 3.2). For data modifications `abstract class AlgorithmBase` is available (chapter 3.3). After the defined algorithms have

processed the data, they are sent to visualization part. Interfaces for common visualization techniques are defined in "renderers" (`IRenderer`, `IRenderer2D`, `IRenderer3D`, `IRendererIso`, `IRendererDVR`, `ITransferFunction` and `ICamera`) – chapter 3.4. There is also an interface for selecting points in displayed data (`IPicking`).

Note that the described pipeline is a general structure and the exact form of pipeline is defined in each program. For example, in some cases no "algorithms" are needed and only visualization functionality is used.

As mentioned earlier, one of our goals (g6) is to minimize time elapsed from execution of pipeline. This is one of the reasons for existence of `interface IThreaded`, which contains prototypes of methods for controlling parallel run of calculations. This interface is used in `IVolume` (because of possibility of loading volume data in another thread) and in `AlgorithmBase` (where executed algorithm can also run in background thread). `IThreaded` is not used in `IRenderer`, because rendering must be usually done in main thread of the application.

3.2 Data Representation

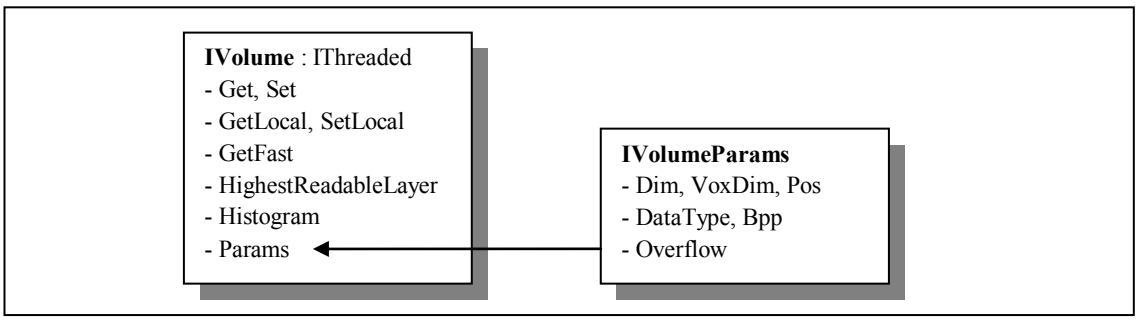


Diagram 7: Most important methods and properties of interfaces IVolume and IVolumeParams

In this chapter we will describe two interfaces: `IVolume` and `IVolumeParams`. They are designed to store the data in 3D matrix and to provide efficient access to that data.

There were several issues, which had to be resolved. The first was checking of access outside the volume, alias *boundary problem*, which is relevant when there are many checks while passing through the whole volume by the algorithm. This problem will be described in the next chapter 3.3.

Second problem is the selection of supported data types for representation of each voxel. We decided to support the following types:

- `byte`
- `ushort`
- `uint`
- `float`

The reason for this is that these data types need not be converted before transmitting them to GPU. That is not true for other data types, which we considered originally – `bool`, `ulong` and `double`.

The third and quite complex problem is data abstraction (g2). What we want is the possibility to store different types of data in a volume, but the processing and visualization tasks should be independent on the incoming data types. Problems with visualization were described in previous paragraph. In processing task we solve this by putting a generics parameter to the deepest point inside `interface IVolume`, i.e. to read and write methods:

```
TYPE Get<TYPE>(int x, int y, int z);
void Set<TYPE>(int x, int y, int z, TYPE value);
```

Code snippet 1: Methods for accessing voxels

Disadvantage of this approach is that it requires to cast field with stored data in these methods, but the code of processing methods defined by user is simpler, then if the generics parameter was on the whole interface `IVolume`, where the casting has to be performed by programmer (if he wants to use some numeric operations).

The last problem, which we want to mention, is dividing volume data into pieces in order to allow computation of multiple parallel algorithms (or loaders and renderers). The idea is that when some piece is e.g. loaded, it can be send to e.g. visualization. This cycle is than repeated until the whole volume is visualized. The major part of the image processing algorithms operate sequentially, so we decided for simple solution of this problem. We record number of layer, for which is true, that all other layers with smaller number are fully filled. This property is stored in `IVolume.HighestReadableLayer`. Figure 7 shows utilization of this property.

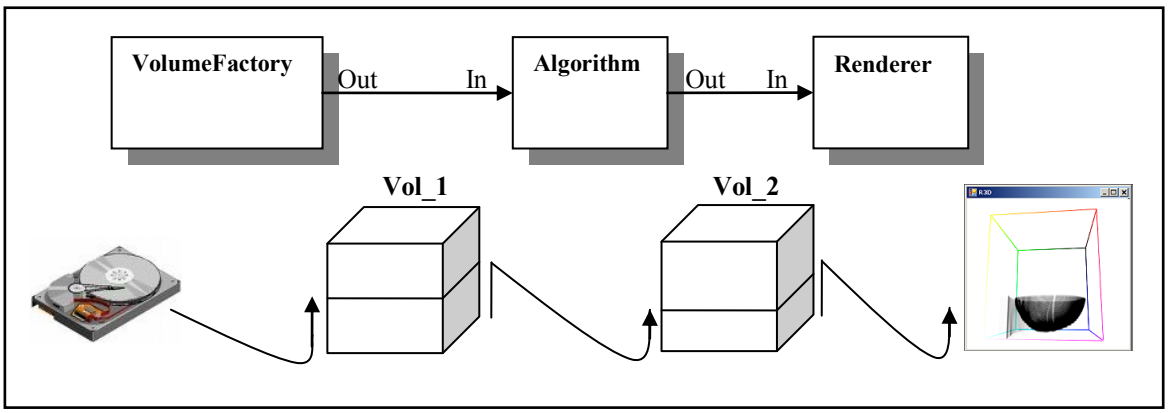


Figure 7: Parallel run of pipeline modules

3.3 Algorithms

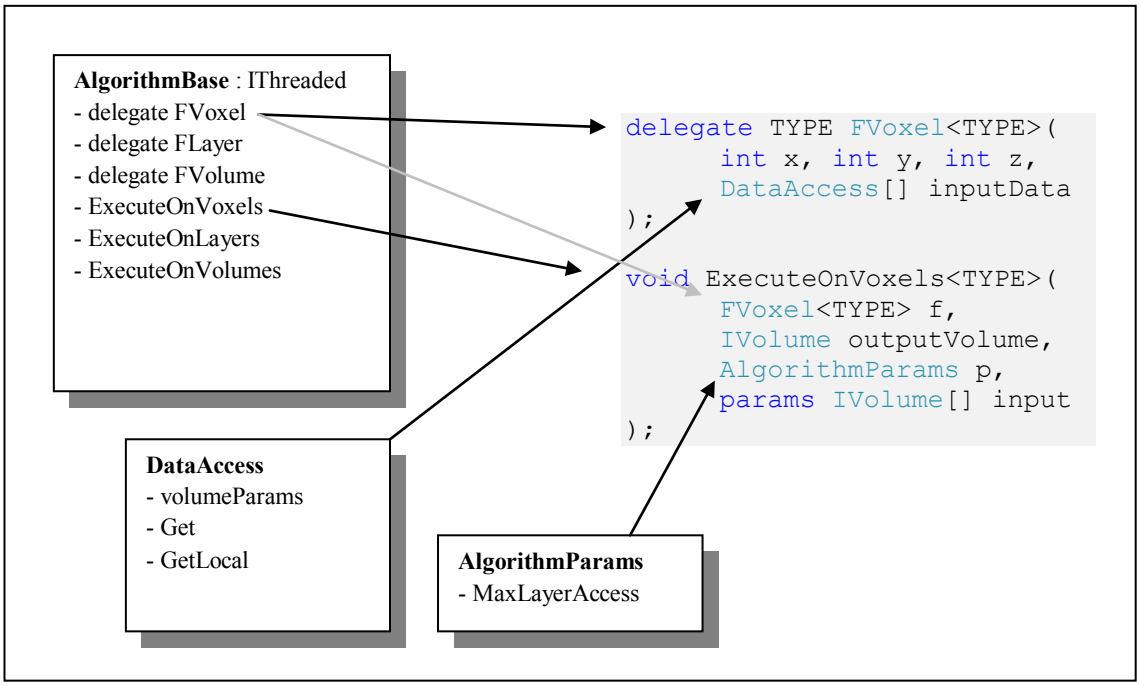


Diagram 8: Set of classes connected with algorithms

This chapter clarifies the functions of algorithms. Classes connected with this topic are `AlgorithmBase`, `AlgorithmParams` and `DataAccess`.

Concept of algorithm is not to define all possible image processing algorithms, but to provide a good interface for writing algorithms (g3). Our solution of that is based on an assumption that most of the algorithms run sequentially through 3D data "voxel by voxel", make some calculations on matching voxels of input volumes and write result to output volume. So, everything, what a user must do in VL.Api, is to write method for one voxel and then only call algorithm for the execution of that method. Running through the voxels is left to the framework, which makes optimizations possible. Delegate definition and corresponding method is shown in the Diagram 8.

Previous paragraph is concerned with a case, when the user algorithm can be divided to independent executions for each output voxel. For other cases there is a method for stepping through the layers, when algorithm must operate on the whole layer. For this case the layer orientation can be specified, but it must be noted that when other than x-y layer is required, the processing pipeline evaluation is paused. Also this case allows parallelization. The last case of run is through the entire volume, but there no optimizations can be done, so it is good to consider carefully, which method should be used.

3.3.1 Algorithm Optimizations

There are two main approaches how to accelerate the algorithm execution.

The first approach is based on brute force, i.e. on utilization of more processors for parallel run of multiple threads. From the previous chapter 3.3 it is obvious, that there is no problem to schedule an execution of the algorithm on as many threads as there are voxels in the volume (for case of `ExecuteOnVoxels`), or as many threads as there are layers (for case of `ExecuteOnLayers`). It is clear, that `ExecuteOnVolume` cannot be automatically parallelized.

Originally we have considered to implement the parallelization on GPU, but that would result in writing the segmentations algorithm in two versions (in two languages), as currently no automatic conversion (from C# to any of languages used on GPU) exists.

The second approach is to make use of two types of `Get` methods from `IVolume`.

```
TYPE Get<TYPE>(int x, int y, int z);
TYPE GetFast<TYPE>(int x, int y, int z);
```

Code snippet 2: Methods for accessing voxel from algorithms

The difference in those two functions is that the first checks a possibility of accessing outside the data bounds (and in a case that someone is attempting to read outside volume it returns value dependent on set up of field `overflow` in `IVolumeParams`), while the second function immediately accesses the data without any overflow checks.

This optimization is possible to implement due to the parameter `MaxLayerAccess` from the `AlgorithmParams` class. When `MaxLayerAccess` added to the voxel's coordinates is always inside the volume, the method `GetFast` is hold up on method `DataAccess.Get`, which can then be securely called by the user algorithm. Otherwise the method `Get` is hold up on method `DataAccess.Get`.

3.4 Visualization

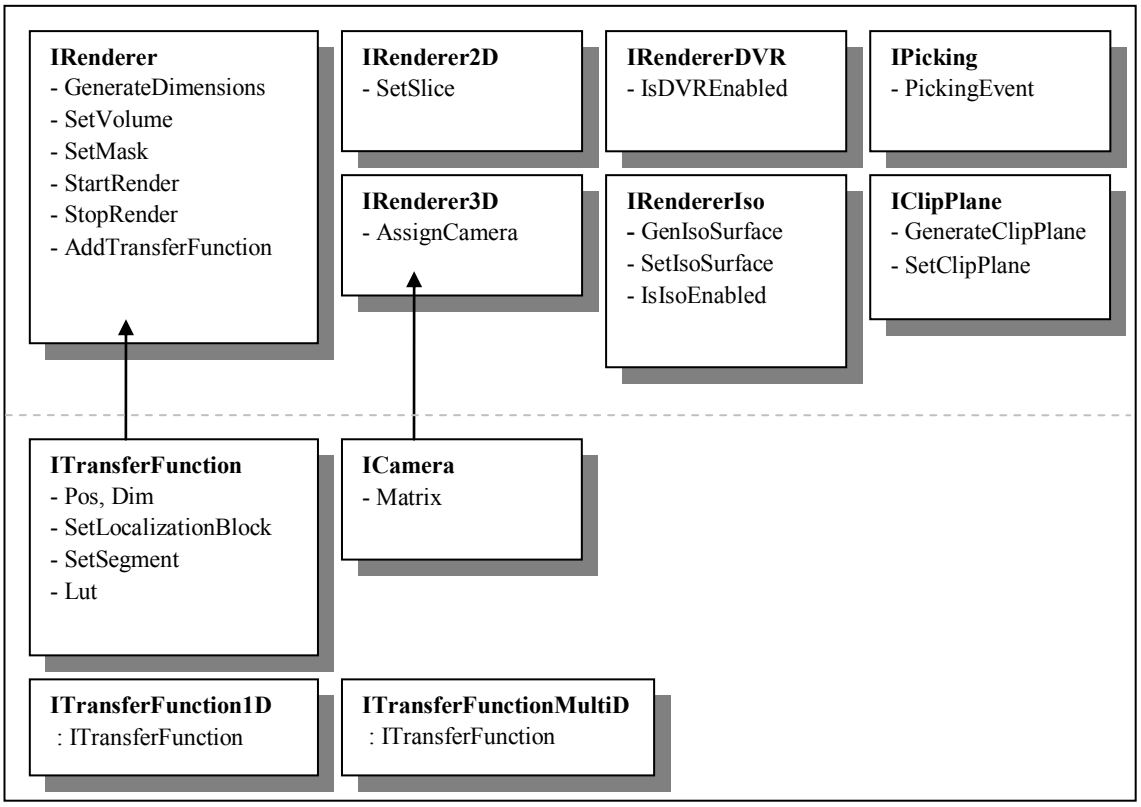


Diagram 9: Interfaces that are connected with visualization

There are several interfaces available for visualization in the API. They are designed to be able to offer implementation in one class, so that implementation programmer can decide which interfaces will be supported by which renderer. Interface `IRenderer` should be implemented in all renderers.

Interfaces shown on Diagram 9 ensure the goal (g4):

- Direct volume rendering – `IRendererDVR`.
- Isosurface rendering – `IRendererIso`.
- Color transfer functions – `IRenderer` and `ITransferFunction1D`.
- Masks – `IRenderer` and `ITransferFunctionsMultiD`
- Multidimensional color transfer function - `ITransferFunctionsMultiD`

The goal (g5) Interactivity is also related to visualization, but we left this goal on implementation.

In our concept of masks and transfer functions there is a rule, that all volumes (main volumes and mask volumes) store one component data and translation to final color (which is obviously multi component value) is left to transfer function. If we want to visualize two component data, it is necessary to store the first component in the main volume and the second component in the mask. Then we must define two-dimensional transfer function, where the first dimension corresponds to values from the main volume

and the second dimension corresponds to values from the mask volume. Each dimension will increase one mask volume and one dimension in the transfer function.

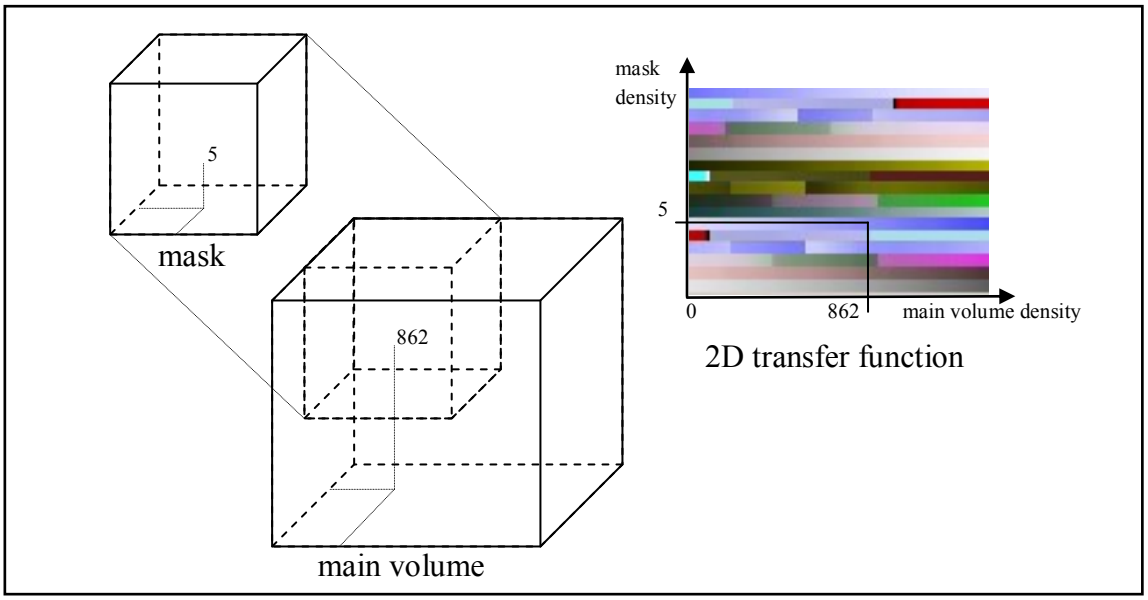


Figure 8: The 2D transfer function

Figure 8 shows the whole situation. There are two volumes, main and a mask. Densities from both volumes are needed for calculation of resulting color of actual voxel. These densities (862, 5) are taken as an index to the two-dimensional color transfer function look-up table.

Figure 8 also shows the reason, why the components are divided between two volumes (instead of using one volume with two components stored in one voxel). Common case of image processing in medicine is that there are detailed data from CT/MRI scanner and the task is to show some segmented organ. The programmer develops a segmentation algorithm, which outputs only a mask with smaller data type, where information about organ is stored. Thus the mask is smaller and of the smaller data type, thus memory usage is optimized.

Previous paragraph implies the necessity of moving the mask volume against the main volume. All volumes have their position, so it is possible to move them along the entire space. The position must be taken into account when reading input components for multidimensional transfer function from volumes. In a case when more components are used and there is no value at the particular position (because the mask volume is moved against the main volume or because the mask volume has smaller dimensions than the main volume), behavior is defined with field `IVolume.Params.Overflow`. There are three possibilities. The first of them is "clamp to border". That means that defined value (density) is used. The second is "clamp to edge", which means that nearest voxel value is used. And the third is "repeat", which means tiling.

Thus there can be more masking volumes and they can be localized. The same is true for transfer functions. In the application specification a possibility is included to localize transfer function by definition of position and dimensions of a block, where

transfer function is used. When more transfer functions are defined for one voxel, the last (meaning the last added) is used. On the other hand, when no function is defined for some voxel, it must be transparent.

3.5 Goals Overview

Here we want to summarize the goals from chapter 1.6 in relation to the presented API.

(g1) Compactness is resolved by using one interface `IVolume` in all parts of the API, and trivially by the fact, that all three main tasks (loading, processing, and visualization) are contained within the API.

(g2) Data abstraction: there is support for four data types (`byte`, `ushort`, `int`, and `float`) in the API and creating of processing pipeline is independent on the volume type.

(g3) Data processing: ease of programming of segmentations and filtrations is designed through the use of delegates on methods. "User programmer" will implement single method, which will be automatically called. There are three types of runs – through each voxel, along layers in defined direction, and for whole volume at once. "User programmer" must decide which type he will use depending on granularity of his problem. The first two possibilities allow the parallelization without any programmer effort ("for free").

(g4) Data visualization: interfaces for 2D rendering and common 3D visualization techniques are contained in the API. Also the color transfer functions and multidimensional color transfer functions are supported.

(g5) Interactivity depends mainly on quality of the renderer implementation, so this goal was not covered by API.

(g6) Temporary results can be handled by `IVolume` property `HighestReadableLayer` and consequent setting of it by loaders and algorithms.

(g7) Written in C# goal is filled trivially.

(g8) Acceleration on GPU is also the goal of the implementation, but the API facilitates it with allowed data types.

Chapter 4

Implementation

This chapter describes certain issues related to our experimental implementation of the application interface `VL.Api`.

Apart from this chapter, there are several other sources of information about VL Implementation:

- VL Programming Guide at the end of this document.
- VL Reference Manual (in Microsoft Help CHM format - .chm) on the attached CD.
- Source code which was developed for introduction and testing of VL features – project "Example" in solution "VL" or in subdirectory "Example_proj" on the CD.
- And there is also source code available for our VL implementation on the CD.

Source code of VL framework is divided into three namespaces:

- `VL`: Main namespace with framework implementation.
- `VL.Utils`: This namespace contains auxiliary classes, mainly for math computations.
- `VL.Api`: In this namespace only `interfaces`, `abstract classes`, and `enums` are defined. For more information see chapter 3 Design of Application Interface.

Overview of main parts of source code is presented in the first section of this chapter. In following sections are implementation details and decisions made while developing VL.

4.1 Classes Overview

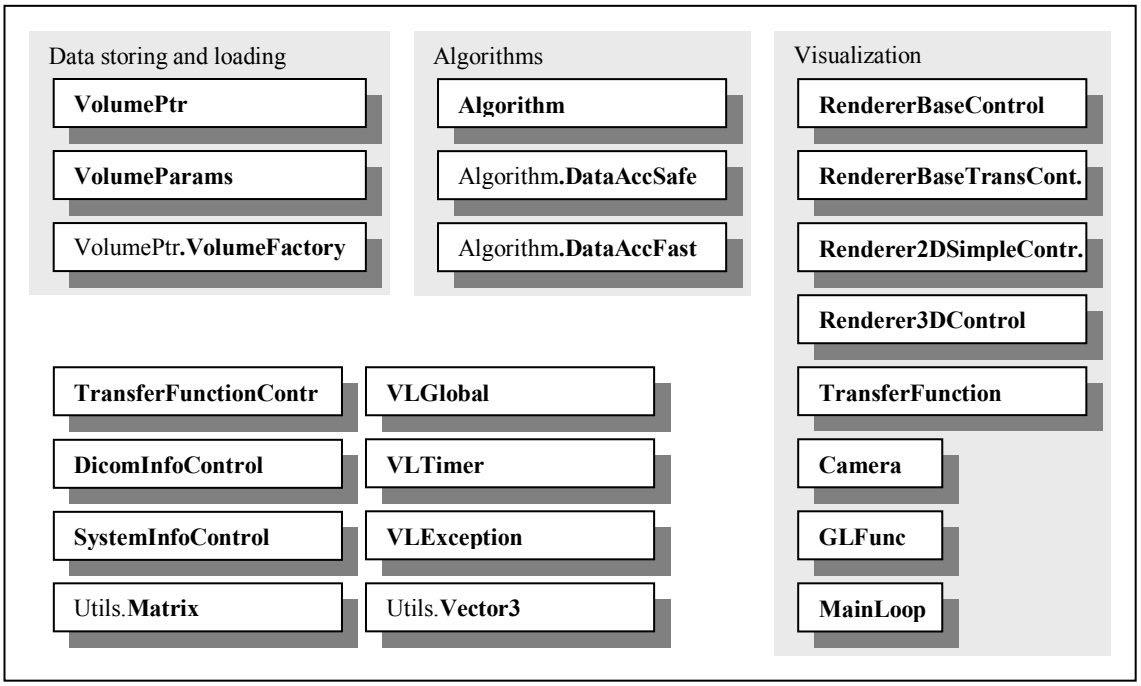


Diagram 10: Most important classes in our VL implementation

Implementation is placed in two namespaces: `VL` and `VL.Utils`. In `VL.Utils` there are auxiliary classes intended especially for mathematical operations with vectors and matrixes.

There are three main groups of classes as shown in Diagram 10. The first group serves for loading data from hard disk drive and their management in the memory. The second group represents classes that solve image processing tasks of VL. And the third group includes classes dealing with visualization.

Other classes shown in the Diagram 10 are auxiliary classes either for comfort of the "end programmer" (classes with word "`Control`" in their name) or classes that are used in the whole project and could not be included in any of the three listed groups (from those the most important is the class `VLGlobal`, which contains global settings of VL library).

4.1.1 Data Storing and Loading

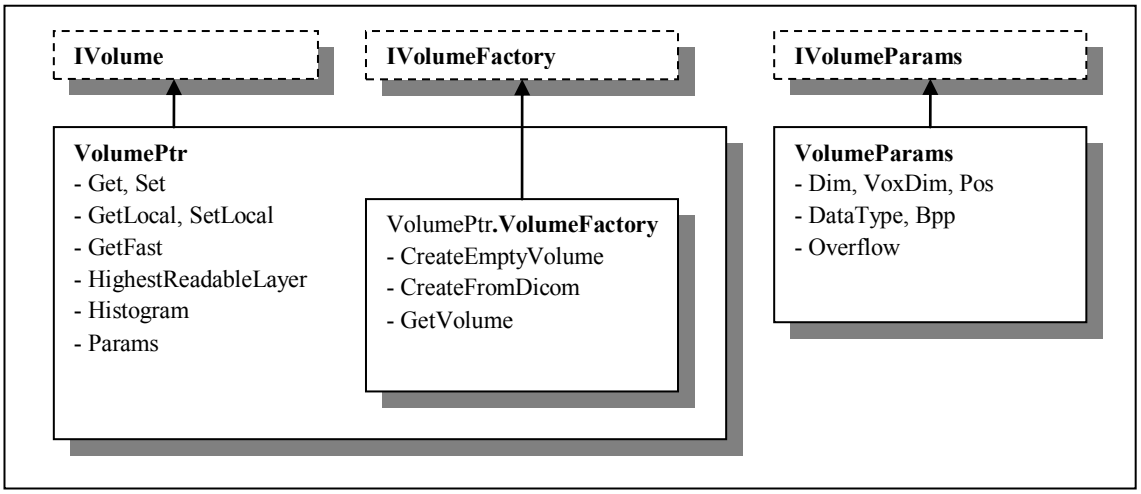


Diagram 11: Inheritance hierarchy and most important methods and properties of classes connected with data storing and loading. Dashed lines indicate interfaces from API.

Connection of classes and interfaces from `VL.Api` is shown in Diagram 11.

There is one feature in the class `VolumePtr`, which should be highlighted. It is obvious that the class `VolumePtr` stores data in the main computer memory, but for visualization it is necessary to store data also on the graphics card. `VolumePtr` is responsible for transfering the data to the VideoRAM for storage and for appropriate handling.

Class `VolumePtr.VolumeFactory` is designed for creating empty volumes and loading volumes from hard disk. In order to simplify work of the "end programmer" as much as possible, there is UserControl `DicomInfoControl`, which is responsible for displaying information stored in DICOM files.

4.1.2 Algorithms

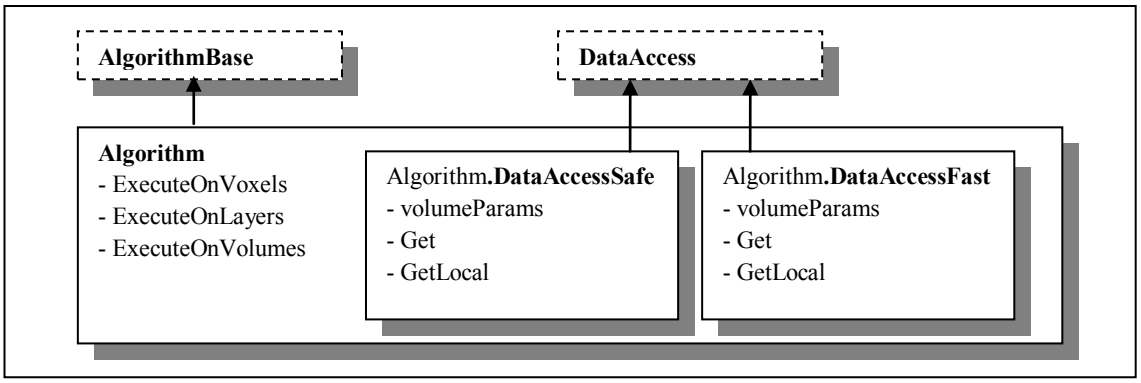


Diagram 12: Inheritance hierarchy of classes connected with Algorithms. Dashed lines indicate interfaces from API.

The algorithm class task is concerned with execution of user defined functions on the input volumes. This class must schedule "work" for the right number of threads,

depending on number of processors in the system and on the number of threads already used by the `VL`. A secondary task of the class is to apply a right method for accessing data of input volumes related to user defined functions. It is done by classes `DataAccessSafe` and `DataAccessFast` derived from `abstract class DataAccess` from `VL.Api`. It is shown in the Diagram 12.

4.1.3 Visualization

It should be stressed (as already noted in chapter 3 Design of Application Interface), that our implementation is using WinForms [23] as a graphical user interface (GUI). That decision affects only classes connected with rendering, because they are designed as the `UserControl`. The main advantage of the `UserControl` is that they can be inserted into destination form and the surrounding environment is not affected, so the developer can personalize appearance of his forms.

Most complicated situation of inheritance lies in visualization group of classes.

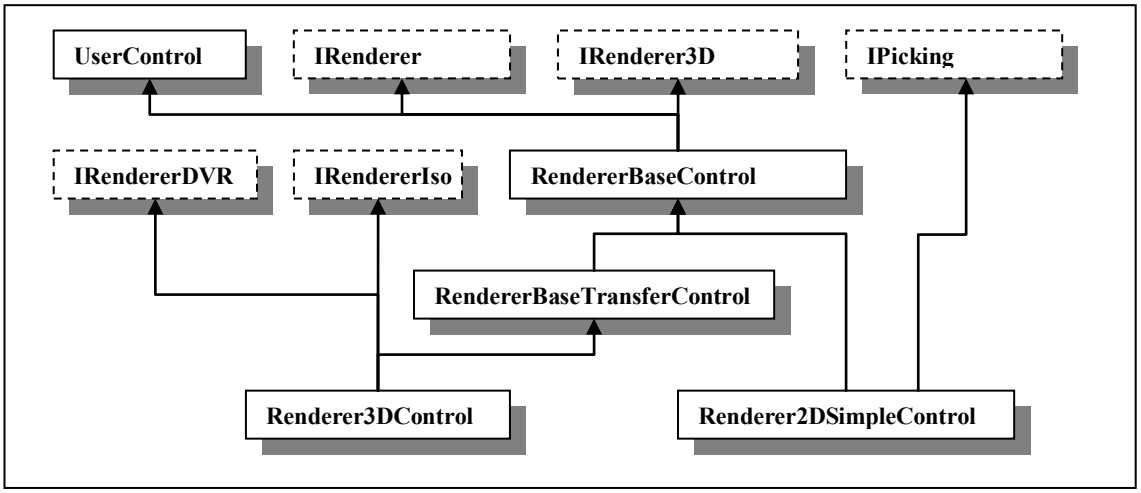


Diagram 13: Inheritance diagram of renderers. Dashed lines indicate interfaces from API.

"Renderers" part of visualization group is shown in the Diagram 13. The results for the "end programmer" are two classes: `Renderer3DControl` and `Renderer2DSimpleControl`. These two classes cover major part of visualization API.

There are two interfaces, which were not implemented.

The first was `IClipPlane`. We have considered it not so important and we preferred to spend time on other parts of the framework.

The second was `IRenderer2D`. The role of this interface was replaced by `Renderer2DSimpleControl`, which is designed for rendering three 2D axis aligned slices. Because the task of rendering common slices is not fulfilled, we decided not to derive this class from `IRenderer2D`. It should be noted, that the class

`R2DSimple` is optimized for the memory requirements and for the speed of rendering. If common 2D slices were implemented, one of those two advantages would be lost.

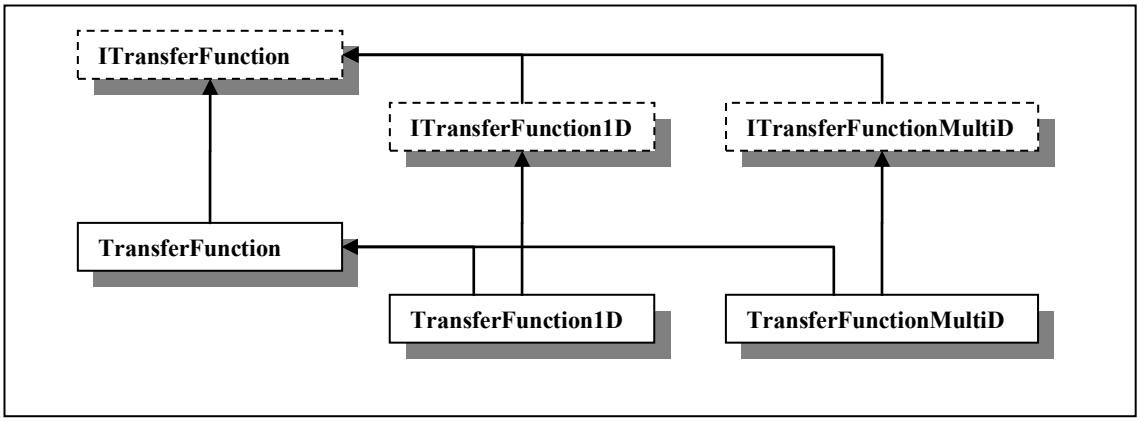


Diagram 14: Inheritance diagram of classes connected with transfer functions. Dashed lines indicate interfaces from API.

Diagram 14 presents inheritance situation of transfer functions. Before the rendering pipeline is executed, transfer functions are assigned to renderers in order to map density values to color.

For user input of the transfer function values there is the `UserControl` `TransferFunctionControl` available, which provides one instance of the class `TransferFunction1D`, which can be used directly in any renderer or can be used to compute another `TransferFunctionMultiD`.

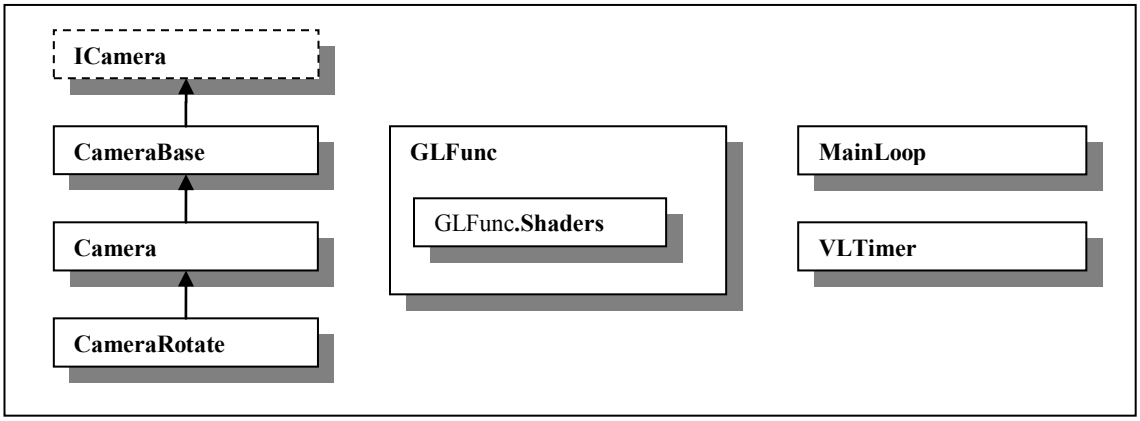


Diagram 15: Inheritance diagram of other classes connected with visualization. Dashed lines indicate interfaces from API.

The Diagram 15 shows the rest of the classes, which are connected with the rendering.

Cameras are classes that define behavior of Renderer3D depending on user input from input devices like mouse and keyboard. `CameraBase` is a class, which is best suitable to derive from if someone wants to have different behavior of Renderer3D than predefined cameras `Camera` and `CameraRotate`. The class `Camera` is used for

simple rotation around x and y axes and the class `CameraRotate` appends continuous turning of the scene around y-axis (this feature is used for measuring the rendering speed).

`GLFunc` and `GLFunc.Shaders` are "singleton" classes with functionality related to OpenGL. It means: initialization, releasing of resources, handling of OpenGL errors, loading of textures and managing of ISO and DVR shaders. An important feature is the rendering context memory sharing, which is necessary when more than one rendering control visualize the same volume, so that the graphics memory is allocated only once.

The task of the class MainLoop is to call renderers for redrawing, when an event arises. That is when new data are loaded or processed and they can be visualized or when camera has changed the position and the scene must be redrawn. VLTimer is "singleton" class for computing a one rendering loop time and counting of frames per second.

4.1.4 Other Auxiliary Classes

Two classes should be mentioned here. The first is `VLGlobal`, where overall library settings are maintained, like setting of output and error window, some threading settings and rendering settings.

To make a list of classes compact, there is the class `SystemInfoControl`, which is designed to display information about the System (processor count, CPU usage, available and used memory, GPU type and OpenGL extensions).

4.2 Rendering

This chapter introduces detailed process of 3D visualization in VL. The process is implemented in the class `Renderer3DControl` and shader `cg/raycasting_shader.cg`.

We use GPU based ray-casting similar to the method presented by Kruger and Westermann in [11]. The main core of the algorithm is to send one ray per screen pixel and trace this ray through the volume. Both techniques, DVR and isosurfaces, are implemented in one fragment program.

First of all, there is initialization of cg [24] and two framebuffer objects (FBO, [25]). In order to generate the necessary rays a trick is used with rendering color cubes, where colors represent coordinates – black means [0,0,0], white means [1,1,1] and so on. First FBO is filled with back faces of cube (Figure 9 left). Before rendering front faces (Figure 9 right) of color cube to second FBO our shader is binded. Finally, the second FBO with ray-casting output is blended to the screen.

In the shader itself, the ray origin (input color to shader, front faces) and ray direction (corresponding color in back face FBO subtracted by ray origin) is than easily

computed. Final color is computed in a single loop, where the shader steps along the ray and recomputes actual color with the volume value translated by the transfer function.

Camera rotation is applied once before rendering color cubes, so that projection matrix and model-view matrix need not be transferred to the shader.

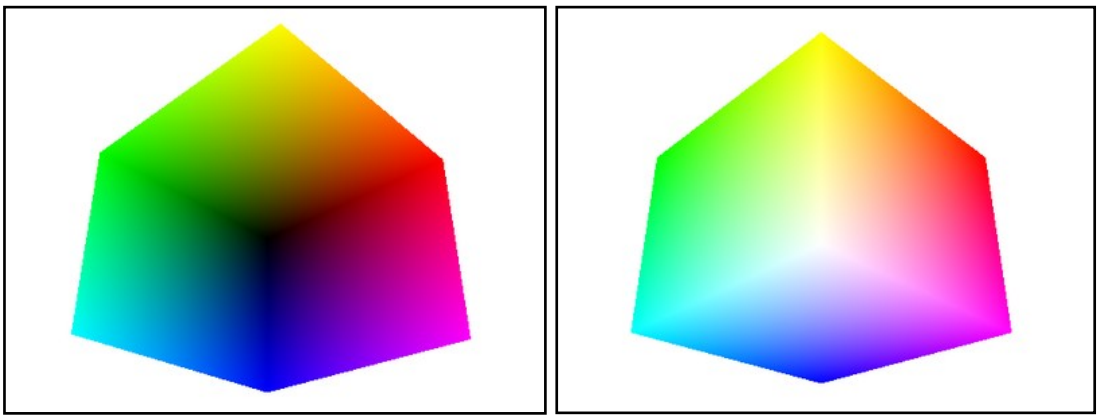


Figure 9: Back faces (left) and front faces (right) of color cube represents coordinates.

The described technique does not support moving camera inside the volume (cube), so we implemented the box-plane intersection from [26] and in place of rendering front faces of cube we render front faces of cube intersected with a plane, which is parallel with the projection plane. The resulting cube is displayed in Figure 10 (left).

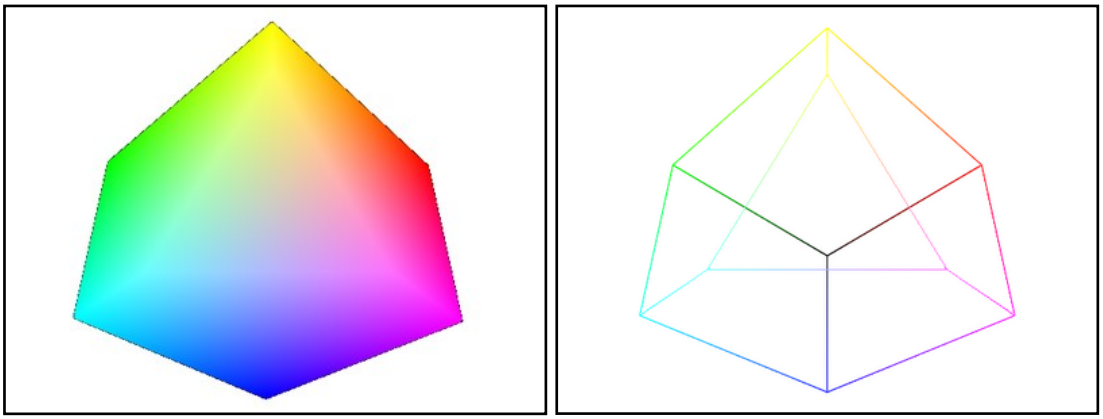


Figure 10: The left image shows color cube combined with intersected polygon. The right image presents a schematic outline of whole color cube.

The way, how the color on each sample is computed when stepping along the ray is expressed by the following commands:

$$C_{dst} = C_{dst} + (1 - a_{dst}) \cdot a_{src} \cdot C_{src}$$
$$a_{dst} = a_{dst} + a_{src} \quad (2)$$

Where

- C_{dst} is destination color
- C_{src} is sample color
- a_{dst} is destination opacity
- a_{src} is sample opacity

4.2.1 GPU Data Transfer

Both data structures volumes and the transfer function look-up tables are stored in GPU memory in form of 3D textures. Because OpenGL behavior is based on states, the transfer of these data must be done on the right place. That is in main thread before the rendering starts.

For each volume only one texture is created, independently on number of renderers, which use this volume. The memory allocation is done in the moment, when the renderer starts, so that if some volume is not visualized (it serves as auxiliary storage for further computations), the data are not transferred to GPU. Once memory for volume is allocated, the slices are transferred as soon as they are available (according to volume property `highestReadableLayer`). This process guarantees the intermediate results.

The 3D textures are used for all transfer functions (`TF1D` and `TFMultiD`) because of simplification of the fragment shader program. The size of texture depends on the size of the transfer function look-up table. Here also the source of limitation of VL on maximum number of volumes that can be visualized together comes.

This behavior is handled by the class `RendererBaseTransferControl`.

Since transfer of data is needed only when transfer function is changed, it allows the real-time editing of the transfer function.

4.2.2 Shaders

An interesting method is used for handling multiple versions of shaders. There must be twenty-one versions of ray casting fragment program altogether. One for DVR rendering, one for isosurface rendering and one for these two together. And each group must be three times (for one, two, or three volumes rendered together). Furthermore there are three possibilities for isosurfaces (ambient, diffuse and specular components of color can be enabled). This generating of code is enabled through the use of runtime compilation by the OpenGL driver.

Because much of the code is same for all of them, we solved this problem with commentaries in the code. Every commentary has its unique identifier and when shaders are being initialized, the right commentary is deleted and thus the line of code over them is enabled.

The Shader Model 3.0 is necessary for run of the fragment program.

4.3 Threads and Synchronization

Threads and parallel processing is used on two places in VL.

The first is in Algorithm class, where the number of threads is calculated and then threads are created and started to perform the calling of user defined functions.

This is not very complicated, but for application programmer it is a great task, because he does not need to care for threads and synchronization at all.

The second task, which threads provide, is parallel run of loading and processing modules. This is done through the use of the volume property `HighestReadableLayer` defined in the `VolumePtr` class. Generally every processing pipeline starts with reading data from hard disk drive, then something is computed and finally visualized. When these tasks run simultaneously, it is very good for utilization of system resources, because disk operations are interlaced with processing operations and rendering operations are performed on GPU.

Intermediate results shown in the application window are the second effect of the parallel processing pipeline.

4.4 External Components

VL uses several external libraries.

Library openDICOM.NET[30] is used for loading of DICOM files.

OpenGL functions are provided through the use of Tao Framework[31].

Cg Toolkit[32] is also accessible through the use of Tao wrapper.

Also our implementation of renderer is based on nice tutorial from Peter Thomsen's Blog[33].

[30] http://www.opendicom.net/
[31] http://www.taoframework.com/
[32] http://developer.nvidia.com/object/cg_toolkit.html
[33] http://www.daimi.au.dk/~trier/?page_id=98

Chapter 5

Results

The Visualization Toolkit (VTK, [18], [19]) is one of the most popular and commonly used framework for image processing and visualization of data. We made extended comparisons to document differences between volumetric features of VTK and our implementation of the VL application interface. The results from that effort are described in this chapter.

Most important criteria for each part of data processing and visualization process were chosen, like loading of data, algorithms (filtering, registration, segmentation and classification), visualization, supported data types, developer point of view and resulting speed. Each group of criteria is pointed out in one of the chapters.

5.1 Programming Language

VL is written in C# under the .NET Framework 2.0 [28], while VTK is written in C++. This implies basic features of both frameworks.

Advantages of C#:

- intended to be simple
- modern
- general-purpose
- object-oriented

Main advantages of C++:

- object-oriented
- templates
- preprocessor macros
- multi-platform

VTK has quite a long history and there were several wrappers made to other languages (than C++). One of them is VTK.NET [29]. First we though, that a comparison with the VTK will be done through the VTK.NET, but subsequently we

discovered, that it was not possible to write the algorithms in the wrapper (the reason was that the inheritance and polymorphism could not be used between the new and wrapped classes). This is one of the grounds for deciding on the selection of VL or VTK.

5.2 Data Types

Subject	VTK	VL
Data Types	`{signed|unsigned} char, short, int, long, long long, float, double`	`byte, ushort, uint, float`

Table 4: Supported data types. In VTK C++ names of types are displayed, while in VL C# names are used.

Regarding this criterion the VTK support more types. VL chooses only unsigned types and largest type is 32 bits large, while VTK support 128 bits (`long long`, depends on a compiler). There are two reasons why VL supports those types. One is that only those types can be transferred to GPU and the other is that in C# preprocessor macros cannot be defined, while largely used in VTK in a code connected with data types.

The data structure used in the common VTK volume processing pipeline is the same as in the VL library (in VTK notation “Image Data”).

5.3 Lines of Code

For the comparison, we wrote simple applications in both frameworks. Immediately after starting of each program the pipeline is created and executed. Pipeline consists of loading of data, making simple threshold segmentation on them, and displaying both volumes (input data and segmented volume) with direct volume rendering. The following Diagram 16 shows pipeline used in the VTK application.

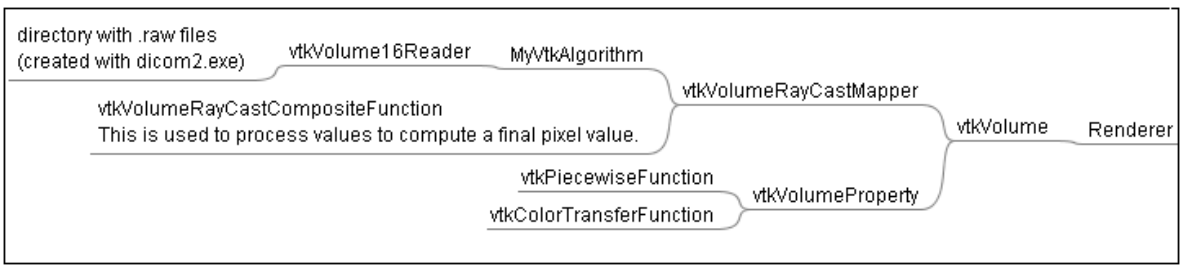


Diagram 16: Pipeline in the VTK testing application

The Diagram 17 shows pipeline used in the VL application. The main difference is in using the mask volume for output from the algorithm, instead of VTK approach

(two components are stored in each voxel). This concept guarantees smaller memory requirements. These two diagrams also show that the VTK universality implies necessity of using more classes and structures in the processing pipeline.

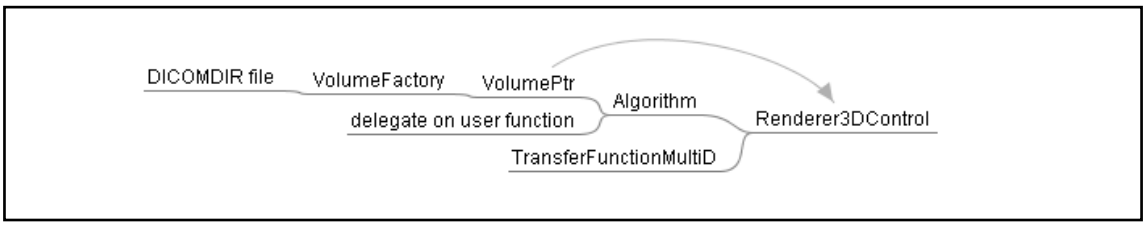


Diagram 17: Pipeline in the VL testing application

The main part of applications with definition of data flow pipeline has essentially the same number of lines. Big difference lies in definition of the desired segmentation. The number of lines is shown in Table 5.

Subject	**VTK**	**VL**
Lines of code – pipeline definition	85	79
Lines of code – single-threaded segmentation	92	12
Lines of code – multi-threaded segmentation	156	12

Table 5: Table with lines of code of simple applications

Results from Table 5 are affected by the fact, that if a user wants to define his own algorithm in VTK, he must create a new class (inherited from `vtkImageAlgorithm`). But in VL it is only necessary to define a single function with segmentation and the framework does the rest.

The same fact also implies that multi-threaded algorithm has the same lines of code in the VL. Multi-threading in VL is left on framework and the end developer has multi-threading "for free".

The difference between VTK and VL codes in segmentation in our example application is so prominent, because our segmentation is very simple. If it were more complex, the ratio between VTK and VL would be reduced (difference should remain the same).

5.4 Speed of Algorithms

Subject	**VTK**	**VL**
Single-threaded segmentation	5.5 s	14.8 s
Multi-threaded segmentation	2.9 s	8.1 s

Table 6: Comparison of our simple segmentation speed

From the Table 6 it can be seen, that concerning computation speed, the VTK is much better than VL. This is due to the fact that accessing voxel values is slower, but more comfortable for a programmer of algorithm in VL.

The VL has also advantage in parallelization. Both features of VL, rendering of temporary results and scheduling of algorithm work, are for end programmer provided without any programmer effort. Furthermore, the visualization does not slow down the algorithm computation - for exact times see the Table 8.

For the task of segmentation, for which we tested the speed, the voxels, which were inside sphere and had a value above some threshold, were selected. There were the following operations:

- Multiplication – 7 times
- Addition – 5 times
- Comparison – 2 times
- Reading input voxel value – 1 time
- Writing to output voxel – 1 time

Data dimensions of input and output volume were 512*512*256 and there were 2 bytes for a voxel.

Hardware and software parameters of the machine:

- CPU: Intel Core 2, 1.8 GHz, dual-core
- GPU: NVIDIA GeForce 8400GS, 256MB
- RAM: 1 GB
- Microsoft Windows XP SP2

5.5 Speed of Loading and Visualization

There are certain difficulties when comparing speed of loading data. First is that VTK itself does not have functionality for loading DICOM data. The way for loading them is either to use some other library, or to transform data to another format before starting the program. We used the second approach and adjusted our data sets with the program dicom2 [30] to raw data. Apart from that, the loading time was practically the same, see Table 7 for exact times.

Since rendering in VL is accelerated on graphics card, it is much faster than VTK, where rendering is computed on CPU.

Subject	**VTK**	**VL**
Loading time	17.5 s	18.2 s
FPS for image with fixed quality – low	2 fps	20 fps
FPS for image with fixed quality – medium	1.5 fps	10 fps
FPS for image with fixed quality – high	0.6 fps	3 fps
Render of first preview image (with loading)	22.1 s	1.4 s

Subject	VTK	VL
Render of first image (with loading)	22.1 s	18.5 s
Can render desired frames per second (in less quality)	Yes	Yes
Temporary results while loading	No	Yes
Multi-dimensional transfer functions	2 or 4 dim*	Yes**
Can render DVR and ISO together	Yes	Yes

Table 7: Comparison of loading and visualization. *One component of voxel can define only one color channel. **In actual implementation of VL the dimension of transfer functions is restricted to three.

Subject	VTK	VL
Render of first preview image (with loading and segm.)	25.0 s	1.9 s
Render of first image (with loading and segmentation)	25.0 s	26.6 s

Table 8: Comparison of segmentation speed 2

We used `vtkFixedPointVolumeRayCastMapper` for rendering in VTK example. There are implementations, which extends VTK with hardware accelerated raycasting ([31] or [32]), but they are not included in the official VTK release and they are not publicly available.

The Figure 11 and the Figure 12 show images of neck with configurations of transfer functions conformable to each other.

When we compare visualization, it should be noted that VL is aimed on visualization of volumes and their masks through multidimensional transfer function. The difference in comparison with VTK is that multidimensional transfer functions cannot read data from more volumes. The result is that data sets have more components in one voxel and thus multidimensionality has smaller possibilities of usage.

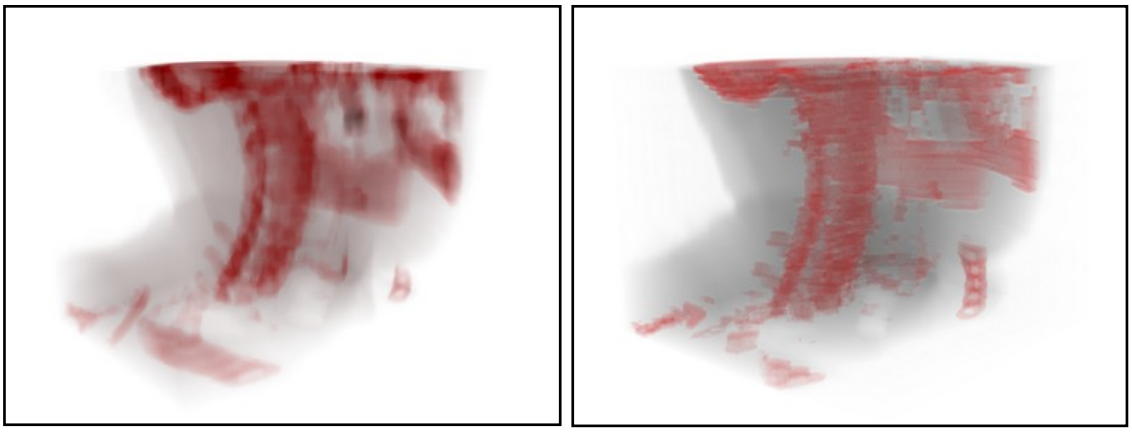


Figure 11: Images of neck rendered with VL 2 fps (left) and VTK 1 fps (right).

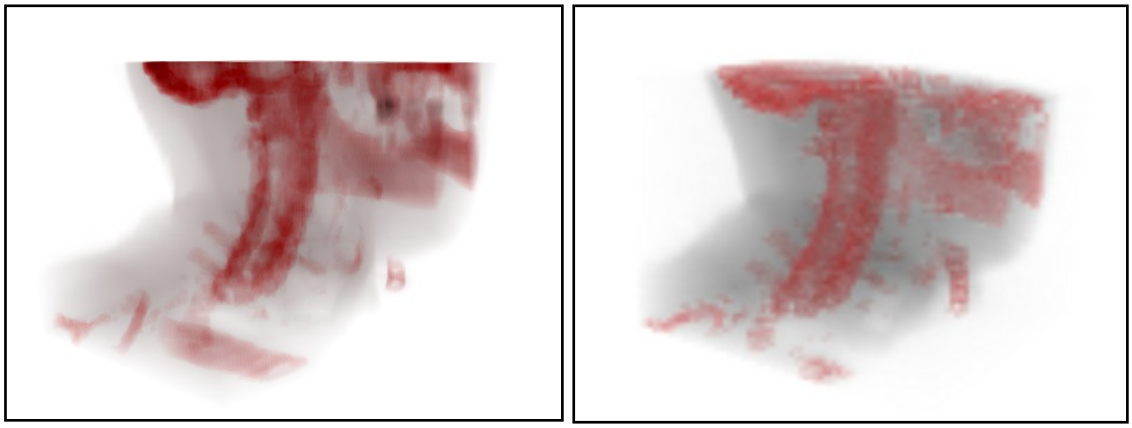


Figure 12: Images of neck rendered with VL 20 fps (left) and VTK 20 fps (right).

5.6 Summary

Subject	VTK	VL
Language	C++	C#
Platform	Multi	Windows
Data types	+	0
Lines of code	0	+
Speed of algorithms	+	-
Ease of writing new algorithms	-	+
Speed of visualization	0	+
Temporary results while loading or segmentation	-	+

Table 9: Results of comparison between VTK and VL. "+" means good support, "0" neutral and "-" bad result or missing feature.

The Table 9 presents the summary results of our comparison. Most negative item of VL is the speed of algorithm. On the other hand, VL can provide all three main tasks (loading, processing and visualization) simultaneously without any line of code by the end programmer and without slowing down the overall computation through the use of all system resources.

This chapter presented differences between VTK and VL and should help in deciding on what tasks are suitable for VTK framework and when it would be better to use our VL framework.

The VTK is suitable for applications developed for multiple platforms and for cases, where priority is given to high performance of algorithms before the end user comfort.

The VL is suitable for cases, where the interactivity of whole application is necessary or for projects aimed for rapid design of new methods, because of both the language and library simplicity.

5.7 VL Testing

This section is aimed at comparison of various settings inside the VL itself.

Data access method	checked	unchecked
`sum += v[0].GetLocalFast<ushort>(i, j, k);`	8.18 s	8.20 s
`sum += v[0].GetLocal<ushort>(i, j, k);`	10.12 s	9.98 s
`sum += v[0].GetFast<ushort>(i, j, k);`	12.26 s	12.53 s
`sum += v[0].Get<ushort>(i, j, k);`	14.40 s	14.61 s

Table 10: The efficiency of the data access methods for algorithms

Table 10 shows results from comparison between the data access methods' speed. We made a loop, which counted values of voxels within data set with dimensions 512^3. The first two methods are for direct indexing of voxels, while the others must subtract the volume position from the index. The first and the third method do not check access outside the input volume, while the others do (these two groups are switched through the use of polymorphism in user defined function). We can see that this optimization speeds up the reading operation for 15-20 percents. The column "checked" means that there was overflow-checking enabled, while the column "unchecked" shows times without overflow-checking. There is obvious, that it has no effect on computation.

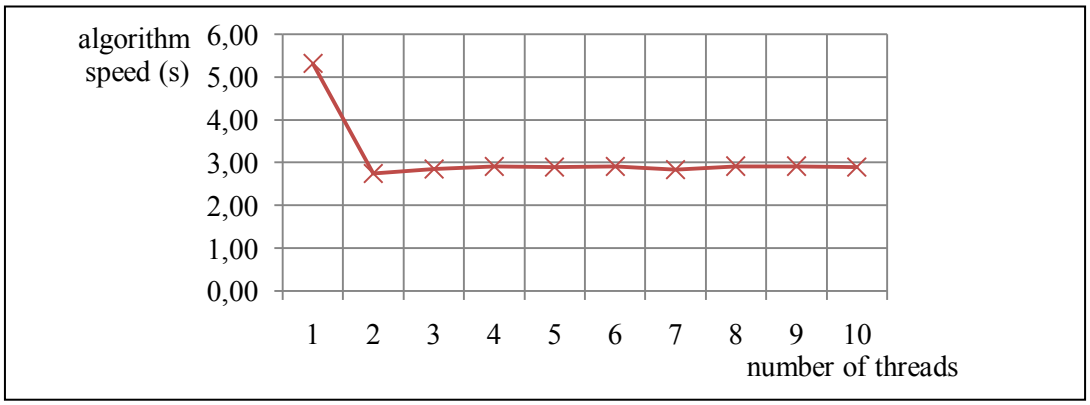


Figure 13: Dependence of the algorithm speed on the number of threads (on dual-core processor)

Figure 13 shows graph with dependency of algorithm speed on the number of threads. Alas only dual-core processor has been used. The figure shows, that the speed of algorithm does not slow down, when there are more threads on a unit.

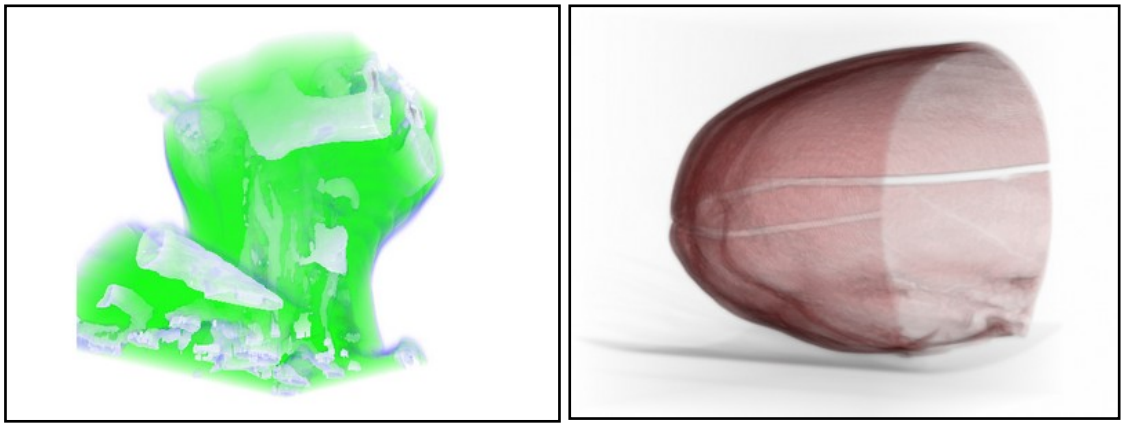


Figure 14: DVR and isosurface rendered together with VL (left), part of skull rendered with VTK (right)

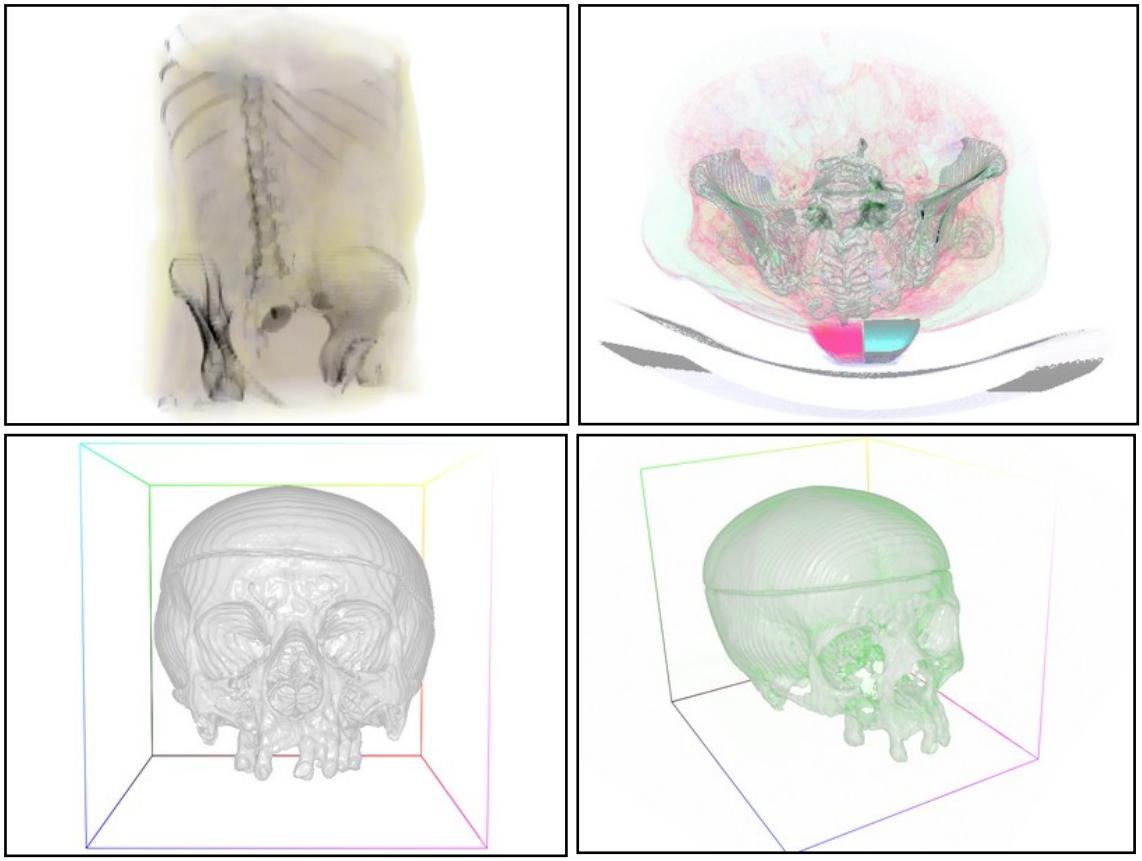


Figure 15: Part of body (top left), pelvis (top right), skulls (bottom). All were rendered with VL framework.

Chapter 6

Conclusion

In this thesis a framework and application interface oriented on processing and visualization of medical volume data sets is presented.

At the beginning, there is a short introduction to all technical aspects dealt with the framework. That is image processing, volume visualization, GPU programming, and the .NET Framework architecture. At the end of the first chapter there are the most important features, which our framework should satisfy.

The second chapter is devoted to existing free and open source software solutions for processing or visualization of volume data sets. It follows from the review, that most compact programs are provided by Kitware Inc., which has been involved in the development of VTK and ITK.

Next a description of the design of application interface and its features is presented. Most important of them is parallelization of multiple modules of the framework and concept of masking volumes, which are used instead of multi-component values in data sets.

Chapter four describes our implementation and its main features. There is an overview of classes and relationship between them. Other interesting problems like rendering process are also included.

Comparison of our framework with The Visualization Toolkit, the leading software in visualization, is made in the fifth chapter. Main differences between corresponding parts of both frameworks are described, such as loading, processing and visualization of data sets.

6.1 Results

The VL is compact volume-specialized framework developed natively in .NET architecture with parallelization, suitable for rapid development of medical image processing algorithms.

Main features of our framework are:

Processing and visualization of three dimensional data sets. The library provides simple environment for all three main tasks (loading, processing, and visualization) in processing pipeline for the end application developer.

Visualization. The ray casting algorithm is implemented on the GPU and makes real-time simultaneous rendering of both direct volume rendering technique and isosurface rendering technique possible.

Oriented on medical purposes. The concept of masking volumes optimizes memory usage and provides more possibilities of usage of our renderer.

Parallelization. The parallel run of data loading and of the processing algorithm provides good utilization of the system resources, because the disk operations interlace the processing operations.

Including into existing projects. The design of library makes possible to simply integrate parts of library to existing software projects.

Main problems of our implementation are:

- The target platform of VL framework is Microsoft Windows only.
- Performance of algorithms written through the use of VL is not as good as in comparable frameworks.
- The number of volumes rendered altogether is restricted to three.
- It is not possible to process data sets that do not fit into the main memory and visualize data sets that do not fit into the graphics memory.

6.2 Future Work

Except for optimization of the current source code, the main direction of the future work might be in implementing the remaining application interface features, primarily completion of the concept of multiple masks volume (our implementation restricts the number of currently visualized volumes to three) and possibility of localized transfer functions.

Next essential improvement would be to make possible of integration of some ITK modules.

Another possibility would be to modify the source code to fit Mono [15], an open source implementation of the Microsoft .NET architecture.

In API the color transfer function interface can be extended in order to handle also isosurfaces.

References

[1] A. C. Kak and Malcolm Slaney: *Principles of Computerized Tomographic Imaging,* Society of Industrial and Applied Mathematics, 2001

[2] J. P. Homak: *The Basics of MRI*, 1996

[3] R. C. Gonzalez, R. E. Woods: *Digital Image Processing*, 3th Edition, 2008

[4] A. B. Ekoule, F. C. Peyrin, C. L. Odet: *A Triangulation Algorithm from Arbitrary Shaped Multiple Planar Contours*, 1991

[5] W. E. Lorensen, H. E. Cline: *Marching Cubes: A high resolution 3D surface construction algorithm*, 1987

[6] L. Westover: *Footprint Evaluation for Volume Rendering*, 1990

[7] P. Lacroute, M. Levoy: *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*, 1994

[8] K. Engel, M. Kraus, T. Ertl: *High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading*, 2001

[9] S. Parker, P. Shirley, Y. Livnat, Ch. Hansen, P. Sloan: *Interactive Ray Tracing for Isosurface Rendering*, 1998

[10] G. Marmitt, A. Kleer, I. Wald, H. Friedrich, P. Slusallek: *Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing*, 2004

[11] J. Kruger, R. Westermann: *Acceleration techniques for GPU-based volume rendering*, 2003

[12] *The OpenGL Graphics System: A Specification 2.1*, http://www.opengl.org/, 2006

[13] *General-purpose computing on graphics processing units*, http://www.gpgpu.org/

[14] *Digital Imaging and Communications in Medicine (DICOM) Standard*, http://dicom.nema.org/

[15] *Mono*, http://www.mono-project.com/

[16] *Insight Segmentation and Registration Toolkit (ITK)*, http://www.itk.org/

[17] Kitware Inc., http://kitware.com/

[18] *Visualization Toolkit (VTK)*, http://www.vtk.org/, 2008

[19] W. Schroeder, K. Martin, B. Lorensen: *The Visualization Toolkit An Object-Oriented Approach to 3D Graphics*, 4th Edition, Kitware, 2006

[20] P. Bhaniramka, Y. Demange: *OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data sets*, 2002

[21] I. Bitter, R. V. Uitert, I. Wolf, L. Ibanez, J.-M. Kuhnigk: *Comparison of Four Freely Available Frameworks for Image Processing and Visualization That Use ITK*, 2006

[22] J. J. Caban, A. Joshi, P. Nagy: *Rapid Development of Medical Imaging Tools with Open-Source Libraries*, 2007

[23] *Windows.Forms reference documentation* (MSDN), http://msdn2.microsoft.com/en-us/library/dd30h2yb.aspx

[24] *Cg Language Specification*, NVIDIA, http://developer.download.nvidia.com/cg/Cg_2.0/2.0.0012/Cg-2.0_Jan2008_LanguageSpecification.pdf

[25] *Framebuffer object (FBO) extension*, http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt

[26] C. Salama, A. Kolb: *A Vertex Program for Efficient Box-Plane Intersection*, 2005

[27] *.NET Framework* (MSDN), http://msdn.microsoft.com/NETFramework/, 2008

[28] *C# Language Specification* (MSDN), http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx, 2008

[29] .NET Wrappers for VTK 5, *VTK.NET*, http://vtkdotnet.sourceforge.net/, 2006

[30] *dicom2*, command line program for converting DICOM data to raw data, http://www.barre.nom.fr/medical/dicom2

[31] J. Allard, B Raffin: *A Shader-Based Parallel Rendering Framework*, 2005

[32] R. Brecheisen: *Real-time volume rendering with hardware-accelerated raycasting*, 2006

[33] B. T. Phong: *Illumination of Computer-Generated Images*, 1973

[34] N. Max: *Optical models for Direct Volume Rendering*, 1995

[35] L. Maršálek: *Lighting using GPU*, 2005

Appendix A: Contents of the CD

There are the following directories on the attached CD:

- Directory **Data** contains four tested data sets. There are anonymous CT images of skull (512x512x49), neck (512x512x27), body (512x512x72), and pelvic (512x512x41)
- Directory **Text** contains electronic version of this document in the PDF format, as well as Microsoft Word 2007 (.docx) version.
- Directory **VL/src** contains the source code of our library and source code of the project developed to test this library (subdirectory **Example_proj**). There is single a solution file (.sln) for both projects for Visual Studio 2008.
- In the directory **VL/bin** there are binaries for the testing project (Example.exe) with necessary libraries. Target platforms are 32 bit versions of Windows XP and Vista.
- In directory **VL/bin64** there are binaries for the testing project (Example.exe) with necessary libraries. Target platforms are 64 bit versions of Windows XP and Vista.
- Directory **VL/doc** contains reference manual for our library in the Microsoft Help CHM format (.chm).
- In directory **VTK** there is the source code of the project used for comparisons made in Chapter 5.

Appendix B: VL Programming Guide

This appendix describes common techniques, which are necessary for developing software with VL library.

VL is designed to be easily included to new as well as to existing projects. All graphical tools are designed as User Controls, which allow simple changing of their appearance and extending them with their own functionality.

For introduction of VL possibilities, a simple form with controls from VL is described.

B.1 Getting Started

Inclusion of VL library to Visual Studio 2008 project consists of several steps.

- First we must add reference on VL library: in "Solution Explorer / *Project_name* / References / right click / Add Reference… / VL.dll"
- Copy all necessary files to our output directory – files with DICOM elements dictionary from data folder and supporting libraries (if they are not in the system already): cg.dll, cgGL.dll, Gobosh.DICOM.dll, opendicom-sharp.dll, Tao.Cg.dll, Tao.OpenGl.dll, Tao.Platform.Windows.dll. These files can be obtained from our testing project.

B.2 Creating 3D Renderer

To create control for 3D volume visualization, we open our form in Designer View and drag `Renderer3DControl` from the Toolbox. Next, we must append following line to constructor of our form:

```
renderer3DControl1.StartRender();
```

When we now build and start the application, it will look like in Figure 16 (left top).

The sign "Powered by Tao" means, that there is no volume assigned to the renderer, so we will now create one:

```
VL.Api.IVolumeParams vp = new VL.VolumeParams(
            256, 256, 256,          // dimensions
            VL.Api.eType.vl_byte    // data type
);
VL.Api.IVolume volume = VL.VolumePtr.Factory.CreateEmptyVolume(vp);
renderer3DControl1.GenerateDimensions(1);
renderer3DControl1.SetVolume(volume);
```

```
renderer3DControl1.StartRender();
```

Code snippet 3: Creating and displaying of the volume

First, we filled in parameters of the new volume and called volume factory to create one empty volume. Result is shown in the Figure 16 (right top). Colored lines mean, that the volume is created, but that it is empty.

Next, we will fill the volume with some data. We create and execute our algorithm defined in function f() immediately after volume is created:

```
VL.Algorithm alg = new VL.Algorithm();
alg.ExecuteOnVoxels<byte>(f, volume, null);

byte f(int x, int y, int z, VL.Api.DataAccess[] inputData,
                                          object userParams)
{
      return (byte)(
            (x < 100 ? 40 : 10) +
            (y < 100 ? 40 : 10) +
            (z < 100 ? 40 : 10)
      );
}
```

Code snippet 4: Definition of function for algorithm

When this is executed, it can be seen that the computation of our algorithm is done in separate thread(s) and that it does not affect a user with any delay. Figure 16 bottom left.

B.3 Creating 2D Renderer

In VL library there is also UserControl for visualization of 2D volume slices. The main difference from 3D renderer is that `Renderer2DControl` can be used for user input in the sense that user can select a point with 3D coordinates. That is done through event handler `PickingEventHandler`, Assigning volumes and transfer functions is the same for both renderers.

B.4 Transfer Function Handling

Handling of the volume color is done through the transfer function. We have one volume, so now we will use `TransferFunction1D`:

```
      VL.TransferFunction1D tf = new VL.TransferFunction1D(8);
      tf.SetSegment(0.0f, 0.2f, new VL.Utils.RGBA(0, 0, 0, 0.2f),
                                new VL.Utils.RGBA(0, 1, 1, 0.2f));
      tf.SetSegment(0.2f, 0.4f, new VL.Utils.RGBA(0, 0, 0, 0.2f),
                                new VL.Utils.RGBA(1, 0, 1, 0.2f));
      tf.SetSegment(0.4f, 1.0f, new VL.Utils.RGBA(0, 0, 0, 1.0f),
```

```
                                new VL.Utils.RGBA(1, 1, 0, 1.0f));

        renderer3DControl1.AddTransferFunction(tf);
```

Code snippet 5: Definition of transfer function

Behavior of transfer function is set by the function `SetSegment()`, which takes four parameters. The first and the second are for volume density and the third and the fourth for final color. See Figure 16 (bottom right) for the result. There are also other functions for setting the transfer function, which are described in the reference manual for VL.

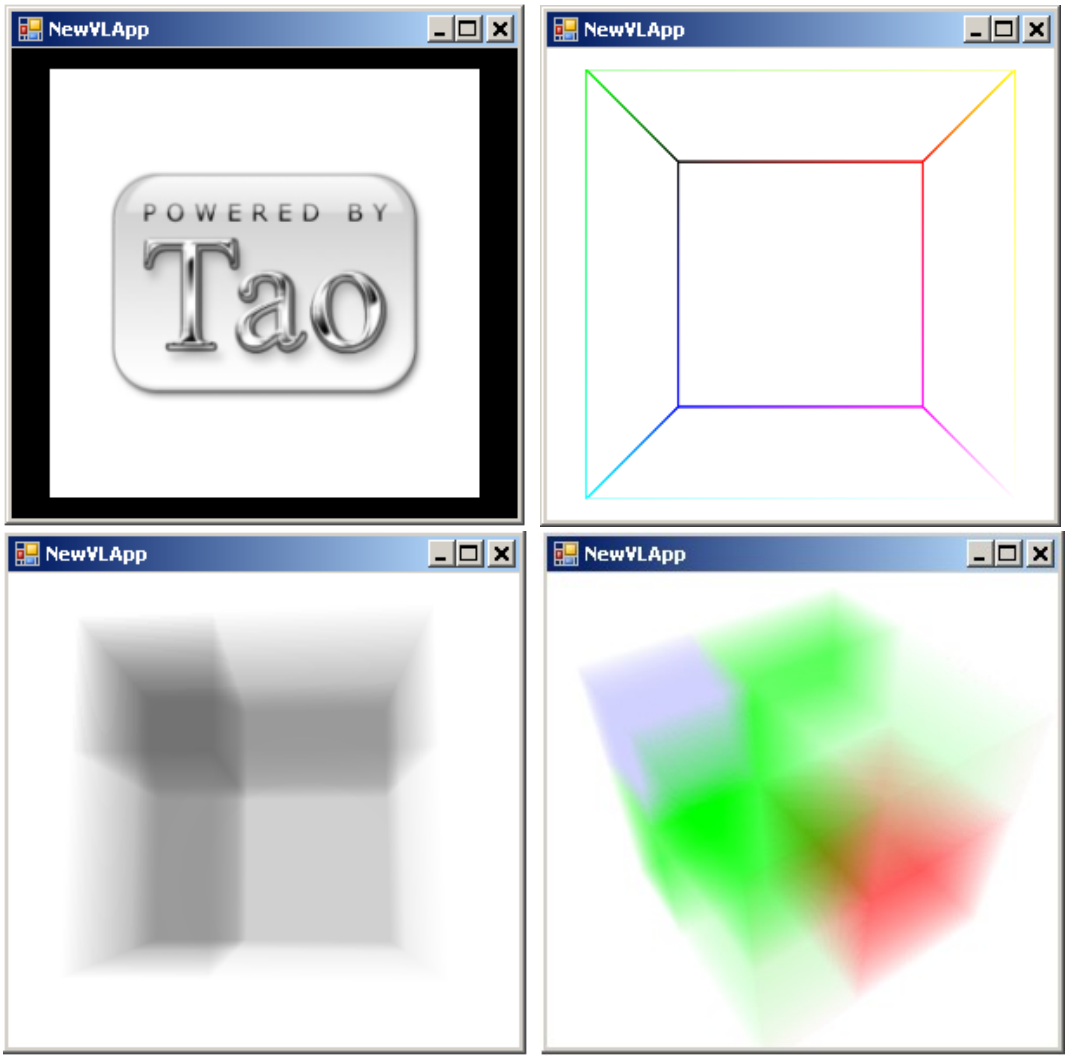


Figure 16: Designing of our form

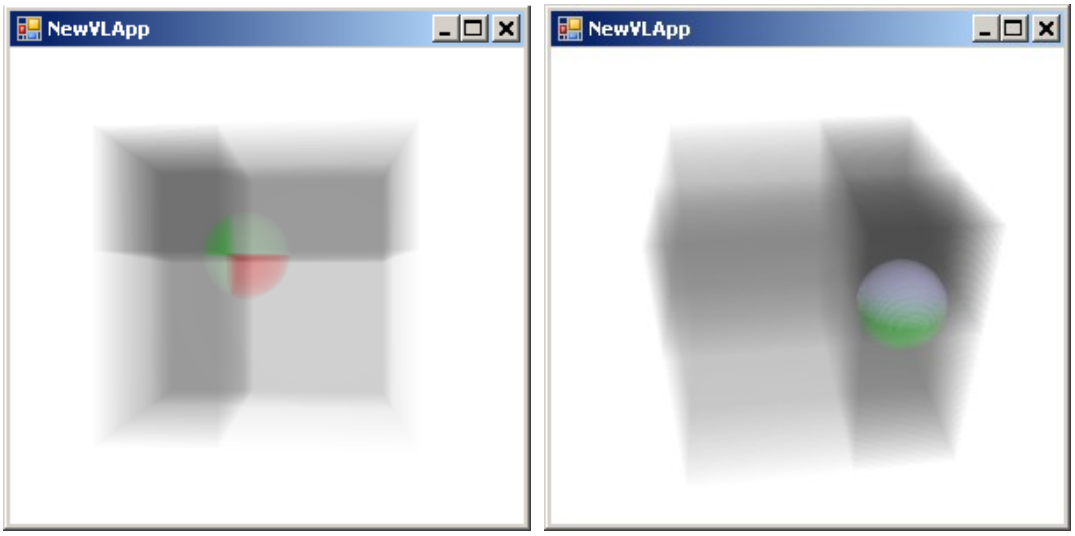


Figure 17: Designing of our form with multidimensional transfer function

Next example, useful to present, is multidimensional transfer function. It means that target color depends on values from more volumes. We must create one more volume, which will stand for a "virtual ball". This ball will be moving across the main volume, so that it will be clear, that ball's color is changing and that it depends on multidimensional transfer function.

```
VL.Api.IVolumeParams vp2 = new VL.VolumeParams(
      80, 80, 80, VL.Api.eType.vl_byte
);
ball = VL.VolumePtr.Factory.CreateEmptyVolume(vp2);
VL.Algorithm alg2 = new VL.Algorithm();
alg2.ExecuteOnVoxels<byte>(f2, ball, null);

byte f2(int x, int y, int z, VL.Api.DataAccess[] inputData,
                                          object userParams)
{
      x -= ball_r2; y -= ball_r2; z -= ball_r2;
      return (byte)(x * x + y * y + z * z < ball_r2 * ball_r2 ?
            byte.MaxValue : 0);
}
```

Code snippet 6: Creating of the volume, which stands for a ball

Function `f2()` changes the values inside the "ball" to 255 and others are left as zeroes. This volume represents a simple mask for our main volume and function `f2()` represents a simple segmentation. Next we must change the transfer function to `TransferFunctionMultiD` and fill in right values:

```
VL.TransferFunctionMultiD tf2 = new VL.TransferFunctionMultiD(8, 1);
tf2.SetSegment(new VL.Api.TFPoint(0, 0), new VL.Api.TFPoint(1, 0),
      new VL.Utils.RGBA(0, 0, 0, 0), new VL.Utils.RGBA(1, 1, 1, 1));
tf2.CopyDimensionFromTF1D(tf, 1);
```

Code snippet 7: Definition of multidimensional transfer function

As we can see, it is quite similar, except places, where the second dimension parameter must be specified. For `TFMultiD` there is also auxiliary function for cooperating with one-dimensional function. For moving the ball the following command is used, which is executed every 50 milliseconds by common timer.

```
ball.Params.Pos += dir;
```

The result is shown in the Figure 17. The whole example is included in the attached CD with some other features, like sample of extending the camera.

There is UserControl `TransferFunctionControl` for simplification of the transfer function definition in the VL library. It can be interactively connected with renderers through its property `tf` and event `TFChanged`. For more information see the reference manual.

Appendix C: User Manual

Appendix C describes a small project "Example", which was developed for testing purposes of the VL library. It should be pointed out, that this project is not the only and main purpose of this work. Here, the VL UserControl's behavior will also be described.

The "Example" is a project for 2D and 3D visualization of DICOM data sets. It can render DVR and isosurface images. It allows definition of transfer function, viewing DICOM file elements, setting rendering quality, and more.

C.1 System Requirements

- Windows XP or Vista operating system.
- .NET Framework 2.0 (Windows Vista contains it by default).
- 256 MB RAM (Depends on used data sets).
- NVIDIA GeForce 6 Series or AMD/ATI Radeon R520 Series or higher (Shader model 3.0 required).
- 256 MB Video RAM (Depends on used data sets).

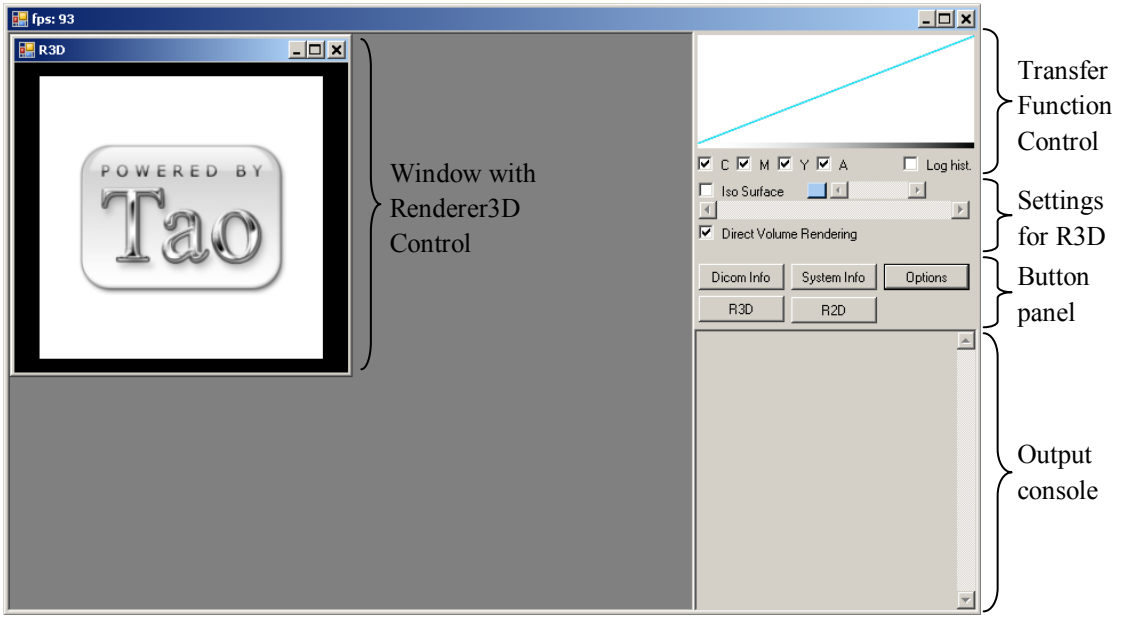
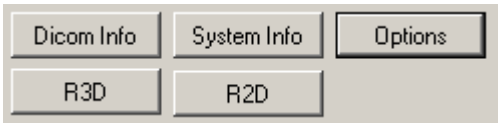


Figure 18: Main parts of "Example" project window.

C.2 The Main Window

The main window of the project is MDI[34] and consists of two parts, the working desktop, and the right panel, as shown in the Figure 18. Working desktop is intended for windows with 2D and 3D renderers.

Most important part of the right panel shows buttons:



- **Dicom Info** serves for opening of DICOM files.
- **System Info** opens panel with system values, like memory usage, processor load, count of processor's cores, and graphics card extensions.
- **Options** serves for setting of advanced mainly visualization options.
- **R3D** opens new window with Renderer3DControl component.
- **R2D** opens new window with Renderer2DControl component.

Below that panel is a place for displaying messages from the VL library, the "output console". Times of some loading, processing and visualization processes, messages from the renderer's initialization and some error messages can be placed there.

In the title bar of the main window the actual count of frames per second is shown.

C.3 Dicom Info

For loading files in the DICOM format there is DicomInfoControl (Figure 19). It consists of two tabs with treeviews. In the first all elements from the file are displayed as they are stored in the file (in same structure and order). The second shows DICOMDIR files and regroups file records in a logical order (patient/study/series/slices). They are marked with colors for better readability – blue means one file or slice, green more of them and red is for record that points on files, which cannot be opened.

After selecting a file with mouse click, loading is started in the last active renderer.

[34] Multi-document interface

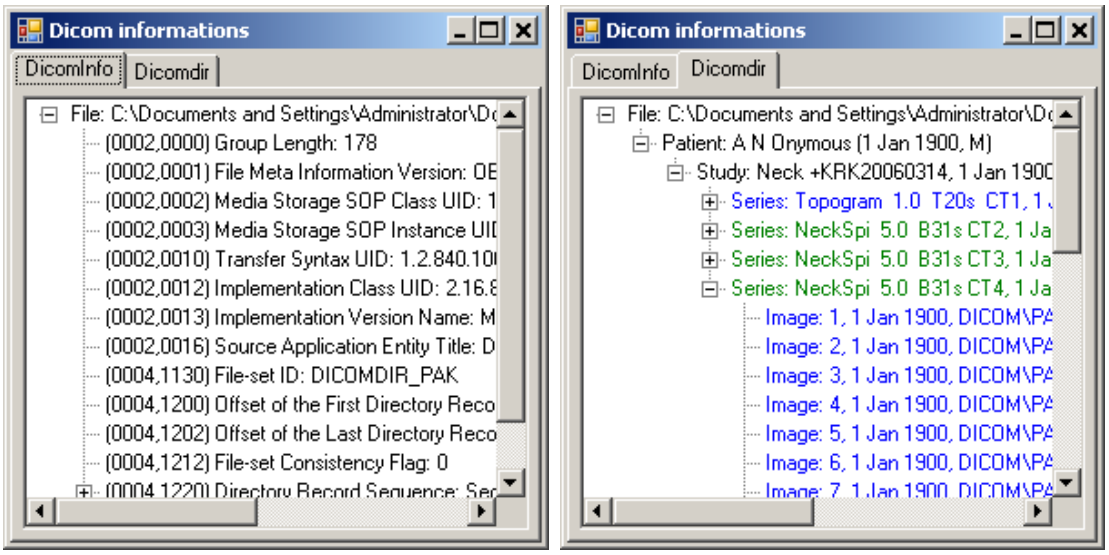


Figure 19: DicomInfoControl

C.4 Options

In the General part of the Options window there are settings for testing and debugging tasks. Parallel computations, printing of algorithm's speed and rotating of camera in 3D renderers can be turned on or off in this window.

In Rendering part of Options accuracy of visualization (at the expense of speed), Lego projection (interpolation type), inverting of colors (for printing purposes), and shading for isosurfaces can be set. Ambient shading means, that there is no dependence of color on isosurface gradient, diffuse is for non-shiny reflection and specular for specular reflection.

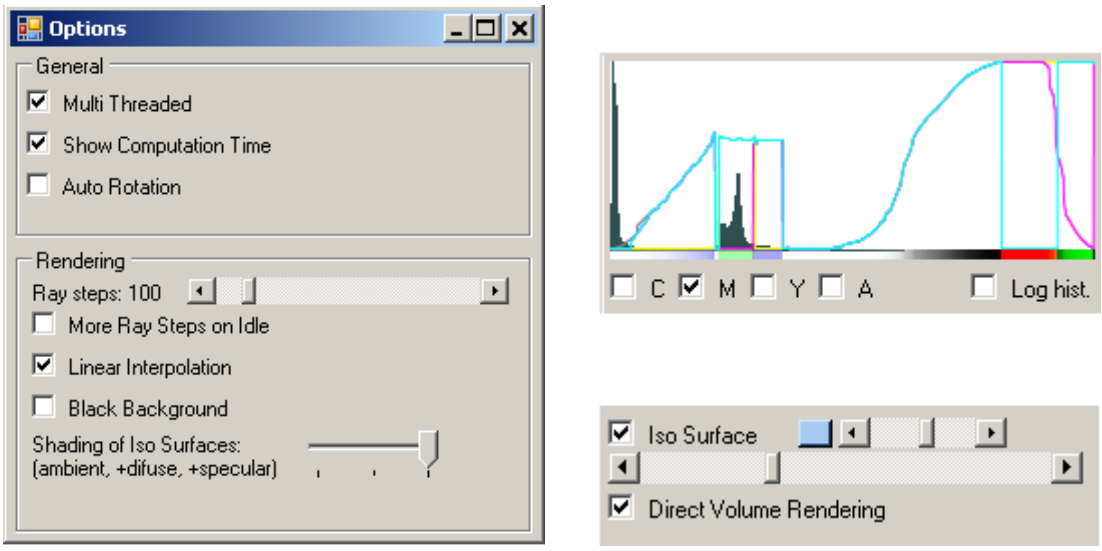


Figure 20: Options (left), TransferFunctionControl (right up), and R3D settings

C.5 R2D a 3D

Renderer2DControl (Figure 21 left) is intended for viewing three 2D axis aligned slices of the volume. Color is set up by the TransferFunctionControl.

TransferFunctionControl (Figure 20 right up) displays five related graphs. In the background there is a histogram from actual volume (dark gray). Vertical scale of histogram can be changed from linear to logarithmic with the checkbox "Log hist". Next four graphs are meant for color channels and opacity. Colors are either red (R), green (G), blue (B), or cyan (C), magenta (M), and yellow (Y), depending on "Black Background" checkbox setting. Under graphs the final color is displayed.

Setting of these four graphs can be done with the left mouse button by directly drawing to background. Modification is done only to those channels, which are checked.

Renderer3DControl (Figure 21 right) relates to DVR and isosurface visualization. These can be checked up with "R3D settings" from the right panel (Figure 20 right bottom). If neither of checkboxes is checked, DVR rendering is used. When isosurface is enabled, color and opacity is set up by a button and a slider right of it. The second slider sets the desired density.

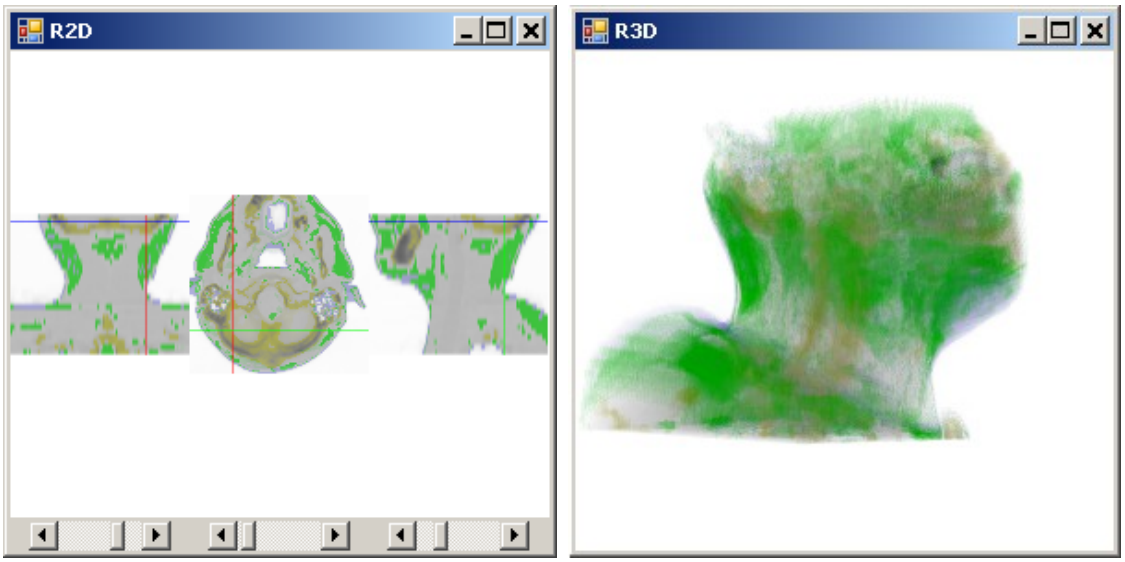


Figure 21: Renderer2DControl (left) and Renderer3DControl (right)