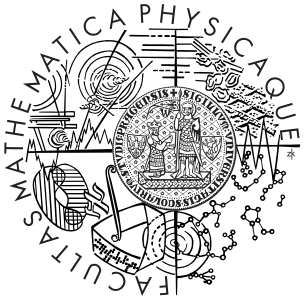
Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS



Vladimír Hrinčár

Volume Visualization of Human Skulls

Department of Software and Computer Science Education

Supervisor: Mgr. Lukáš Maršálek,
Study Program: Computer Science, Software Systems

2008

Most of all I would like to thank my supervisor and team leader of this project, *Mgr.Lukáš Maršálek*, for his time, patience, feedback and for many valuable advices, which gave me a lot of new ideas. His cooperation on this project was one of the best I have ever experienced.

Next I would like to thank all my friends, who were always ready to help me with every problem I have asked and I am grateful to all my fellow colleagues at Laboratory Imaging, for their patience with me and for providing great working environment.

I also want to thank my parents and family for their support, especially during last few weeks when I was not able to come home, but their cheers made me very happy and helped me greatly in those times.

I declare that I wrote the thesis by myself and listed all used references. I agree with making the thesis publicly available.

Prague, August 8, 2008 Vladimír Hrinčár

Contents

List of Figures

List of Tables

Názov práce: Vizualizácia CT snímkov lebiek
Autor: Vladimír Hrinčár
Katedra (ústav): Kabinet software a výuky informatiky
Vedúci diplomovej práce: Mgr. Lukáš Maršálek
e-mail vedúceho: lukas.marsalek@mff.cuni.cz

Abstrakt: Zobrazovanie objemových dát je dôležitý nástroj pre skúmanie a pochopenie trojrozmerných vedeckých dát, ako napríklad tých, ktoré sú získané pomocou CT skenovania. Nedávne pokroky v oblastiach hardvéru a softvéru obnovili záujem o interaktívne a vysokokvalitné zobrazovanie objemových dát, pretože sa momentálne stávajú dostupnými dokonca aj na domácich počítačoch. V tejto práci prezentujeme novú aplikáciu určenú k vizualizácii CT snímkov ľudských lebiek. Zameriavame sa hlavne na kvalitu zobrazenia, pričom pridávame možnosti pre porovnávanie lebiek a vizualicáciu ich rozdielov. Naše riešenie je postavné na .NET aplikácii, ktorá sprostretkúva uživateľské rozhranie a externej dynamickej knžnici obsahujúcej samotný renderer. K implementácii vykosokvalitného zobrazovania izoplôch sme použili algoritmy založené na metódach interaktívneho vrhania lúčov a ďalej v nej obsiahli možnosť presvitnosti izoplôch a vrhania ich tieňov. Taktiež sme pridali podporu pre zobrazovanie viacerých objemovych dát a ďalej predstavujeme nové zobrazovacie módy pre vizualizáciu ich rozdielov. Náš systém používa technológiu NVidia CUDA, čím ukazuje ako je možné využit urýchlenie pomocou moderných grafických procesorov. Naše výsledky ukazujú, že sme schopní predložit nástroj určeny na interaktívnu vizualizáciu, ktorý do značnej miery može pomôc v oblastiach ako je napríklad antropológia.
Klúčové slova: CT, CUDA, GPU, izoplocha, lebka, vrhanie lúča, zobrazovanie objemu

Title: Volume Visualization of Human Skulls
Author: Vladimír Hrinčár
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Lukáš Maršálek
Supervisor's e-mail address: lukas.marsalek@mff.cuni.cz

Abstract: Volume visualization is an important tool for exploration and understanding of complex 3D scientific data, as those acquired for example by CT scanning. Recent advancements in hardware and software renewal interest in interactive, high-quality volume rendering as it now becomes feasible even on desktop computers. In this work we present new application designed for visualization of CT scans of human skulls. We focus on image quality and add new features for skull comparison and difference visualization. Our solution is based on .NET application, which provides user interface(GUI) and external volume renderer as dynamic link library. We have implemented complex renderer for high-quality isosurfaces using algorithms based on interactive ray-marching methods including self shadowing and transparency. We also add support for multiple volumes and introduce new rendering modes for difference visualization. In addition our system uses NVidia CUDA technology, showing how to harness the power of modern GPUs for accelerating this complex task. Our results show that we are able to deliver interactive visualization tool, which can greatly help in areas like anthropological research.
Keywords: CT, CUDA, GPU, isosurface, ray-casting, skull, volume rendering

+

Chapter 1

Introduction

1.1 Motivation

Many years have past since the time when real-time volume rendering techniques required system with so enormous computing power that it was affordable only by some specialized medical institutions or universities. Huge progress in the computing performance of microprocessors and increasing availability of low-cost multi-core systems in the last few years brought us massive use of specialized software implementing various volume visualization methods interactively and with high-quality results. Also the source data can now be acquired much more easily and cheaper than in the past as originally medical imaging methods like *Computed tomography* (CT) are common in use by many other institutions and some CT scanners are now parts of their equipment.

Increasing demand for miscellaneous visualization applications from other organizations and science departments lead to developing full-scale commercial software solutions. Using the recent hardware, several methods have been upgraded to take advantage of being accelerated by modern graphics cards thus gaining significant speedup. These applications come with plenty of features and rendering techniques but they not provide specific features accessible together in one tool. Also none of them is targeted purely for anthropological research, in which we want to contribute by delivering our visualization solution.

1.2 Volume rendering

1.2.1 Overview

Volume rendering is a technique used to display a two-dimensional projection of a three or four-dimensional discretely sampled data set, a scalar or vector field. This fundamentally differs from standard visualization of objects, that are represented and displayed by geometric primitives. These techniques found their use in many areas like medicine, chemistry, archeology or microscopy. Most applications work with 3D grids of input data, but there are also some that work with more dimensions like with time sequences of volumetric data. Representations by scalar field prevail over vector fields but in areas like meteorology, it is common to have grid points with multiple quantities, stored as vector fields.

Typical data set for volume visualization is acquired by *Magnetic Resonance Imaging* (MRI) or *Computed Tomography* (CT). There are also examples of synthetic datasets, like generated textures of wood, marvel, etc. They can be used and visualized together with data from empirical measurements to provide mixed visualization when the system displays most appropriate information from each data set at each point. However such a implementation requires to have all input datasets properly registered.

1.2.2 Volume Visualization Methods

There are fundamentally two types of algorithms for volume visualization - *Isosurface extraction* and *Direct volume rendering* (DVR). First class of algorithms is based on extracting surfaces of equal values from the volume and then rendering them as polygonal meshes. These surfaces are called isosurfaces and are defined by their threshold value and color (possibly with opacity). Since the surface is generated before the visualization as a preprocessing, the rendering process is then usually very fast, but on the other hand surface mesh data have to be regenerated again every time the isovalue changes. Direct volume rendering takes different approach, where DVR method maps volume data directly to the screen space without using any geometrical primitives. This allows to display entire volume as a block of data and it is based on real physical models for light absorption or accumulation. Another difference is that the volume traversal has to be performed every time for every new image. DVR methods produce the highest possible image quality but at a high price, because such a rendering is computationally intensive task.

1.3 Computed tomography

Computed tomography is a technique that generates a three-dimensional image of an object from a series of two-dimensional X-ray images of cross-sections of that object. According to the type of scanner used, it is possible to produce images with different resolution. Standard resolution of the single slice image may vary between 256x256 to 4096x4096 pixels. Pixel is a two-dimensional unit based on the matrix size and the field of view. When the CT slice thickness is also factored in, the unit is known as a Voxel, which is a three-dimensional unit. Voxel's width and height are usually equal (may be from 0.5mm to 2mm) but its thickness can be different so it is not guaranteed that the voxel represents a regular cube. This distance between two slices is called *collimation* and its minimum value is defined by the type of the scanner, where the latest medical scanners are able to produce slices with distances like 0.5mm. Higher resolution can be achieved by specialized industrial scanners and for example synchrotron X-ray tomographic microscopy techniques produces scans with details as fine as 1000th of a millimeter.

Each voxel stores a value that represents its radiodensity, which is measured mean attenuation of the material that corresponds to that voxel. It can be any value within a interval from -1024 to +3071 (assumed 12-bit data) and it is mapped on the *Hounsfield scale*. Water has an attenuation of 0 Hounsfield units (HU) while air is -1000 HU, cancellous bone is typically +400 HU, cranial bone can reach 2000 HU or more and values over 3000 are typical for X-ray almost non-permeable materials, like metals.

1.4 CT in anthropology and archeology

Computed tomography as nondestructive evaluation (NDE) technique has already been recognized by some archaeologists and museum curators as an efficient tool for nondestructive studying of archaeological artifacts. Different NDE method based on radiofrequency pulses, *Magnetic Resonance Imaging* (MRI) on the other hand useless for such kind of research, because this technique is sensitive to hydrogen nuclei spin orientation. Mineralized bone delivers none or only weak signals and thus best applicable specimens are those containing water like brain or other organ tissue.

With the help of CT, an archaeologist can research for example the cuneiform texts sealed in clay envelopes without a need to destroy the outer envelope as they had to do it in the past. Also another revelations like the

evidence of repairs and information about ancient manufacturing techniques that could help to verify the authenticity of certain artifacts has been discovered on the scans of bronze Chinese Urns when using industry CT scanner.

These advanced imaging techniques have been successfully applied to anthropological research as well and the computed tomography became the ideal research tool to access the internal structures of various precious fossils without even touching, let alone damaging them. Additional post-processing may also help as many fossils are filled with stone encrustations and therefore can not be examined properly. Digital reconstruction techniques were also successfully used for visualizing most missing parts of some partially destroyed anthropological specimens. These new imaging and post-processing techniques gave a birth to a new field of anthropological research - *Virtual Anthropology*, which is characterized as a multidisciplinary approach to studying anatomical data representations in 3D or 4D, particularly humans, their ancestors, and their closest relatives.

We believe that all these mentioned examples of the usage of CT as the nondestructive tool proved to be useful in preliminary studies of many other areas of archaeological and anthropological research.

1.5 Opportunity

Faculty of Science of Charles University in Prague and more precisely it's Department of Anthropology is one of the next candidate which is about to expand possibilities in their future research concerning human skulls by collaborating with Computer Graphics Group at Faculty of Mathematic and Physics. This would be a great opportunity to test some of the volume visualization methods previously used in other areas and by other institutions with data obtained from the fields of research like anthropology or even archeology. Also modifying existing solutions and developing new ones which are more suited for this specific use will be tested in practice. Comparison of multiple skulls and visualizing them so differences can be examined is one of the example of the new method.

This thesis covers only a small part of the whole problem and it is the first step in the future teamwork project itself. In this first phase several requirements have to be accomplished and at the end results should be delivered. Our primary objectives are described in the next section about the goals of the thesis.

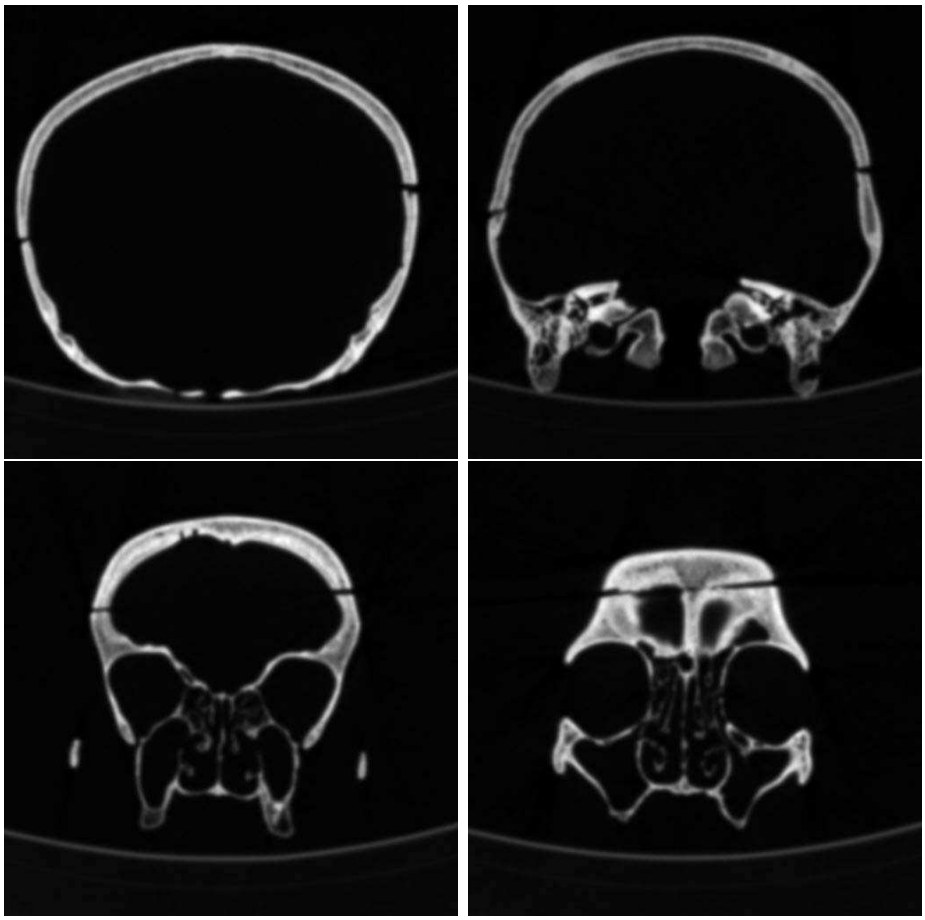


Figure 1.1: Examples of human skull CT slices scanned by Department of Anthropology at Faculty of Science of Charles University in Prague

1.6 GPU acceleration in volume visualization

Huge progress has been made not only in the area of general-purpose processors (CPU), but also the graphics processors (GPU) increased their performance rapidly during last few years. Recent development in the programmability of graphics hardware brought many applications, which tried to take advantage of modern graphics cards and their powerful GPUs. When enhanced by graphics accelerator, many volume visualization algorithms performed a lot faster compared to their CPU implementation. Therefore most research in the area of volume rendering led to exploiting all possibilities of accelerating volume visualization process and updating its algorithms to suitable form for GPU implementation. Nowadays GPU solution is considered preferred and affordable option for real-time rendering on desktop

platform.

The latest evolution of high-level GPU programming was introduced by NVIDIA, when their new technology called CUDA was presented in late 2006. CUDA is specific form of solution for general-purpose computing on graphics processing (GPCPU) available on all recent NVIDIA cards. When working with algorithms, which are highly convenient for parallel computing, like these in image processing, flow simulation or in our case 3D rendering, CUDA allows us to take advantage of high performance computation power of modern GPU with multiple cores suitable for massive parallelism. Theoretical computational peak can reach over 1TFLOP/s on recent GPUs. Just for the comparison, the maximum performance of Intel Core 2 Quad CPU is around 100GFLOP/s. This gives us the great tool to solve most complex compute-intensive tasks more effectively on GPU. With CPU left relatively unoccupied, we are able to perform additional tasks, which otherwise can not be executed for the background processing.

1.7 .NET framework

The Microsoft .NET Framework is a software technology, that is available with several Microsoft Windows operating systems as a managed code programming model recommended for creating new applications for the Windows platform. It was created to take over some major responsibilities that are usually handled by programmer and to focus on rapid application development, platform independence and network transparency. All .NET applications are executed in the framework's virtual machine software environment that manages the program's runtime requirements. Such a behavior is in some sense similar to Java Virtual Machine, key component of the Java Platform. Because of this, programmer doesn't need to consider the capabilities of the specific CPU that will execute the program. Framework's runtime environment also provides additional services for .NET applications like memory management, and exception handling.

Another great advantage brought by .NET, is an included large library of precoded solutions for many programming problems. This collection of precoded solutions is called Class Library and by combining it with own code, programmer can easily gain access to areas like user interface (GUI), numeric algorithms, data access, network communications, etc.

Managed code however runs more slowly and requires more resources than a well written conventional Windows program. But all advantages brought by .NET together with the fact that new CPUs and memory mod-

ules will always get cheaper, make the Microsoft .NET framework as another preferred option for current developers.

1.8 Goals of the thesis

Main task of this thesis is to create a tool for the interactive isosurface visualization of CT scans of multiple human skulls for potential use in the anthropology research and public displays in the museums. Additional work should be concentrated on the visualization of their differences. This should greatly help anthropologist to analyze numerous specific differences in the sets of human skulls and to compare them in the real-time.

Core renderer would be based on the interactive ray marching methods implemented entirely on GPU. For this purpose compromise between the high-quality output and the rendering speed has be chosen to meet desired requirements on satisfactory level. In the end a stand-alone testing application is to be created and compared with similar systems. This testing application should base its GUI on the .NET framework, both to provide easy-to-use interface for the rendering library and to give a proof of concept that managed GUI environment can be effectively used even in high-performance, time-critical applications.

1.9 Structure of the text

The thesis is divided into several chapters each one being shortly described in this section.

Chapter 1 provides motivation for visualization of human skulls and specifies the part of it covered by this thesis.

Chapter 2 is a brief overview of methods concerning volume and isosurface rendering as they were chronologically presented in the history of volume visualization. Also specific advantages and disadvantages for our use are discussed here.

Chapter 3 contains theoretical background of all methods which were used in our implementation. This part suggests some not yet implemented techniques, which can be considered worthy of interest and can be implemented in the future work as well.

Chapter 4 is all about implementation details of previously described methods used in testing application and partially serves as a programming documentation. This part contains some in-depth information about the core components in the renderer and also deals with additional optimization steps. *CUDA* architecture is described here too.

Chapter 5 is the part where results are presented, compared with similar rendering systems and discussed in the terms of quality and performance.

Chapter 6 provides conclusion, overall summary and reviews the achieved goals together with future work discussion.

Appendix A is formulated as a basic user guide for the presented testing application.

Appendix B is a list of files and directories found on enclosed DVD.

Chapter 2

Previous work

Volume rendering as the computer graphics term has been used many years since the first attempts to visualize 3D datasets. Constant development in this field during those years brought several approaches how to accelerate rendering process and also improve overall visual quality. This chapter is a summary of the most important methods presented during the historical evolution of volume rendering and since we focus mainly on isosurface rendering, this part has been also dedicated to analysis related to previous research on rendering of isosurfaces.

2.1 Volume visualization

Hardware acceleration is a major key element in the volume rendering so many researches tried to take advantage of the hardware support as much as possible. Nowadays this trend continues and most likely will not change in the future as the technology and all its usage possibilities with new graphics cards grow every year.

2.1.1 Texture based slicing

One of the first systems exploiting hardware support were based on texture slicing. In those days only features like hardware 3D texturing were commercially available through OpenGL extensions which were then supported by workstations from SGI. Those systems re-sample volume data, represented as a stack of 2D textures or as a 3D texture mapped onto the polygons placed into the volume. This set of polygons is called proxy geometry and

can be can be either aligned with the data, aligned orthogonal to the viewing direction or aligned in other configurations (such as spherical shells).

Early work about this technique was published by Cullip and Neumann [7] where they discussed the necessary sampling schemes as well as axis aligned or view-oriented sampling planes. Later on the method was further enhanced by Cabral et al. [4] and they were the first to introduce an implementation of volume reconstruction which required special hardware with 3D texture acceleration capabilities. They managed to present simple way for interactive volume rendering and this texture based slicing technique soon became very popular.

Additional works improving this approach were subsequently released. Method for direct volume rendering with shading via three-dimensional textures was published by Van Gelder and Kim [10], or new technique, which enables shading as well as classification of the interpolated data was published by Meißner et al. [21]. This technique came up with accurate lighting for a one directional light source, semi-transparent classification and correct blending, with all of these algorithms performed within the graphics pipeline.

Another work was presented by Rezk-Salama et al. [32] in which they proposed new rendering techniques that significantly improve both performance and image quality of the 2D-texture based approach and showed how multitexturing capabilities of modern consumer PC graphic boards are exploited to enable interactive high quality volume visualization on low-cost hardware. Furthermore they demonstrated how multi-stage rasterization hardware can be used to efficiently render shaded isosurfaces and to compute diffuse illumination for semi-transparent volume rendering at interactive frame rates.

Method which brought great improvement in the speed and quality of texture slicing was then presented by Engel et al. [9] as they introduced pre-integrated volume rendering, which achieves the image quality of the best post-shading approaches with far less slices. They implemented various specific algorithms using the novel technique like direct volume rendering, volume shading and arbitrary number of isosurfaces or mixed mode rendering.

All these works proved worthy and so many systems and applications were created using texture slicing for volume visualization method. It was very easy because it required only hardware with 3D texture support. However as the method is very simple it brings some significant drawbacks considering the performance issue because many pixel blending operations, texture fetches or lighting calculations and others doesn't have any impact in

the final result. Because of that it is convenient to use some common acceleration techniques which will minimize all of these unnecessary operations. Early ray termination and empty space skipping are suggested acceleration methods in volume rendering (see [8, 17, 38]). Unfortunately when using texture based slicing for rendering, it is quite difficult to implement any of these improved algorithms.

2.1.2 Volume ray-casting on graphics hardware

After the arrival of DirectX 9.0 in 2002, Shader Model 2.0 was introduced as a new feature of Direct3D API. Soon both major graphics chip manufacturers ATI and NVIDIA presented their DirectX 9 compliant cards where ATI Radeon 9700 was released in 2002 and NVIDIA GeForce FX 5800 in early 2003. This was the first time when GPU accelerated volume ray-casting could be implemented on consumer hardware as it was then possible to utilize better programmability in the fragment stage of the graphics pipeline. One of the most significant work discussing this technology and ray-tracing on programmable graphics hardware was done by Purcell et al. [29]. They evaluated latest trends in programmability of the graphics pipeline and explained how ray tracing can be mapped to graphics hardware.

Multi-pass technique

Later on several implementations from various groups were published, using multi-pass techniques and Shader Model 2.0 API for their algorithms. Work by Krüger and Westermann et al. [16] can be considered as innovative in a way of the integration of acceleration techniques like early ray termination and empty-space skipping into texture based volume rendering on graphical processing units. Their GPU implementation achieved performance gains up to a factor of 3 for typical renditions of volumetric data sets comparing to the standard slice-based volume rendering. Additionally work by Röettger et al. [30] presented their implementation of pre-integration technique, volumetric clipping, and advanced lighting together with space leaping and early ray termination in GPU ray-caster. Another work which was the first implementation of a volume ray casting algorithm for tetrahedral meshes was then published by Weiler et al. [41], where they also guaranteed accurate ray integration by applying pre-integrated volume rendering.

Single-pass technique

Until the next big evolution in programmability of fragment graphics pipeline there was no way to implement volume ray-casting algorithm in a single pass. This all changed in the late 2004 when new version of DirectX 9.0c

was introduced. It defined new Shader Model 3.0, which was highly anticipated as it brought significant improvements over the last version. It greatly increased maximum count of program instruction, vertex texture fetches, floating point textures and frame buffers dynamic. But the most relevant feature in pixel shader was dynamic flow control with loops and branches. The first graphics cards which fully supported DirectX Pixel Shader 3.0 API were from NVIDIA GeForce 6 family. They performed very well comparing to previous generation when GeForce 6800 Ultra was 2 to 2.5 times faster than NVIDIA's previous top-line product (the GeForce FX 5950 Ultra).

One of the first presentations that showed implementation on latest hardware was presented by NVIDIA in their Shader Model 3.0 developer presentation where they included short example of single-pass volume rendering running on NVIDIA GeForce 6800 [25]. Using this new graphic card, Stegmaier et al. [36] presented flexible framework for GPU-based volume rendering based on single-pass volume ray-casting approach. They didn't used DirectX Pixel Shader 3.0 API but rather its OpenGL equivalent, NVIDIA's `NV_fragment_program2` extension [24]. Framework which they presented was highly flexible and showed some high-quality standard and non-standard volume rendering techniques including translucency, transparent isosurfaces, refraction, and reflection or self-shadowing. They also compared performance results of slice-based volume renderer with single-pass volume ray-casting solution but it was approximately only half as fast as the reference implementation. It was due to the fact that used graphics processor didn't performed very well with dynamic flow control and branches or loops as it was only the first generation with Shader Mode 3.0 support and further improvement was expected in the next generation. They also mentioned that implementing accelerating technique for early ray termination in most cases gained only very little performance benefit because of the mentioned dynamic flow control instruction. Short time after this work, Klein et al. [14] published another work on accelerating GPU based ray-casting by empty-space–leaping technique. Simply by exploiting frame-to-frame coherence they experienced a speed-up of more than a factor of two comparing to the basic GPU ray-casting solution. Additionally they demonstrated selective super-sampling based antialiasing because the achieved speed-up allowed further image quality improvement.

2.2 Isosurface rendering

This work is mainly related to visualizing human skulls and so we also made specific research concerning isosurface rendering. The method called *March-*

ing Cube algorithm published by Lorensen and Cline [19] was the first technique widely used for visualizing isosurfaces. It was relatively easy and required only hardware capable of triangle rendering. The reconstruction of the surface was approximated by triangle mesh and therefore it produced large amounts of triangles even for moderately complex datasets. Several improvements of the basic method were described and algorithms like *Regularised Marching Tetrahedra* (see Treece et al. [37]) were published. However these extraction methods don't provide good solution for interactive work as the triangle mesh must be regenerated every time the isovalue has been changed. Additional visual effects like global illumination or shadows are also very difficult to apply onto the generated isosurface.

As discussed in previous sections about volume visualization, several methods can be modified to perform volume rendering together with isosurface rendering without major and difficult upgrades. Westermann and Ertl [42] showed ways to use 3D textures for the rendering of lighted and shaded isosurfaces in real-time without extracting any polygonal representation. Engel et al. [9] improved their texture based method for isosurface rendering and discussed specific problems in implementation of gradient interpolation and lighting for isosurfaces. They also showed an intermixing of semi-transparent volumes and isosurfaces when performed by a multi-pass approach that first renders a slice with a pre-integrated dependent texture and then renders the slice again with an isosurface dependent texture. Advanced multi-pass isosurface ray-caster was also included in the work by Krüger and Westermann et al. [16] where they just modified shader program for the passes from 3 to N with first two unchanged. They showed how both opaque and transparent can be implemented and pointed out the fact that gradient reconstruction and illumination has to be always performed even if no surface has been hit, because fragment shaders have not yet supported conditional execution of expressions. Also their acceleration techniques proves worthy for standard datasets and opaque isosurfaces. Another implementation done by Sigg et al. [35] employed tri-cubic filtering of a scalar volume for real-time high-quality reconstruction of an isosurface, its gradients, and second order derivatives. They presented that with their method an advanced shading such as high-quality reflection mapping, solid texture filtering, and non-photorealistic effects based on implicit surface curvature are possible in real-time. As mentioned on previous part, next very interesting techniques for isosurface rendering were showed in the framework by Stegmaier et al. [36]. Besides common transparent and opaque isosurfaces they included shaders for isosurface scattering, self shadowing, sphere-mapping or incorporated additional volume clipping. Until recently this way of single-pass SM 3.0 method for isosurface rendering was consid-

ered the best solution.

2.3 High-quality techniques

In most applications where the isosurface visualization is key element, methods with the best possible output quality are employed. All previously mentioned methods have one attribute in common and it's the way they compute intersections with the isosurface. They are based on approximation of the surface-ray intersect point by linear interpolation and are highly dependent on the chosen sampling precision. But for the exact intersection this is not enough and therefore works like Marmitt et al. [20] presented alternative solution for accurate ray intersections which on the other hand are suitable for interactive applications because of the higher performance compared to exact algebraic intersections algorithms. Their approach showed great suitability for a SIMD implementation as well, so in our work we have chosen this algorithm for visualizing high-quality isosurfaces.

2.4 Recent evolution in GPU architecture

With the arrival of the 8th generation of graphic cards from NVIDIA released in November 2006, NVIDIA's first unified shader Direct3D 10 Shader Model 4.0 / OpenGL 2.1 architecture was presented. It was the major shift from separate concept of pixel and vertex shader to the more common model where all shader types have almost the same capabilities as the instruction set is consistent across all shader types. Thanks to the array of floating processors called *Stream Processors* every shader can thus perform more universal set of tasks. Additionally with the G8X GPUs, NVIDIA released their *Compute Unified Device Architecture* know as CUDA. It is designed as parallel programming model and software environment and allows programmer to write highly parallel programs which will execute directly on GPU's stream processors. These programs can be written in C language which is very convenient.

After the release of CUDA Toolkit which is a C language development environment for CUDA-enabled GPUs, there were several attempts on implementing generic ray-tracer (see Rollins [31]). Using the first versions of CUDA Toolkit to develop volumetric ray-waster was on the other hand quite difficult due to the lack of 3D texture support that caused much lower performance comparing to previous SM 3.0 implementations. This changed when

CUDA toolkit 2.0 beta was released and when new SDK which came with a simple example of volume renderer using newly added 3D texture support.

In our work we try to continue to follow this path and therefore we developed purely CUDA based full-scale ray-caster for isosurface visualization and volume rendering. All previous methods proved as a good solution too, but so far we have always seen great progress in the GPU industry so with that in mind we anticipate all brand new hardware in near future which will be even more suitable for general purpose computing. Also recently announced info about next version of DirectX included speculations about its new shader technology which should re-position GPUs as general-purpose parallel processors.

Chapter 3

Theoretical Background

3.1 High-quality isosurfaces

3.1.1 Overview

One of the most important factor for some specific users working with the volume visualization software is to have an ability to render as accurate isosurfaces as possible. Common reason could be for example that every small detail is very important and thus high-quality output is necessary. Since we are interested only in the real-time algorithms we do not consider methods where computation time doesn't play an important role. Also we will focus only in the area of ray casting-based algorithms.

Standard ray-caster based isosurface renderer can find surface intersections just by marching along the ray and sampling values of density which are then compared to the preselected isovalue. Very important task is to compute exact or just the most possible accurate position of that intersection and also try not to miss any.

Simple and common way to achieve accurate results is to increase ray marching sampling rate. By doing this we are able to detect more isosurface intersections but on the other hand we loose performance advantage since sampling rate has direct impact on it. But still this is not the ultimate solution as the behavior of the scalar function could have even higher frequency so some intersections will be missed during the ray marching process no matter how high sampling frequency is. Although many algorithms work this way quite well our aim to is achieve correct results by using method which should be fast and accurate at the same time.

Assuming that source data come from CT scanning and so discrete density volume information has been stored equally into the vertices defined by voxel structure of the dataset. Outside the vertices density is generally calculated as trilinear interpolation from the values in the voxel's reference vertices. This implies that the function describing the density along any ray passing through the voxel is a cubic polynomial. The whole problem of density calculation at any point of the volume has been additionally described in the following section.

3.1.2 Density calculation

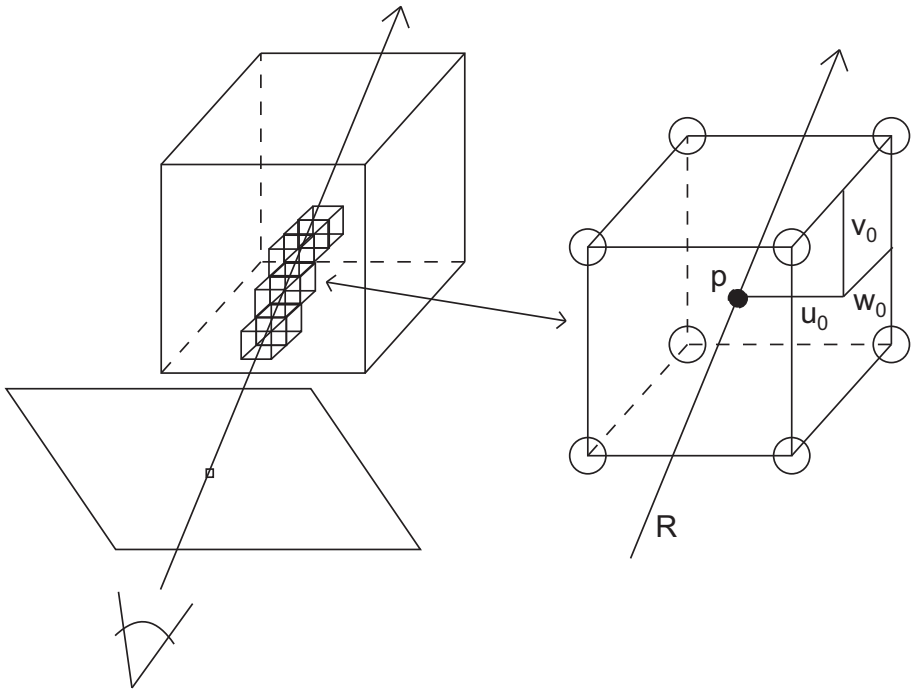


Figure 3.1: Ray cast into the grid of voxels.

Let's have point p with the coordinates (u, v, w) where $u, v, w \in \langle 0, 1 \rangle$ are located in the volume space defined by eight neighboring vertices. Then using

$$\begin{aligned} u_0 &= 1 - u,\ u_1 = u, \\ v_0 &= 1 - v,\ v_1 = v, \\ w_0 &= 1 - w,\ w_1 = w, \end{aligned}$$

density $\rho(p)$ at this point can now be calculated from trilinearly interpolated values ρ_{ijk} at the bounding vertices as

$$\rho(u,v,w) = \sum_{i,j,k\in\{0,1\}} u_i v_j w_k \rho_{ijk}.$$

General case has the point $p(p_x, p_y, p_z)$ located in the voxel whose vertices have coordinates (x_{ij}, y_{ij}, z_{ij}) $(i,j \in \{0,1\})$ and therefore we must first perform transformation into unit coordinate system and only then previous form can be applied. According to the exact notation used in Marmitt et al. [20] we get

$$p_0 = (u_0^p, v_0^p, w_0^p) = (\frac{x_1 - p_x}{x_1 - x_0}, \ldots),$$

$$p_1 = (u_1^p, v_1^p, w_1^p) = (\frac{p_x - x_0}{x_1 - x_0}, \ldots)$$

and final form as

$$\rho(p_x, p_y, p_z) = \sum_{i,j,k\in\{0,1\}} u_i^p v_j^p w_k^p \rho_{ijk}.$$

But our situation also requires calculating density along the ray and hence using parametric ray representation $R(t) = a + bt$ is the most suitable way to do it easily. In that context a is the origin point for the casted ray and b is its direction vector.

When this representation is used with the transformed trilinear interpolation form defined above then density along that ray $\rho(R(t))$ can be calculated as

$$\rho(t) = \sum_{i,j,k\in\{0,1\}} (u_i^a + tu_i^b)(v_j^a + tv_j^b)(w_k^a + tw_k^b).$$

After additional rewrite by multiplying to the polynomial form we get

$$\rho(t) = At^3 + Bt^2 + Ct + D$$

which refers to cubic polynomial function as mentioned previously. Individual coefficients are thus defined as

$$\begin{aligned}
A &= \sum_{i,j,k\in\{0,1\}} u_i^b v_j^b w_k^b \rho_{ijk} \\
B &= \sum_{i,j,k\in\{0,1\}} (u_i^a v_j^b w_k^b + u_i^b v_j^a w_k^b + u_i^b v_j^b w_k^a)\rho_{ijk} \\
C &= \sum_{i,j,k\in\{0,1\}} (u_i^b v_j^a w_k^a + u_i^a v_j^b w_k^a + u_i^a v_j^a w_k^b)\rho_{ijk} \\
D &= \sum_{i,j,k\in\{0,1\}} u_i^a v_j^a w_k^a \rho_{ijk}.
\end{aligned}$$

When considering the initial task to find the intersections of the ray with the isosurface defined by $isovalue := \rho_{iso}$ then problem can be accomplished by finding roots of the function $f(t) = \rho(t) - \rho_{iso}$ in every voxel intersected by the ray. Next few sections are dedicated to specific methods how to compute those or approximate roots.

3.1.3 Analytic method

Simple and somehow naive method is to compute the roots directly by solving the cubic equation. Analytic method was historically published first by G. Cardano [3] in the 16th century. Since then various implementations based on his work have been presented (e.g. see [33, 18]). But according to the research done by Herbison [12] most algorithms for solving cubic equations have been proposed with aims of elegance, generality or simplicity rather than error minimization or overflow avoidance. These interesting facts can be found in his conclusion about operation counts of the best combination of stabilized algorithms for non-special cases summarized in the table 3.1. But when considering problems related to the fact that some implementation computes exclusively with single precision floating point numbers so cumulated error can have even greater impact on final results, then this method cannot be recommended. Therefore in the next sections some other non-analytic methods based on iterative approach are introduced and their characteristics described and compared.

	Additions and Subtractions	Multiplications and Divisions	Functions e.g. Root, Sine	Tests
Best	8	10	2	18
Worst	13	15	3	19

Table 3.1: Operation counts for a best combination of stabilized algorithms.

3.1.4 Approximation methods

Given entry and exit points t_{in}, t_{out} on the ray $R(t)$ passing through the voxel we would like to approximate the location of the roots of the function $f(t) = \rho(t) - \rho_{iso}$ inside given voxel. The very simple way to do it at a low price is to test function values at the entry and exit points and if they differ in sign, which implies that the function has root inside the interval. Then

compute it by linear interpolation from those bounding points. Example in pseudo code taken from Marmitt et al. [20] has been listed below as algorithm 1.

Algorithm 1 Pseudo code for linear interpolating the intersection position.

```
// linear interpolation
ρ_in := ρ(R(t_in)); ρ_out := ρ(R(t_out))
if sign(ρ_in − ρ_iso) = sign(ρ_out − ρ_iso) then
    return NO_HIT
end if
return t_hit := t_in + (t_out − t_in) (ρ_iso − ρ_in) / (ρ_out − ρ_in)
```

This simple approach assumes that density function along the ray between the boundary points has linear progress. But as we previously showed this requirement is not always satisfied. The only case when the algorithm works can be seen in the first situation on the Figure 3.2 in which the density function has only one intersection with the isosurface.

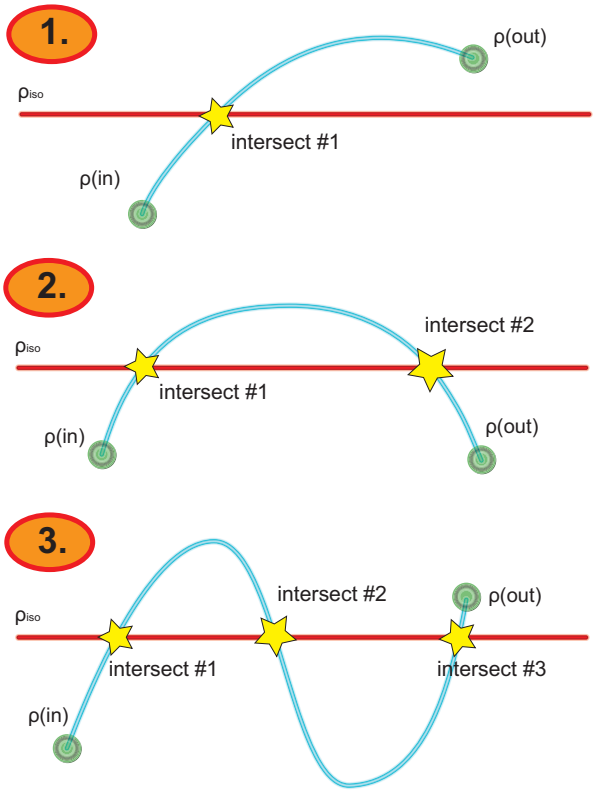


Figure 3.2: Different cases of ray intersections with isosurface.

When considering situation 1, then simple linear interpolation can be improved by finding the intersection using the iterative method. This was suggested by Neubauer [23] (see algorithm 2) because it gives great improvement in the output quality as the intersections is computed with precision very similar to the results achieved by analytic method. Additionally numerical stability, which is here much better plays great role when computing with single-precision floating point numbers. Although total time complexity of the algorithm has increased, since the linear interpolation step has to be performed several times (2-3 iterations are often sufficient) this method can be still considered simple and useful solution for increasing quality in isosurface rendering.

Algorithm 2 Pseudo code for Neubauer's algorithm using repeated linear interpolation.

```
// Neubauer: repeated linear interpolation
t_0 := t_in; t_1 := t_out
ρ_0 := ρ(R(t_0)); ρ_1 := ρ(R(t_1))
if sign(ρ_0 − ρ_iso) = sign(ρ_1 − ρ_iso) then
  return NO_HIT
end if
for i = 1..N do
  t := t_0 + (t_1 − t_0) (ρ_iso − ρ_0)/(ρ_1 − ρ_0)
  if sign(ρ(R(t)) − ρ_iso) = sign(ρ_0 − ρ_iso) then
    t_0 := t; ρ_0 = ρ(R(t))
  else
    t_1 := t; ρ_1 = ρ(R(t))
  end if
end for
return t_hit := t_0 + (t_1 − t_0) (ρ_iso − ρ_0)/(ρ_1 − ρ_0)
```

Let's analyze another two possible situations shown on the Figure 3.2. Situation 2 shows density function $\rho(R(t))$ having two intersections with the line defined by value ρ_{iso}. When test where entry and exit values of the function are compared to the *isovalue* is used, then those intersections will be incorrectly missed. Although it is true that $sign(\rho_0 - \rho_{iso}) = sign(\rho_1 - \rho_{iso})$, density function reaches its extrema inside the interval $[t_{in}, t_{out}]$ and therefore two of its intersections are incorrectly missed. The last situation 3 shows the very special case when there are all tree intersection inside the interval $[t_{in}, t_{out}]$. In this case the test performed by the algorithm will be successful. But since we are interested in finding the first intersection on the ray and the returned result can be in fact the third intersection then this is

another example when the algorithm can not be considered satisfactory for accurate isosurface rendering.

3.1.5 Advanced iterative root finding

New Algorithm 3 introduced and presented in Marmitt at al. [20] can be considered as improved version of Neubauer's method. The main advantage of this algorithm is that while still using repeated linear interpolation as root finding element, it is also capable of handling situations 2 and 3 showed in Figure 3.2. Thus it surpasses the previous approximative algorithms in terms of completeness and correctness, while retaining their speed. Also its accuracy is at least comparable to the approach and due to better numerical stability can be in practice even better. Therefore it was chosen for our application as a reliable high-quality approximated isosurface intersection test.

The task of this new algorithm is to find the first root of the function f defined as

$$f(t) = \rho(t) - \rho_{iso}.$$

Notice that the failure of previous algorithm was caused by the fact that it can only operate on intervals where the function has monotonic progress. By monotonic we mean that it has to be *nondecreasing* or *nonincreasing*. A function $f(x)$ is said to be nondecreasing on an interval I if $f(x) \leq f(y)$ for all $x \leq y$, where $x, y \in I$. Conversely, a function $f(x)$ is said to be nonincreasing on an interval I if $f(x) \geq f(y)$ for all $x \leq y$ with $x, y \in I$. On the other hand the function is monotonic if its first derivative (which need not be continuous) does not change sign. Therefore monotonic progress can be guaranteed only for intervals defined by points where the function reaches its extrema.

In our case the task can be solved easily because cubic polynomial function may have local extrema only in two points at most. This is equivalent to finding $t \in I$ where $f'(t) = 0$ which can be accomplished by finding the roots of the function

$$f'(t) = 3At^2 + 2Bt + C,$$

thus solving simple quadratic equation.

Let's analyze situation shown on Figure 3.3. We have the function with both its extrema e_1, e_2 inside interval $[t_{in}, t_{out}]$, which divide it into three subintervals I_1, I_2 and I_3. As it can be seen the test for the first interval will fail because no intersection occurs. Then we advance to the second interval

Algorithm 3 Pseudo code for intersection algorithm introduced by Marmitt et al.

```
// Marmitt: Extrema finding with repeated linear interpolation
t_0 := t_in; t_1 := t_out; f_0 = f(t_0); f_1 = (t_1)
// Find extrema by looking at f'(t) = 3At^2 + 2Bt + C
if f' has real roots then
  e_0 = smaller root of f'
  if e_0 ∈ [t_0, t_1] then
    if sign(f(e_0)) = sign(f_0) then
      // Advance the ray to the second segment
      t_0 := e_0; f_0 := f(e_0)
    else
      t_1 := e_0; f_1 := f(e_0)
    end if
  end if
  e_1 = second root of f'
  if e_1 ∈ [t_0, t_1] then
    if sign(f(e_1)) = sign(f_0) then
      // Advance the ray to the third segment
      t_0 := e_1; f_0 := f(e_1)
    else
      t_1 := e_1; f_1 := f(e_1)
    end if
  end if
end if
if sign(f_0) = sign(f_1) then
  return NO_HIT
end if
// now, know we have got a root in t_0, t_1
// find it via repeated linear interpolation
for i = 1..N do
  t := t_0 + (t_1 - t_0) (-f_0)/(f_1 - f_0)
  if sign(f(R(t))) = sign(f_0) then
    t_0 := t; f_0 = f(R(t))
  else
    t_1 := t; f_1 = f(R(t))
  end if
end for
return t_hit := t_0 + (t_1 - t_0) (-f_0)/(f_1 - f_0)
```

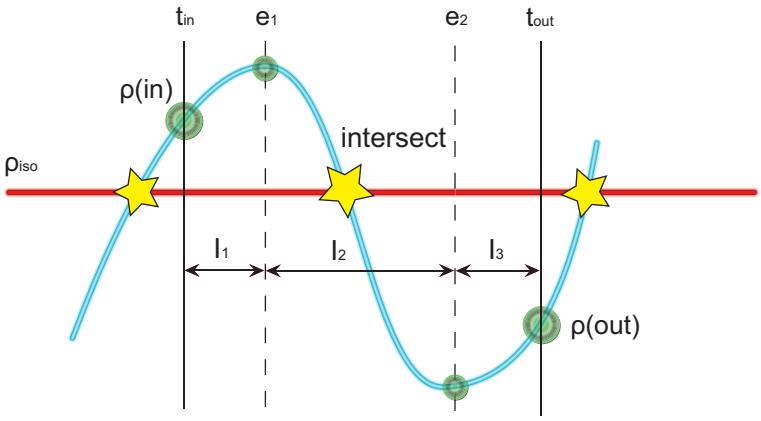


Figure 3.3: Original interval divided into three subintervals where monotonic progress for the function is guaranteed.

where the test will pass and intersection itself will be found by repeated linear interpolation as in previous algorithms.

As shown above this algorithm can be considered an alternative way to finding intersection of the ray with the isosurface in specific voxel delivering sufficiently accurate results with performance much higher than direct algebraic solution. It is perfectly suited for applications which require higher quality output. In our application we used both approaches by implementing this algorithm for high-quality mode and the simple linear interpolation algorithm for much common usage when the accurate isosurfaces are not top priority.

3.2 Voxel traversal

The previous section discussed specific algorithms for computing accurate isosurface intersections, and this section deals with voxel traversal. Because the computations are done separately in each voxel, we need to find all these voxels that are intersected by the ray and also compute points where the ray enters and exits specific voxel. For this purpose we employed algorithm which is similar to three-dimensional DDA line algorithm. It consists of two parts: first is the initialization and the second is traversal loop itself. Input parameters are start position of the ray with its direction vector ($R(t) = A + Dt$) and the output parameters are entry and exit positions $tIn, tOut$ on that ray for every intersected voxel.

The first part serves for initialization of all variables needed for subse-

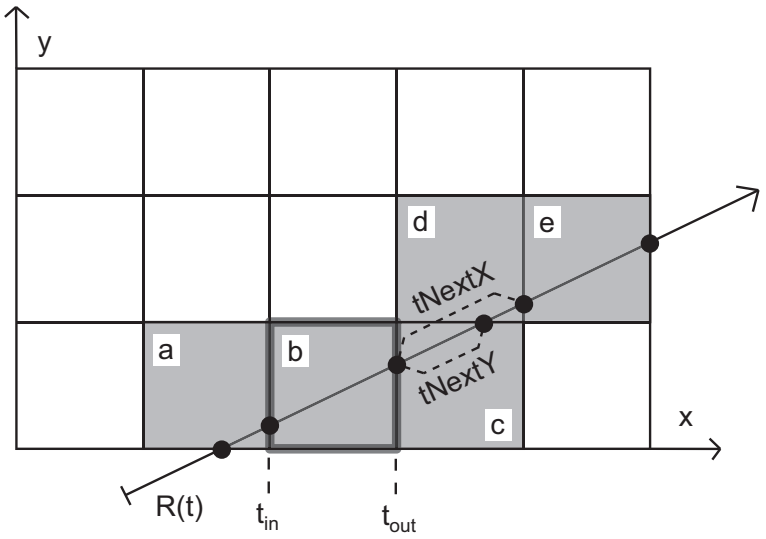


Figure 3.4: Scheme for voxel traversal algorithm.

quent successful traversal. First of all we determine point where the ray enters the volume (tIn) and also the position of entry voxel ($voxX, voxYvoxZ$). Then we compute the position where the ray crosses the first boundary between neighbor voxel in X axis and store that parameter as variable $tNextX$. We do the same computation also for another two variables. As the ray can move from voxel to voxel in each of three axis we have to compute increments of t which are needed for transition to next voxel in the all three directions. We store these values in variables $tIncX, tIncY$ and $tIncZ$.

After the initialization we approach to the traversal part which is now very simple. In each step we move to the next voxel in the direction which has the lowest value of $tNext$ parameter. This parameter also specifies new exit point $tOut$. Then we update other two parameters and perform test where is a voxel for the next step, since the ray may have left the volume.

Pseudo code for basic version of the main loop for voxel traversal is listed in Algorithm 4. Increments used for moving to the next voxel have to be set dynamically for general case because they can be positive or negative, depending on ray direction.

Algorithm 4 Pseudo code for the main loop of voxel traversal

```
outside := false
repeat
  if tNextX < tNextY and tNextX < tNextZ then
    voxX := voxX + 1
    tOut := tNextX
    tNextY -= tNextX; tNextZ -= tNextX; tNextX := tIncX
    if voxX = maxX then
      outside := true
    end if
  else if tNextY < tNextX and tNextY < tNextZ then
    voxY := voxY + 1
    tOut := tNextY
    tNextX -= tNextY; tNextZ -= tNextY; tNextY := tIncY
    if voxY = maxY then
      outside := true
    end if
  else
    voxZ := voxZ + 1
    tOut := tNextZ
    tNextX -= tNextZ; tNextY -= tNextZ; tNextZ := tIncZ
    if voxZ = maxZ then
      outside := true
    end if
  end if
  // process current voxel
  tIn := tOut
until outside
```

3.3 Self shadowing

Shadowing is one of the advanced techniques used for better spatial perception of visualized object. There are several methods how to do it with each one of them suitable for different situation. In our case we decided to apply sharp shadows but as we describe it will not be a problem to switch to soft shadowing mode. An example of skull rendered with and without self shadows enabled can be seen on Figure 3.5.

Because our isosurfaces are rendered using ray-casting method, it is very simple to integrate additional self shadowing. Rendering process is then divided into two phases. First phase is the same as used in standard opaque

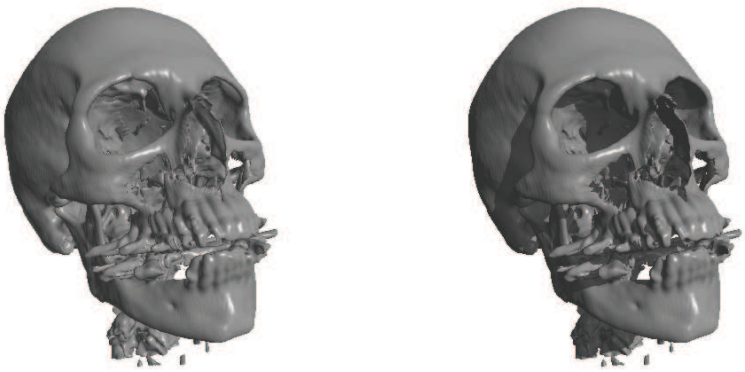


Figure 3.5: Examples of rendering where self shadowing is disabled (left) together with the feature enabled (right).

isosurface rendering. Primary ray is used for finding the first isosurface intersection with the ray and after that the lighting is performed for this point. Second phase stands for ray-cast of shadowing ray. The ray starts at the intersection point and has the direction of the light vector. This ray is also tested for intersection with the isosurface but the procedure is much more simplified as we just want to know if there is any intersection or not. The iterative root finding part described in the previous section is therefore omitted. If no intersection is found, then no change is done, because that point doesn't lie inside a shadow, but if there is, final color is then altered by shadowing factor F_{shadow} like this:

$$C_{dst} = F_{shadow}(F_{amb}C_{amb} + F_{diff}C_{diff})$$

Here C_{dst} stands for output color and $F_{amb}, C_{amb}, F_{diff}, C_{diff}$ are ambient and diffuse colors with their corresponding factors as the parameters of the light.

Two possible situations are showed in Figure 3.6 where intersection P1 was found for the primary ray R1 and then self shadow test was done by secondary ray S1. Since it has not intersected the object, surface color was not changed. On the other hand isosurface point P2 lies in the shadow because secondary ray S2 intersected with the object.

We have applied only one point light source in our application but simple modification can be done to support multiple lights of this sort. But that comes with high performance loss because for every new light, another shad-

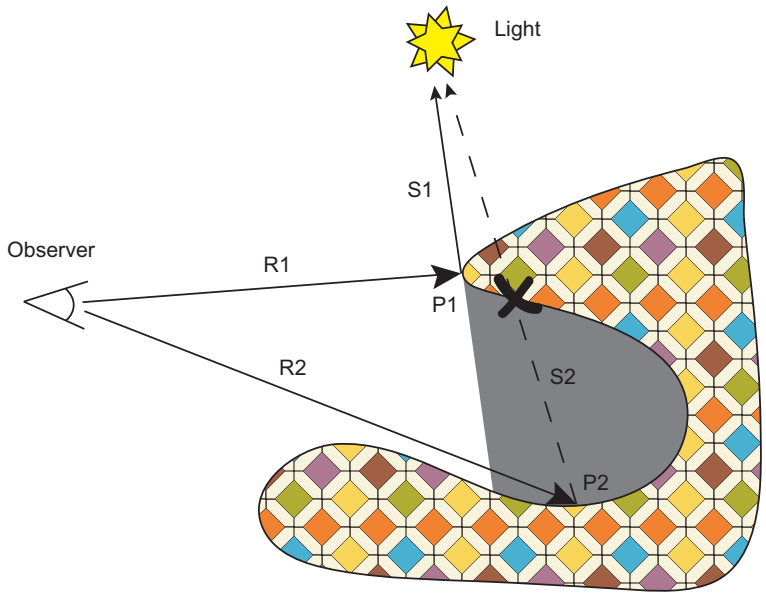


Figure 3.6: Ray-cast with shadows. Point 1 lies outside the shadow. Point 2 lies inside because its secondary shadowing ray intersected with the object.

owing ray must be tested for intersection. Rendering with soft shadows is very similar to this approach, where the light is sampled from several points and therefore this high-quality shadow rendering is much slower.

3.4 Difference visualization

3.4.1 Skull comparison introduction

One of the common and the most time consuming procedure in the anthropological research concerning human skulls is to have them all correctly examined, compared and every important difference or resemblance properly described. This comparison is done all between each other pieces, or if it is needed, then only to the one reference skull. Comparison itself can be realized old-fashion way just by manual examination and measurement. Although only specialized persons perform this difficult task, the results are sometimes inaccurate due to the restrictions, which made many things impossible with this manual method. Such a process has also another disadvantage because repeated daily work can after some time seriously damage certain skulls and is not even possible for many precious historical pieces.

This is where CT is taken into consideration and therefore all skulls are scanned into digital form. By doing this, set of slices are acquired for each skull which can be later used for any other research and original pieces can be safely stored for potential future use. Resultant slices are good start point for checking differences which cannot be revealed in manual examination method. Software which is capable of displaying multiple isosurfaces has to be used in the first place. We designed several modes and techniques that were integrated into our tool to provide the most suitable way for the skull comparison and their additional difference visualization.

3.4.2 Slice view mode and DVR

This mode is similar to standard manual method based on the comparison of individual CT slices but it is more enhanced. Main idea is to bring an option for comparing various slices, not just those acquired from scanning process or axis-aligned slicing. This mode is in fact a simple ray-caster for volumetric data with a clipping plane as an additional feature. Researcher can examine visualized volume data along that plane, which is just like to work with CT slices but it allows much more freedom.

Applied front to back ray-marching requires besides color accumulation also alpha opacity accumulation for the each step of the main loop. Destination color C_{dst} and alpha A_{dst} are updated like this:

$$C_{dst} = C_{dst} + (1 - A_{dst})A_{src}C_{src}$$

$$A_{dst} = A_{dst} + (1 - A_{dst})A_{src}$$

.

C_{src} and A_{src} stands for color and opacity of the new sample point on the ray. In the each step inside the main loop we move further away from the start in the direction of the ray. This step size therefore defines overall rendering quality. When the opacity A_{dst} oversteps preselected limit (eg. a value 0.95) then ray-marching is complete because there is no need to continue as every new accumulated sample will have only minimal effect on the resultant color. This is a common way to integrate early ray termination optimization technique. Also with clipping plane enabled, part of the volume is clipped in advance, so the segment of ray that intersects the volume is much shorter and needs fewer steps to compute the final color.

3.4.3 Overlay mode

When two or more isosurfaces are rendered at once, it is not easy to see and understand their spatial location. This is due to the fact that all closer opaque isosurfaces cover others and some important parts are therefore hidden. Rendering with transparency enabled can be considered useful but on the other hand, many visible transparent isosurfaces create chaotic overall visualization. Because of that we decided to design new rendering mode that has all advantages of the rendering with transparent isosurfaces but also doesn't display too much information at once and only important isosurfaces are rendered.

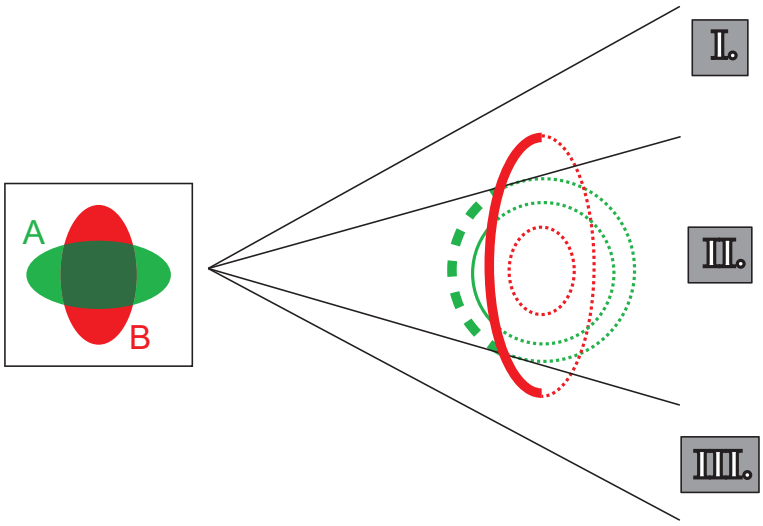


Figure 3.7: Different situations for individual isosurface locations and their rendering style in Overlay mode. Only surfaces drawn with thick line will be rendered.

In this mode we render both opaque and also transparent surfaces so that's why is it called Overlay Mode. Some possible cases which can occur during the rendering are shown in the Figure 3.7. It shows two isosurfaces which has been drawn with different pen styles so different rendering modes can be distinguished. All those parts of the surface that are drawn with full thick line are rendered as opaque and those represented by thick dashed line are rendered as transparent. Full thin line represents skipped surfaces and dotted thin line stands for all those surfaces that are not rendered because they are located behind opaque surface.

In the first sector we can see that all rays intersect only isosurface B with two their intersections. Because of that, only surface corresponding to the first intersection will be rendered and shaded as opaque. The situation in

the third sector is the same. Little bit complicated spatial arrangement can be seen in the sector II. Most rays intersected the surface A twice before the intersections with the surface B. This gives us the first intersection of surfaces A rendered as transparent (the second is skipped) and again the first intersection of surface B rendered as opaque.

3.4.4 Ray cast mode

With overlay mode enabled, two skulls can be compared in very convenient way. But for users who desire more information about the overall arrangement of all isosurfaces we added yet another mode for visualizing local differences. It is done by ray-casting so called *Difference Ray* and displaying additional info about all its intersections with all isosurfaces in the right order. This significantly helps to understand some spatial structure of examined skulls. An example of this difference ray is shown in Figure 3.8.

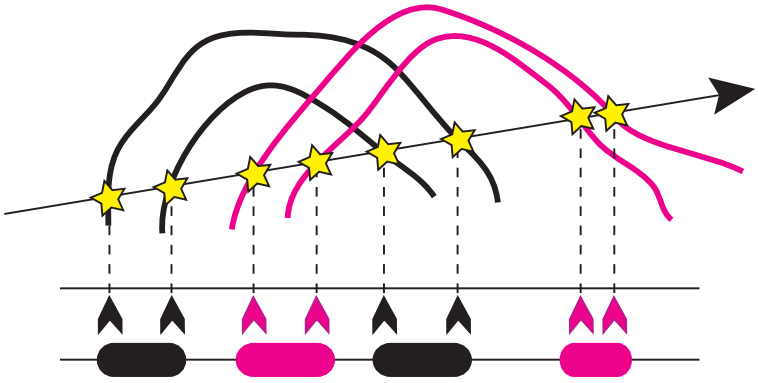


Figure 3.8: When Ray cast Mode is enabled, user can cast *Difference Ray* and see all its intersections by both isosurfaces visualized together with marked ray segments, which represent the volume inside the bone.

Another important information is to display which segments between those intersections represent bone and which just some empty space. We know that scanned input volume data store density information and therefore areas representing bones have higher density than areas of empty space, which have the lowest value equal to density of the air. The problem is to determine if the intersection is the entry point and the ray points into the bone or it is the exit point and the ray points out of the bone. It can be solved by comparing the values of density function in points [t_{in} and t_{out}] (used in previously described iterative root finding algorithm) but we decided to use gradient information as the key to the solution.

The fact is that the gradient vector calculated at the isosurface intersection points in the direction of the greatest rate of increase of the density scalar field. After computing its dot product with the direction vector of Difference Ray we can tell if that intersection is the entry or exit point for the bone. With all intersections marked like this, it is no problem to visualize all bone segments along the ray.

In the situations when the source data are calibrated and this mode enabled, additional local measurement feature is possible. Voxel dimensions, which are therefore very important input parameter, serve as primary form of calibration. Then we can measure distances between individual intersections on the Difference Ray and display the values next to corresponding ray segment. Such a measurement tool is very useful for researchers who besides difference visualization, desire also quick and easy way to perform local measurements.

Chapter 4

Implementation

4.1 CUDA

4.1.1 Scalable parallel programming model

CUDA stands for *Compute Unified Device Architecture* which is parallel programming model and software environment created by NVIDIA. CUDA is designed to overcome the challenge to create application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to multi-core GPUs with widely varying numbers of cores. CUDA brings all these advantages while maintaining a low learning curve for programmers familiar with standard programming languages such as C. (see CUDA Programming Guide [27] for more information)

Since CUDA is developed by NVIDIA, it only works with recent generations of NVIDIA GPUs, first time introduced by G8X series. Later on, all newer NVIDIA cards are compatible with CUDA, including GeForce, Quadro and the Tesla line. As for AMD/ATI, there is another similar GPGPU technology, which is called AMD Stream. It took different approach from that seen in CUDA and since we didn't choose AMD Stream for our implementation, this thesis will not deal with it. For more information about AMD Stream computing refer to [1].

CUDA allows programmer to write functions called *kernels*, which are executed in multi-threaded way. Unlike normal C function, set of threads is assigned for simultaneous execution of the kernel code. Programmer can specify how many of these threads will be created and how these threads will be organized. Simple example of using CUDA kernel function for vector

pair-wise addition operation is written below.

```
__global__ void vecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
int main()
{
// Kernel invocation
vecAdd<<<1, N>>>(A, B, C);
}
```

As mentioned before, the kernel executes multiple threads, which can be distinguished real-time in the code by internal `threadIdx` variable. This three component vector allows defining one-dimensional, two-dimensional or three-dimensional interpretation of the *thread block*. Number threads per block depends on how many shared resources like memory is used by all threads in the block. Currently predefined constant 512 is the maximum allowed number, but it may change in the future. Subsequently thread blocks can form a grid, which can be one-dimensional or two-dimensional. These dimensions and the block position inside the grid can be accessed through blockDim and blockIdx built-in variables. The dimension of the grid is specified by the first parameter of the `<<<...>>>` syntax.

The kernel function itself is declared as standard C function with `__global__` keyword added before its declaration. Calling convention is slightly different from other functions, where kernel parameters are written inside `<<<...>>>` tag. In this example we called the kernel with one-dimensional block with the size N and the grid was degenerated to single thread block.

CUDA software stack (see the Figure 4.1) consists of several host layers, which are all above device part. Lowest layers capable of accessing the device via application programming interface are the *CUDA Driver* and its runtime. There is also layer with two mathematical libraries - CUFFT (Fast Fourier Transform implementation) and CUBLAS (Basic Linear Algebra Subprograms CUDA implementation) above that. Finally an Application layer is the place, where the programmer can access any of those layers to write specific CUDA application.

CUDA memory hierarchy can be described as a set of different memory spaces with different characteristic, which are designated for specific usage in specific situations. Each CUDA thread has its own private *local memory*

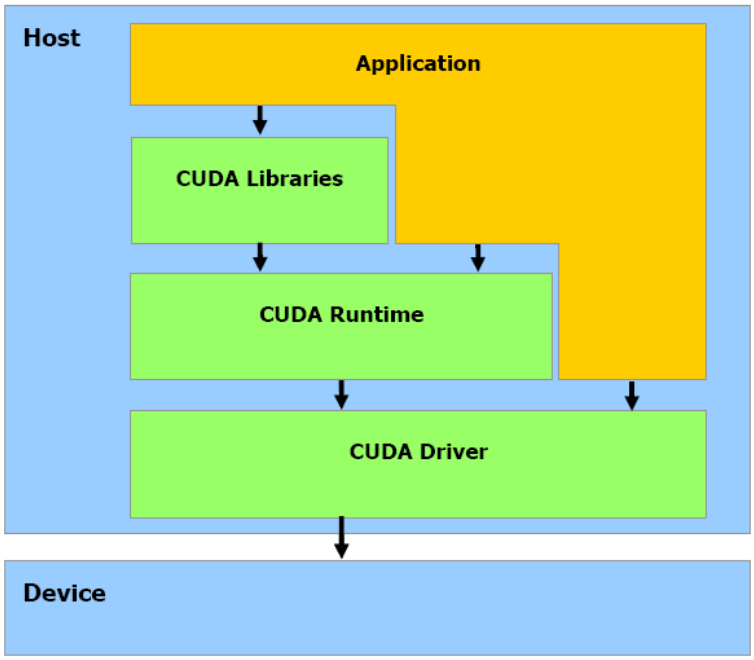


Figure 4.1: Compute Unified Device Architecture Software Stack.

space. *Shared memory* space is shared between threads from the same block and all threads can access *global memory* space. These memory spaces are available only during the thread execution and to them only. On the top of that, there are two additional memory spaces with read-only access available to all threads: the *constant* and *texture memory* spaces. The global, constant and texture memory spaces are persistent across kernel launches by the same application.

4.1.2 A set of SIMT multiprocessors with on-chip shared memory

When CUDA program launches its kernel then individual blocks of threads are scheduled to run on Stream Multiprocessors. Each multiprocessor consists of eight Scalar Processor (SP) cores, two special function units for transcendentals, a multi-threaded instruction unit, and on-chip shared memory. They employ new SIMT (single-instruction, multiple-thread) architecture. Threads are executed on scalar cores independently and are scheduled by multiprocessor in groups of 32 parallel threads called warps. All threads from the same warp start together but can finish independently because

every one can take different code path. Every instruction step, the SIMT unit selects one warp, which is ready to run and executes next instruction with its threads. Maximum efficiency can be then accomplished only when all warp threads take the same code path and no divergent branches occur. On the other hand the executions of threads from different warps are not dependent and have no effect on execution performance.

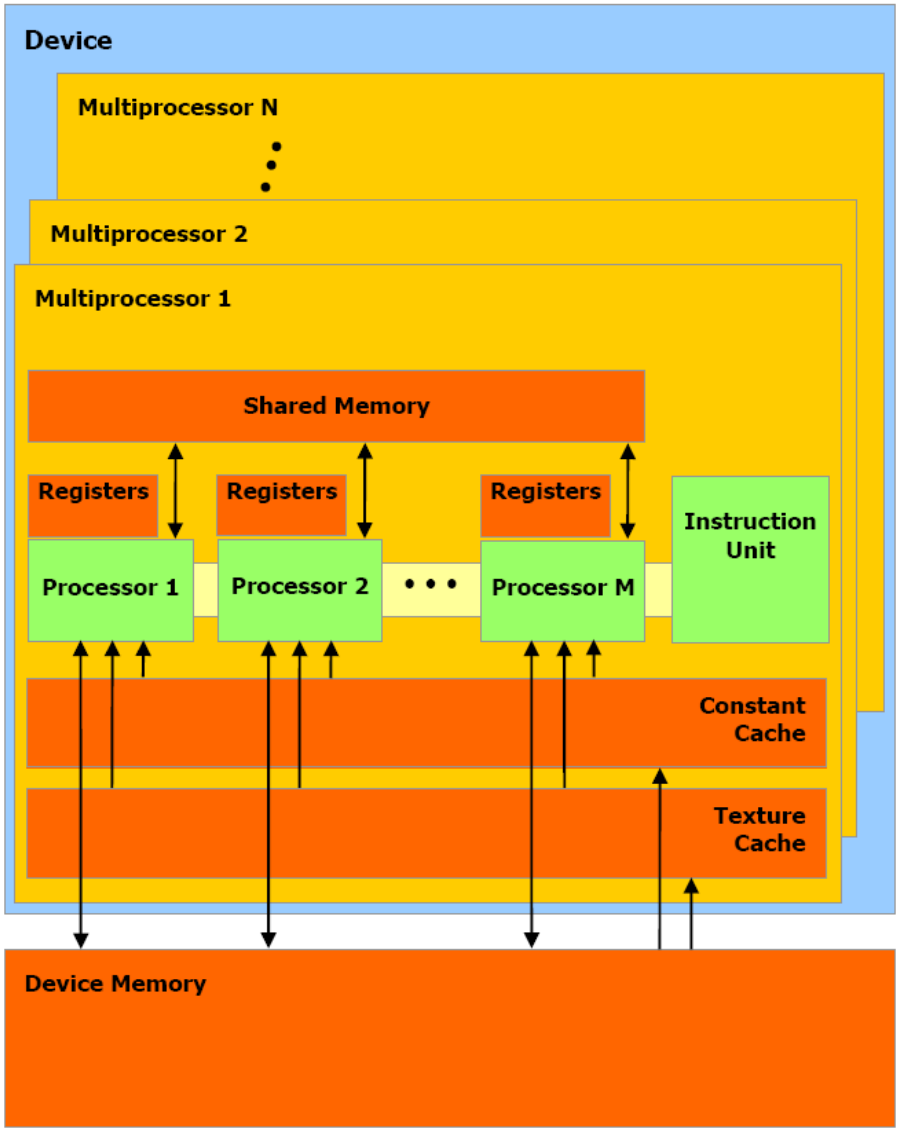


Figure 4.2: A set of SIMT multiprocessors with on-chip shared memory.

SIMT architecture is similar to general SIMD vector architecture but differs in several ways. The main difference is that SIMT instructions reveal

execution and branching behavior of a single thread but SIMD vector organizations expose only their SIMD width to the software. Knowing those advantages of SIMT architecture the programmer can write thread-level parallel code for independent scalar threads, as well as data-parallel code for coordinated threads.

As illustrated in Figure 4.2, each multiprocessor has on-chip memory of the four following types:

- One set of local 32-bit *registers* per processor,
- A parallel data cache or *shared memory* that is shared by all scalar processor cores and is where the shared memory space resides.
- A read-only *constant cache* that is shared by all scalar processor cores and speeds up reads from the constant memory space, which is a read-only region of device memory.
- A read-only *texture cache* that is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region of device memory. Each multiprocessor accesses the texture cache via a texture unit that implements the various addressing modes and data filtering (see [27] for more information).

All these parameters together with kernel complexity and the size of the thread block define maximum count of blocks processed by one multiprocessor because resources like registers and shared memory are limited for use. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.

4.2 CUDA - final thoughts

It is important to understand, that while CUDA brings easy tool for developing multi-threaded algorithms, which take advantage of GPU multiprocessor architecture, also another aspect of this approach has to be mentioned. Most programs written for general-purpose CPU are not suited for SMIT architecture and sophisticated algorithms perform relatively slowly. One of the main reason is the divergent branching problem and low capacity of fast on-chip memory like registers and shader memory. Also with recursion not supported due to the missing stack, many algorithms like some acceleration techniques do not deliver sufficient performance and have to

be omitted form GPU implementation. However some of them can be upgraded and greatly optimized for SMIT architecture, but this challenging operation is very time-consuming even with the help of available CUDA profiling tools. More information about optimization can be found in the section about CUDA specific optimization.

On the other hand, properly written and optimized program greatly outperforms any of its CPU implementations even on the latest high-end multi-core processors. We also assume that further improvements of next generations of NVIDIA graphics cards will bring much better support for CUDA programming and also some new features available in the next CUDA Toolkit. With that in mind, we will continue to experiment with more CUDA-powered implementation of the algorithms used in the area of volume visualization.

4.3 WisS Application

WisS (Volume Visualization Software) is a testing application designated to be a first-approach software solution used for volume visualization of human skulls and their differences. It implements algorithms for isosurface rendering and specific difference methods described in the Chapter 3. Next few sections are dedicated to individual parts of the program and cover description of their implementation details or partially serve as basic programming documentation.

Simple software model of WisS application is shown on Figure 4.3, which is in fact the structure of two main parts: .NET Windows Form Application and CUDA Renderer Library accessible via CULIB API. First part is focused mainly on GUI features and handles user control together with all option setting for internal renderer, which is implemented in the second part as dynamic link library.

4.4 Renderer library

4.4.1 Helper structures and objects

Addition helper structures were chosen for the representation of matrices and vectors in the renderer. Classes `CMatrix4x4` and `CVector4` are implementations of those structures. They were created for handling all matrix and vector operations. As their internal representation was made simple

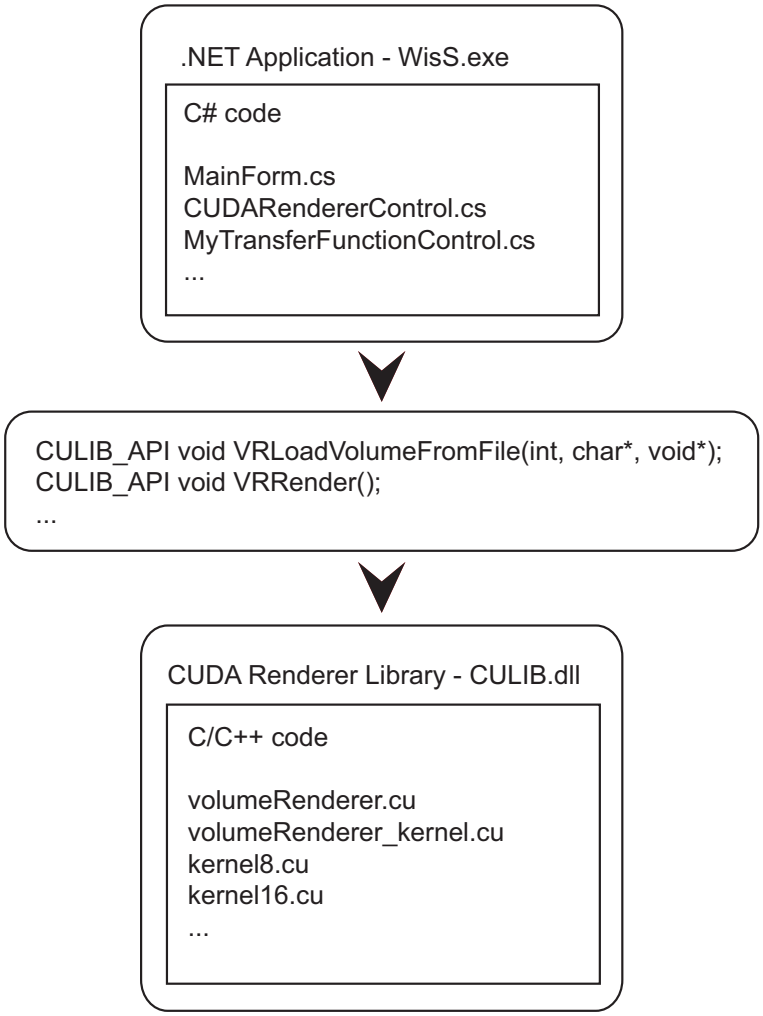


Figure 4.3: Wiss Application software model.

with no complicated member structures, it is very convenient to use them with compatible OpenGL vector and matrix functions or CUDA internal vector types like `float4`.

Console text standard output was chosen for displaying various debugging messages. But the Renderer Library as native Win32 module doesn't come with console after its startup. Therefore it was necessary to invoke console explicitly and redirect there all calls for standard output. This is done at the initialization stage of the Renderer Library by `RedirectIOToConsole` function.

4.4.2 Application programming interface

This library is used by main application for loading the volume data and consequent rendering process. Renderer can be accessed by calling CULIB API functions only. It uses OpenGL for rendering the output image so the main application is responsible for creating, handling and destroying OpenGL rendering context and its window.

The simplest example of how to use the library for rendering a single frame can look like this:

```
...
// initialize rendering engine
VRInit();
// specify output dimensions
VRReshape(512, 512);
// load volume data into memory
VRLoadVolumeFormFile(...);
// render into OpenGL window
VRRender();
// close engine
VRClose();
...
```

At the beginning, the engine must be initialized by calling `VRInit` function. Then `VRReshape` should specify the dimensions of output window. From now on, the renderer is ready and we can load all raw volume data from single file into the memory by calling `VRLoadVolumeFromFile` with desired parameters. Finally the function `VRRender` is self explanatory and serves for rendering the output image into current OpenGL window. After that, we must free all data and close the renderer by calling `VRClose` function.

Next part is an quick summarization of all library API functions divided into groups according to their common usage.

Renderer engine core

```
void VRInit();
void VRClose();
void VRReshape(int x, int y);
void VRRender();
```

```
void VRLoadVolumeFromFile(int volID, char *fileName,
                          SVolData *volData);
```

All functions have been used and described in the previous example so `SVolData` is the only one new structure and contains a definition of volume dataset size, voxel extent and the number of bits per component in the source data:

```
struct SVolData
{
   int3     datasetSize;
   float3   voxExtent;
   int      srcBpc;
};
```

Camera control

```
void  VRRotate(int posX, int posY, float x, float y, float z);
void  VRMove(float x, float y, float z);
float VRGetZPos();
void  VRSetZPos(float z);
```

This library also handles simple camera control so all mouse input is translated into the calls of these functions and the camera can be moved or rotated by specific desired offset. For performing an *arcball* rotation style, application must send mouse cursor position relative to output window as additional parameter in `VRRotate` function.

Isosurface settings

```
void VRSetIsoValue(int volID, float value);
void VRSetIsoSurfaceColor(int volID, float r, float g, float b,
                          float a);
```

Isosurface is defined by specifying its isovalue and RGBA components of a color used in the isosurface visualization. Isovalue must be within the range of $[0, 1]$ where 1 represents the highest possible density in the volume data.

DVR settings

```
int  VRSetTransferFnc(int volID, float* data);
void VRSetDVRSteps(int steps);
void VRSetDVRAccCoef(float coef);
```

These functions are used for setting the count of ray-marching steps and specifying color accumulation coefficient in the Direct Volume Rendering mode and also for setting user-defined transfer function as one-dimensional lookup table for the each volume.

Clip Plane

```
void VRResetClipPlane();
void VRMoveClipPlane(float offset);
void VRRotateClipPlane(float x, float y, float z);
void VRShowClipPlane(int show);
```

Clip plane, which is only used in DVR mode, is controlled by functions allowing movement, rotation and visibility of the clip plane.

Miscellaneous functions

```
void VRSetRenderMode(int mode);
void VRShowBBox(int show);
void VREnableDiffRay(int enable);
void VRShowDiffRay(int show);
void VRSetDiffRayPos(int x, int y);
void VREnableVolume(int volID, int enable);
int  VRGetHistoData(int volID, float *data);
```

The last group includes functions for visibility flags of bounding box, Difference Ray or flags for enabling specific volume for rendering. Rendering mode can be changed too. Also both transfer function controls retrieve all histogram data via the last function from this group.

4.4.3 Textures

Textures are one of the most important data structures used in the volume rendering implementation. At least one three-dimensional texture is required

for storing source volume data and when performing DVR, another texture is needed for storing data sampled from the transfer function.

CUDA supports a subset of the texturing hardware that the GPU uses for graphics to access texture memory, which brings several performance benefits over a case when using global memory. Kernel function can access texture data through CUDA functions for texture fetches. These functions operate with specific type of CUDA object called *texture reference*. It defines which part of texture memory is fetched. Each texture reference must to be bound with region of device memory, called texture, before it can be used by a kernel.

A texture reference has several attributes. The most important attribute specifies its dimensionality. It defines how it is addressed via texture coordinates. For example single texture coordinate is required for addressing one-dimensional array and three coordinates are required to address three-dimensional array. Anther attribute specifies the type of data that is returned when fetching the texture. This can be for example single-precision floating-point four component vector of just one integer scalar value. There are also several run-time texture attributes, which specify whether texture coordinates are normalized or not, the addressing mode and the texture filtering.

CUDA allows to allocate device memory either as linear memory or as *CUDA arrays*. CUDA arrays are opaque memory layouts optimized for texture fetching. Because of that, it common way to use them for storing the volume data in three-dimensional array. CUDA arrays are only readable by kernels through texture fetching and may only be bound to texture references with the same number of packed components. Writing into the array can be performed only in host code and only through the special memory copy functions from CUDA API.

We use these texture references in our implementation:

- `texA8` 8-bit 3D texture for primary volume data,
- `texA16` 16-bit 3D texture for primary volume data,
- `texB8` 8-bit 3D texture for secondary volume data,
- `texB16` 16-bit 3D texture for secondary volume data,
- `transferDVRTex` 2D four-component texture for transfer function for both volumes

Not all of them have are bounded to the CUDA array. Secondary volume texture is used only when comparison of two skulls is required and their differences are visualized. Also 16-bit texture is used only for multi-byte source data so half of allocated device memory can be freed otherwise.

4.4.4 Volume data loading

Loading new volume data into CUDA arrays is performed in several steps by function `VRLoadVolumeFromFile`. Input parameters specify filename, dimensions, voxel size and other information about loaded volume dataset. Loading process is also slightly different for secondary volume, because it its required, that all parameters must be the same as defined by primary volume. Because of that, current version of our application doesn't support primary and secondary volume with different dimensions.

When loading the primary volume, we first reset all global parameters and update them with the new values. If we have already loaded any previous volume data, all this memory is freed (`FreeVolume`) and new one is allocated (`CreateVolume`). After that, main loading process is started. Data are read from source file to CUDA array slice by slice and for each one slice, all its data values are scaled to the interval from 0 to $2^{bpc} - 1$ by the function `CopyRawToFinalSliceData`. Histogram calculation is done there too. Function `SetVolumeSliceData` then performs all host-to-device memory copy operations. In this function we have to perform additional memory transfer from host to page-locked memory, because current beta version of CUDA Toolkit suffers from a bug concerning memory copy operations. This known issue should be fixed in the public release.

4.4.5 Rendering

Rendering into OpenGL window is performed by `VRRender` function. This process consists of several phases, which are responsible for specific tasks used for the rendering of final image.

1. Set view transformation matrix and clipping plane
2. Update required global parameters
3. Map pixel buffer object to CUDA and call kernel function for selected rendering mode
4. Render back faces of bounding box and clip plane if enabled

5. Use PBO to draw textured full screen quad over it
6. Render front faces of bounding box and clip plane if enabled
7. Call kernel function for Difference Ray and display its information if enabled

4.4.6 Global parameters

Due to the fact that single kernel function can only have limited number of parameters (defined by maximum shader memory for parameters, see [27]), most global parameters accessed by kernel are defined as `__constant__` variables. Because of that, relatively fast access is guarantied by CUDA constant memory cache.

All these CUDA constants have their relevant equivalents in host variables. Their values are copied to the CUDA constants at the initialization stage when loading data, like to these parameters

```
float3     c_volSize,
float3     c_voxExtent,
float3     c_volBoxSize,
```

or if it is required, they are updated every time before the rendering process. Three such a self explanatory global parameters are listed below.

```
float3x4   c_invViewMatrix,
float      c_stepSize,
float      c_isoValue[MAX_VOLUMES]
```

Several parameters have their purpose in minor optimization, like to minimize floating point division operations. For example volume dimensions are frequently used in many calculations so we added these two global parameters as inverted values:

```
float3     c_volSizeInv,
float3     c_volBoxSizeInv
```

4.5 Kernel functions

There are four different types of kernels functions designed to perform specific tasks.

- `d_RenderDiff` - basic renderer for transparent isosurfaces and Overlay difference mode
- `d_RenderShadow` - advanced renderer for high quality opaque isosurfaces with self shadowing
- `d_RenderDVR` - simple direct volume renderer for Slice view mode
- `d_DiffRay` - single ray cast for Difference Ray mode

All kernel functions perform some kind of ray casting technique and except the last one, all produce rendered output image. Because of that, output buffer and its dimensions are passed as function parameters to these kernels. Specific output buffer is also used for calculating all Difference Ray information that is filled by the fourth kernel.

4.5.1 Isosurface renderer

Basic isosurface renderer is implemented by `d_RenderDiff8` and `d_RenderDiff16` kernel functions. It supports multiple (two) transparent isosurfaces and besides standard rendering it does also Overlay mode.

Rendering process is divided into these main parts:

1. Initialization
2. Voxel traversal
3. Intersection test
4. Isosurface shading

Initialization part is responsible for several calculations, which have to be done before the start of following voxel traversal loop. First of all, parameters like direction vector for the ray are computed according to internal kernel variables `threadIdx` and `blockIdx`. After that, ray and volume bounding box intersection test is performed. If it is not successful, kernel

function ends and no pixel is rendered. Otherwise we get two ray intersections - volume entry and exit points. Then according to initialization phase for voxel traversal algorithm, described in the Chapter 3, we determine entry voxel position and compute increments of t which are needed for transition to next voxel in the all three directions together with such a closest positions. This part of code contains multiple branches, but as it is just the initialization step, which is executed only once per each ray, it is not necessary to be concerned about this.

Voxel traversal is implemented as the main loop in the rendering algorithm. In the each step, next voxel intersected by the ray is processed and its entry and exit points are computed. Voxel traversal is aborted as soon as the ray leaves the volume or the opacity of the rendered pixel exceeds the threshold value (set to 0.95).

Intersection test is the core of the isosurface rendering. As described in Chapter 3 about iterative root finding, we use Neubauer's algorithm. Isosurface intersect it then calculated via repeated linear interpolation. This is not the accurate test, but it is sufficient enough for this mode. We can switch to better quality test anytime by uncommenting `#define EXACT_TEST`. This test is performed for each enabled volume.

Isosurface shading is the last important part of the isosurface renderer. After surface intersection is found (and its position calculated), function `DoShading` performs its shading. We use Phong reflection model and therefore normal, reflection and light vectors are calculated for the intersection point. Then ambient, diffuse and specular lighting are applied and final shading is done to the pixel. When Overlay mode is enabled, shading is done only for the first intersects for each isosurface and these two colors are blended together.

4.5.2 Renderer with self shadowing

This renderer, which is implemented by `d_RenderDiff8` and `d_RenderDiff16` kernel functions, is different form the previous one in three key elements: renders always high-quality isosurfaces, doesn't support transparency and includes self shadowing.

The process of this renderer is divided into two parts, where the first part has identical structure as the whole rendering process in the previous kernel. However high-quality intersection test algorithm, that is described in Chapter 3, is always used instead of repeated linear interpolation algorithm. Also when the first intersection is found and shaded as opaque, voxel

traversal stops and the second phase is started.

The second phase performs ray-cast of shadow ray and therefore it can use the same algorithm as the first phase. At the beginning, shadow ray direction is calculated and intersection point found by the first phase is set as a start position for the shadow ray. For the each step of voxel traversal loop, isosurface intersection test is reduced. Exact position of the intersection is not needed so only those calculations, which can confirm its presence are performed. Because of that, computation time of the second phase is much faster than in was in the first phase. If the intersection is found, then the color of rendered pixel must be changed according to the shadowing factor.

4.5.3 DVR

Although the first part of this kernel is similar to that used in other kernels, direct volume rendering kernel is completely different from those used for isosurface rendering. No voxel traversal is performed but instead, we use simple ray marching with a constant step. Also after the initial ray-volume bounding box test we do another one for a clipping plane, which is always enabled in this mode.

Main part of this kernel is the ray marching loop. Current position on the ray is updated in the each step by moving along the ray by defined step size. This position is then used to sample scalar density data from the 3D texture and it is done for each enabled volume. These values are then used as coordinates into transfer function texture and retrieved RGB color is then blended with color accumulated in previous steps. Opacity of the pixel is also updated. Main loop ends after the ray marching reached its volume exit point or after successful early ray termination maximum opacity test.

4.5.4 Difference Ray

This kernel is almost the same as for high-quality isosurface renderer, but it actually doesn't produce any output image, but instead, only fills buffer with information about ray intersections. This is later used for the final visualization of Difference Ray info.

Output buffer structure is very simple, when the first two values form a header, followed by a set of trio values for each intersection. Specific order of the first few items in the output buffer looks like this:

0. Intersection count

1. Length of the ray segment from to volume entry to its exit point
2. Position of the first intersection
3. Flag that specifies if the first intersection is the bone's entry or exit point
4. Volume ID of the first intersection
5. Position of the second intersection
6. ...

Gradient vector calculation is performed to determine the flag if the intersection point represents entry or exit for a bone segment on the ray.

4.6 Optimization

Optimization is one of the most important process in the implementation stage. It is the process of modifying a system to make it work more efficiently or use fewer resources. In our case, we aim for maximum rendering speed, measured in frames per seconds (FPS). Because we work with large datasets, we also don't want to waste much more memory than is defined by minimum memory requirements for volume data.

Optimization can occur at a number of levels. The highest level stands for system design and handles better resource management, which can be beneficial for used algorithms. These algorithms will benefit from good quality code, which depends on compiler optimization. The lowest level can be achieved by modifying program's assembly directly and it is considered last possible option for all those cases, when optimization done by compiler is not sufficient. However recent compilers are able to produce highly optimized code and in the most cases the programmer can not achieve much better performance gain with his hand-written assembly optimization.

The process of the optimization is based on finding *bottleneck*, which is critical part of the code that consumes most resources like CPU or GPU time. The most performance improvement is gained after optimization of this part. It is not always possible to achieve this at satisfactory level, because improvement of one component done by optimization, can cause some degradation for other components of the program. This *trade-off* is common factor in optimization phase and programmer must decide which components should be improved and which can be therefore less effective.

We have dealt with optimization of our program since its design, but with that in mind we also tied to avoid any premature optimization as it is considered *"root of all evil"* (D. Knuth [15]).

4.6.1 CUDA specific optimization

Development of CUDA programs requires enormous time, that has to be spent with CUDA specific optimization. SIMT architecture allows us to include various types of optimization and in this paragraph we follow some of advices given by Green [11] in his presentation about CUDA Performance Strategies. After each group of optimization type, we report how it is done in our application and provide other comments.

Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity
- Sometimes it's better to recompute than to cache. GPU spends its transistors on ALUs, not memory.
- Do more computation on the GPU to avoid costly data transfers. Even low parallelism computations can sometimes be faster than transferring back and forth to host.

Our ray-tracing algorithm is parallelized on ray level. For each ray, new thread is created and its corresponding pixel is rendered. These threads work independently and only one data transfer is performed for each rendered frame. We can also confirm that multiple computation operations outperformed single texture read in one special case of `ComputeValue` function. This function was frequently used in the kernel for isosurface renderer, so we used proposed approach (*it's better to recompute*) and successfully optimized this part of code. To maximize arithmetic density we tried some modifications to code flow, like reducing number of branches. But the overall complexity of implemented algorithms doesn't allow many changes for achieving better arithmetic intensity.

Optimize Memory Coherence

- Coalesced vs. Non-coalesced, Local vs. Global
- Optimize for spatial locality in cached texture memory

- In shared memory, avoid high-degree bank conflicts

Volumetric data are stored in multiple 3D textures. Our first implementation used one texture of `uchar4` type and one of `ushort4` for different source data (8bit/16bit). Because of that, we have supported up to four volumes, each one stored in one texture component. However we have only used two of them for skull volumetric data and one for additional color for isosurfaces. This changed in late development phase and single component textures are used instead. This allowed us use to spare some free memory and use it for loading larger primary dataset when the secondary volume is not required. We noticed that performance has decreased a little. It is due to the fact, that when both volumes are used for rendering, two texture fetch operations are performed instead of one. This is a good example of trade-off.

Spatial locality in cached texture memory is partially satisfied, because of the method for voxel traversal and isosurface rendering. Only the first ray, that intersects a voxel, performs full texture fetch of eight values from its vertices, but all others use cached data. But we have keep in mind, that when using two single component textures, "4-byte read" is not satisfied, and it can have some impact on memory read performance.

Take Advantage of Shared Memory

- Hundreds times faster than global memory
- Threads can communicate through shared memory
- Use one/a few threads to load/compute data shared by all threads
- Use it to avoid non-coalesced access. Stage loads and stores in shared memory to re-order non-coalesceable addressing.

The nature of our ray-casting algorithm doesn't consider any thread communication via shared memory, nor any thread-wise pre-computation. On the other hand, we understand the importance of shared memory and we have tried to use it as replacement for local memory. In the most complex kernel, `d_RenderShadow` number of free registers is lower than its memory requirements so some local memory is allocated. Local memory is very slow (latency) comparing to registers and shared memory, and therefore we thought that some performance can be gained. Unfortunately we were not successful because after this optimization, there was no change in allocated size of local memory. We think, it is the great complexity of this kernel, that doesn't allow further replacement for local memory or some problem

with the compiler (we have found many of them so far). However it is possible that some local memory latency can be hidden by higher number of running threads, but this has to be tested. On the other, in the much simpler `d_RenderDVR` kernel, we successfully reduced used registers and gained some additional speedup.

Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy (many threads, many thread blocks)
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor (registers, shared memory)

We have studied generated *cubin* files and used *CUDA Visual Profiler* [6] or even *decuda* [5], external disassembler tool, to analyze our CUDA kernels. We have discovered some important information like number of used registers, size of allocated constant, shared and local memory or number of divergent branches. These information are very important and we have referred them as useful indicators during the optimization process.

At the beginning we started to run kernels with block of threads with dimensions 32x16, which means, that we used all 512 possible threads. However as the complexity of some kernels grew enormously, we were forced to decrease the number of threads to 128 with the block of threads downsized to 16x8. This gave us 64 available registers and 128 bytes of shared memory per thread. With these resources available, each kernel showed different requirements in cubin file. Used constant memory doesn't really have any significant impact on overall performance so we focused mainly on used registers and local memory. `d_RenderDVR` kernel showed as the one with the lowest register consumption (20) and `d_RenderShadow` as the one with the highest requirements (60 regs. + 68 local mem.). That means 10 times lower multiprocessor occupancy, which we consider key factor for overall kernel performance.

4.6.2 Global optimizations

It application design stage, we hoped for many possible accelerating techniques, which can be used in our volume renderer. Most of all we intended to use techniques like early ray termination or efficient calculation of isosurface intersections. We also thought about optimizing pixel transfer, when rendering the final image.

- **Early ray termination**: This common acceleration technique was implemented without any problems. It is focused on reduction of unnecessary pixel operations in DVR or rendering of transparent isosurfaces. As described in the previous sections, we perform front-to-back ray-casting and because of that, we can interrupt ray marching process after opacity has reached predefined threshold - further color accumulation is not necessary. Gained speedup was not measured, but it is clear that this technique brought us significant improvement.

- **Voxel traversal**: First version of our voxel traversal algorithm was rather slow and ineffective. We noticed, that it was necessary to optimize this part of code especially for large datasets, where there are many intersected voxels. Some optimization was done to original algorithm but at the end we have written completely new version of voxel traversal algorithm. It is much faster and also the code is much simpler. Detailed description of the new algorithm can be found in Chapter 3.

- **Isosurface intersections**: The most critical part of our renderer is the code for calculating isosurface intersections. We use two different algorithms for those calculations: Neubauer's algorithm for repeated interpolation and accurate intersection finding algorithm by Marmitt et al. Both are described in Chapter 3. Accurate test is much more complicated and therefore we first aimed for reducing the number of those cases, when this test will fail because no intersection is found. For each processed voxel we fist look at its eight vertices. If all density values are below or above defined isovalue, we already know, that there is no isosurface intersection and there is no need to perform additional accurate test. This optimization gained some noticeable performance improvement and was considered successful. However when we optimized Neubauer's algorithm, we saw even greater difference of FPS, but in negative axis. This was quite a shock. We think, that the main reason could be the fact that both algorithms are totally different. One algorithm (though complicated) can take advantage of well written optimization and another (simpler) just doesn't need it because in the end it makes it slower.

- **Empty space skipping**: This global optimization was intended to be used only in a case of poor and insufficient performance. We have always thought that our renderer can perform better with empty space skipping technique. But after some time programming in CUDA, it showed that implementation of kd-tree or octree hierarchy is not really convenient to use with CUDA. Recursive structures and tree traversal

are good choice for CPU renderer but our case requires alternative approach. This is a example of optimization designated for future work.

- **Pixel transfer**: We noticed, that volume renderer example, supplied with new CUDA SDK, used `glReadPixels` for uploading final image from PBO to screen. This function is considered as relatively slow for pixel transfer operations and therefore we decided to use faster alternative. Instead of `glReadPixels`, we first use `glTexSubImage2D` for uploading data from PBO to texture and then draw full-screen quad using this texture. This idea seamed to be good option, but we have discovered that performance improvement was only about 1% above the results achieved by original method.

4.7 .Net Application

Main application is a single window program developed under .NET using Windows Forms. This main window is handled by `MainFrom` class. There is no message loop like in native windows application when using WIN API. Instead of that, all actions are handled by callback methods, which are assigned to specific events. Methods like `OnPaint` and `OnClick` are examples of these callbacks.

One of the most important callback is that for *idle* event, which is handled by `OnApplicationIdle` method. It occurs when the application finishes processing and is about to enter idle state. We use this method to update the renderer and also update our special timer, which performs calculations of frames per seconds (`FPSTimer` class). Other callbacks process numerous events from all main GUI controls like main menu, right control panel of transfer function controls. Every action is then translated into Renderer Library API function call. User can perform various actions like load new volume data, toggle bounding box visibility, set specific color for isosurface or change its isovalue. Complete list of supported actions is equivalent to the list of functions described in the previous section about Renderer Library API.

4.7.1 VL framework

VL framework was designed by J. Hlaváček (see [13]) as a complex framework dealing with accelerating and simplifying the development of systems

for processing and visualization of medical volume data in C#. We intended to use VL as much as possible or even integrate our CUDA renderer into VL. But at the end we decided to use only a small part of all its features - its high level support for transfer functions. We use VL class `TransferFunction1D`, which is responsible for all operations handling one-dimensional transfer function.

4.7.2 Transfer function control

This control is represented by `MyTransferFunctionControl` class and it is inspired by a similar control used in experimental implementation of volume renderer, which was included in the first release of VL framework. This custom control is responsible for managing the transfer function for specific volume. It was designed to provide user friendly interface for visualizing and modifying this transfer function. We also implemented feature for loading predefined lookup tables (LUTs) from a text file. Renderer Library API function `VRGetHistoData` is used to fill all histogram data from assigned volume dataset. Whenever user interacts with the control and its transfer function is changed, renderer is updated through `VRSetTransferFnc` API function.

4.7.3 Renderer control

User control, which wraps CUDA Renderer Library and defines output rendering window is handled by class `CUDARendererControl`. OpenGL rendering context for its window is created during the initialization part, when the control is created. This context is destroyed by parent window when the control is about to be disposed. After OpenGL window and Renderer Library are initialized, it is possible to load new volume data and start with the rendering by calling `VRRender` function.

User interaction is handled by processing various mouse and key events by assigned callback methods. `OnMouseDown`, `OnMouseMove` and others process user input and translate it into specific Renderer Library API function like `VRRotate` or `VRMove`. When this happen, renderer switches into *interactive mode* for smoother rendering and output resolution is scaled down to 1/4 of window size. After predefined time of user inactivity (1000ms) it will switch back to normal rendering mode.

This control was intended to be compatible with VL interface API for user DVR and isosurface renderer classes. Unfortunately VL framework

doesn't provide suitable compatibility level with Renderer Library. It is because our CUDA solution requires different interface than it was showed in experimental SM 3.0 renderer implementation for VL. Of course we are still interested in integrating our renderer into VL and therefore it can be considered future work.

Chapter 5

Results

5.1 Performance comparison

We have performed several benchmarks to provide overall information about rendering performance of our application. Standard desktop PC with following configuration was used for the tests:

- **OS:** Windows XP SP2, 32-bit
- **CPU:** AMD Athlon 64 3000+
- **System Memory:** 2048 MB
- **VGA:** NVIDIA GeForce 9600GT, 512 MB Video Memory, Driver version 177.35b

We believe that this combination of single core processor and mainstream GPU is representative for most mid-range users and so are our performance results.

Because our program was designed to run as windowed application (although full-screen is also possible), default size of renderer window was set to acceptable 512×512. Most of our tests were performed in this resolution.

The first set of tests was executed with five different datasets and four different rendering modes. Each dataset is specific in its size and structure. We believe that such a combination of various tests is necessary to provide overall perspective of the rendering performance.

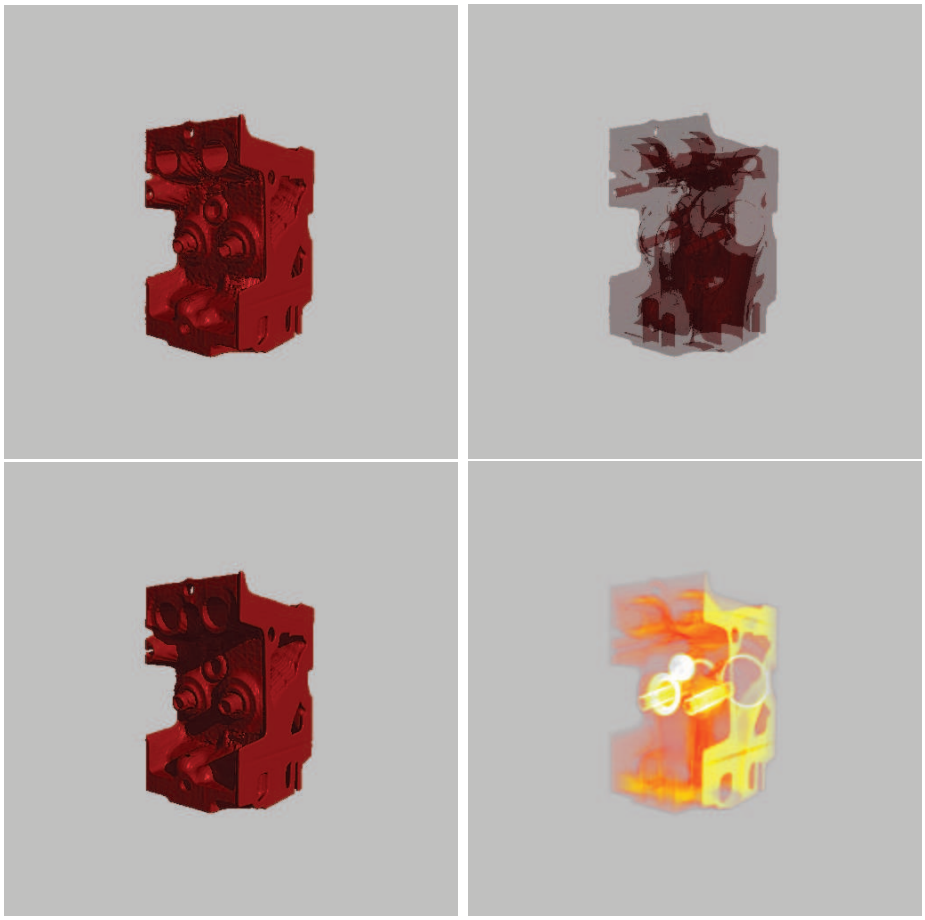


Figure 5.1: Engine dataset, as rendered in different modes. Left to right, top to bottom: opaque and transparent isosurfaces, self shadowing and direct volume rendering.

The application supports several rendering modes (see Figure 5.1). Transparent isosurface rendering mode is our default mode. It was used for opaque isosurface rendering as well. Tests for high-quality rendering were performed in self shadowing mode and direct volume rendering was also used for testing. Difference Ray and Overlay mode were not included in the benchmarks, because the first one computes only one ray (which is not interesting for testing), and the second one is performed in fact by the same kernel as used for transparent isosurfaces.

The Figure 5.2 shows the rest of the datasets used in our benchmarks. The first one is *Bucky ball*, which is also a dataset with the smallest size (32^3). We used sample bucky ball provided by NVIDIA CUDA SDK [28] volume renderer example. The second dataset is down-sampled version of *Bonsai*

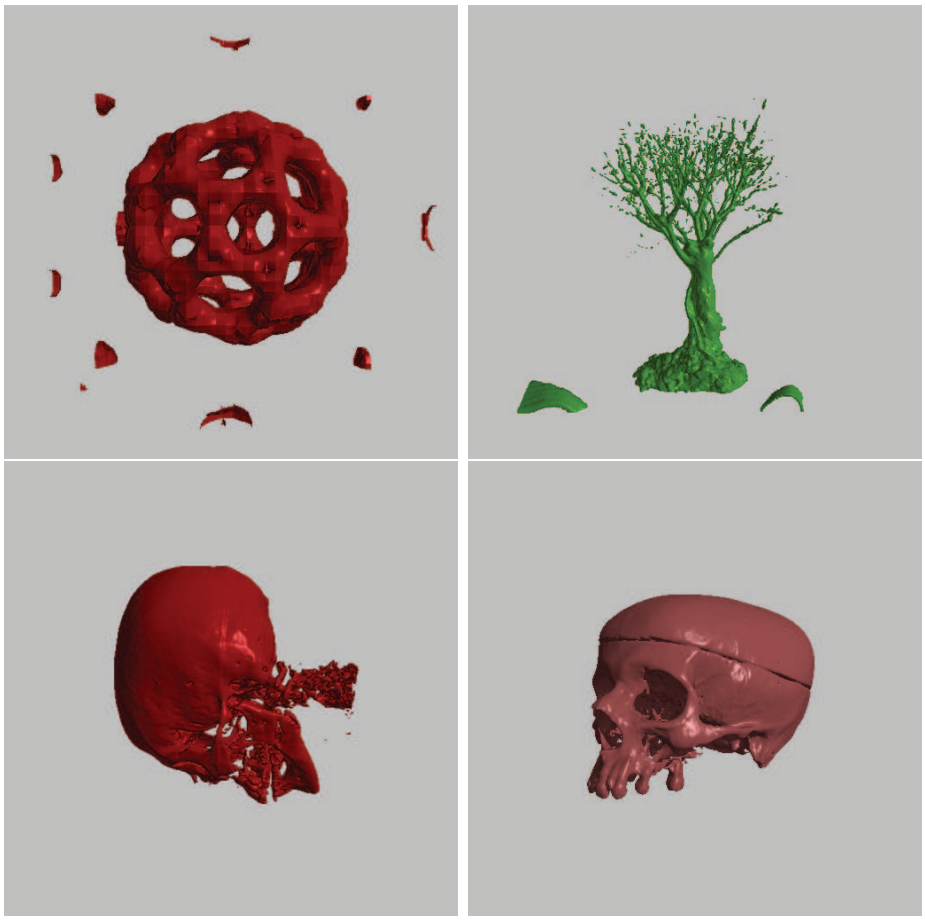


Figure 5.2: Datasets used for performance testing: Bucky ball(top-left), Bonsai tree (top-right), CT head bottom-left and Anthropological skull (bottom-right).

tree ($256^2 \times 128$) from Stefan Röttger's Volume Library. CT scan of human *head* ($256^2 \times 225$), which was acquired from Simple and Flexible Volume Rendering Framework (SPVolRen), developed by Stegmaier et at [36] is the third dataset. This framework is also used as the reference renderer, which our solution is compared to. This comparison is performed in all four rendering modes, and with the mentioned datasets. The last testing sample is an anthropological *skull* ($512^2 \times 648$), which is our largest dataset. Unfortunately were not able to load it into SPVolRen framework because of unknown loading error.

Final results for all benchmarks are shown in the Table 5.1. We have performed measurements of minimum, maximum and average count of rendered frames per seconds (fps). The output window was resized to resolution of 512×512 pixels. During this benchmark, fps calculations were performed

Test 1: Opaque Isosurfaces				
	min. fps	max. fps	avg. fps	avg. fps SPVolRen
Bucky	90.0	93.0	90.3	30.0
Engine	31.5	37.0	33.1	60.2
Bonsai	16.5	22.6	18.4	30.03
Head	14.3	18.4	16.0	29.1
Skull	3.3	10.8	5.3	N/A
Test 2: Transparent Isosurfaces				
	min. fps	max. fps	avg. fps	avg. fps SPVolRen
Bucky	73.1	76.0	74.2	4.2
Engine	21.6	26.5	23.6	7.9
Bonsai	15.5	21.6	17.2	8.8
Head	12.0	15.1	13.3	5.1
Skull	2.1	8.1	3.4	N/A
Test 3: Self Shadowed Isosurfaces				
	min. fps	max. fps	avg. fps	avg. fps SPVolRen
Bucky	19.4	22.0	20.8	29.3
Engine	7.5	11.6	8.5	37.5
Bonsai	4.8	7.3	5.4	20.1
Head	3.8	5.4	4.5	24.75
Skull	0.7	2.9	1.1	N/A
Test 4: Direct Volume Rendering				
	min. fps	max. fps	avg. fps	avg. fps SPVolRen
Bucky	54.0	57.0	55.3	29.9
Engine	81.1	92.3	85.5	41.0
Bonsai	59.0	60.2	59.4	28.2
Head	44.7	61.2	49.3	23.2
Skull	7.4	34.0	11.2	N/A

Table 5.1: Comparison between our application and SPVolRen framework in four different rendering modes.

and measured while rotating tested volume around its y-axis. All 360 corresponding frames are rendered.

These results clearly show, that the complexity of used datasets is almost equally scaled in all rendering modes with one little exception - bucky ball

performance in DVR rendering. Differences between minimum and maximum fps fluctuated between 2% increase, up to over 300%. High values were however measured with absolute difference of 6 fps. We can see them almost exclusively in those tests, which were performed with the largest dataset.

Next we compare the results from the first test with the second test. This shows, how much is the algorithm for opaque isosurfaces faster than the version, which renders more isosurfaces with transparency enabled. Overall speedup is about 20% and it means that additional computations for transparent isosurfaces bring only moderate slowdown. These two modes are using the same renderer so the difference is caused only by those additional computations for each new intersected isosurface.

Self shadowing showed clearly as the most time-consuming rendering mode. However, it was caused mainly by its part of the code where high-quality isosurface test is performed. It increased overall complexity of the renderer to higher level. Although it produced the best quality of rendered isosurfaces, it performed slower than others. Value of minimal fps for the Skull dataset was below 1, and because of that it can not be used for interactive work with very large volumes.

The last group of tests benchmarked direct volume rendering mode. We used default transfer function a sampling rate of 512 samples for the volume ray-marching. The results showed the best performance among all rendering modes. This was expected, because its rendering kernel is the most simple implementation among others.

Our CUDA renderer was also compared with alternative SM 3.0 implementation of SPVolRen framework to show relative difference in performance for all compatible rendering modes. This comparison showed us that in the some cases SPVolRen performed better as our solution. Transparent isosurfaces and direct volume rendering was on the other hand faster with our renderer. It is also important so say that we use complex ray-caster based on voxel traversal and SPVolRen is based on sampling ray-caster, which is much simpler. We also found their renderer more balanced, when average fps for each dataset was not so much dependent on tested dataset. However we noticed small abnormality in their shader for rendering transparent isosurfaces. Its poor results were observed with all used datasets. The average performance drop over the previous mode for opaque isosurfaces was measured by factor of six. Compared to our renderer, it is much slower.

Final results for the last set of benchmarks are showed in the Table 5.2. We measured average fps of the renderer for opaque isosurfaces again, but this time, we report how it scales with different resolutions of rendering

	128^2	256^2	512^2	1024^2
Bucky(32^3)	398.0	219.0	92.0	34.8
Engine($256^2 \times 110$)	138.0	82.7	33.1	12.6
Bonsai($256^2 \times 128$)	93.3	51.9	18.4	6.1
Head($256^2 \times 225$)	64.8	40.0	16.0	5.7
Skull($512^2 \times 648$)	23.3	9.0	5.3	3.2

Table 5.2: Performance of opaque isosurface rendering in different output resolution.

window. Four tests were performed and the results showed that average fps scaled by factor of 3 in the best case. This shows that our implementation scales better than is expected, because average FPS dropped at most by a factor of 3, while the number of pixels increased by factor of 4. We also report that performance scales very good with dataset size.

5.2 Output quality

This section refers to images acquired from different rendering modes, which are compared and discussed in the terms of output quality. First of all we compare three different algorithms for isosurface rendering. Two of them are implemented in our isosurface render and the third is method used in SPVolRen framework. These algorithms are: voxel traversal with repeated linear interpolation isosurface test (Neubauer [23]), voxel traversal with accurate isosurface test (Marmitt et al. [20]), and ray-marching with constant step and single linear interpolation test. The Figure 5.3 shows output images of *Head* dataset using all three renderers. We see the difference between the first and second image where the exact intersection test outperformed simple interpolation method. We have especially selected these images for the comparison, because we can see how the linear interpolation test combined with voxel traversal produce unwanted grid artifacts. However the Figure 5.5 shows images, where these artifacts are not so evident. SPVolRen and its ray-marching method was able to produce sufficient quality of rendering only with high sampling rate (at least 2000 samples were needed) and it therefore performed relatively slow.

Furthermore we have tested quality of images achieved by our implementation of direct volume rendering. Results are shown in the Figure 5.4. We have performed three tests with different sampling rates. These results show, that we need to perform ray-marching with at least 200 samples to

achieve acceptable image quality. We have also observed that sampling rate with more than 400 samples in the volume does not bring much greater improvement.

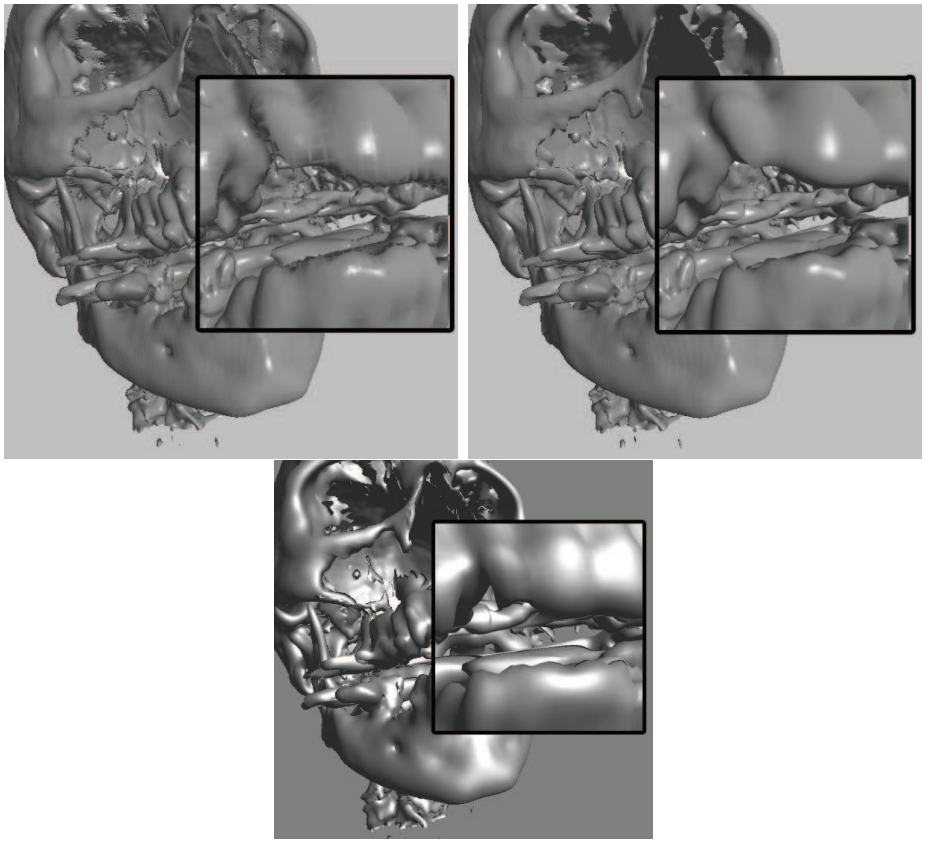


Figure 5.3: Examples of image quality of three different algorithms for isosurface rendering: voxel traversal with repeated linear interpolation (top-left), accurate test (top-right), constant sampling step in SpVolRen(bottom).

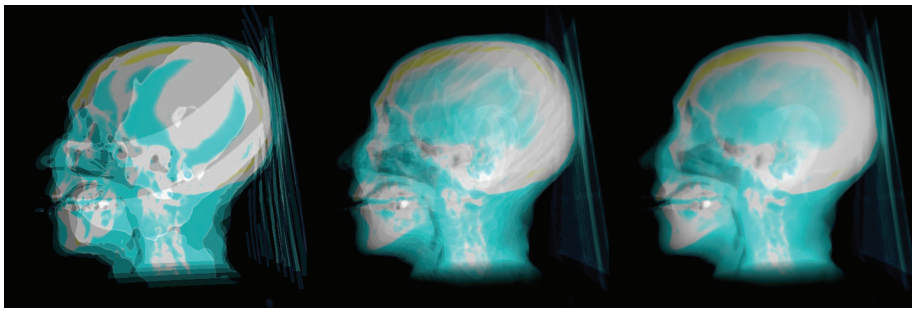


Figure 5.4: Direct volume rendering with different ray-marching steps. From left to right: 25 steps, 100 steps, 400 steps.

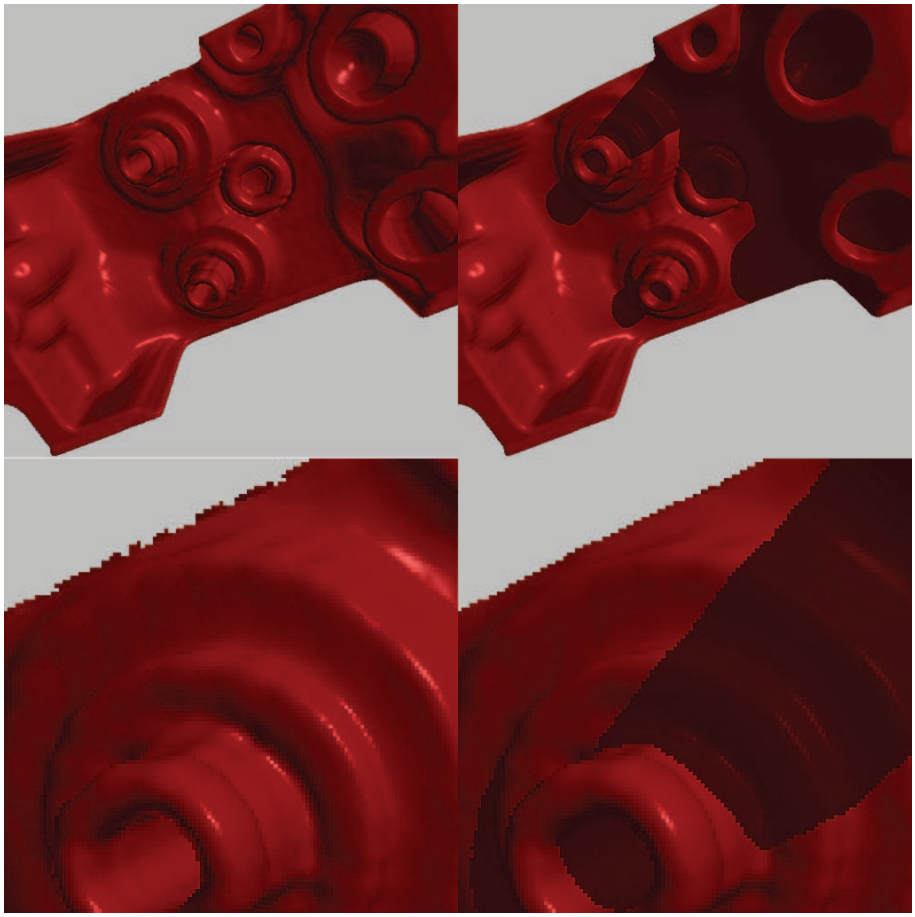


Figure 5.5: High-quality isosurfaces with self shadowing compared to standard rendering.

5.3 Comparison with other applications

We have compared performance and quality results with SPVolRen framework in the previous part. In this section we discuss other packages, which offer solution for volume visualization like Voreen [40], MRIcro [22] and NIS Elements [26], etc. Each of this applications are designed for different users and provide various kind of features regarding volume rendering or isosurface rendering.

For example NIS Elements is a good choice for live cell imaging where its implemented volume renderer is able to visualize multi-component spectral z-stacks. This software supports rendering of 3D time-lapse sequences and generated isosurfaces from segmented images. It comes also with the integrated tool for creating video sequences of visualized volume data.

Voreen is considered simple tool for volume visualization with advanced transfer function control, standard rendering modes and lighting or material settings. MRIcro program comes with many features concerning region of interest and supports working with overlays.

However none of these packages are useful for anthropological research, where the researchers demanded tool for difference visualization. Our application is the answer to this request and brings features, which were not found in any of other tested programs.

We have obtained the set of scanned skulls from the anthropologists, which they have prepared for comparison. These skulls were used as the testing datasets to show new capabilities of our application. We present results of renderer, which supports multiple volumes and such a comparison and difference visualization is possible. Examples of final images when using different rendering modes are shown in the Figure 5.6. This set of images includes examples of high-quality isosurfaces, transparent rendering and Overlay Mode together with Difference Ray information bar. Last two images show results of slice view in direct volume rendering mode.

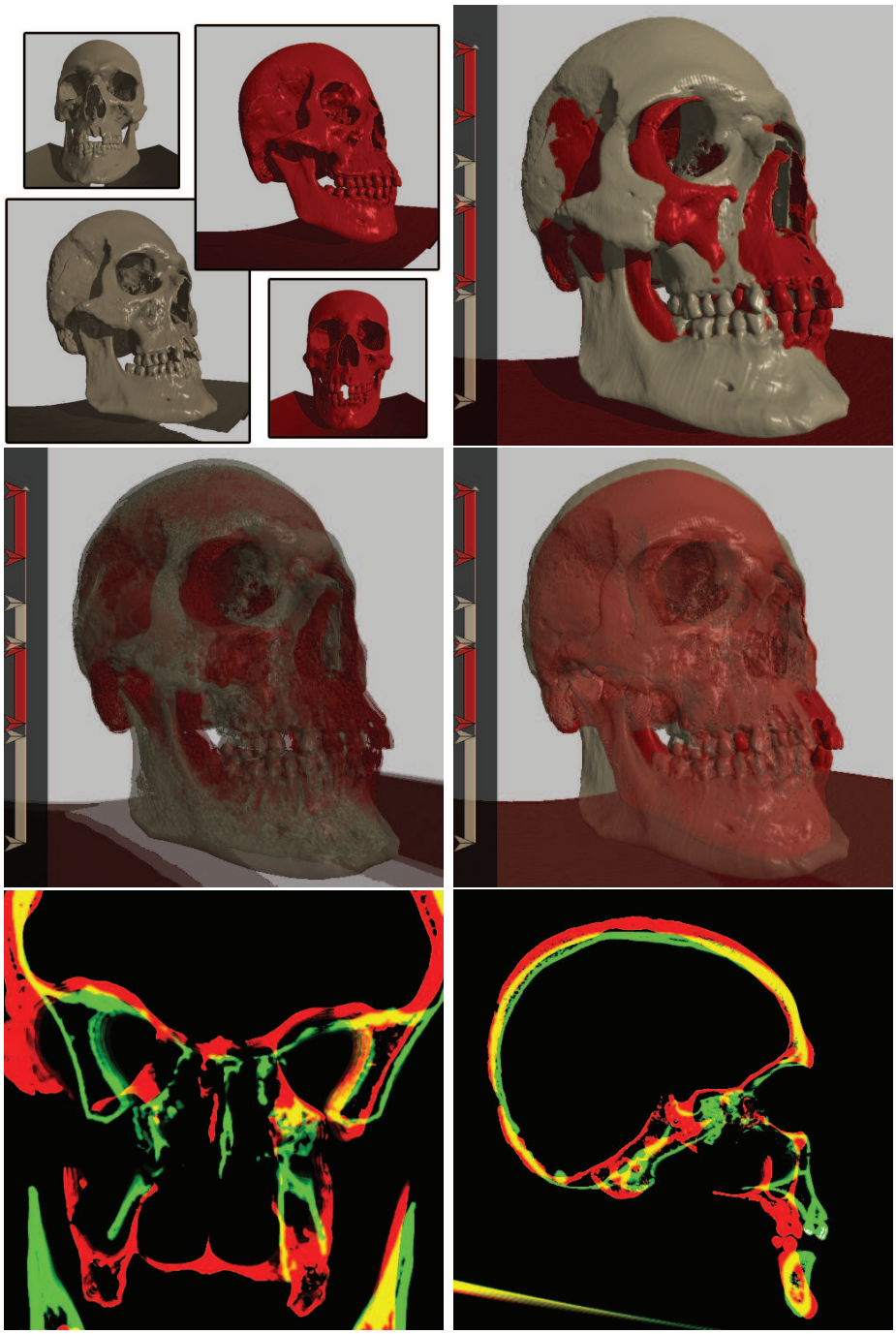


Figure 5.6: Examples of rendering modes used for difference visualization.

Chapter 6

Conclusion

6.1 Summary

We have developed prototype application designed for visualization of human skulls. This work started with many preparations including literature search, especially in the area of isosurface rendering. We have selected this visualization method as the best option for visualizing objects like human skulls, which are common research specimens used in anthropology and archeology. Our colleagues from these departments demanded tool, that will help them with their research and they saw potential advantage in modern nondestructive visualization methods like CT scanning. We have discussed several key features, which had to be included in such a tool. Additional support for displaying multiple skulls and interactive difference visualization were selected as the main features. Furthermore they intended to use this visualization tool for public presentations and high-quality rendering was thus another key element.

We have decided to use algorithms for real-time high-quality isosurface rendering ([23, 20]) and implement them entirely on GPU. Our application is considered the first CUDA implementation combing techniques for rendering of accurate isosurfaces together with selfshadowing or transparency, while adding support for multiple volumes. All these algorithms are based on ray-casting and therefore such a combination is possible. Additionally we have implemented standard direct volume rendering using ray-casting with arbitrary clipping plane.

Second part of this work has focused on difference visualization. We have included several options for researchers to compare, visualize and then analyze differences of human skulls. Special Overlay mode allows to visualize

these differences in higher level than for example transparent rendering and gives better perception of spatial structure of compared skulls. DVR slice view mode together with Difference ray mode are another two useful options for visualization and even basic measurement. Single ray can be casted into the volume and information about its intersections with all isosurfaces can be displayed.

After the implementation phase, we have done several tests regarding rendering performance and quality of our application. These results were presented in the previous chapter and additional comments were provided about implemented difference visualization features.

With all these features currently available in our application, we are looking forward to participate in further cooperation with Department of Anthropology and to test the application with real data. So far we have received only a small part of these data and all usable datasets still required additional preprocessing before they could be used in the program.

6.2 Goals achievement

We have delivered application for interactive visualization of CT scans of human skulls with difference highlighting, which was our primary goal. This main objective can be divided into these partial accomplishments:

- We have implemented **high-quality** isosurface renderer using algorithms based on ray-marching methods on GPU and employed NVIDIA CUDA technology to deliver modern GPGPU solution for this implementation.
- We were able to perform **interactive** rendering in 512×512 window with some algorithms and showed that further improvement is possible with additional optimization.
- We have successfully integrated support for **multiple volumes**, which currently allows two volumes to be loaded and visualized at once. Only a simple change of code is then required for adding even more datasets.
- Three modes for **difference visualization** were developed and implemented and we have shown how they can be used with real data.
- We have also performed **comparison with similar systems** and have presented our results.

Because of these accomplishments, we declare this work successful and hope for more future improvements and any subsequent use of our application.

6.3 Future work

We believe that this application is complete, but some currently implemented features can be further upgraded and optimized. It has been considered prototype implementation since we are the beginning of the collaboration with the Department of Anthropology. It is only the first step of much larger project. In the beginning of this teamwork, we have discussed many features, which were divided into several groups. One of these group is covered by this work but some other features are related to our application as well. Their integration into our solution is natural decision. We plan to or we have already started to work on some of these features:

- **CUDA specific optimization**
  As we have shown in the previous part about optimization, it is necessary to continue with optimization process. Performance results achieved with the current version of CUDA volume renderers can be further improved after intensive work on code optimization for SIMT architecture. In the near future we anticipate the arrival of release version of CUDA Toolkit 2.0, which should also bring some additional tweaks and improvements to the current beta version. Also we would like to perform tests on multi GPU solutions (SLI) and report their results.

- **New mode for difference visualization**
  This is the second most wanted enhancement. We have already done some side work regarding next-generation of difference highlighting, but further research is required before any kind of implementation. This problem is related to the fact, that we still need to fully understand what is really important as a *difference* in the anthropological point of view. We plan to upgrade current Overlay Mode to higher level, where all differences will be more visible and easier to work with.

- **Different rendering techniques**
  After we have taken initial research concerning isosurface rendering we decided to implement specific ray-casting algorithms, which we have described in Chapter 3. On the other hand, alternative approach, presented by Sigg et al. [35], made us also interested in their method using

cubic reconstruction filter for high-quality isosurface rendering. This completely different rendering technique should be properly studied for possible new CUDA implementation.

- **Large datasets**
  Large dataset are partially supported by our application, but due to the limited access to graphic cards equipped with more video memory, we were not able to test it with some really large datasets. We also wanted to test CUDA support for large 3D textures (2048^3), but we currently don't have any available hardware capable of this test.

Another group, which was not directly related to this work, included various interesting features. Since we hope that is it possible to integrate them to our application, future work can be done in these areas:

- Isosurface texturing
- Volume measurement
- 3D image registration
- Advanced camera path control and replay
- Output video encoding

We hope for intensive co-operation with anthropologists to receive additional feedback and therefore we provide first-approach solution in the form of a tool, which should be soon tested by group of researchers. This feedback will greatly help us understand, if we have made right choices and if we are getting closer to the desired final solution, or if we have made some mistakes. This is the matter of time and only the future will tell about our success with this collaboration project. However in the end, we all who participated in this "opportunity" can say, that everybody gained great amount of precious experiences, while working on interesting research.

Bibliography

[1] AMD Stream Computing, http://ati.amd.com/technology/streamcomputing/

[2] J. Amanatides and A. Woo :*A Fast Voxel Traversal Algorithm for Ray Tracing*, Eurographics '87, Eurographics Association, 1987

[3] G. Cardano: *Ars Magna*, Nurmberg, 1545

[4] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, p. 91–98, New York, NY, USA, 1994

[5] Cubin Utilities, decuda, http://www.cs.rug.nl/w̃ladimir/decuda/

[6] CUDA Visual Profiler 2 beta, http://www.nvidia.com/object/cuda_develop.html

[7] T. J. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical report, Chapel Hill, NC, USA, 1994

[8] J. Danskin, and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *ACM Workshop on Volume Visualization '92*, p. 91–98, 1992

[9] K. Engel, M. Kraus, T. Ertl :*High- Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, Los Angeles, 2001

[10] A. V. Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. In VVS '96: *Proceedings of the 1996 symposium on Volume visualization*, p. 23–ff., Piscataway, NJ, USA, 1996

[11] S. Green :*Introduction to CUDA, CUDA Performance, CUDA toolchain*, ARCS 2008 GPGPU and CUDA Tutorials, February 2008

[12] D. Herbison-Evans. *Solving Quartics and Cubics for Graphics*, Technical Report, Sydney, July 2005

[13] J. Hlaváček: *GPU-accelerated processing of medical data*, 2008

[14] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting Frame-to-Frame Coherence for Accelerating High-Quality Volume Raycasting on Graphics Hardware. In *IEEE Visualization 2005 (VIS'05)*, Minneapolis, Minnesota, USA, pp.223-230, 2005

[15] D. Knuth: *Structured Programming with go to Statements*, ACM Journal Computing Surveys, Vol 6, No. 4, p. 268, December 1974

[16] J. Krüger and R. Westermann. Acceleration Techniques for GPU based Volume Rendering. In *Proceedings of IEEE Visualization 2003*

[17] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9(3)* , p. 245–261, July 1990

[18] D.E. Littlewood. *A University Algebra*, Heineman, London, p. 173, 1950

[19] W. E. Lorensen and H. E. Cline :*Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, Computer Graphics (Proceedings of ACM SIGGRAPH '87), p. 163–169, 1987

[20] G. Marmitt, A. Kleer, I. Wald, H. Friedrich, P. Slusallek :*Fast and Accurate Ray Voxel Intersection Techniques for Iso-Surface Ray Tracing*, Vision, Modeling and Visualization (VMV) 2004, Stanford (CA), USA, 2004

[21] M. Meißner, U. Hoffmann, and W. Straßer. Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering using OpenGL and Extensions. In *VIS '99: Proceedings of the conference on Visualization '99*, p. 207–214, San Francisco, CA, United States, October 1999

[22] MRIcro, www.mricro.com

[23] A. Neubauer, L. Mroz, H. Hauser, R. Wegenkittl: *Cell-based first-hit ray casting*, Proceedings of the symposium on Data Visualisation 2002, May 27-29, Barcelona, Spain, 2002

[24] NVIDIA's NV_fragment_program2 OpenGL Extension specification, http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_fragment_program2.txt

[25] Shader Model 3.0, NVIDIA Developer Technology Group, 2004

[26] NIS Elements, Laboratory imaging, www.nis-elements.com

[27] NVIDIA CUDA Programming Guide 2.0, June 2008

[28] NVidia CUDA 2.0 Beta SDK Volume Rendering Demo.

[29] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *Computer Graphics SIGGRAPH 98 Proceedings*, p. 703–712, 2002

[30] S. Röettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the Eurographics Symposium on Data Visualisation*, 2003

[31] E. Rollins. *Real-Time Ray Tracing with NVIDIA CUDA GPGPU and Intel Quad-Core*, http://eric_rollins.home.mindspring.com/ray/cuda.html, 2007

[32] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, p. 109–118, 2000

[33] J. Schwarze :*Graphics gems*, Academic Press Professional, Inc., San Diego, CA, USA, p. 404-407, 1990, ISBN:0-12-286169-5

[34] M. Segal, K. Akeley :*The OpenGL Graphics System: A Specification Version 2.1*, www.opengl.org/documentation/specs, December 1, 2006

[35] Ch. Sigg, M. Hadwiger, M. Gross, K. Bühler. *Real-Time High-Quality Rendering of Isosurfaces*, Technical Report VRVis, 2004

[36] S. Stegmaier, M. Strenger, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. In *Proceedings of Volume Graphics*, p. 187–195, 2005

[37] G. Treece, R. Prager, and A. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics 23(4)*, p. 583–598, 1999

[38] R. Yagel, and Z. Shi. Accelerated Volume Animation by Space-Leaping. In *Proceedings IEEE Visualization '93*, p. 62–69, 1993

[39] Y. Uralsky. Practical Metaballs and Implicit Surfaces, NVIDIA Developer Technology on Game Developer Conference, San Jose, CA, USA, March 2006

[40] Voreen, Volume Rendering Engine, www.voreen.org

[41] M. Weiler, M. Kraus, M. Merz, and Th. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proc. Visualization '03*, p. 333-340, 2003

[42] R. Westermann and T. Ertl, Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, p. 291–294, New York, NY, USA, 1998

Appendix A

User Guide

In this appendix we provide simple user guide to our testing application. We describe functionality of all GUI controls included in the main application window. Overall screenshot of this window is shown in the Figure A.1.

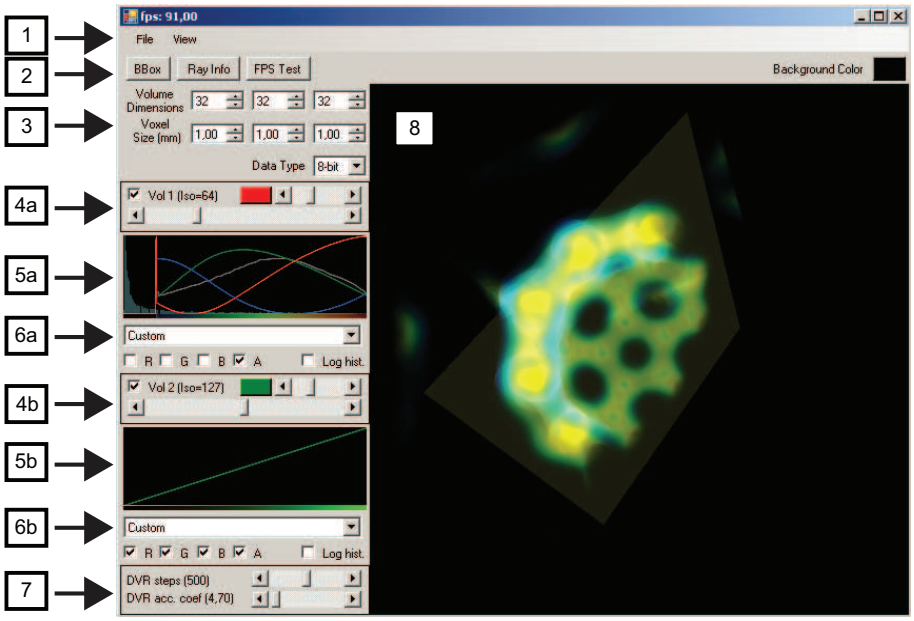


Figure A.1: User guide screenshot of our application.

The application window is divided into four main parts: main menu with top toolbar, right panel and renderer window. These parts contain more GUI elements, which are described in the following summary.

1. **Main menu**
   Commands for loading new primary and secondary volume data are available through the *File* popup menu. The *View* popup menu provides switching between the rendering modes.

2. **Top toolbar**
   It contains buttons which enable and disable volume bounding box and info about Difference Ray mode, together with the button for enabling FPS performance test. User can also change the background color used in the render window.

3. **Dataset info**
   User must there specify the complete information about volume dimensions, voxel extent and data type before loading new dataset. These information about each dataset are located in the *.vh* header files.

4. **Isosurface settings**
   These controls provide settings for primary and secondary isosurfaces and allow changing their color and specifying their isovalue.

5. **Transfer function control**
   This control provides easy way for modification of the transfer function through separate setting of its RGBA lookup table.

6. **Transfer function presets**
   Several predefined presets for transfer function can be selected from this combo box control. User can also add new preset by defining its transfer function in the *predefined.lut* file located in the main application directory.

7. **DVR settings**
   Ray-marching sample rate and color accumulation factor used in direct volume rendering mode can be changed by this control.

8. **Renderer window**
   This window displays rendered image and processes all mouse and key input from the user. Volume rotation is performed with mouse move and the left mouse button, the right mouse button is for moving the volume and the mouse wheel allows zooming. While in DVR mode and the *CRTL* key is pressed, user can rotate the clipping plane with left the mouse button or place it into the default position by the right mouse button and move the plane with the mouse wheel.

Appendix B

Contents of DVD

bin – Executable version of our program

data – Sample datasets in raw format

doc – PDF and PS versions of this thesis

src – Source files needed for compiling the application

- **CULib** – CUDA rederer library
  - **src** – C++ source code and CUDA kernels
  - **lib** – Glew library
- **Properties** – Resource files
- **src** – .NET application C# source code
- **testing** – Debug directory for testing