VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

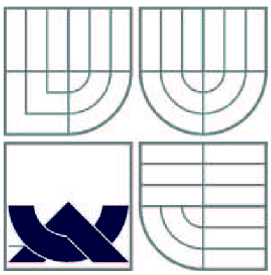
VIZUALIZACE OBJEMOVÝCH DAT POMOCÍ VOLUME RENDERINGU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

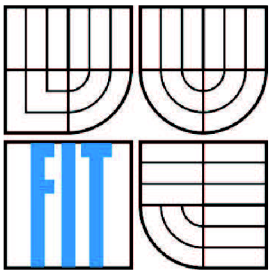
AUTOR PRÁCE
AUTHOR

Bc. PAVEL NĚMEČEK

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VIZUALIZACE OBJEMOVÝCH DAT POMOCÍ VOLUME RENDERINGU

3D VOLUME RENDERING DATA VISUALIZATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVEL NĚMEČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. PŘEMYSL KRŠEK, Ph.D.

BRNO 2010

Zadání diplomové práce/4029/2009/xnemec46

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2009/2010

Zadání diplomové práce

Řešitel: **Němeček Pavel, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Vizualizace objemových dat pomocí volume renderingu**
3D Volume Rendering Data Visualization
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s problematikou vizualizace objemových dat.
2. Analyzujte problematiku vizualizace objemových dat.
3. Navrhněte systém pro vizualizaci objemových dat.
4. Implementujte a otestujte navržený systém.
5. Zhodnoťte dosažené výsledky a stanovte další vývoj projektu.

Literatura:

1. Žara J., Beneš B., Felkel P.: Moderní počítačová grafika. 1. vyd. Praha, Computer press 1998, 448 s., ISBN 80-7226-049-9

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splňte první tři body zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kršek Přemysl, doc. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 21. září 2009
Datum odevzdání: 26. května 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

První část práce je zaměřena na teoretický rozbor metod zobrazování objemových dat. Analyzovány jsou jak metody zobrazující objemová data pomocí převedu na síť trojúhelníků, tak metody přímého zobrazení objemových dat. Podrobně je představena metoda ray casting. Možný způsob její realizace pomocí shader programu grafické karty je předmětem implementační části. Práce uvádí několik metod, které lze při ray castingu aplikovat, a dosáhnout tak různých výsledků vizualizace nad stejnými daty. Práce je také zaměřena na vytvoření grafického uživatelského rozhraní, které umožní interaktivní vytváření vizualizací.

Abstract

The first part of this project is focused on theoretical analysis of methods for rendering volume data. Both methods are analyzed showing the volume data using triangle mesh, and methods for direct volume rendering. Ray Casting is presented in detail. Possible way of its realization using graphics card is the subject of implementation part. The paper presents several methods that could be applied to ray casting and achieve different results of visualization of the same data. The work also aims to create a graphical user interface that allows interactive visualizations.

Klíčová slova

Vizualizace, objemová data, ray casting, OpenGL, vertex a fragment shader, GLSL, Qt, MIP, SIP, LUT, gradientní stínování, přímé zobrazování objemů.

Keywords

Visualization, volume data, ray casting, OpenGL, vertex and fragment shader, GLSL, Qt, MIP, SIP, LUT, gradient shading, direct volume rendering.

Citace

Pavel Němeček: Vizualizace objemových dat pomocí volume renderingu, diplomová práce, Brno, FIT VUT v Brně, 2010

Vizualizace objemových dat pomocí volume renderingu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Doc. Ing. Přemysla Krška, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Němeček
26.5.2010

Poděkování

Tímto děkuji vedoucímu práce za kvalitní vedení a pomoc při vypracovávání diplomové práce.

Obsah

1 Úvod

Pojem vizualizace, jako techniky k zobrazování jisté informace, sahá do dávných dob, kdy lidé neuměli psát a své myšlenky vyjadřovali graficky. Dnes je pojem vizualizace chápán ve spojitosti se zobrazováním velkého množství dat, které by bylo jen obtížné chápat jinak než v grafické podobě. Příkladem může být meteorologická mapa, kterou v grafické podobě viděl snad každý z nás. Odborníkům by jistě čísla stačila, ale pokud chceme výsledky své práce, či pozorování prezentovat i nezasvěceným osobám, je z mé vlastní zkušennosti grafická podoba srozumitelnější. Uplatnění vizualizace najdeme v mnoha oblastech. V této práci jsem se rozhodl věnovat vizualizaci objemových dat – volume renderingu.

Za zdroje objemových dat můžeme považovat cokoli, co na výstupu produkuje diskrétní vzorky v trojrozměrné mřížce. Velmi často vizualizovaná jsou data získaná měřením reálných objektů. K jejich pořízení slouží výpočetní tomografie (Computed Tomography, CT) nebo magnetická rezonance (Magnetic Resonance Imaging, MRI). Výsledkem je sada 2D řezů, které jsou umístěny za sebe do trojrozměrné mřížky. Nezanedbatelnou část vizualizovaných objemových dat tvoří také výstupy z počítačových simulací.

Díky nárůstu výpočetního výkonu se rozšířily možnosti i ve vizualizaci – především v možnosti interaktivní práce s objemovými daty. Týká se to možností rotace objemových dat, přepínání pohledů na objemová data, zvětšení, zmenšení, skrytí částí mimo zájem pozorovatele, zvýraznění objektů v okruhu zájmu. Nárůst výkonu také přináší i zvýšení obrazové kvality a možnosti vizualizovat rozsáhlá objemová data.

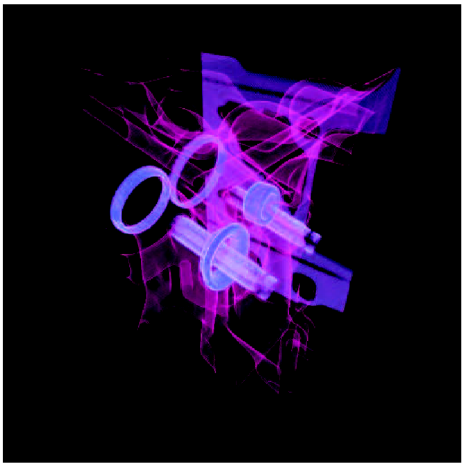
Metod pro vizualizaci objemových dat je poměrně hodně a nelze říci, která je nejlepší. Každá má své charakteristické vlastnosti. Nejčastěji používaným metodám se věnuji v kapitole druhé, v teoretickém rozboru. Detailněji je představena metoda raycasting přímého volume renderingu. Funguje na principu vrhání paprsku objemovými daty. Uvádím možnosti implementace raycastingu pomocí grafického akcelerátoru. Je to metoda, která v této době zažívá rozvoj a to především kvůli nárůstu výpočetního výkonu grafických akcelerátorů. Do teoretického rozboru jsou také zahrnuty hlavní metody aplikované při zobrazení objemových dat. Třetí kapitola vznikla spojením částí návrhu a implementace aplikace s grafickým uživatelským rozhraním pro vizualizaci objemových dat. Předmětem čtvrté kapitoly se stala analýza rychlosti raycastingu v závislosti na zvolené metodě vizualizace a slouží také k prezentaci výsledků. Práci zakončuje kapitola pátá, která zhodnocuje vytvořenou aplikaci a uvádí možnosti jejího dalšího vývoje.

2 Teoretický rozbor

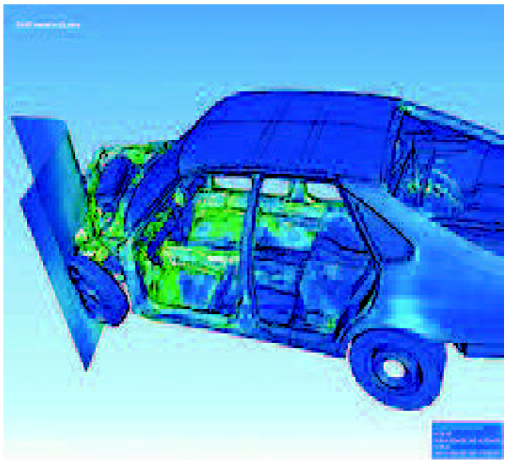
Cílem této kapitoly je seznámit čtenáře s možnostmi vizualizace objemových dat. Pojem vizualizace chápeme jako sadu nástrojů a postupů, sloužících k vizuální analýze dat, tedy o celý proces zkoumání dat a informací po jejich převedení do grafické podoby. Cílem vizualizace je pochopení zkoumaných jevů a jejich vnitřních vztahů. Prostředkem však nejsou tabulky čísel, jako u numerické analýzy, ale zobrazení, v maximální míře interaktivní [1]. Vizualizaci můžeme rozdělit do několika tříd, v závislosti na datech, které zobrazujeme [2]:

- realistické zobrazování 3D modelů (architektura, filmy, animace, design, atd.)
- zobrazení vícerozměrných dat (fyzika, matematika, simulace, atd.)
- zobrazení negrafických dat (grafy, vztahy, procesy atd.)

Jako zdroje při zobrazování vícerozměrných dat nám mohou sloužit například výstupy ze simulací (obrázek 2.2) nebo snímky z výpočetní tomografie a magnetické rezonance (obrázek 2.1). Častým jevem po získání dat je jejich úprava či předzpracování pomocí technik počítačové grafiky (převzorkování, interpolace, filtrování atd.).

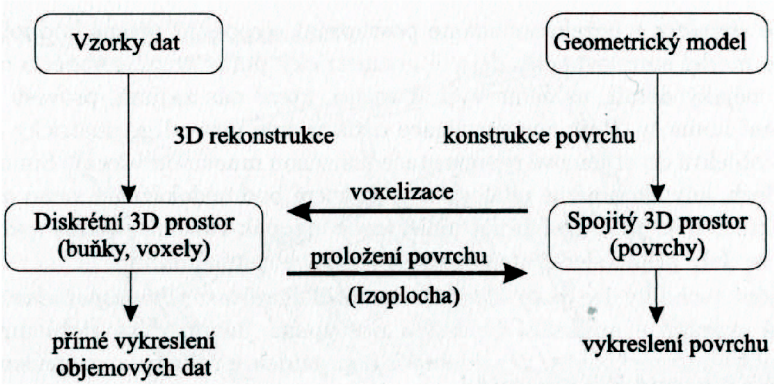


Obrázek 2.1: Vizualizovaný blok motoru získaný z řezů CT [4].



Obrázek 2.2: Vizualizace deformace auta v asymetrickém nárazu [3].

V celém následujícím textu se budeme zabývat metodami a algoritmy pro vizualizaci 3-dimenzionálních dat. Lze je rozdělit na algoritmy zobrazující povrchy (kapitola 2.1) a na algoritmy



Obrázek 2.3: Metody zobrazování objemových dat [1].

přímého zobrazování (kapitola 2.2). Vztah mezi těmito přístupy je znázorněn na obrázku 2.3. Nadále budeme uvažovat pravidelně rozmístěné vzorky dat v 3D mřížce zachovávající konstantní vzdálenosti. Prostorový vzorek dat budeme označovat pojmem voxel.

2.1 Algoritmy zobrazující povrchy

Algoritmy zobrazující povrchy (v anglické literatuře označované surface fitting) vytvoří z objemových dat nejprve aproximaci povrchu v místech s konstantní hodnotou vzorků, nejčastěji ve formě sítě trojúhelníků. Takovou reprezentaci lze s výhodou zobrazit na grafických kartách, kde je vykreslování trojúhelníkové reprezentace optimalizované. Zobrazují tedy data nepřímo. Převod na trojúhelníkovou síť probíhá buď ve fázi předzpracování dat (rozsáhlé modely), nebo během vykreslování, kde o příslušnosti hodnot k ploše rozhodne klasifikační funkce.

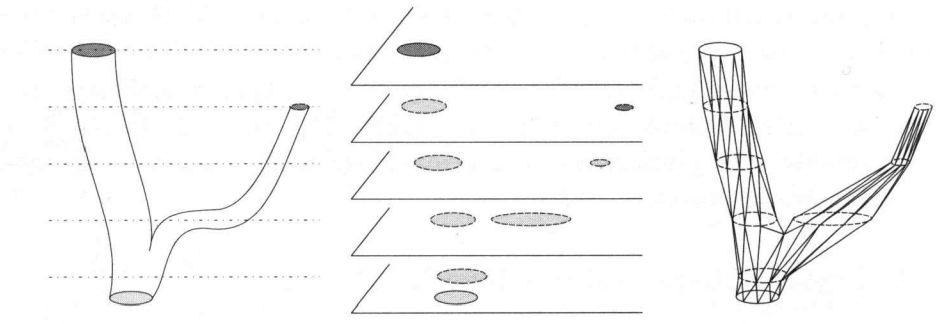
Při zobrazování dat z výpočetní tomografie (CT) se data označují pomocí jednoduchého kritéria, například rozmezí intenzit voxelů, nebo se ve fázi předzpracování vytvoří objem stejné velikosti jako data, ve kterém jsou zaznačeny příslušnosti k povrchu (povrchům). Při vytváření povrchové reprezentace se pak přistupuje do obou souborů dat. Vytváření pomocného objemu dat lze také využít jako masku, která vyřadí voxely z dalšího zpracování.

Následují čtyři kapitoly zachycují nejpoužívanější postupy vytváření povrchové reprezentace objemových dat.

2.1.1 Propojování kontur

Kontura (u složitějších těles s více větvemi a dírami množina kontur) je průnikem hranice tělesa s rovinou [1]. Příklad je uveden na obrázku 2.4 uprostřed. Konturu objektu lze získat výstupem z algoritmů zpracování obrazu.

Vstupem algoritmu je množina kontur, které popisují hranice objektu v každém objemovém řezu. Výstupem je rekonstruovaný trojúhelníkový povrchový model objektu, který ten původní aproximuje. Jádrem rekonstrukce je propojení sousedních kontur sítí trojúhelníků. Čím menší je vzdálenost řezů, tím je rekonstruovaný model přesnější. Grafické znázornění metody je na obrázku 2.4.



Obrázek 2.4: Prostorový objekt, kontury v rovnoběžných řezech a rekonstrukce opláštěním sítí trojúhelníků [1].

2.1.2 Povrchové kostky

Základem tohoto algoritmu jsou objemové elementy – krychličky. Objemová data jsou podle pravidelné mřížky rozděleny na krychličky stejné velikosti. Tato velikost nemusí být nutně v souladu se vzdáleností řezů. Důležité je, aby objemový element reprezentoval jednu hodnotu intenzity. Krychličky jsou následně procházeny a podle zvoleného prahu jsou označeny na povrchové a vnitřní či vnější. Při vykreslování jsou povrchové objemové elementy převedeny na šestici čtyřúhelníků, které se zobrazují již klasickými metodami.

Nevýhodou tohoto algoritmu je výsledná zřetelná kostičkovanost modelu. Efekt se dá zmírnit odhadem normál vrcholů podle původních dat. Tato metoda je především vhodná pro náhledy objemových dat.

2.1.3 Pochodující kostky

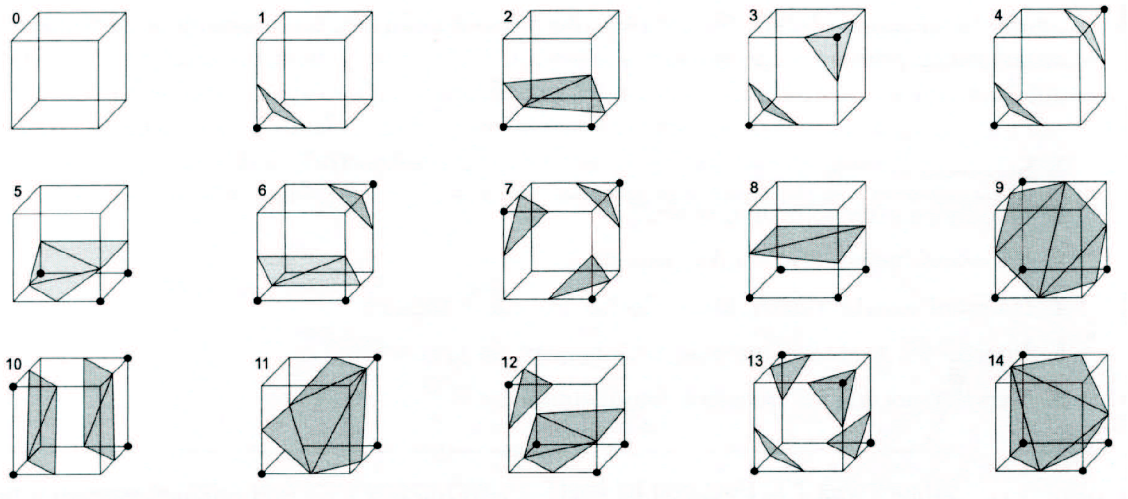
Asi nejznámější a nejrozšířenější algoritmus převodu objemových dat na povrchovou reprezentaci. V této podkapitole zevrubně popíšu algoritmus, tak jak je uveden v [1]. Detailnější informace lze nalézt v původním článku na [13].

Vstupem algoritmu jsou objemová data a hodnotící funkce (práh). Výstupem je množina trojúhelníků popisující povrch tělesa. Kroky algoritmu jsou následující:

1. Definování krychliček – vstup zde tvoří 2 mřížky, které mají ve svých vrcholech hodnoty vzorků volumetrických dat. Plochy se propojí kolmicemi, které spojují vrcholy mřížek, tím se definují krychličky.
2. Ohodnocení vrcholů – k tomuto účelu nám poslouží hodnotící funkce. Vrcholy hodnotí binárně dvěma možnými způsoby – buď jako vnitřní, nebo jako vnější. Pokud jsou všechny vrcholy ohodnoceny jako vnitřní, pak je krychlička uvnitř objektu, pokud jako vnější, je krychlička vně objektu. V obou případech jsou vyřazeny z dalšího zpracování. Zajímají nás tedy jen krychličky se smíšenými vrcholy – protíná je tedy povrch objektu.
3. Sestavení indexu do tabulky případů a nalezení seznamu hran – z osmi binárních hodnot ve vrcholech je vytvořen index do tabulky s 256 možnými variantami, jak může povrch objektu krychličkou procházet. Díky tomu, že je krychle symetrické těleso, lze 256 variant redukovat na 15, které zobrazuje obrázek 2.5.
4. Výpočet souřadnic vrcholů trojúhelníků – poloha vrcholů trojúhelníků na hranách se dopočítá pomocí výsledku hodnotící funkce (předloží se ji vrcholy hrany) lineární interpolací.
5. Výpočet normál – normály ve vrcholech trojúhelníků se dopočítají lineární interpolací normál ve vrcholech krychličky. Vektory ve vrcholech jsou rovny gradientu dat, který se spočítá symetrickou diferencí hodnot okolních intenzit objemových dat.

Převod na síť trojúhelníků touto metodou není bez problémů. Jedním je velké množství generovaných trojúhelníků. Možným řešením je použít algoritmy eliminace hran nebo vrcholů. Dalším problémem je, že se trojúhelníky generují na základě jedné krychličky bez ohledu na sousední. Může a dochází tak k nespojitosti trojúhelníkové sítě, které se projevují jako díry ve výsledném modelu.

Modifikovanou variantou algoritmu pochodujících kostek je pochodující čtyřstěn. Detaily je možno nalézt v [1].



Obrázek 2.5: Základní případy konfigurace vrcholů krychle u algoritmu pochodující kostky a způsob, jakým povrch objektu protíná krychli. Tečky odpovídají vrcholům uvnitř tělesa a neoznačené vrcholy odpovídají vrcholům vně tělesa [1].

2.1.4 Rozdělení kostek

Vznikl jako řešení rasterizace velkého počtu malých trojúhelníků. Na výstupu algoritmu jsou trojúhelníky nahrazeny povrchovými body s normálou. Velikost povrchového bodu odpovídá jednomu pixelu na obrazovce. Tím je dosaženo, že se generuje minimum bodů a negenerují se zbytečné detaily, které by se při rasterizaci zahodily. Důsledkem toho však je, že při přiblížení v modelu vznikají díry.

2.2 Metody přímého zobrazování

Obecně metody označované v anglické literatuře jako DVR (direct volume rendering). Tyto metody nevytvářejí síť trojúhelníků. Jejich výstupem je barevný obraz (barvy pixelů), který je uživateli zobrazen přímo na zobrazovací rovinu.

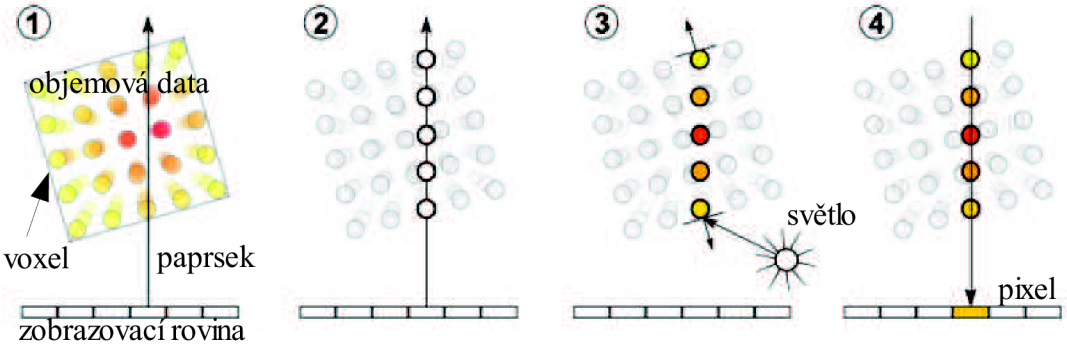
S rozvojem grafických karet, s nárůstem jejich výkonu a funkcí, které jsou schopny poskytovat, je snaha tyto metody implementovat nebo alespoň urychlovat pomocí grafických karet.

2.2.1 Ray casting

Ray casting je technika založená na fyzikální podstatě světla. V počítačové grafice se proces šíření světla modeluje obráceným postupem. Paprsek je promítán ve směru pohledu skrz pixel v zobrazovací rovině. Pro paprsek pak určíme, zda prošel nějakým objektem. Pokud paprsek při průchodu scénou narazil na nějaký objekt, a pozice tohoto objektu je ze všech nalezených nejblíže k pozorovateli, pak nastavíme barvu příslušného pixelu na barvu objektu. Při výpočtu výsledné barvy pixelu lze uplatnit osvětlovacího modelu, např. Phongova [1]. Tak by se dal ve stručnosti popsat princip ray castingu, v české literatuře označován také jako zpětné sledování paprsku prvního řádu.

Značnou výhodou ray tracingu je možnost jeho paralelizace. Každý pixel obrazu lze počítat samostatně. Když se k této výhodě připojí i fakt, že algoritmus není nijak složitý, nabízí se implementace ray castingu pomocí shaderů grafické karty, kdy je fixní zobrazovací řetězec nahrazen vlastním. Více o shaderech v kapitole 3.

Při použití ray castingu pro zobrazování volumetrických dat je do scény umístěn kvádr, reprezentující objemová data a světlo pro vyhodnocení osvětlovacího modelu. Princip získání barvy pixelu je znázorněn na obrázeku 2.6.

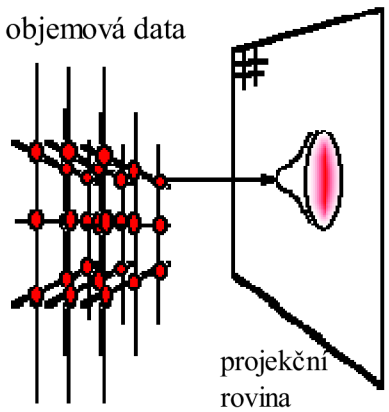


Obrázek 2.6: Čtyři kroky ray castingu pro zobrazování objemových dat. Vržení paprsku (1), vzorkování (2), stínování (3) a zápis výsledné barvy (4) [6].

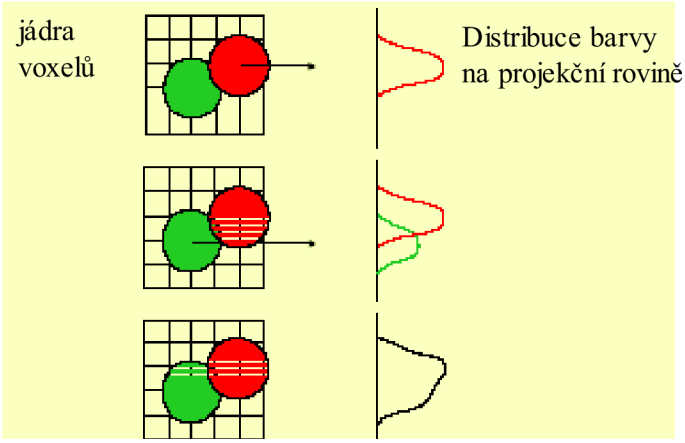
Při zápisu výsledné barvy lze, pomocí aplikace různých funkcí na vzorky podél paprsku, na stejná data nahlížet různě. Například zobrazovat jen voxely určité intenzity.

2.2.2 Voxel splatting

Lee Westover tuto metodu představil v roce 1989. Do češtiny by se vexel splatting dal přeložit jako "naplácávání voxelů". A skutečně toto je i principem metody. Vykreslování objemových dat pomocí voxel splattingu probíhá zpracováním řezů objemu zepředu dozadu ve směru pohledu. Daný voxel se promítne na rovinu obrazu, přičemž zanechá stopu s Gaussovým rozložením. Další parametry promítnutého voxelu je jeho barva, průhlednost, velikost atd.. Postup promítání voxelů je na obrázku 2.7 níže.



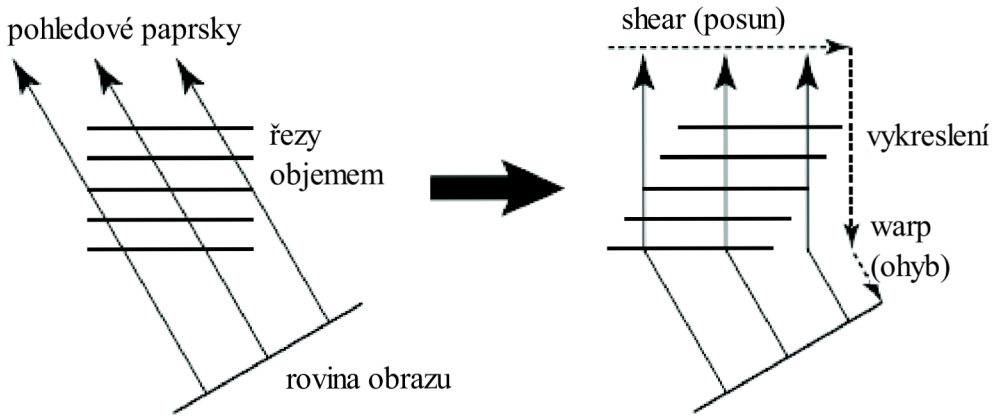
Obrázek 2.8: Princip voxel splattingu [14].



Obrázek 2.7: Stopy voxelů na projekční rovině [14].

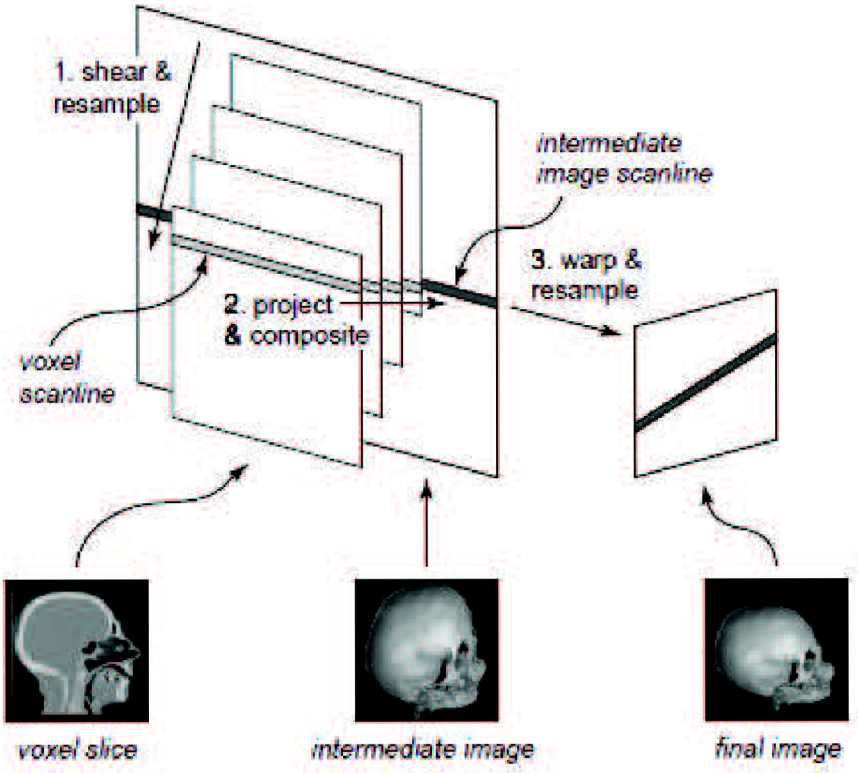
2.2.3 Shear warp

Tato metoda, kterou zpopularizovali Lacroute a Levoy, pracuje na principu posunu řezů objemu tak, aby paprsky směřovaly stále kolmo na tyto řezy viz obrázek 2.9.



Obrázek 2.9: Transformace objemových řezů při paralelní projekci [5].

Na obrázku 2.9 výše je uvedena paralelní projekce. Pokud by čtenáře zajímala i perspektivní projekce, nalezne její popis v [5]. Jen nastíním, že princip je zhruba stejný, jen se k posunu řezů přidá ještě jejich zvětšení či zmenšení.



Obrázek 2.10: Tři základní kroky algoritmu shear warp [5].

Při vykreslování bodů na obrazovku je jim opět nastavena barva, průhlednost, překrývání, normály atd., podobně jako u předchozí metody. Na obrázku 2.10 výše jsou tři hlavní kroky, které algoritmus shear warp při vykreslování objemových dat dělá:

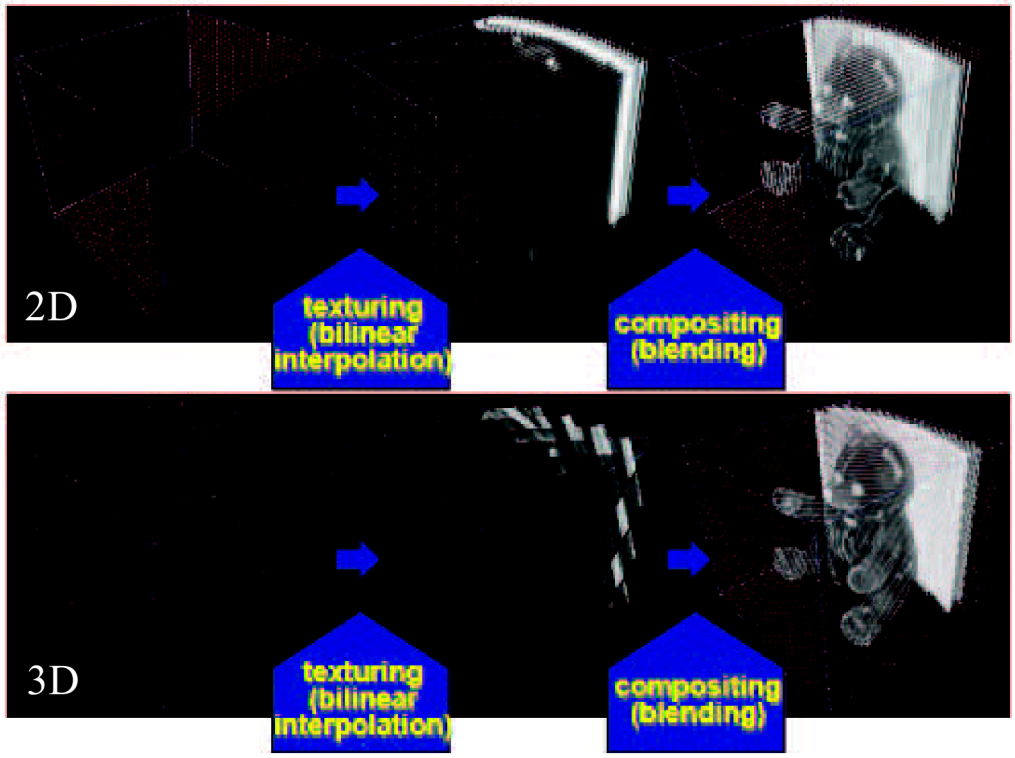
1. Posun a převzorkování řezů.
2. Vkreslí převzorkovanou a právě zpracovávanou řadu voxelů do dočasné řady.
3. Zalomí dočasnou řadu a vykreslí ji na rovinu obrazu.

2.2.4 Texture mapping

Tento postup se snaží využít grafický akcelerátor k vykreslení volumetrických dat za pomoci texturování polygonů a blendingu (technika OpenGL, která v texturách definuje průhlednost). Existují dvě varianty:

1. 2D texture mapping – používá 2D textury k uchovávání dat. Tato varianta je paměťově náročná, protože si potřebuje v paměti uchovávat tři kopie objemových dat „nařezaných" podle os x, y a z. Vykreslování pak probíhá otexturováním polygonů jednotlivými řezy ve směru nejvíce přivrácené strany. Obrázek 2.11 nahoře.
2. 3D texture mapping – k uchování dat využívá 3D texturu. Oproti 2D přístupu jsou objemová data v paměti uloženy jen jednou. Vykreslování pak probíhá otexturováním polygonů kolmých na směr pohledu. Obrázek 2.11 dole.

Jednou z největších výhod texture mappingu je rychlost vykreslování. Dokáže zachovat interaktivní práci s objektem i při větších rozměrech dat.



Obrázek 2.11: 2D (nahoře) a 3D (dole) texture mapping [14].

2.3 Metody zobrazení

Jak již bylo zmíněno v kapitole 2.2.1 je při přímém volume renderingu možné při zobrazování dat aplikovat různé metody a tím i na data nahlížet různě. Lze je rozdělit do dvou základních skupin. Na metody nehledající povrchy a na metody zobrazující povrchy.

V této kapitole se zaměřím na metody jednoduše použitelné při zobrazování dat pomocí volume ray castingu, jehož implementace je cílem této práce. Přehled metod lze nalézt v [1]. Uvedu zde jen několik z nich a budou to ty, které jsem použil při návrhu aplikace.

2.3.1 Metoda MIP

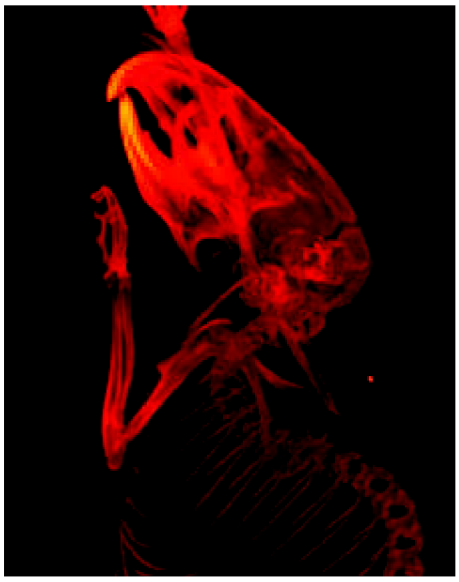
Metoda MIP (maximum intensity projection) hledá na dráze paprsku voxel s největší intenzitou. Intenzita pixelu na projekční rovině je dána vztahem [1]:

$$I = \max_{i \in J}(I_i) \tag{1}$$

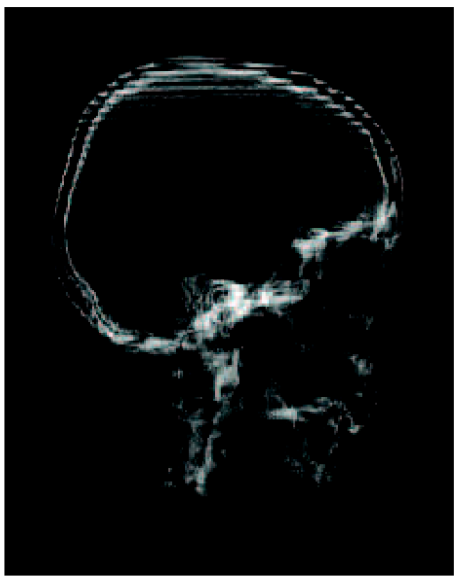
kde I je výsledná intenzita, J je množina voxelů na dráze paprsku a I_i je intenzita i-tého voxelu.

Tato metoda je především vhodná pro rychlý náhled, protože je výpočetně velmi jednoduchá. Ovšem výsledné zobrazení nám poskytuje málo informace na to, abychom si dokázali představit umístění zobrazovaných dat v prostoru. Proto se často vytváří animace rotovaných dat, které dokáží zlepšit naši 3D představu zobrazovaného objektu.

MIP se používá k detekci plicních uzlin na snímcích pořízených výpočetní tomografií [15].



Obrázek 2.13: Objemová data vizualizované pomocí MIP[15].



Obrázek 2.12: CT data pomocí metody SIP.

2.3.2 Metoda SIP

Jas pixelu získaný touto metodou se vypočítá jakou součet intenzit voxelů podél paprsku, které leží v zadaném intervalu. Zkratka pro tuto metodu (SIP) je odvozena z anglického „summed intensity projection“. Vzoreček pro výpočet má tvar [1]:

$$I = \sum_{i \in J} (I_i) \tag{2}$$

kde význam proměnných je stejný jako u metody MIP.

Při aplikaci této metody se snažíme o napodobení snímku pořízených rentgenem. Srovnáme-li postup jakým je pořízen rentgenový snímek s metodou SIP nalezneme podobnost. Při ozáření objektu krátkými rentgenovými pulsy dopadají nepohlcené paprsky na radiografický film. V místech objektu s větší elektronovou hustotou se záření pohltí. Po vyvolání filmu odpovídají černá místa vyšší expozici (např. tkáním) a bílá místa naopak nižší (např. kostem). Podobně, čím více bude na dráze paprsku voxelů hledané intenzity, tím bude výsledek v tomto místě světlejší, protože výsledná barva je dána jejich sumou.

2.3.3 Osvětlení povrchů

Jak jsem zmínil na začátku kapitoly, metody zobrazování jde rozdělit do dvou skupin, nyní se budeme zabývat tou druhou – metodami hledajícími povrch objektů v objemových datech. Díky metodám hledající povrch a použitím osvětlovacího modelu si lze daleko lépe představit prostorovou strukturu zobrazovaných dat a přidávají na realističnosti zobrazení.

Na objemová data lze aplikovat lokální či globální osvětleni. Ve zbytku kapitoly se budeme zabývat lokálním osvětlením. Více informací o způsobech globálního osvětlení lze nalézt v [16].

Tradiční lokální osvětlovací modely pro osvětlení povrchů mohou být aplikovány i na objemovou reprezentaci. Lokální osvětlovací modely používají normálových vektorů, které popisují lokální orientaci povrchu. Osvětlovací modely počítají odražené světlo jako funkci normály, směru pohledu, úhlu dopadu paprsku a materiálových vlastností povrchu. Téměř všechny osvětlovací modely mohou být použity v zobrazování objemových dat substitucí normály povrchu za normalizovaný gradient povrchu objektu v objemu [16].

Při výpočtu gradientu lze postupovat dvěma cestami. Gradienty si přepočítat a uložit k objemovým datům a nebo gradient počítat při vykreslování pro každý bod za „běhu“. Hlavní rozdíl mezi těmito metodami je ten, že přepočítané gradienty jsou spočítány v mřížce o velikosti originálních dat a při vykreslování jsou gradienty interpolovány, zatímco v druhém případě jsou gradienty spočítány již z interpolovaných hodnot, například ve fragment shaderu.

Způsobů odhadnutí gradientu z objemových dat je hned několik a jsou uvedeny v [1] či [16]. Nejpoužívanější přístup (použil jsem jej i při návrhu) je metoda central differences [16]. Metoda centrálních rozdílů spočítá složky třírozměrného vektoru gradientu. Výpočet probíhá podle vzorce [1]:

$$\nabla f(x,y,z) \approx \begin{pmatrix} f(x+h,y,z) - f(x-h,y,z) \\ f(x,y+h,z) - f(x,y-h,z) \\ f(x,y,z+h) - f(x,y,z-h) \end{pmatrix} \tag{3}$$

Ze vzorce vidíme, že pro každý bod, ve kterém chceme spočítat gradient povrchu, je potřebných 6 okolních vzorků ve vzdálenosti h od daného bodu. Hodnota h je zvolena s ohledem na vzdálenost dat v objemové mřížce. Při výpočtu gradientu předem je h rovno této hodnotě, při výpočtu za běhu je h zvoleno někde kolem této hodnoty. Při implementaci je potřeba počítat i s tím, že gradient v místě s malou variabilitou dat může být (0,0,0).

Nyní, když už je znám výpočet normály povrchu, zaměříme se na lokální osvětlovací model. Ve zbytku kapitoly si rozebereme osvětlovací model Blinn-Phong. Jedná se o upravený Phongův

osvětlovací model. Je to model používaný v OpenGL a pro některé materiály dává lepší výsledky [19]. Podobně jako u Phongova osvětlovacího modelu se světlo odražené od objektu vypočítá jako součet tří složek: ambientní, difusní a spekulární [16]:

$$I = I_{ambient} + I_{difuse} + I_{specular} \tag{4}$$

Ambientní osvětlení je světlo, které bylo rozptýleno prostředím natolik, že nelze určit jeho směr [18]. Je to osvětlení, které vnímáme při pohledu na nepřímo osvětlené objekty. I přesto, že objekt není přímo osvětlen, vnímáme jeho barvu a povrchovou strukturu. Bez ambientního osvětlení by se každý nepřímo osvětlený objekt jevil jako černý, což vypadá nerealisticky. V lokálních osvětlovacích modelech se ambientní osvětlení nahrazuje konstantní hodnotou. Ambientní světlo snižuje kontrast a dynamický rozsah obrázku [16]. Výpočet ambientní složky [16]:

$$I_{ambient} = k_a M_a I_a \tag{5}$$

kde k_a je konstanta, která uvádí, kolik z dopadajícího ambientního světla je odraženo, M_a je ambientní barva materiálu objektu a I_a je intenzita ambientního osvětlení.

Difuzní složka je světlo, které přichází na povrch z jednoho směru, z jednoho konkrétního zdroje světla [18]. Po dopadu na povrch se rovnoměrně rozptýlí do všech směrů. Takto osvětlený povrch se jeví jako matný a stejně osvětlený, ať se díváme z jakéhokoliv směru. Světlost povrchu závisí pouze na úhlu dopadu φ mezi směrem dopadajícího světla L a normály povrchu N [16], obrázky 2.14 a 2.15. Výpočet difuzní složky [16]:

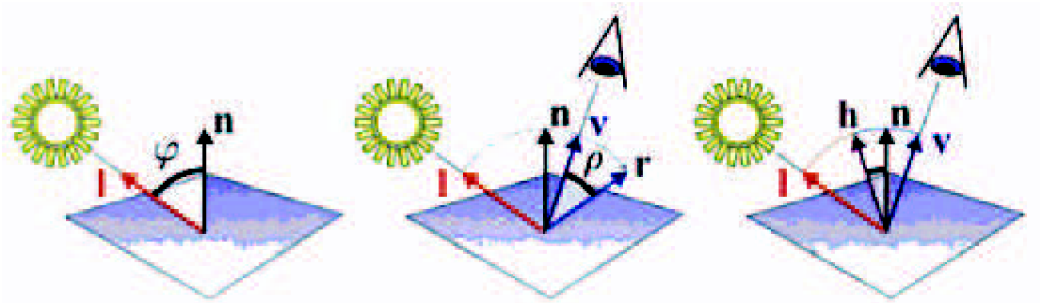
$$I_{difuse} = k_d M_d I_d \max(L \cdot N, 0) \tag{6}$$

k_d, M_d a I_d podobně jako u ambientní složky, akorát pro složku difuzní. Maximum je uvedeno z důvodu, pokud by byl úhel mezi L a N větší jak 90 stupňů.

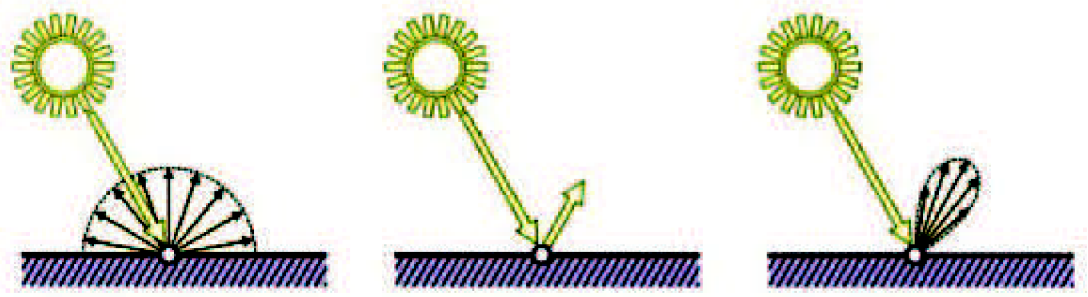
Spekulární složka přidává osvětlenému materiálu dojem lesklosti a je závislá na směru paprsku odraženého od povrchu k pozorovateli. Čím je úhel mezi odraženým paprskem a paprskem dopadajícího světla menší, tím je spekulární složka výraznější. Při výpočtu se odražený paprsek nahrazuje vektorem H v polovině mezi V a L, což je rozdíl oproti Phongovu modelu [16]:

$$I_{specular} = k_s M_s I_s \max(H \cdot N, 0)^n \tag{7}$$

k_s, M_s a I_s podobně jako u ambientní složky s významem pro spekulární složku, N je normála povrchu, H je poloviční vektor mezi V a L a n je parametr materiálu, který kontroluje velikost odlesku.

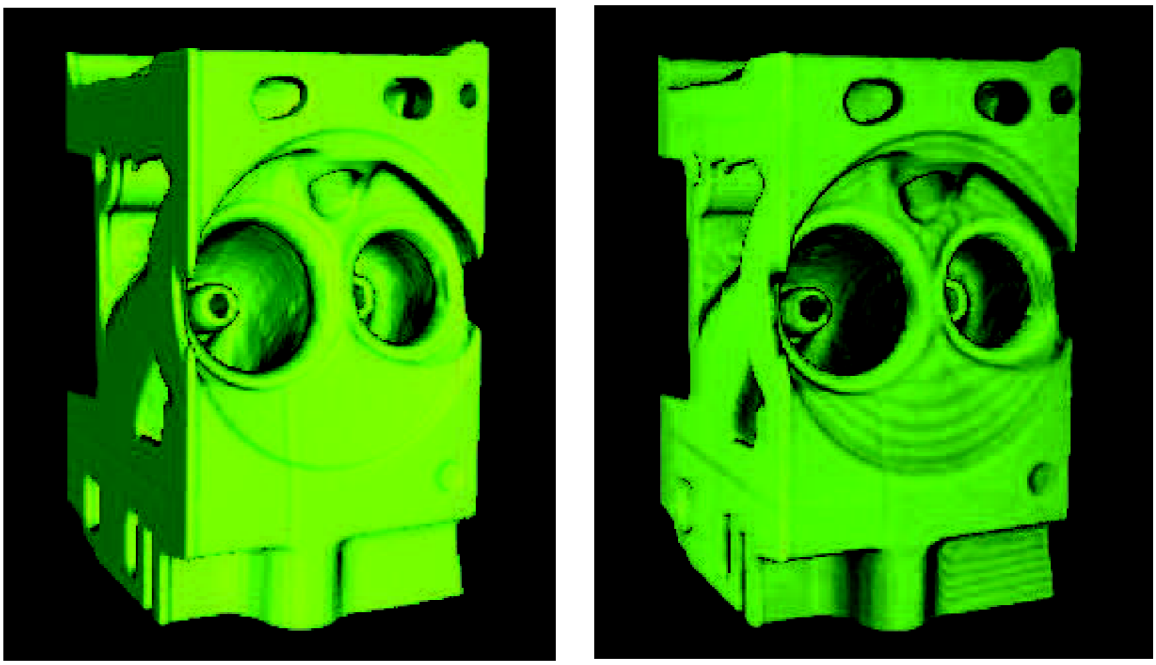


*Obrázek 2.14: Schéma difuzního osvětlení (vlevo), Phongův model spekulárního osvětlení založený na úhlu **ρ** mezi odraženým paprskem **r** a směru pohledu **v** (uprostřed) a spekulární složka Blinn-Phong modelu závisejícího na úhlu mezi normálou **n** a vektorem **h** (vpravo) [16].*



Obrázek 2.15: Způsoby odrazu: všemi směry stejným způsobem (vlevo), perfektní odraz v jednom směru (uprostřed) a spekulární odraz ve směru prefektního odrazu (vpravo) [16].

Závěrem kapitoly ještě uvedu, přímý rozdíl (obrázek 2.16) mezi zobrazovací metodou, která osvětluje povrch objektu objemových dat a metodou, která zobrazuje data s vhodně nastavenou přenosovou funkcí (kapitola 2.4).



Obrázek 2.16: Vizualizace objemových dat pomocí osvětlení povrchů (vlevo) a pomocí vhodně zvolené přenosové funkce (vpravo).

2.4 Přenosová funkce

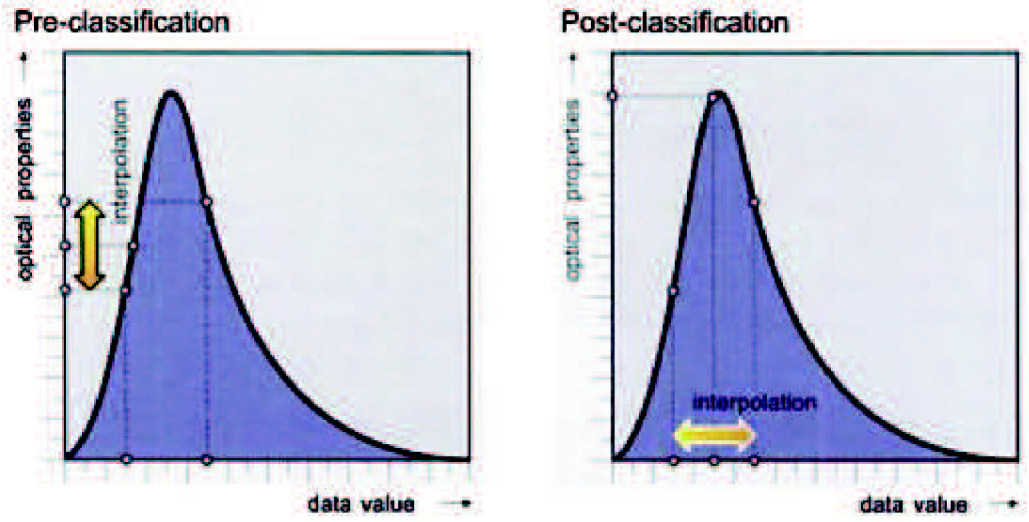
Při zobrazování objemových dat metodami MIP a SIP jsme uvažovali pouze hodnotu hustoty (intenzity) zaznamenanou při pořizování CT. K tomu, abychom mohli simulovat skutečný průchod světelného paprsku objemovými daty, potřebujeme znát optické vlastnosti, jako vyzařovací a absorpční koeficienty v každém místě uvnitř objemu. Tuto informaci však nejsme schopni z objemových dat přímo získat. Je tedy na uživateli, aby intenzitám voxelů přiřadil jejich optické vlastnosti a tím definoval, jak by měly ve výsledku vypadat. Toto mapování se nazývá přenosová funkce a proces nalezení odpovídající přenosové funkce se nazývá klasifikace.

V kontextu vizualizace objemových dat je klasifikace definována jako: proces identifikace příznaků zájmu založených na abstraktních hodnotách dat [16]. Obvykle je to funkce, která určitou skupinu vstupních dat transformuje na hodnoty barev RGB a alfy (průhlednosti). Tato funkce se nejčastěji reprezentuje pomocí 2D vyhledávací (look-up) tabulky. Přenosové funkce se zdokonalují a ty pokročilejší dokáží materiálu přiřadit vlastnosti jako průsvitnost či odrazivost. Více o těchto pokročilých přenosových funkcích je uvedeno v [16].

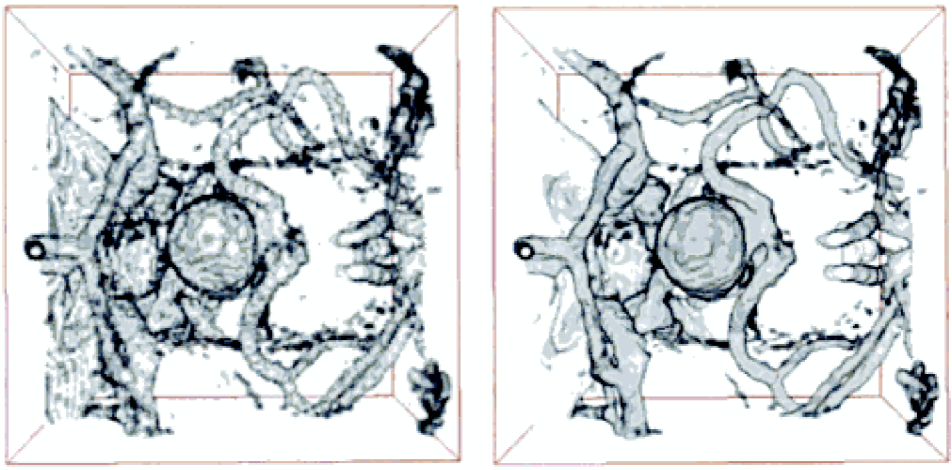
Návrh přenosové funkce je ruční, zdlouhavá práce a vyžaduje detailní znalosti prostorových struktur vyskytujících se v objemu. Z tohoto důvodu je důležitá nejlépe okamžitá zpětná vazba, kdy uživatel hned vidí, zda postupuje správným směrem. Bohužel přenosové funkce ve většině případů nelze aplikovat na více dat ani stejného typu. Většinou je potřebná aspoň minimální úprava. Za zmínku stojí i to, že existují přístupy, jak automaticky generovat přenosové funkce. Více v [16].

Přenosová funkce může být použita přímo na diskrétní data, ještě před rekonstrukcí struktury zachycené v datech (angl. pre-interpolative) nebo až po rekonstrukci (angl. post-interpolative). Rozdíl mezi těmito metodami zachycuje obrázek 2.17. Pre-interpolative přenosové funkce mapují barvu z 2D tabulky před nebo během rasterizace a tedy interpolují barvu z 2D tabulky pro daná skalární data. Naopak Post-interpolative přenosové funkce nejprve interpolují skalární data a snaží se zrekonstruovat původní spojitý signál a až poté jim přiřadit barvu (optické vlastnosti) z 2D tabulky. Srovnání, která metoda je lepší si lze udělat pohledem na obrázek 2.18.

Cílem přenosové funkce je oddělit objekty uvnitř objemových dat v závislosti na jejich skalární hodnotě. Protože jsou data vzorkovány s určitou přesností (konečnou), hranice mezi objekty (odpovídají vysokým frekvencím) nejsou zachyceny s dostatečnou přesností a tedy i při rekonstrukci pomocí přenosové funkce dochází k nepřesnostem (artefaktům) a to především na hranicích objektů. Detailněji je tento problém rozebrán v [16].



Obrázek 2.17: Rozdíl mezi přiřazením optických vlastností před (vlevo) a nebo až po rekonstrukci signálu (vpravo) [16].



Obrázek 2.18: Rozdíl mezi klasifikací vzorků před rekonstrukcí signálu (vlevo) a po (vpravo). Oba obrázky byly generovány se stejnou přenosovou funkcí a za stejných podmínek [16].

2.4.1 Před-integrovaná přenosová funkce

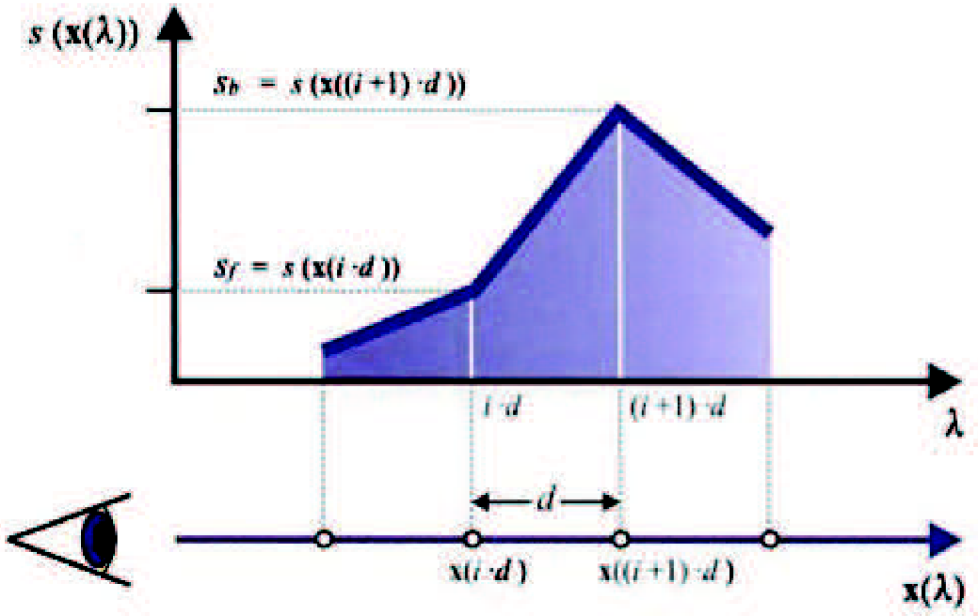
Jak bylo zmíněno výše při vizualizaci objemových dat mohou vznikat artefakty na hranicích objektů zapříčiněnými vysokými frekvencemi. Do jisté míry lze toto odstranit a to přidáním počtu vzorků na dráze paprsku, tedy zmenšení kroku vzorkování objemových dat. Do jisté míry to zlepší obrazový vjem, ovšem na úkor výpočetní náročnosti – obrázek 2.19.

Obrázek 2.19: Objemová data vizualizovaná pomocí 128 řezů a post-klasifikace (vlevo), 284 řezů a post-klasifikace (uprostřed) a 128 řezů a před-integrované přenosová funkce (vpravo) [17].

Řešením, jak potlačit artefakty, je použití před-integrované přenosové funkce. Jejím cílem je spojit přenosovou funkci s co nejpřesnější aproximací volume rendering integrálu, tedy jak se daný světelný paprsek zachová na určitém úseku objemových dat. K názorné představě může posloužit obrázek 2.20. Paprsek vzorkuje objemová data v krocích vzdálených o d. Výsledkem jsou v každém kroku hodnoty s_f a s_b. Nás nyní zajímá, jakou hodnotu má volume rendering integrál mezi těmito dvěma body. Přesný výpočet je uveden v [16]. Důležité však je, že se tyto hodnoty dají před počítat a výsledkem je 2D tabulka pro všechny možné hodnoty s_f a s_b, které se mohou v objemových datech

vyskytnout. Pro představu u 8-bitových dat je velikost tabulky 256x256 hodnot, u 12-bitových 4096x4096 hodnot.

Před integrovaná přenosová funkce slouží k přesnější aproximaci volume rendering integrálu při zachování stejného počtu vzorků na dráze paprsku.



Obrázek 2.20: Schéma určení barvy a průhlednosti i-tého segmentu paprsku [16].

2.5 Ray casting pomocí GPU

V posledních pár letech se ray casting pomocí GPU dostal do popředí metod používaných pro zobrazování objemových dat. Je to především proto, že v minulosti grafické karty nenabízely takovou funkcionalitu, aby bylo možné v nich ray casting efektivně implementovat. S nástupem moderních grafických karet (rychlých, programovatelných a s velkou pamětí) se tato situace obrátila.

Grafické karty poskytují určité výhody pro real-time zobrazování objemových dat. Mezi hlavní patří možnost programovat vykreslovací řetězec krátkými programy, které jsou vykonávány rychle a efektivně přímo v čipu grafické karty (shadery). Algoritmus ray castingu je jednoduchý a proto se přímo k implementaci nabízí. Další výhodou je rychlá a s přibývajícími roky stále se zvětšující dedikovaná paměť, která umožňuje rychlé čtení a zpracování objemových dat.

Výkon volume renderingu naprogramovaného pomocí shaderů GPU je ovlivňován více stran a nalezení optima mezi rychlostí a kvalitou je netriviální problém. Kvalita výsledného zobrazení je většinou určující a rozhoduje o tom, zda daný systém bude mít úspěch či nikoliv. Chceme-li dosahovat kvalitního až realistického zobrazení potřebujeme přesnější (více bitů na vzorek) a rozměrově větší data, která zabírají více paměti, ovšem nabízí možnost lepší rekonstrukce původního objektu. S přibývajícími daty se při vykreslování zvyšuje počet přístupů do paměti a ten nepříjemně ovlivňuje výkon celého systému. V literatuře [16] je tato problematika rozebrána do větších detailů. Autoři se zde zabývají konkrétní grafickou kartou a vysvětlují, jak uložení dat může

ovlivnit rychlost zobrazování. Výsledky jsou zajímavé, ovšem typ použité grafické karty (NVIDIA GeForce 6800 GT PCIe x16) se v době psaní této práce už stal téměř minulostí, a výsledky pro současné karty mohou být odlišné. Nicméně závěr z načerpaných informací je významný. Znalost hardware, na kterém bude výsledná aplikace spouštěna, je velmi důležitá.

Způsob uložení dat není jediným faktorem ovlivňující konečnou rychlost systému. V následující kapitolách se pokusím vysvětlit hlavní myšlenku a přínos několika dalších vylepšení k dosažení zrychlení vykreslování.

2.5.1 Předčasné ukončení paprsku

V anglické literatuře nazývané early ray termination, je velmi jednoduchá optimalizace. Když prochází paprsek objemem (odpředu dozadu), akumuluje v sobě výslednou barvu daného pixelu výsledného obrazu. Barva může být složená ze 4 složek: RGBA. Při této optimalizaci nás nejvíce zajímá složka čtvrtá – průhlednost. Pokud při průchodu nabude stavu plné neprůhlednosti, můžeme vzorkování podél paprsku ukončit, protože budoucí vzorky nebudou mít na barvu výsledného pixelu žádný vliv a počítají se zbytečně.

2.5.2 Vyřazení dat mimo zájem pozorovatele ze zpracován

Uvažujme situaci, kdy v objemových datech máme na dvou různých místech data stejného typu. Například vlevo-vpravo, nahoře-dole atp. Uživatele ovšem zajímají jen data kupříkladu vlevo. Proč by se tedy měly vykreslovat data vpravo a konzumovat tak výkon grafické karty? Vyřadíme je tedy ze zpracování a to tak, že omezíme délku a počátek generovaného paprsku pro ray casting. Některé paprsky, v závislosti na úhlu pohledu a výběru dat, nemusíme dokonce vůbec generovat.

2.5.3 Snížení rozlišení vykresleného snímku

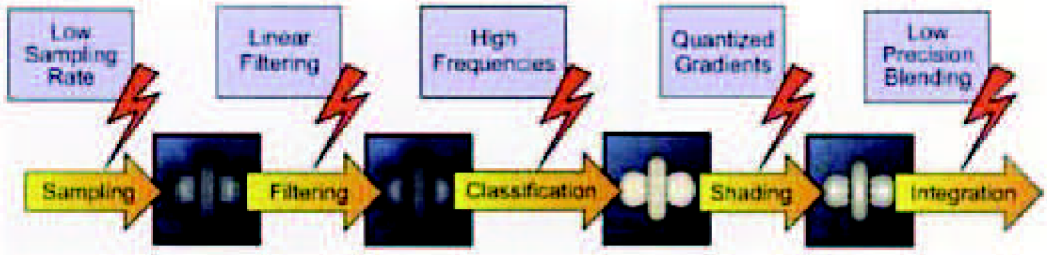
Předchozí dvě zmíněné optimalizace přinášely zrychlení zobrazování při zachování stejné kvality. V případech, kdy nemáme k dispozici tak výkonný hrardware a předčasné ukončení paprsku ani vyřazení dat mimo zájem pozorovatele rychlost vykreslování neurychlí na přijatelnou úroveň, se nám nabízí snížení rozlišení vykresleného snímku nebo snížení počtu vzorků na dráze paprsku. Cílem obou metod je snížit počet přístupů do paměti a udělat to nezávisle na promítaných datech a podle potřeby grafické karty až na snesitelný počet snímků za vteřinu. Pokud snížíme rozlišení snímku na polovinu, např.: z 512 x 512 bodů na 256 x 256 bodů, klesne počet vržených paprsků raytracingu na čtvrtinu, což už je výrazné odlehčení grafickému hardwaru. Výsledný obraz můžeme zpětně zvětšit na původní velikost a tím dosáhnout rozumného poměru mezi kvalitou a rychlostí zobrazování.

Snížení počtu vzorků na dráze paprsků je drastičtější a vede ke vzniku artefaktů spojených s podvzorkováním viz následující kapitola 2.5.4. Úpravu na správný počet vzorků můžeme provést vždy při zastavení manipulace s objemovými daty a vygenerovat tak korektní snímek splňující vzorkovací teorém.

2.5.4 Artefakty

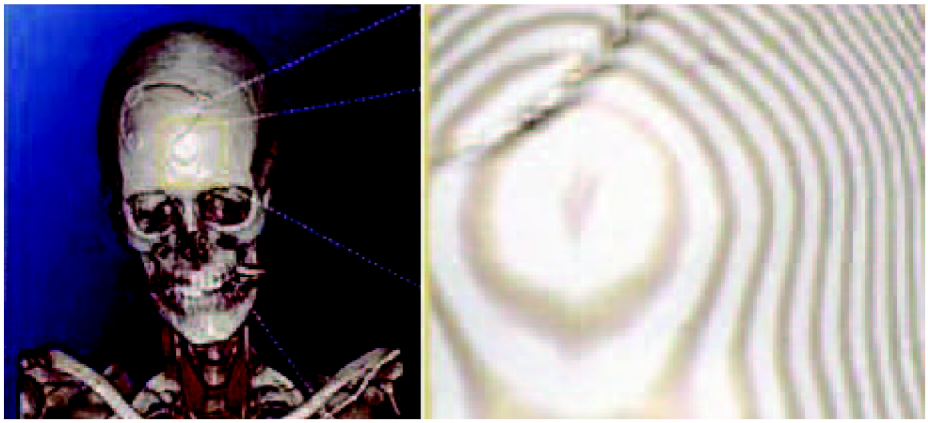
Artefakty jsou nepřesnosti, kdy některé detaily nejsou zobrazeny, nebo se naopak objeví tam, kde v datech nejsou [1]. Tyto nepřesnosti se mohou v procesu vykreslování objemových dat objevit hned

na několika místech. Samozřejmě, že artefakty jsou nechtěné a chceme, aby jich bylo ve výsledném obrazu co nejméně a pokud to jde, tak žádné. Vyznačení míst vzniku artefaktů je prvním krokem k jejich odstranění. Obrázek 2.21 názorně ukazuje místa vzniku artefaktů a zároveň udává jejich příčinu. V následujícím textu je postupně rozeberu a uvedu možnosti jejich odstranění.



Obrázek 2.21: Řetězec volume renderingu a místa vzniků artefaktů [16].

Na začátku ray castingu jsou vrženy paprsky skrz objemová data. Vzdálenost mezi navzorkovanými body ovlivňuje přesnost s jakou jsme schopni zobrazit objemová data. Nízká vzorkovací frekvence (velké vzdálenosti mezi vzorky) vedou k artefaktům zvaným wood-grain [16], obrázek 2.22. Na otázku jaká je správná vzorkovací frekvence dává odpovědět Nyquist-Shannonův vzorkovací teorém. V procesu převodu analogového signálu na digitální (vytvoření snímků CT) je podle teorému nutné, aby vzorkovací frekvence byla alespoň dvakrát vyšší než je maximální frekvence v původním signálu. Pak je možné původní signál z navzorkovaných dat zrekonstruovat přesně. Fázi vzorkování většinou nejsme schopni ovlivnit a pracujeme již s daty, které někdo pořídil a počítáme, že při tomto procesu došlo k jisté ztrátě informace. Pokud budeme uvažovat, že nejvyšší frekvence, která se může v datech vyskytnout je *1 / nejmenší vzdálenost mezi voxely*, pak pro přesnou rekonstrukci signálu z diskrétních dat potřebujeme pořídit aspoň dva vzorky na nejmenší vzdálenosti mezi voxely. Větší počet vzorků má dopad na výkon systému. Některé techniky urychlení byly uvedeny v kapitolách 2.5.1 – 2.5.3.



Obrázek 2.22: Wood-grain artefakty zapříčiněné nízkou vzorkovací frekvencí [16].

Postoupíme-li v řetězci volume renderingu k dalšímu kroku, dostaneme se k filtrování – fáze, kdy se podle navzorkovaných dat rekonstruuje původní signál. Hodnoty se získávají aplikováním diskrétní konvoluce se zadaným jádrem filtru. Bylo dokázáno, že nejlepší jádro poskytuje filtr sinc [16]. Ten ovšem implementovaný přímo v grafických kartách nenajdeme. Máme tedy dvě možnosti, buď použijeme filtry vestavěné v grafických kartách (trilineární – pro 3D textury) a nebo se pokusíme

implementovat filtr vlastní v shaderech grafické karty. Vlastní implementace filtru znamená další výpočty navíc a tím i snížení rychlosti vykreslování.

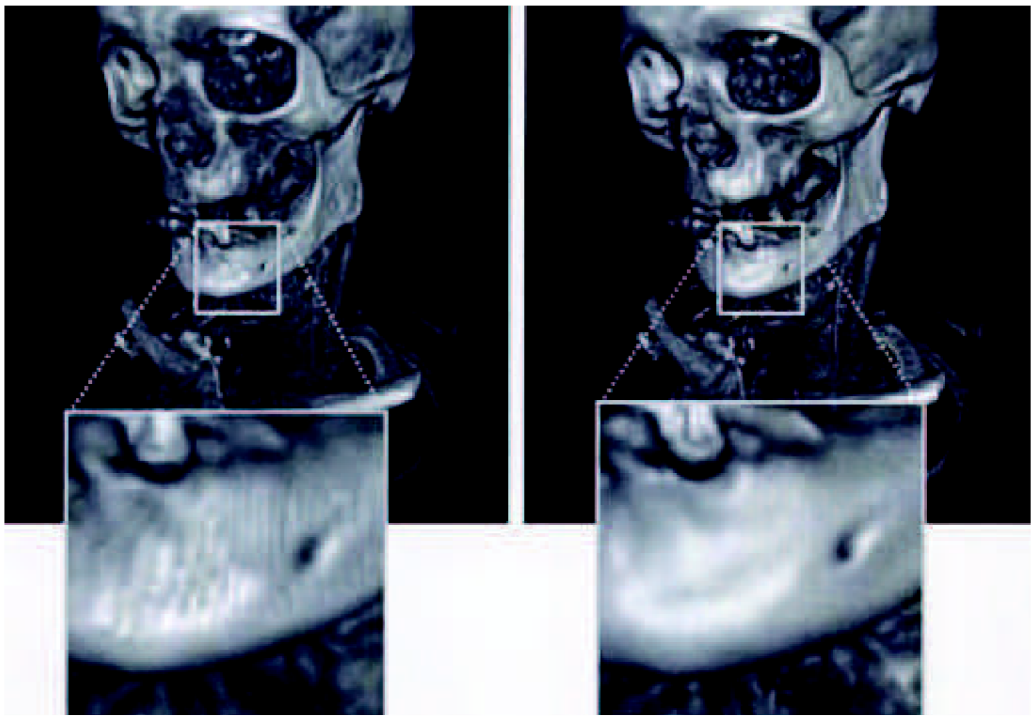
V pořadí třetí případný zdroj artefaků je klasifikace. Přiřazuje objektům optické a materiálové vlastnosti. Má tedy velký na to, co na konečném snímku vidíme. Klasifikací a přenosovými funkcemi se zabývá kapitola 2.4, kde jsou zmíněny i okolnosti vzniku artefaktů.

Nalezení a osvětlení povrchů objektů v objemových datech podporuje naši představu, že se díváme na skutečný trojrozměrný objekt. Při osvětlení povrchů je určující výpočet gradientu (kapitolo 2.3.3 osvětlení povrchů). Z pohledu vzniku artefaktů je lepší výpočet za „běhu“, kdy se gradient počítá pro filtrovaná data a s dostatečnou přesností. Přináší to s sebou opět nárůst výpočetní náročnosti, ale výsledek je o poznání lepší (obrázek 2.23), nehledě na to, že se ušetří místo v paměti grafické karty, kterou by zabíraly před počítané gradienty.

Poslední možný bod vzniku artefaktu je způsob výpočtu výsledné barvy. Výpočet pro šíření paprsku zepředu dozadu se provádí pomocí vztahu [16]:

$$C_{dst} = C_{dst} + (1 - \alpha_{dst}) C_{src} \quad , \tag{8}$$

C_{dst} je postupně se akumulující barva pixelu, C_{src} je hodnota volume rendering integrálu pro úsek mezi vzorky a α_{dst} a α_{src} jsou alfa kanály obou barev. Pokud jsou hodnoty alfa kanálu velmi malé (velmi průhledné objekty), dochází při výpočtu k zaokrouhlování a to je důvod vzniku artefaků. Dnešní grafické karty už počítají s vyšší přesností a minimalizují tak výskyt tohoto typu artefaktu. Míra přesnosti závisí na typu grafické karty [16].



Obrázek 2.23: Srovnání před počítaných (vlevo) a za „běhu“ počítaných gradientů (vpravo) [16].

3 Návrh a implementace

Kapitolu návrhu a implementace aplikace jsem spojil, protože spolu úzce souvisí. Chtěl jsem, aby čtenář, který se zajímá nejen o návrh, ale i o implementaci nemusel mnoho listovat a pro danou část návrhu hned viděl, jak jsem ji v práci realizoval.

3.1 Požadavky na řešení

Cílem práce bylo navrhnout aplikaci s grafickým uživatelským rozhraním, která bude zobrazovat objemová data metodou ray castingu. Umožní načítání dat ze souboru a interaktivní práci se zobrazovanými daty (rotace, zvětšení, zmenšení, různé metody pohledu na data). Měla by být nezávislá na operačním systému a hardwarovém vybavením počítače.

3.1.1 Prostředky zvolené pro realizaci

Určujícím bodem pro výběr nástrojů pro implementaci návrhu byl požadavek na grafické uživatelské rozhraní (dále jen GUI). Má-li být přenositelné, bylo nutné zvolit knihovnu, která je multiplatformní. Další požadavek na knihovnu byl, aby podporovala OpenGL, která je standardním grafickou knihovnou pro tvorbu multiplatformních hardwarově akcelerovaných aplikací. Výsledkem požadavků byla knihovna Qt. Spolu s knihovnou Qt je distribuované vývojové prostředí, které při implementaci usnadňovalo práci. Při implementaci raycastingu budou využívány rozšíření OpenGL. Z tohoto důvodu je použita knihovna GLEW [12].

Implementačním jazykem byl zvolen jazyk C++, který je podporovaný všemi zmíněnými knihovnami. Pro implementaci shaderů jsem zvolil jazyk GLSL.

Instalace Qt toolkitu byla bezproblémová, jak se dnes u většiny vývojových prostředí dá očekávat. Knihovna GLEW je distribuovaná ve dvou verzích – jen zdrojové kódy nebo spolu s binárními soubory knihovny. V mém případě se binární soubory ukázaly jako nekompatibilní a bylo nutné je znovu zkompilovat. Kompilace je v přiloženém souboru dobře popsána a byla opět bezproblémová.

Do programového vybavení jsem ještě zařadil aplikaci glslDevil [21], která mi v začátcích pomáhala při vytváření a testování jednoduchých shader programů.

3.2 Cílová platforma

Při výběru nástrojů implementace byl kladen důraz na multiplatformní řešení. Aplikace by měla být spustitelná na počítačích s operačním systémem Windows, Linux a Mac OS X, které mají grafický akcelerátor podporující minimálně OpenGL verze 2.0 nebo vyšší. Při nižší verzi nejsou podporovány všechny potřebné rozšíření pro implementaci ray castingu pomocí shaderů. V době psaní práce OpenGL 2.0 podporují i integrované grafické karty v noteboocích.

Vývoj aplikace probíhal na notebooku s parametry:

Operační systém: Windows Vista SP2 32-bit

Procesor: Intel T4200 @ 2.0 Ghz

Operační paměť: 2 GB

Grafická karta: Intel Mobile 4 Series, GPU GM 45, 32 MB RAM

Druhou sestavou, na které probíhalo testování aplikace, byla:

Operační systém: Window Vista SP2 32-bit SP2

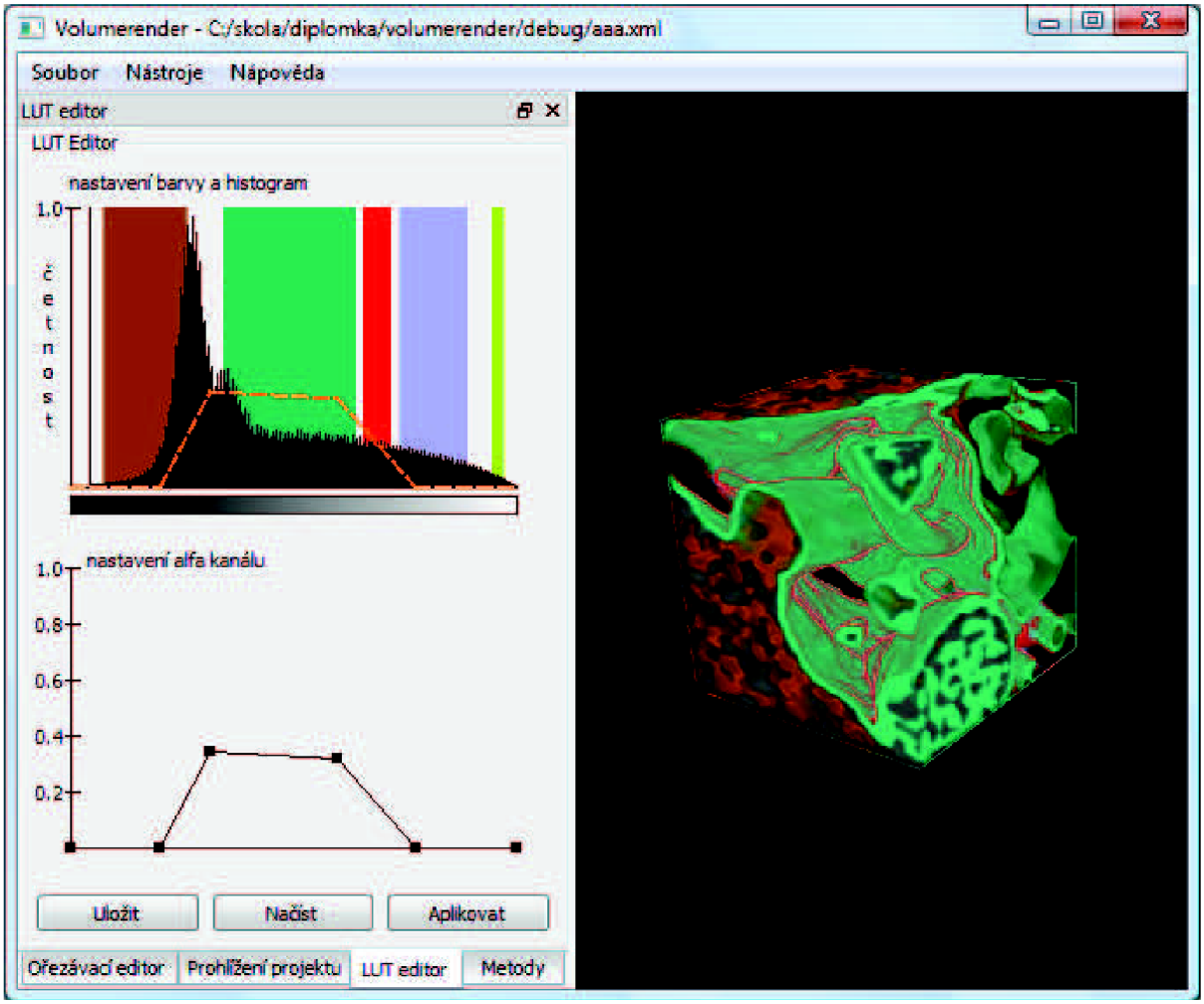
Procesor: AMD Athlon 64 X2 4200+ @ 2.2 Ghz

Operační paměť: 2GB

Grafická karta: NVIDIA GeForce 8600 GT, GPU G84, 256 MB RAM

3.3 GUI

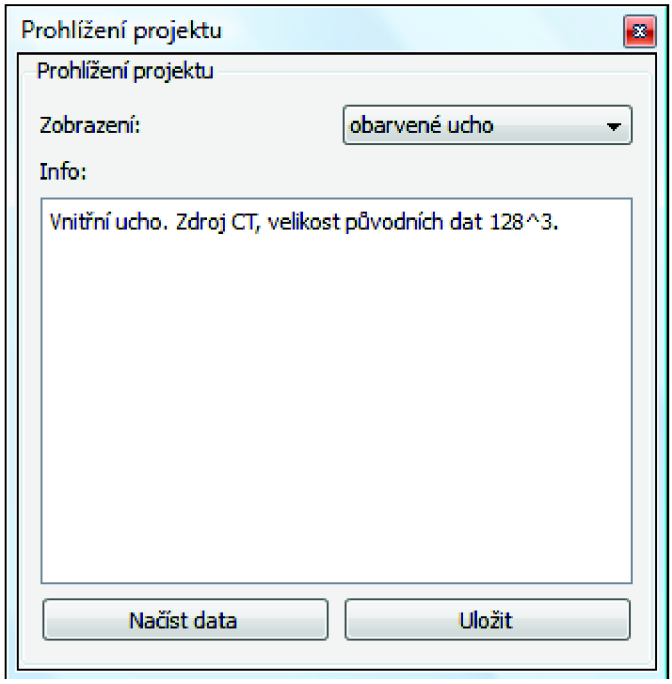
V počátku vývoje aplikace bylo GUI implementované pomocí knihovny Qt velmi jednoduché. Skládalo se z komponenty zobrazující výsledek volumerenderingu a menu, kde se daly přepínat metody zobrazení. S postupujícím vývojem narůstala potřeba zadávat více parametrů a nakonec jsem dospěl k podobě GUI, jenž je na obrázku 3.1. Skládá se ze dvou částí – panelu s nástroji a komponenty zobrazující výsledek volumerenderingu. Panel s nástroji lze z okna vyjmout v podobě samostatných oken a tím vytvořit větší prostor pro vyrenderovaná data. V následujících kapitolách karty a komponentu zobrazující výsledek volumerenderingu popíši.



Obrázek 3.1: Návrh grafického uživatelského rozhraní aplikace.

3.3.1 Karta prohlížení projektu

Ukládání parametrů metod zobrazení jsem se rozhodl ukládat jako projekty. Každý projekt se vztahuje k právě jedněm objemovým datům, nad kterými lze vytvořit libovolný počet zobrazení s využitím jedné ze čtyř implementovaných metod (MIP, SIP, LUT a Gradientní stínování). Ke každému zobrazení lze připojit komentář, kde si uživatel může například poznamenat, v čem má napříště pokračovat. Uživateli je poskytnuta i možnost vyměnit objemová data pro celý projekt. Tím si lze vyzkoušet, zda by se již nějaké vytvořené zobrazení dalo využít pro prohlížení jiných dat.



Obrázek 3.2: Karta prohlížení projektu volumetrického zobrazení.

Projekty jsou ukládány do souborů ve formátu XML. Struktura dokumentu je zvolená tak, aby zasvěcený uživatel byl schopen editovat zobrazení i pomocí úpravy atributů v XML souboru.

Navržená struktura dokumentu:

```
<?xml version="1.0" encoding="UTF-8"?>
<volumerender_projekt volume_data="CTA_inOhr_1_128_char.raw">
  <nastaveni_zobrazeni jmeno="obarvené ucho">
    <info>Vnitřní ucho. Zdroj CT, velikost původních dat 128^3.</info>
    <metoda>LUT</metoda>
            .
            .
            .
  </nastaveni_zobrazeni>
</volumerender_projekt>
```

Hodnota elementu metoda je závislá na použité metodě zobrazení a podobně i oblast tří teček. Této části konfiguračního souboru se věnuje kapitola 3.3.2.

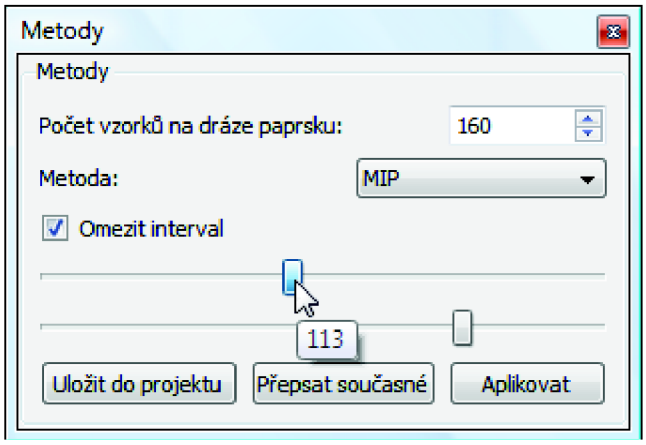
Tato i ostatní karty jsou implementovány jako instance třídy `QDockWidget` s vlastností `Qt::AllDockWidgetAreas`, která umožňuje „připnout“ kartu na všechny strany okna aplikace. Rozmístění komponent je pomocí třídy `QGridLayout`.

3.3.2 Karta metody

Karta metody sdružuje nastavení zobrazovacích metod. Viditelnost elementů pro zadávání parametrů je závislá na zvolené metodě, až na pole počtu vzorků, který je společný všem metodám. Pomocí této karty se do projektu vkládají nebo přepisují zobrazení. Aplikace nového nastavení se projeví až po stisknutí tlačítka `Aplikovat`, kdy se parametry předají shader programu.

Podoba karty pro metodu MIP je na obrázku 3.3. Pokud chce uživatel pro metodu MIP použít jen určitý interval z rozsahu 8 bitových vstupních dat (0 – 255), může to provést pomocí táhel. Část XML souboru pro metodu MIP:

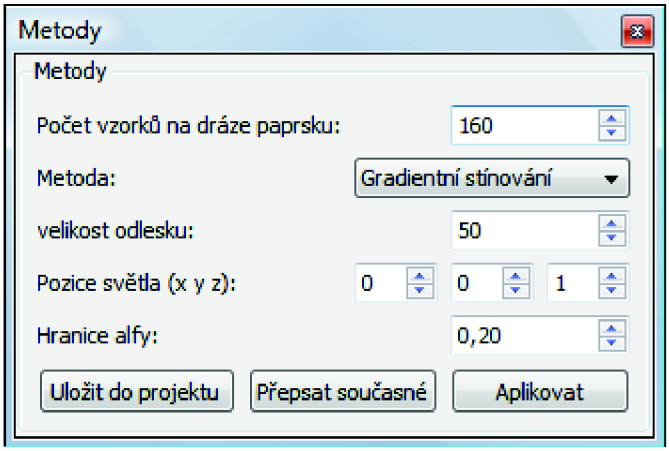
```
<metoda>MIP</metoda>
<vzorku>160</vzorku>
<orez x1="0" x2="100"
      y1="0" y2="100"
      z1="0" z2="100"/>
<interval i1="113"
          i2="193"/>
```



Obrázek 3.3: Podoba karty metody pro metodu MIP.

Pro metodu SIP v kartě přibude jen pole pro vložení konstanty. Proto nepovažuji za podstatné kvůli tomu uvádět její podobu. Při změně přepínače na metodu LUT nebo Gradientní stínování dojde k vyvolání samostatné karty pro vytváření přenosových funkcí. Této kartě jsem se rozhodl věnovat samostatnou kapitolu 3.3.3.

U metody gradientní stínování má uživatel možnost nastavit více parametrů než jen přenosovou funkci, jak je tomu u metody LUT. Může ovlivnit lesklost materiálu, relativní pozici světla vůči přednastavenému směru (0,0,6) a hranici alfa průhlednosti. Alfa průhlednost slouží k definování povrchů za pomocí přenosové funkce. Pokud tedy nastavíme například hranici alfy na 0,20, tak by se měl gradient počítat ve všech bodech, jejichž alfu přenosová funkce určila jako vyšší než 0,20.



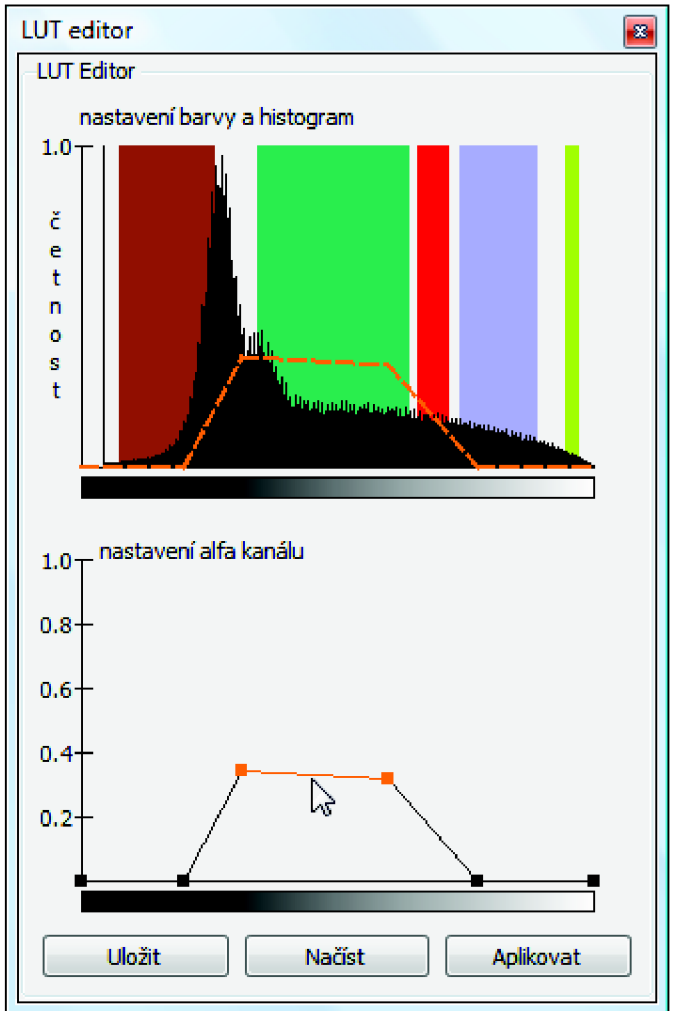
Obrázek 3.4: Podoba karty metody pro metodu Gradientní stínování.

Podoba elementů v XML souboru je následující:

```
<metoda>GRAD</metoda>
<vzorku>160</vzorku>
<orez x1="0" x2="100" y1="0" y2="100" z1="0" z2="100"/>
<odlesk>50</odlesk>
<svetlo x="0" y="0" z="1"/>
<hranice_alfy>0.2</hranice_alfy>
```

3.3.3 Karta pro vytváření přenosových funkcí

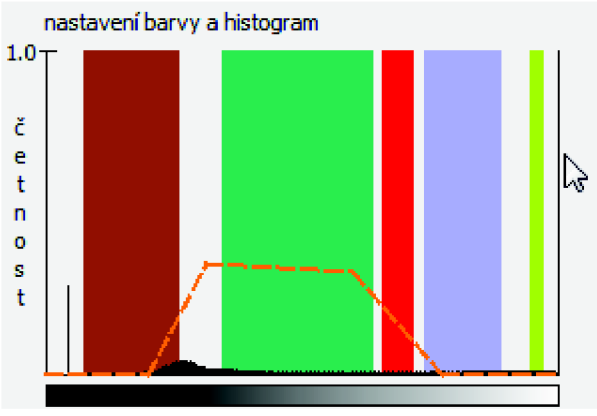
Karta je tvořena dvěma komponentami. První zobrazuje histogram objemových dat a umožňuje jim přiřazovat barvu. Histogram (vykreslen černou barvou) udává počet výskytů voxelů dané intenzity vůči počtu voxelů s nejčastější intenzitou. Četnost 1.0 má tedy voxel, který můžeme v datech nalézt nejčastěji. U objemových dat je to voxel nebo voxely, které většinou obklopují náš objekt zájmu. Jsou často bezvýznamné, ale nepěkným způsobem histogram zkreslují. Hledal jsem tedy způsob, jak histogram upravit, aby byl lépe čitelný. Zkusil jsem jej zlogaritmovat, ale výsledný histogram byl příliš plochý. Poté mě napadlo použít funkci sinus v intervalu $0, \pi$ a výsledek se konečně dostavil. Funkce potlačí voxely s nízkou a vysokou intenzitou a zvýrazní střed histogramu. Histogram na obrázku 3.5 nahoře již po aplikaci funkce sinus a histogram na obrázku 3.6 týkající se stejných objemových dat bez aplikace funkce sinus.



Obrázek 3.5: Karta pro vytváření přenosových funkcí.

Barvu objemovým datům lze přiřadit označením oblasti histogramu. Dvojklikem na plochu s barvou je možná její editace pomocí standardního dialogu pro výběr barvy. Smazání obarveného region lze provést klávesou `delete` v momentě, kdy je kurzor myši nad danou oblastí.

Druhá komponenta na kartě slouží k editaci alfa kanálu přenosové funkce (na obrázku 3.5 dole). Čím jsou hodnoty nižší, tím je průhlednost voxelů dané intezity větší. Lomenou čáru uživatel může upravovat táhnutím bodů zlomu nebo celé přímky. Mazání bodů je opět pomocí klávesy `delete` v okamžiku, kdy je kurzor nad bodem označující zlom. Přidávání bodů zlomu se provádí kliknutím myši na danou pozici.



Obrázek 3.6: Histogram (černě) bez aplikace funkce sinus. Voxely s největší četností jsou označeny šipkou.

Uživateli je nabídnuta možnost si přenosovou funkci uložit nebo načíst jako samostatný XML soubor. Struktura dokumentu je následující:

```
<?xml version="1.0" encoding="UTF-8"?>
<LUT_nastaveni>
    <color_editor>
        <oblast X1="0.34375" X2="0.64453" color="4278233727"/>
            .
            .
    </color_editor>
    <alfa_editor>
        <bod X="0" Y="0"/>
            .
            .
        <bod X="1" Y="0"/>
    </alfa_editor>
</LUT_nastaveni>
```

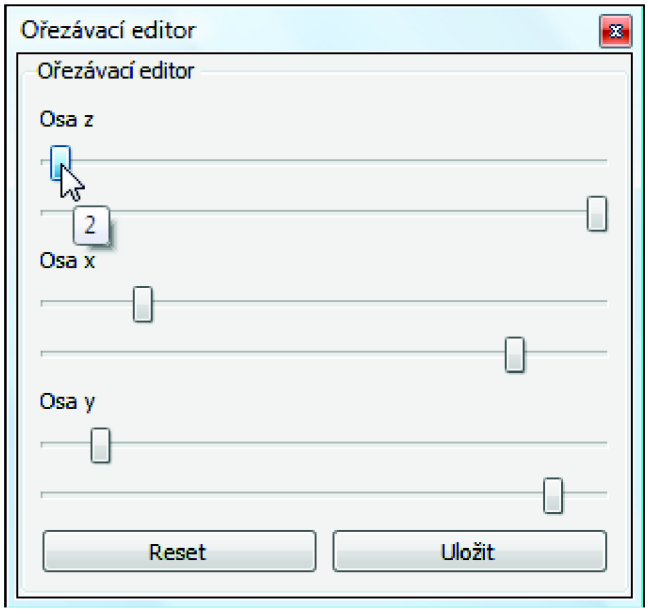
Každé nastavení přenosové funkce se skládá z 0 až N barevných oblastí a lomené čáry s počátkem X = 0 a koncem X=1 s požadovaným počtem zlomů. Barva je číslo typu int ve formátu AARRGGBB, kde A, R, G a B jsou hexadecimální číslice. Složka alfa je při načítání/ukládání nastavení ignorována.

Společný předek obou komponent je třída `QWidget`, která je základem prvků GUI. Komponenty jsou implementovány v samostatných třídách, aby je bylo možné v případě potřeby znovu použít.

3.3.4 Karta ořezávacího editoru

Možná jste si v předcházejících kapitolách v XML zápisu nastavení metody všimli elementu `<orez>`. Toto nastavení kontroluje, podle které osy (x, y, z) a o kolik budou při vykreslování objemová data ořezána. Nastavení je aplikováno hned a tak je zaručena vysoká interaktivita. Uživateli je nabídnuta možnost si toto nastavení uložit ke stávajícímu zobrazení.

Implementaci jsem provedl pomocí třídy `QDockWidget`. Aby bylo možné karty skládat i přes sebe (obrázek 3.1 vlevo dole), použil jsem metodu třídy hlavního okna `tabifyDockWidget`.



Obrázek 3.7: Karta ořezávacího editoru.

3.3.5 Komponenta volumerenderingu

Komponenta zobrazující výsledek volumerenderingu (obrázek 3.1 vpravo) se ovládá pomocí myši. Stiskem a táhnutím levým tlačítkem dojde k rotaci vyrenderovaného objektu kolem středu. Zvětšení

nebo zmenšení je možné provést pomocí stisknutí a tahu pravým tlačítkem myši. Při rotaci je použit zadaný počet vzorků na dráze paprsku v kartě metody. Po zastavení je obrázek přegenerovaný s dvojásobným počtem vzorků. Vede to k lepšímu vizuálnímu dojmu.

Komponenta je založena na třídě `QGLWidget`, která umožňuje zobrazování grafiky OpenGL v okně Qt aplikace.

3.3.6 Jazyková lokalizace

Qt poskytuje nástroj a mechanizmus pro snadný překlad aplikace do cizích jazyků. Nástrojem je aplikace Qt Linguist, která je distribuovaná s vývojovým prostředím Qt Creator. Mechanizmem, který překlad v konečném důsledku provádí, je třída `QTranslator`.

V průběhu implementace jsem všechen uživateli viditelný text psal v anglickém jazyce a pomocí nástroje Qt Linguist jsem jej přeložil do češtiny. Vytvoření dalších jazykových verzí není tedy problémem.

Prvním krokem za úspěšným překladem je vyhledání všech řetězců ve zdrojových kódech aplikace, které se budou překládat. Programátor takové řetězce v kódu předává metodě `tr`, kterou implementuje každý objekt knihovny Qt. Vyhledání za nás provede konzolová aplikace `lupdate`, také distribuovaná s vývojovým prostředím. Pro správný běh `lupdate` je nutné do souboru s projektem vyvíjené aplikace (aplikace.pro) přidat řádek `TRANSLATIONS = jmeno.ts`. Soubor `jmeno.ts`, který vyprodukoval `lupdate`, otevřeme v aplikaci Qt Linguist, vytvoříme textový překlad frází a uložíme. Nyní na řadu přichází konzolová aplikace `lrelease`, která ze `jmeno.ts` vyprodukuje soubor `QM` použitý k finálnímu překladu za běhu aplikace.

Realizaci překladu provádí instance třídy `QTranslator`, která si pomocí metody `load` načte soubor `QM`. Vyvíjené aplikaci se instance přiřadí voláním funkce `installTranslator`. Nyní zavoláním metody `tr` dostaneme přeložený řetězec:

```
QString s = tr("Czech language"); // s obsahuje "Český jazyk"
```

Pokud by jste chtěli vědět více k nástroji Qt Linguist, manuál v anglickém jazyce je dostupný na [22].

3.4 Vstupní data

Data určená k vizualizaci mohou být různá stejně jako způsob jejich pořízení. Rozhodující pro aplikace však je kolik bitů je věnováno na jeden voxel, tedy s jakou přesností jsou navzorkována. V programu budeme uvažovat jeden 8 bitový vzorek na voxel. Důvody, proč bylo zvoleno právě 8 bitů na voxel, jsou dva. Prvním je velikost předintegrované mapy použité při metodě LUT. S přibývajícím počtem bitů na vzorek se textura předintegrovaná mapy zvětšuje do velkých rozměrů (pro 12 bitová data se jedná již o texturu o rozměrech 4096x4096) a ne každý grafický akcelerátor je schopen texturu takové velikosti pojmout. Druhým důvodem je samotná editace a tvoření předintegrované přenosové funkce, jak bylo zmíněno v kapitole 3.3.3.

Data určená k vizualizaci budou načítány přes GUI. Formát vstupních dat bude typu RAW - vzorky objemových dat budou v souboru uloženy bez hlavičky a v pořadí jako na obrázku 3.8. Ke každému souboru s objemovými daty bude existovat soubor, který bude udávat rozměry dat v osách

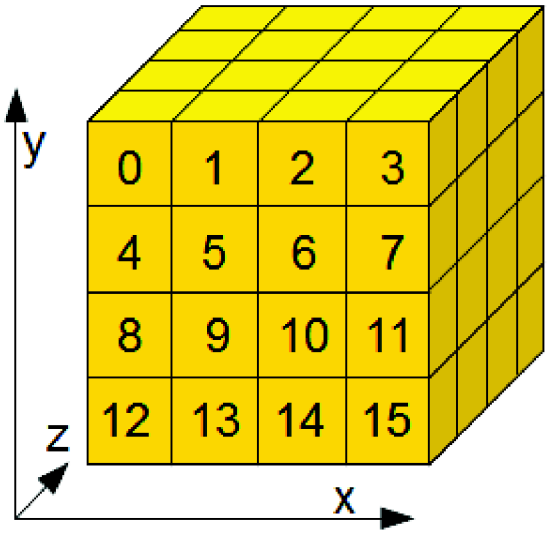
x, y a z, s jakou přesností byla data pořízena (kolik bitů na vzorek) a rozestupy vzorků v objemové mřížce v každém ze směru x, y, z (typický rozestup je 1).

Pro implementaci volumerenderingu pomocí GPU je podstatné, aby se objemová data dala umístit do paměti GPU. Pokud by mělo dojít k načtení dat o větší velikosti než je maximální velikost 3D textury, budou objemová data zmenšena právě na maximální velikost. To umožní vizualizovat i rozsáhlá objemová data, pochopitelně musíme počítat se ztrátou kvality. Existuje přístup, jak rozměrná data vizualizovat přímo – brickíng. Pro interaktivní práci je téměř nepoužitelný, protože vykazuje nízký počet snímků za vteřinu viz práce [26].

Zdrojem testovacích objemových dat mi byly internetové stránky [7], [20]. Zde lze najít jak výstupy simulací (např.: vstřikování paliva nebo simulace prostorového rozdělení pravděpodobnosti elektronu v atomu vodíku), tak výstupy z počítačové tomografie (čajová konvice nebo blok motoru).

3.4.1 Implementace zmenšování rozsáhlých objemů

Při implementaci jsem využil toho, že vstupní objemová data, pokud je budeme brát po řezech, jsou podobná šedotónovým obrázkům (8 bitů na jeden pixel). Zmenšení probíhá ve dvou krocích – nejprve ve směru osy z a poté ve směru osy x. Implementace je provedena pomocí knihovny `Qt` třídou `Qimage`, které je zadána posloupnost bytů (jeden řez). Konstruktor třídy vytvoří šedotónový obrázek, který je poté metodou `scaled` upraven na požadovaný rozměr. Metodou `bits` je získána posloupnost bytů již zmenšeného obrázku. Poměr stran je zachován a je zvolena rychlá trasformace bez vyhlazování `Qt::FastTransformation`.



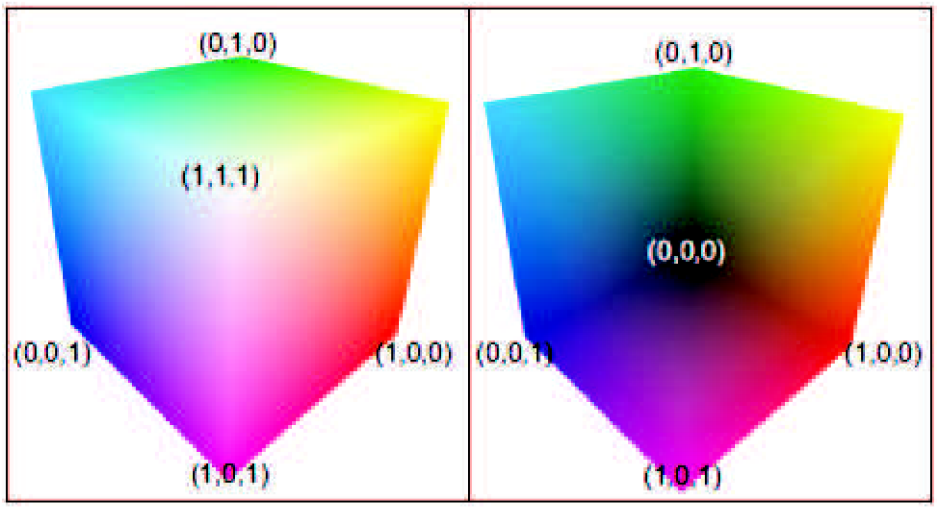

```
ObjectFileName: Cube.raw
Resolution:     4 4 4
SliceThickness: 1.0 1.0 1.0
Format:         UCHAR
```

Obrázek 3.8: Uložení objemových dat v souboru (vlevo) a příklad souboru popisující data typu raw (vpravo).

3.5 Ray casting

Asi nejzásadnější problém při vykreslování objemových dat pomocí ray castingu je nalezení bodu, kde paprsek do objemových dat vstupuje a jakým směrem se šíří. Tyto dvě základní informace musíme umět vypočítat pro každý pixel výsledného obrazu a to při libovolném natočení objemových dat, ať už se jedná o paralelní či perspektivní projekci. V článku [9] je uveden princip, jak tyto informace elegantně získat. Uvedený způsob se mi zalíbil svou jednoduchostí a elegancí a na jeho základě jsem aplikaci vystavěl.

Pro nalezení bodu, kterým vstupuje paprsek do objemových dat z libovolného pohledu, a směru jeho šíření nám poslouží RGB krychle, která je umístěna ve scéně a tvoří obalové těleso objemových dat (obrázek 3.9). Velikost krychle není důležitá, dokonce to nemusí být ani krychle viz kapitola



Obrázek 3.9: Vyrenderovavá přední (vlevo) a zadní (vpravo) strana RGB krychle, která obaluje objemová data [9].

3.5.3. O mnoho podstatnější je barva přiřazená vrcholům. Na obrázku 3.9 je hodnota barvy ve vrcholu označena trojicí (R,G,B) a je v rozsahu 0 - 1 pro každý kanál. Představme si, že trojice (R,G,B) představuje místo barvy texturovací souřadnice pro naše objemová data. Díky interpolaci barvy grafickým akcelerátorem mezi vrcholy dostaneme, nejen hladký přechod mezi barvami, ale především texturovací souřadnice každého bodu RGB krychle (našeho obalovacího tělesa). Texturovací souřadnice (barvu RGB) nám grafický akcelerátor vypočítá velmi rychle, v každém natočení, v perspektivní i paralelní projekci. Nyní již známe kudy do objemových dat paprsek ray castingu vstupuje (texturovací souřadnice). Informací, která nám ještě schází zjistit, je jakým směrem se šíří. Bodem ***i*** si označme bod vstupu paprsku na přední straně RGB krychle (z pohledu pozorovatele). Bodem ***o*** si označme bod, který protíná paprsek vycházející z krychle. Pro získání směru paprsku $\vec{v}$ nám stačí už jen odečíst bod ***o*** od bodu ***i***:

$$\vec{v} = o - i \qquad (9)$$

Pokud vektor $\vec{v}$ normalizujeme a vynásobíme konstantou, o kterou chceme, aby se v následující kroku zvětšil, dostaneme přírůstek ***p***. Tento přírůstek pak stačí v cyklu přičítat k bodu ***i***, dokud nedojdeme do bodu ***o***. V každém kroku cyklu tedy známe souřadnici voxelu na dráze paprsku a můžeme aplikovat některou z metod zobrazení.

Generování paprsků volumerenderingu provádím vykreslením RGB krychle do scény. Každý barevný pixel přední strany krychle výsledného obrazu je zdrojem paprsku raytracingu. Abych mohl určit směr paprsku, potřebuji znát bod, kde paprsek krychli opouští. V jednom průchodu to pro mne byla obtížná úloha a nepodařilo se mi jí rozluštit. Vydal jsem se tedy cestou článku [9] a vykreslování RGB krychle jsem implementoval dvouprůchodově. V prvním průchodu se vyrenderuje zadní část krychle pomocí ořezu přední strany:

```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```

Vyrenderovaný snímek si uložím do textury. V druhém průchodu nastavím ořezání zadních stran a opět provedu vyrenderování, nyní již s shader programem požadované metody zobrazení. Shader programu předám snímek zadní strany, aby ji mohl použít při určování směru paprsku. Návrhu a implementaci shaderů pro raycasting se věnuje kapitola 3.6.

Renderování snímku do textury není obtížná záležitost. Prvním krokem je založení objektu FBO (Frame Buffer Object):

```
GLuint fbo;
glGenFramebuffersEXT(1, &fbo);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
```

Následuje vytvoření textury a paměti hloubky o požadované velikosti. V našem případě je to velikost komponenty zděděné od QGLWidget. Paměť hloubky je připojena, aby bylo možné použít ořezání předních stěn.

```
GLuint textura;
glGenTextures(1, &textura);
glBindTexture(GL_TEXTURE_2D, textura);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,  sirka, vyska, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);

GLuint pametHloubky;
glGenRenderbuffersEXT(1, &pametHloubky);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, pametHloubky);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT,
                         sirka, vyska);
```

Proces je ukončen spárováním objektu FBO s texturou a pamětí hloubky.

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
         GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, textura, 0);

glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
          GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT, pametHloubky);
```

Nyní, když budeme chtít renderovat do vytvořené textury, stačí provést příkazy:

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo); // připojení FBO
  // teď se renderuje do textury
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0); // odpojení FBO
```

Ukázky zdrojového kódu jsem uvedl, protože založení FBO objektu je zásadní pro správný chod celé vykreslovací smyčky volumerenderingu. Pro zájemce o obsáhlejší výklad používání FBO objektů mohu doporučit zdroj [10].

3.5.1 Uložení objemových dat

Objemová data jsou ukládána do 3D textury v podobě GL_LUMINANCE, protože obsahují jen intenzitní hodnoty. Aby bylo možné data do textury nahrát musí splňovat 2 vlastnosti:

1. Nejdelší hrana objemových dat musí být menší nebo rovna rozměru maximální velikosti textury, kterou lze do grafického akcelerátoru nahrát. Pokud toto není splněno, jsou objemová data zmenšena. Nejdelší hrana objemových dat je zmenšena na maximální velikost textury. Ostatní hrany jsou zmenšeny tak, aby zachovaly původní poměry mezi hranami.
2. Každý rozměr objemových dat, před nebo po zmenšení, musí mít hodnotu rovnou 2^m, kde m je celé nezáporné číslo. Pokud toto není splněno, jsou objemová data zvětšena na nejbližší větší hodnotu 2^m. Intenzita voxelů, o které jsou objemová data doplněna, je rovna 0 a při vykreslování se chovají jako prázdný prostor. Při přidávání voxelů dbám na to, aby po úpravě byla původní objemová data uprostřed. Je to z důvodu, aby střed otáčení ležel ve středu původních dat.

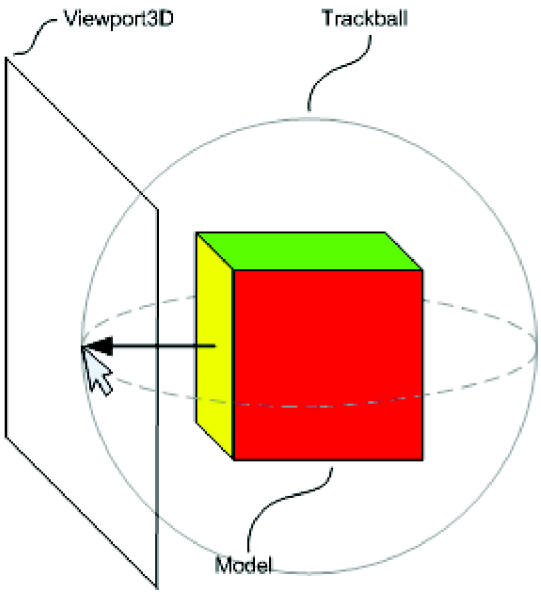
Při nahrávání textury nesmíme zapomenout nastavit filtrování. Definuje, jak se budou získávat hodnoty voxelů, jejichž souřadnice nejsou přesně na objemové mřížce a hodnotu voxelu tedy nelze získat přímo. V implementaci jsem zvolil filtrování GL_LINEAR, které použije vážený lineární průměr z pole 2x2x2 voxelů, které leží nejblíže středu požadovaného voxelu [18].

3.5.2 Rotace obalového tělesa

Pro rotaci obalového tělesa jsem zvolit tzv. virtuální trackball. Jde o snahu napodobit chování zařízení trackball pomocí tahů myši. Trackball je zařízení s kuličkou, kterou uživatel pohybuje. Zařízení si zaznamenává polohu a točí s objektem stejně, jako uživatel s kuličkou. Ukázka zařízení trackball je na obrázku 3.11. Výhodou je přesnost, s jakou lze rotace provádět, a intuitivní ovládání. Virtuální trackball se snaží výhody napodobit tím, že převádí 2D pohyby myši na 3D rotace. Dělá to pomocí projekce pozice myši na imaginární kouli (obrázek 3.10). Při pohybu myši se rotuje objektem tak, aby bod pod kurzorem označoval stále stejné místo na kouli [24].

V aplikaci jsem použil implementaci z demonstračního příkladu dinospin toolkitu GLUT [25].

Obrázek 3.11: Ukázka zařízení trackball [23].



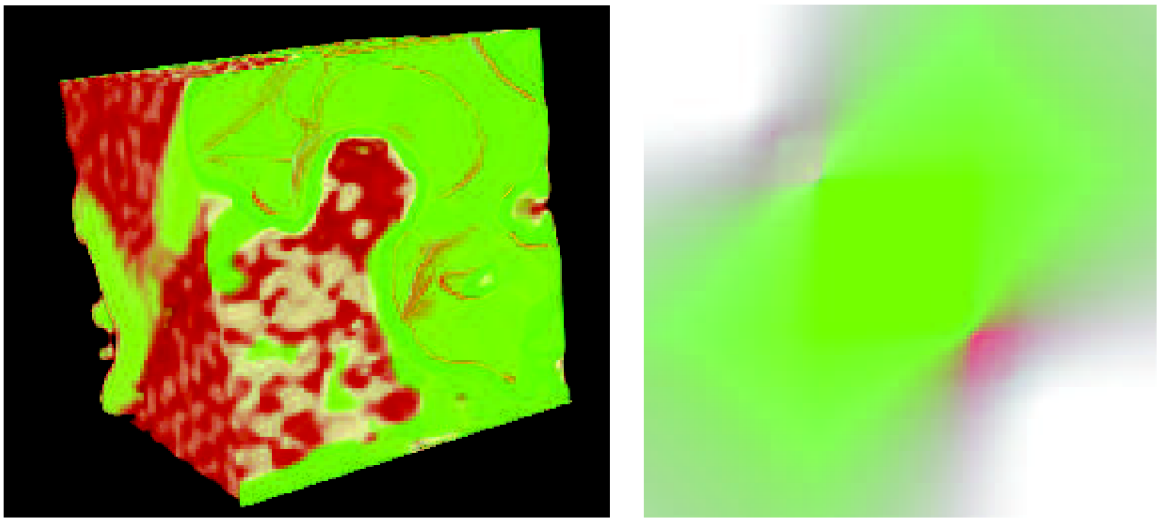
Obrázek 3.10: Boční pohled ilustrující mapování pozice myši na povrch koule [24].

3.5.3 Řešení ořezání vizualizovaných dat

Ořezání provádím pomocí RGB krychle. V požadovaném směru (x, y, z) zmenšuji její rozměr a úměrně s ním i hodnotu barvy ve vrcholech. Pokud bychom jen upravovali rozměr kostky, docházelo by k deformaci zobrazovaných objemových dat. V podstatě zkrátíme dráhu paprsku objemem podle požadavků.

3.5.4 Vytvoření přenosové funkce

Přenosová funkce je shader programu předávána jako 2D textura. Textura je vytvořena pomocí funkce převzaté z [16]. Vstupními parametry funkce jsou barvy a alfa hodnoty přiřazené intenzitám voxelů objemových dat. Výstupem je textura o rozměrech 256x256. Ukázka předintegorvané přenosové funkce a výsledek její aplikace je na obrázku 3.12.



Obrázek 3.12: Vizualizovaná dat (vlevo) pomocí předintegrované přenosové funkce (vpravo).

3.6 Realizace ray castingu pomocí GPU

Abychom mohli realizovat ray casting přímo v grafickém akcelerátoru, musíme změnit program fixního vykreslovacího řetězce a upravit si ho podle vlastních požadavků. Od verze OpenGL 2.0 můžeme do vykreslovacího řetězce zasáhnout na dvou místech, a to ve zpracování vrcholů (vertex shader) a fragmentů (fragment shader). Pro ray casting je zásadní vytvoření nového fragment shaderu, který bude implementovat hlavní smyčku vykreslování a tvoří základ všech zobrazovacích metod. Nevyhneme se však implementaci i nového vertex shaderu. Důvodem není nedostatečná funkcionalita, nýbrž fakt, že do fragment shaderu potřebujeme předat informace, které nejsou při použití původního vertex shaderu ve fragment shaderu dostupné. Jaké vstupní informace jsou nám v shaderech dostupné a v jakém místě vykreslovacího řetězce shadery leží, je velmi dobře popsáno v knize [15], kterou mohu doporučit. V následujících kapitolách se budu zabývat návrhem a implementaci raytracingu a zobrazovacích metod. Informace týkající se programování shaderů

uvedu v takové míře, aby čtenář byl schopný porozumět uvedeným částem shader programů. Podrobnější studium nechám na čtenáři, kterému mohu ještě doporučit tutoriál [11].

3.6.1 Zavedení shaderů, předávání parametrů

Vytvoření nového verex a fragment shaderu je podobné jako vytváření programu v běžném programovacím jazyce, jako je například C. Vstupní podmínkou je zápis zdrojového kódu shaderu. Kroky ke spuštění jsou následující (provádí se za běhu aplikace):

1. Kompilace zdrojových shadrů – vytvoří objekt shaderu.
2. Vytvoření programu shaderu.
3. Přidání objektů shaderu do programu shaderu.
4. Zpracování všech objektů shaderů v programu a vygenerování spustitelného programu.
5. Spuštění programu shaderu.

Kompilace a spojování shaderů se provádí pomocí ovladačů grafické karty. Protože se kompilace provádí za běhu aplikace, je důležité kontrolovat, zda v krocích 1-5 nedošlo k žádné chybě. V aplikaci si takto připravím všechny shadery metod a za běhu je už jen přepínám podle potřeby.

Při vytváření shader programů jsem řešil umístění zdrojových kódů. Šlo o to, zda je uložit v souboru nebo je včlenit do zdrojového programu aplikace. Nakonec jsem zvolil variantu první – uložit je do samostatných souborů. Zkušenější uživatel si tedy může shader programy upravit bez nutnosti kompilace aplikace.

Data z OpenGL aplikace lze předat fragment i vertex shaderu. Provádí se pomocí spárování proměnných deklarovaných v shader programu a proměnných v OpenGL programu. K tomuto účelu slouží modifikátory u proměnných deklarovaných v shaderech:

1. Modifikátor attribute – Používá se s proměnnými ve vertex shaderech. Určuje, že proměnná obsahuje informace týkající se vrcholů OpenGL objektů.
2. Modifikátor uniform – Hodnota tohoto typu musí být po celou dobu vykonávání shaderu neměnná. Uniformní proměnné jsou mezi vertex a fragment shadery sdílené a musí být deklarovány jako globální.
3. Modifikátor varying – Proměnné tohoto typu musí být deklarované ve vertex i fragment shaderu a slouží pro předávání informací.

V shaderech jsem použil jen modifikátory varying a uniform.

3.6.2 Základní shader program raycastingu

Navržený základní shader program tvoří kostru zobrazovacích metod. Zpracovává problém šíření paprsku a vzorkování objemových dat. V následujícím textu uvedu zdrojový kód shaderu s patřičnými komentáři.

Vertex shader je jednoduchý a slouží pouze pro získání interpolované barvy na přední straně RGB krychle, která je předána do fragment shaderu. Každý sebejednodušší vertex shader musí

```
varying vec4 frontColor;      // barva předaná fragment shaderu

void main(){
  frontColor = gl_Color;      // vestavěná proměnná definující barvu
  gl_Position = ftransform(); // defaultní fransformace vrcholu
}
```

definovat hodnotu vestavěné proměnné gl_Position určující pozici vrcholu. Použil jsem funkci, která vrchol transformuje stejně jako fixní vykreslovací řetězec.

Fragment shader je o poznání složitější a tak ho neuvedu naráz, ale po logických částech. První úsek tvoří proměnné předávané z OpenGL aplikace a vertex shaderu:

```
// barva pixelu na přední straně RGB krychle
varying vec4 frontColor;
// vyrenderovaná zadní strana RGB kostky předaná jako textura
uniform sampler2D backSide;
// rozměry textury s vyrenderovanou zadní stranou krychle
uniform float width, height;
// objemová data, předaná ve formě 3D textury
uniform sampler3D volume;
// počet vzorků na dráze paprsku
uniform float samples;
```

Část funkce main, která vypočítá, kde do objemových dat paprsek vstupuje a jakým směrem se šíří:

```
void main(){
// souřadnice počátku paprsku vstupujícího do objemových dat
// viz kapitola 3.5
vec4 start = frontColor;

// souřadnice paprsku opouštějícího objemová data
// gl_FragCoord udává pozici aktuálně zpracovávaného pixelu
// v okně OpenGL. Podělíme-li je šířkou a výškou aktuálního okna,
// dostaneme souřadnice pixelu, v textuře vyrenderované zadní stěně
// krychle. Barva tohoto pixelu odpovídá souřadnicím, kde paprsek
// opouští objemová data.
vec4 end  = texture2D(backSide,vec2(gl_FragCoord.x/width,
gl_FragCoord.y/height));
// směr šíření paprsku objemem
vec3 dir = end.xyz - start.xyz;
// délka paprsku skrz objem, slouží k ukončení cyklu ray castingu
float len = length(dir.xyz);
// normalizace směrového vektoru
vec3 normDir = normalize(dir); // normalizace vektoru
// počet vzorků na dráze paprsku, pokud to jde, měl by být dodržen
// vzorkovací teorém
float delta = 1.0/samples;
// přírůstek paprsku v jednom kroku iterace šíření.
vec3 deltaDir = normDir * delta;
// délka přírůstku
float deltaDirLen = length(deltaDir);
// začátek paprsku, bude se k němu přičítat přírůstek
vec3 vec = start.xyz;
// uražená dráha paprsku
float lengthAcc = 0.0;
// zarážka maximálního počtu vzorků, 1.74 = sqrt of 3, jedná se
// o tělesovou úhlopříčku. Maximální počet vzorků je zaveden, protože
// kvůli zaokrouhlovací chybě může dojít k zacyklení hlavní smyčky
// raycastingu. Zarážka tomuto stavu předchází.
float maxSamples = (1.74*samples);
// počítadlo provedených vzorků
float cycles = 0.0;
```

Pokračování funkce main. Vzorkování dat podél dráhy paprsku:

```
// proměnná na vzorek na dráze paprsku
vec4 colorSample;

// Cyklus průchodu paprsku objemem
while(lengthAcc < len && cycles < maxSamples){
    // získej intenzitu voxelu na dané pozici
    colorSample = texture3D(volume,vec);

    //  místo pro implementaci metod zobrazení

    // přičtení přírůstku k paprsku (kde se bude v příštím kroku
    // vzorkovat)
    vec += deltaDir;

    // připočtení délky přírůstku
    lengthAcc += deltaDirLen;

    // inkrementace čítače počtu průchodů
    cycles++;
}
```

Fragment shader by měl definovat barvu výstupního pixelu zápisem do vestavěné proměnné gl_Color. Pokud tak neučiní, je barva nedefinovaná. Zbývající část funkce main a zápis výsledné barvy:

```
// výsledná barva je rovna intenzitě posledního vzorku
gl_FragColor = colorSample;

// konec funkce main
}
```

Základní shader program neplní žádnou smysluplnou funkci, jen vzorkuje objemová data na drahách paprsků a výslednou barvu nastaví na barvu zadní stěny objemových dat. V další kapitole si postupně rozebereme shadery navržených zobrazovacích metod.

3.6.3 Shader programy zobrazovacích metod

V teoretickém rozboru (kapitola 2.3 a 2.4) byly představeny čtyři metody možného zobrazení objemových dat – MIP, SIP, LUT a Gradientní stínování. V práci jsem si vzal za cíl navrhnout a implementovat shadery těchto metod.

Kostru metod tvoří základní shader program, do kterého jsou přidány řádky zdrojového kódu v závislosti na metodě, a kostra se tak bude opakovat ve všech shaderech metod. Kód tedy bude obsahovat známky redundance. V podstatě by bylo možné úseky shaderů metod implementovat v samostatných funkcích a k základnímu shader programu je připojit. Na druhou stranu nám jde především o rychlost a každé volání funkce nebo větvení programu výpočet zpomaluje. Rozhodl jsem se upřednostnit rychlost a tuto redundantnost kódu jsem připusti.

Metoda MIP

Pro připomenutí je to metoda, která hledá voxel s největší intenzitou na dráze paprsku. Při návrhu GUI jsem přidal možnost definovat interval, ze kterého se maximum vybírá. Interval je ohraničený zdola 0 a shora 255 včetně, protože v aplikaci počítám s 8 bitovými intenzitními daty.

Program vertex shaderu zůstává stejný jako u základního shader programu. Do fragment shaderu jsou z hlavního programu předány zmíněné hodnoty intervalu výběru maxima:

```
uniform float min, max;
```

Pro uchovávání dočasného maxima je zavedena nová proměnná typu vec4 (vektor o čtyřech složkách typu float – reprezentují barvu ve tvaru RGBA).

```
vec4 maximum = vec4(0.0,0.0,0.0,0.0);
```

Složky RGB proměnné colorSample jsou si rovny, protože jsme při vytváření 3D textury s objemovým daty nastavili typ na GL_LUMINANCE. Je tedy jedno, na kterou ze složek vzorku se dotazujeme. Pokud je intenzita v daném intervalu a překročila dosavadní maximum, zapíše se nová hodnota maxima.

```
if (min <= colorSample.r && colorSample.r <= max
                          && colorSample.r > maximum.r){
  maximum = colorSample;
}
```

Hodnotu maxima pak zapíšeme jako výslednou barvu pixelu výsledného obrazu:

```
gl_FragColor = maximum;
```

Metoda SIP

Metoda sčítá intenzity vzorků na dráze paprsku. Podobně jako u metody MIP lze definovat interval intenzit, v kterém se budou vzorky sčítat. V návrhu GUI jsem přidal možnost zadat konstantu. Konečná podoba vzorce pro výpočet intenzity pixelu:

$$I = \sum_{i \in J} (I_i * k), \tag{10}$$

kde I_i je *i-tý* vzorek z intervalu J a k je násobící konstanta. K zavedení násobící konstanty mě vedly příliš světlé výsledky vizualizace. Navíc, to kolik se bude sčítat vzorků, je ovlivněno počtem vzorků na dráze paprsku. Zvedne-li se vzorkovací frekvence, dojde k dalšímu zesvětlení výsledků. Pokud ale dáme do správného poměru násobící konstantu a počet vzorků, dosáhneme slušného zobrazení. Porovnat rozdíly můžete v kapitole výsledky.

Vertex shader je stejný jako u základního shader programu. Do fragment shaderu předávám z hlavního programu proměnné minima, maxima intervalu a násobící konstantu:

```
uniform float min, max, mulConst;
```

Ve funkci main přidávám proměnnou součtu intenzit na dráze paprsku:

```
vec4 colAcc = vec4(0.0,0.0,0.0,0.0);
```

Jazyk GLSL definuje mimo jiné operaci násobení vektoru s typem float, tudíž je zápis vzorce velmi efektivní:

```
if (colorSample.r > min && colorSample.r < max)  {
  colorSample *= mulConst;
  colAcc += colorSample;
}
```

Na konci funkce main už jen stačí zapsat hodnotu výsledného pixelu:

```
gl_FragColor = colAcc;
```

Metoda LUT

Tato metoda používá pro zobrazování objemových dat předintegrovanou přenosovou funkci. Je to zlom v zobrazování, protože výsledkem předcházejících funkcí byly doposud jen šedotónové obrázky. Předintegrovaná přenosová funkce přináší nejen možnost přiřadit barvu a průhlednost voxelům, ale také lepší aproximaci volume rendering integrálu a tím i kvalitnější výsledky. Příprava přenosové funkce probíhá v hlavním programu a do fragment shaderu je předána jako 2D textura s kanály RGBA:

```
uniform sampler2D preint;
```

Do funkce main je přidána proměnné pro akumulaci výsledné barvy a vektor reprezentující intenzitu minulého a aktuálního vzorku (s_f a s_b na obrázeku 2.20).

```
vec4 colAcc = vec4(0.0,0.0,0.0,0.0);
vec2 scalar = vec2(0.0,0.0);    // x je minulý a y je aktuální vzorek
```

Aktuální a minulý vzorek jsou souřadnicemi do textury přenosové funkce. Na pozici udané souřadnicemi se nachází hodnota volume rendering integrálu odpovídající úseku mezi aktuálním a minulým vzorkem objemových dat.

```
colorSample = texture3D(volume,vec);
scalar.y = colorSample.r;
colorSample = texture2D(preint,scalar.xy);
```

Typická kompozice výsledné barvy pro směr šíření paprsku zepředu dozadu se řídí vzorcem [16]:

$$\begin{aligned} C_{dst} &= C_{dst} + (1-\alpha_{dst})C_{src} \\ \alpha_{dst} &= \alpha_{dst} + (1-\alpha_{dst})\alpha_{src} \end{aligned} \tag{11}$$

C_{dst} je postupně se akumulující barva pixelu, C_{src} je hodnota volume rendering integrálu pro úsek mezi vzorky a αdst a αsrc jsou alfa kanály obou barev. Implementace ve fragment shaderu odpovídá řádku:

```
colAcc = colAcc + (1.0-colAcc.a) * colorSample;
```

Pro další průchod si hodnotu intenzity aktuálního vzorku uložíme na pozici minulého:

```
scalar.x = scalar.y;
```

Finální barva pixelu je uložena v colAcc a stačí ji předat jako výsledek fragment shaderu.

```
gl_FragColor = colAcc;
```

Ani v této metodě jsem vertex shader základního programu nijak neupravoval.

Metoda Gradientní stínování

Metoda gradientní stínování vychází z poznatků z kapitoly 2.3.3 o teorii osvětlování povrchů. Mým cílem bylo poznatky zpracovat a navrhnout shader program, který uplatní lokální osvětlovací model a zobrazovaným objemovým datům přidá na realističnosti. V momentě zapnutí této metody do scény přibude bílé bodové světlo na pozici (0,0,6), svítí tedy z pohledu pozorovatele. Pokud by uživatel toužil po tom změnit směr světla, umožnil jsem mu to přes GUI, kde se světlem může pohybovat v rozmezí -4 až 4 v každém směru. Pozice (0,0,6) není určena náhodně. Polohu jsem zvolil s ohledem na dobré nasvícení objemových dat.

Pro definic povrchů a jejich barvy si zavádím přenosovou funkci v podobě jednorozměrné RGBA textury. Intenzita voxelu je indexem do textury. Zde nacházející se složky RGB definují difuzní barvu materiálu a složka A slouží pro rozhodnutí, jestli se jedná o voxel patřící povrchu objektu v objemových datech. Rozhodnutí závisí na prahové hodnotě alfy předané z GUI. Je-li

A složka větší než práh, pak se jedná o povrch objektu a následuje výpočet osvětlovacího modelu. Přenosová funkce se vytváří v GUI na kartě LUT editor.

Při implementaci shaderu jsem čerpal z knihy[16]. Jako první opět uvedu vertex shader. Nyní se již nevyhneme změnám oproti původnímu programu. Pro výpočet rovnic 5 – 7 potřebujeme znát směr pohledu pozorovatele, který získáme vynásobením matice gl_ModelViewMatrix s vrcholy obalového tělesa:

```
varying vec3 V; // deklarace proměnné sdílené mezi shadery

V = vec3(gl_ModelViewMatrix * gl_Vertex);
```

Do fragment shaderu potřebujeme předat navíc matici gl_NormalMatrix, která nám poslouží pro zjištění směru normály na povrchu objektu v lokálních souřadnicích. Matice je uložena ve vestavěné proměnné vertex shaderu a proto si ji musíme uložit:

```
varying mat3 glNormalMatrix; //deklarace proměnné sdílené mezi shadery
// uložení vestavěné proměnné gl_NormalMatrix
glNormalMatrix = gl_NormalMatrix; // uložení
```

Navržený fragment shader z hlavního programu přebírá i rozměr textury s objemovými daty. Tento parametr bude potřeba při výpočtu normály v bodě dopadu paprsku na povrch.

```
uniform sampler1D preint;
uniform vec3 texStep; // rozměr textury s objemovými daty
uniform float alpha;

// hodnoty předávané z vertex shaderu
varying vec3 V;
varying mat3 glNormalMatrix;
```

Výpočet barvy je proveden sekvencí:

```
// hodnota z přenosové funkce definující barvu a povrch
colorSample = texture1D(preint,colorSample.r);

// pokud je alfa složka větší jak práh, jedná se o povrch a proběhne
// výpočet osvětlovacího modelu
if(colorSample.a > alpha){

  // N udává normálu povrchu v místě průsečíku paprsku s objektem
  // v objemových datech
  vec3 N =  normalize(glNormalMatrix * getNormal(vec));

  // L udává pozici světla
  vec3 L = normalize(gl_LightSource[0].position.xyz );

  // zápis výsledné barvy pixelu
  colAcc.rgb = shading(N,V,L, colorSample.rgb);
  break;
}
```

V předcházejícím úryvku kódu funkce getNormal přebírá souřadnice aktuálního vzorku na dráze paprsku a vrací normálu v tomto bodě. V podstatě se jedná o gradient vypočítány podle vzorce 3 metodou centrálních rozdílů. Vektor h udává vzdálenosti dvou vzorků v textuře s objemovými daty:

```
vec3 getNormal(vec3 point){
// centrální rozdíly
vec4 Gx = texture3D(volume,vec3(point.x+texStep.x,point.y, point.z)) -
          texture3D(volume,vec3(point.x-textStep.x,point.y, point.z));
vec4 Gy = texture3D(volume,vec3(point.x,point.y+textStep.y, point.z))-
          texture3D(volume,vec3(point.x,point.y-texStep.y, point.z));
vec4 Gz = texture3D(volume,vec3(point.x,point.y,point.z+textStep.z))-
          texture3D(volume,vec3(point.x,point.y, point.z-textStep.z));
// nenormalizovaný gradient
return  vec3(Gx.r,Gy.r, Gz.r);
}
```

Přenosová funkce definuje difuzní barvu materiálu. Barva světla, stejně jako ambientní a spekulární složka barvy jsou předdefinované. Kód funkce pro výpočet Blinn-Phong osvětlovacího modelu jsem převzal z [16]:

```
vec3 shading(vec3 N, vec3 V, vec3 L, vec3 Kd){
  vec3 Ka = vec3(0.1,0.1,0.1);  //ambientní složka
  vec3 Ks = vec3(0.2,0.2,0.2);  //spekulární složka

  // barva světla a ambientní barva světla
  vec3 lightColor = vec3(1.0,1.0,1.0);
  vec3 ambientLight = lightColor * 0.4;

  // výpočet vektoru halfvektor
  vec3 H = normalize(L+V);

  // výpočet ambientního členu
  vec3 ambient = Ka * ambientLight;

  // výpočet difuzního členu
  float diffuseLight = max(dot(L,N),0.0);
  vec3 diffuse = Kd*lightColor*diffuseLight;

  // výpočet spekulární složky
  float specularLight =
                    pow(max(dot(H,N),0.0),gl_FrontMaterial.shininess);
  if(diffuseLight <= 0.0) specularLight = 0.0;
  vec3 specular = Ks * lightColor*specularLight;

  // výslednou barvu tvoří všechny tři složky
  return ambient + diffuse + specular;
}
```

Nakonec zápis výsledné barvy pixelu:

```
gl_FragColor = colAcc;
```

3.6.4 Optimalizace

V základní variantě ray castingu paprsek prochází celým objemem. V závislosti na použité metodě lze průchod paprsku ukončit předčasně, ještě než dorazí k zadní stěně objemových dat. Kriteria metod pro okamžik ukončení jsou:

1. Jas zkoumaného pixelu u metody MIP dosáhl maxima.
2. Součet intenzit na dráze paprsku je roven bílé barvě 255.
3. Alfa (průhlednost) složka u metody LUT je rovna 1.0 a vzorec 11 následující hodnoty barvy neovlivní.

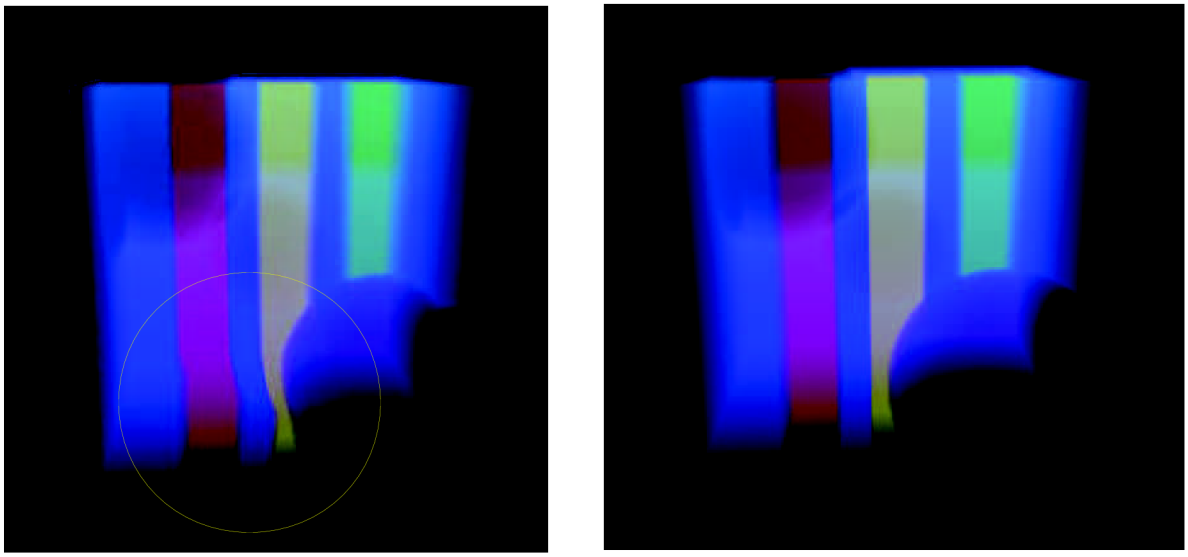
4. Paprsek metody gradientního stínování narazil na povrch.

Předčasné ukončení paprsku ušetří výkon viz výsledky, přičemž implementace je velmi jednoduchá. Do cyklu šíření paprsku stačí přidat podmínku ukončení. Například u metody LUT, pak cyklus while vypadá taktko:

```
while(lengthAcc < len  && colAcc.a < 1.0){
  // příkazy těla cyklu zůstavají
}
```

3.6.5 Problémy při realizaci

V raných stádiích implementace jsem se setkal s problémy týkající se určení směru paprsků. Tak, jak je aplikace navržena, spoléhá na kvalitní způsob interpolace barvy mezi vrcholy a na správný překryv přední a zadní strany RGB krychle. Nejsou-li podmínky dodrženy, směry paprsků se odchýlí a při zobrazení vznikají artefakty v podobě pokřivení objektu v objemových datech. Pro ověření správného směru paprsků jsem použil objemová data s pravidelnými geometrickými objekty. Zdrojový kód pro vygenerování 3D textury jsem převzal z [8]. Na obrázku 3.13 můžeme pozorovat, jak se špatné generování paprsků projeví na geometrii objektu.



Obrázek 3.13: Pokřivení objektu (označené kružnicí) v objemových datech. Špatné generování paprsků ray castingu (vlevo) a správné (vpravo).

Špatné generování paprsků jsem opravil implementací jednoduchého shader programu, který je použit pro vykreslování zadní stěny RGB krychle do textury.

Vertex shader vypočítá interpolovanou barvu mezi vrcholy:

```
varying vec4 frontColor;
void main(){
  frontColor = gl_Color;
  gl_Position = ftransform();
}
```

Fragment shader zapíše pixelu předanou barvu z vertex shaderu:

```
varying vec4 frontColor;
void main(){
  gl_FragColor = frontColor;
}
```

4 Výsledky

Při vyhodnocování aplikace jsem se zaměřil hlavně na testování rychlosti zobrazení objemových dat, protože rychlá interakce s uživatelem byla prioritním požadavkem při volbě metody volumerenderingu – ray castingu na GPU.

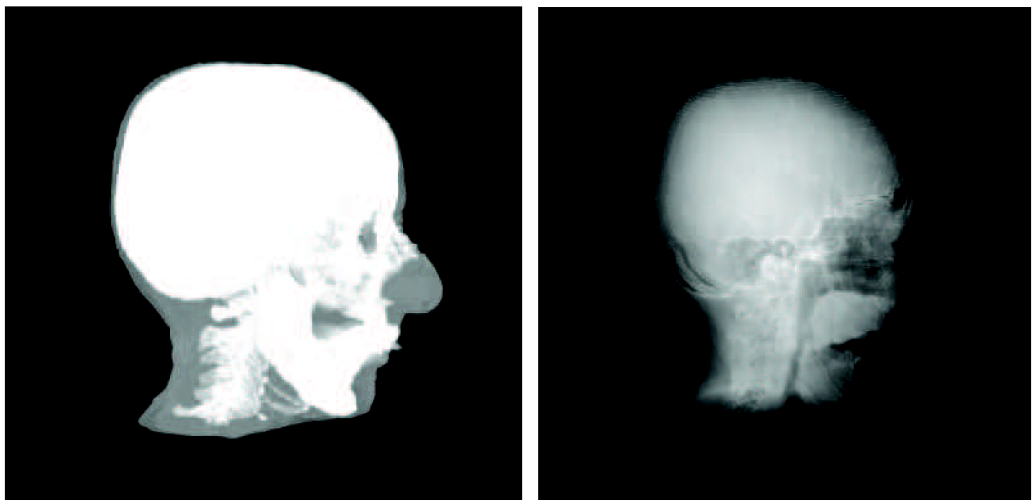
4.1 Testování rychlosti zobrazování

Testování, podobně jako vývoj aplikace, probíhalo na dvou sestavách. Konfigurace sestav je uvedena v kapitole 3.2. Pro testování rychlosti jsou důležité parametry grafických akcelerátorů. Jak se při testování ukázalo, zatížení procesoru bylo v rozmezí 5% - 50%, v závislosti na velikosti zobrazovacího plátna a počtu snímků za vteřinu. Pěti procentní zatížení odpovídá velikosti plátna 512x512px při 37 snímcích za vteřinu. Padesáti procentní zatížení odpovídá 35 snímkům za vteřinu s velikostí plátna 1680x1050px. Při testování aplikace byl výkon procesoru vždy dostačující. Grafické karty sestav použitých při testování byly:

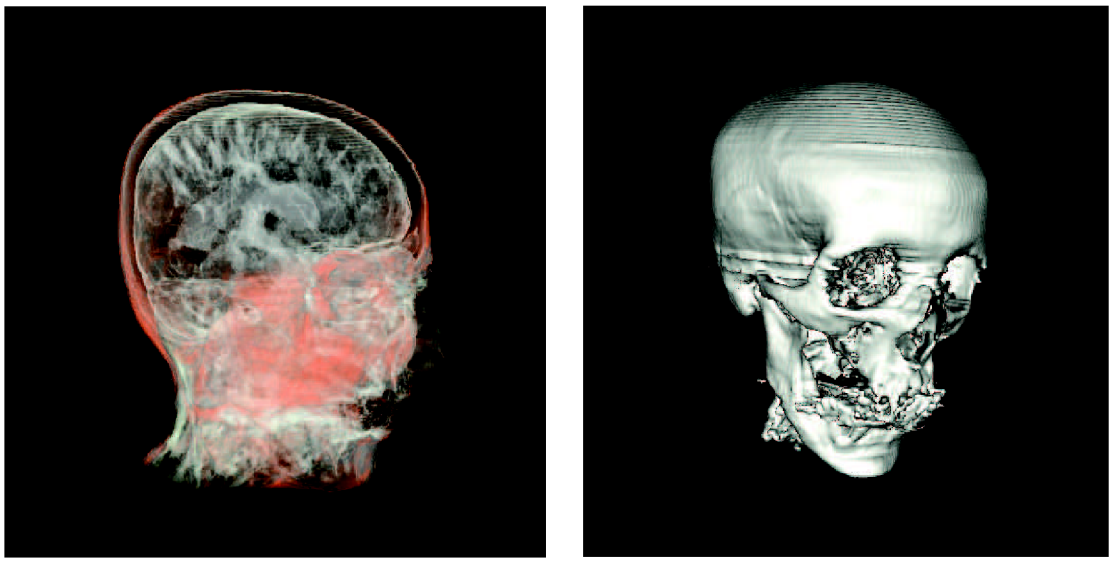
1. NVIDIA GeForce 8600 GT, GPU G84, 256 MB DDR3 RAM, šířka sběrnice 128 bitů, takt GPU 600 MHz, 32 unifikovaných shader procesorů, maximální velikost 3D textury 2048.
2. Intel Mobile 4 Series, GPU GM 45, 32 MB DDR2 RAM, šířka sběrnice nezjištěna, takt GPU 475 MHz, 10 unifikovaných shader procesorů, maximální velikost 3D textury 128.

Testoval jsem všechny čtyři metody s různou velikostí obalového tělesa objemových dat a s různým počtem vzorků na dráze paprsku. Velikost obalového tělesa určuje rozměr vizualizovaného objektu v pixelech. Objemová data použitá při testování byla CT snímek hlavy o rozměrech dat 256x256x106. Zvolené parametry metod pro testování byly:

1. MIP – celý interval intenzit, obrázek 4.1.
2. SIP – interval intenzit v rozmezí 147-224, násobící konstanta 0,01, obrázek 4.1.
3. LUT – pomocí lut editoru byly vyzdvihnuty tváře a mozek, obrázek 4.2.
4. Gradientní stínování – pomocí lut editoru byl vybrán povrch lebky, obrázek 4.2.



Obrázek 4.1: Nastavení parametrů zobrazení CT snímků halvy metod MIP (vlevo) a SIP (vpravo) pro účely testování počtu snímků za vteřinu.



Obrázek 4.2: Nastavení parametrů zobrazení CT scanu halvy metod LUT (vlevo) a Gradientní stínování (vpravo) pro účely testování počtu snímků za vteřinu.

Metoda zobrazení	Počet vzorků na dráze paprsku	Počet snímků za vteřinu v závislosti na rozměrech obalového tělesa.				
		128x128x128 px	256x256x256 px	384x384x384 px	512x512x512 px	768x768x768 px
MIP	256	60,0	60,0	40,0	26,0	17,5
	512	60,0	30,5	21,5	14,5	9,0
SIP	256	60,0	56,0	38,0	26,0	17,5
	512	59,0	32,0	24,0	15,0	10,0
LUT	256	60,0	37,0	25,0	18,0	12,0
	512	38,0	21,0	14,0	10,0	6,5
Gradientní stínování	256	60,0	38,0	26,0	19,5	13,0
	512	45,0	22,0	15,0	11,0	7,0

Tabulka 1: Naměřené hodnoty počtu snímků za vteřinu pro sestavu číslo 1.

Počet vzorků na dráze paprsků je zvolen s ohledem na vzorkovací teorém. Tedy pokud data mají velikost 256x256x106, je nutné na dráze paprsku učinit nejméně 512 vzorků, aby došlo ke korektní rekonstrukci původního objektu. U druhé testovací sestavy byla maximální velikost objemových dat, které šly do paměti grafické karty nahrát, rovna 128^3. Aplikace objemová data zmenšila na 128x128x53 a při testování jsem počet vzorků na dráze paprsku stanovil na potřebných 256.

Do testování jsem zahrnul i poloviční počet vzorků na dráze paprsku než je optimální hodnota. Dojde sice k pod vzorkování, ale zobrazená objemová data jsou stále v dostačující kvalitě.

V průběhu testování se ukázalo, že počet snímků za vteřinu je závislý také na rotaci objemových dat. Při pohledu ve směru osy z je rychlost zobrazování vyšší než z ostatních pohledů.

U metody LUT rozdíl činí až 10 % . Urychlení ve směru osy z je způsobeno uložením a čtením objemových dat z paměti. Testy zabývající se vlivem uložení objemových dat na rychlost zobrazování lze nalézt v [16]. Při testování jsem se snažil pohled zvolit tak, aby odpovídal běžné práci s aplikací.

Metoda zobrazení	Počet vzorků na dráze paprsku	Počet snímků za vteřinu v závislosti na rozměrech obalového tělesa.				
		128x128x128 px	256x256x256 px	384x384x384 px	512x512x512 px	768x768x768 px
MIP	128	55,0	23,0	13,0	7,0	2,0
	256	32,0	13,0	7,0	4,0	4,0
SIP	128	66,0	26,0	16,5	9,0	2,0
	256	36,0	15,0	8,5	4,5	4,5
LUT	128	60,0	27,0	16,0	9,0	2,5
	256	30,0	15,0	9,5	5,0	5,0
Gradientní stínování	128	65,0	28,0	16,0	9,5	2,5
	256	35,0	15,5	7,0	5,0	5,0

Tabulka 2: Naměřené hodnoty počtu snímků za vteřinu pro sestavu číslo 2.

Z naměřených hodnot v tabulkách 1 i 2 plyne závěr, že rychlost zobrazování závisí nejen na rozměrech obalového tělesa objemových dat, ale také na zvolené metodě i počtu vzorků na dráze paprsku. Po hardwarové stránce to znamená, že rychlost vykreslování je přímo úměrná počtu přístupů do paměti a počtu shader procesorů.

4.1.1 Testování optimalizací

Do návrhu jsem zahrnul dvě optimalizace, které by měly rychlost vykreslování zrychlit. Jednalo se o předčasné ukončení paprsku a ořezání objemových dat, které nejsou předmětem zájmu. Testování jsem opět prováděl na CT snímku hlavy o rozměrech 256x256x106 s použitím sestavy číslo 2.

Testu vlivu předčasného ukončení paprsku na zrychlení vykreslování jsem podrobil metody MIP, SIP, LUT s nastavením jako na obrázcích 4.1 a 4.2. Metoda Gradientní stínování již implicitně předčasné ukončení paprsku předpokládá v okamžiku, kdy narazí na povrch v objemových datech, a do testů tedy není zahrnuta.

Metoda	Rozměr obalového tělesa	Počet vzorků na dráze paprsku	Počet snímků za vteřinu	
			Bez předčasného ukončení	S předčasným ukončením
MIP	384x384x384 px	256	38	40
SIP			40	38
LUT			25	25

Tabulka 3: Vliv předčasného ukončení paprsku na počet snímků za vteřinu.

V tabulce 3. stojí za povšimnutí výsledky pro metodu SIP. Při testovaných došlo ke zpomalení zobrazování, což je důsledek nastavených parametrů zobrazování. Test na ukončení paprsku totiž nebyl úspěšný ani v tolika případech, aby vyrovnal ztráty výkonu při zapnutí testování ukončení paprsku. V ostatních případech rychlost zobrazení zůstává na stejné úrovni nebo se zlepší.

Objemová data často na svých okrajích obsahují nevýznamná data. Tato data lze oříznout a ke zpracování předložit jen nezbytnou část s objektem našeho zájmu. Tuto techniku spojenou s předčasným ukončením paprsku jsem aplikoval na testovací objemová data halvy z kapitoly 4.1 (obrázky4.1 a 4.2). Testoval jsem na sestavě 1. i 2.

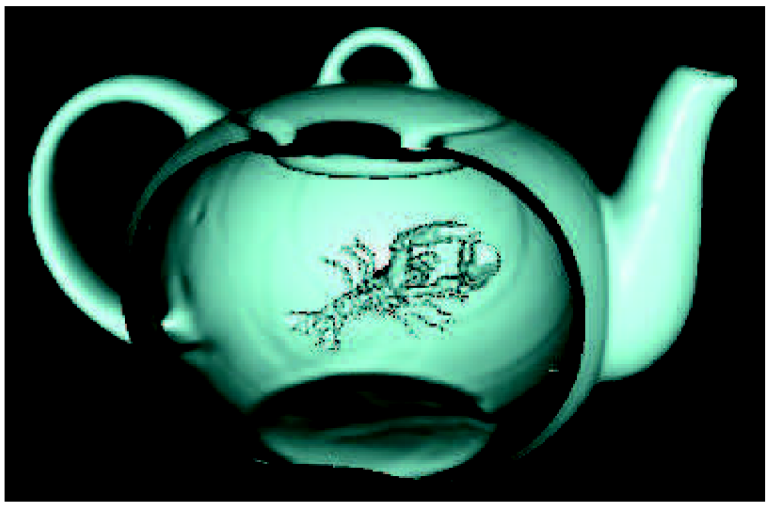
Metoda	Rozměr obalového tělesa	Oříznuté obalové těleso	Počet vzorků na dráze paprsku	Počet snímků za vteřinu			
				Sestava 1.		Sestava 2.	
				Před	Po	Před	Po
MIP	512 x 512 x 512	338 x 400 x 466	256	26,0	56,0	4,0	8,5
SIP				26,0	52,0	4,5	9
LUT				18,0	33,0	5	9,5
Gradientní stínování				19,5	43,0	5	15,5

Tabulka 4: Vliv ořezání nevýznamných dat na počet snímků za vteřinu.

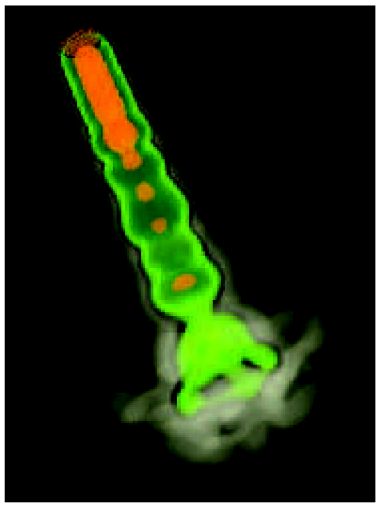
Vlivem oříznutím se zmenší objem zpracovávaných dat o 53%. Z tabulky 4 je znatelný nárůst výkonu o zhruba 50% procent, což je pádný důvod se o ořezání nevýznamných dat při nastavování parametrů zobrazování zajímat.

4.2 Ukázky výstupů

Aplikaci jsem testoval na řadě objemových dat. Jednalo se o snímky z CT i o data ze simulací, které jsou volně ke stažení na internetová stránce [7]. Možnosti implementované aplikace jsem se snažil zachytit na následující obrázcích (GRAD je zkratka pro gradientní stínování).



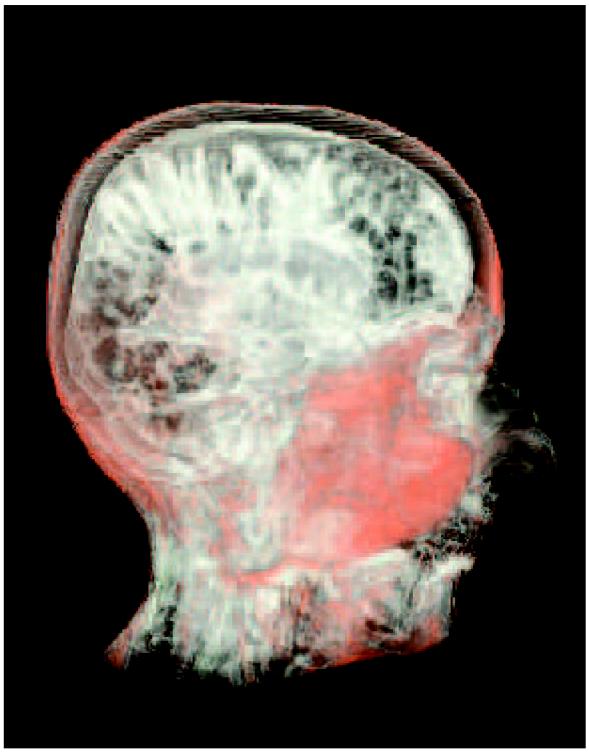
Obrázek 4.3:Konvice s humrem, CT 256x256x178, GRAD.



Obrázek 4.4:Palivo, 64^3, LUT.

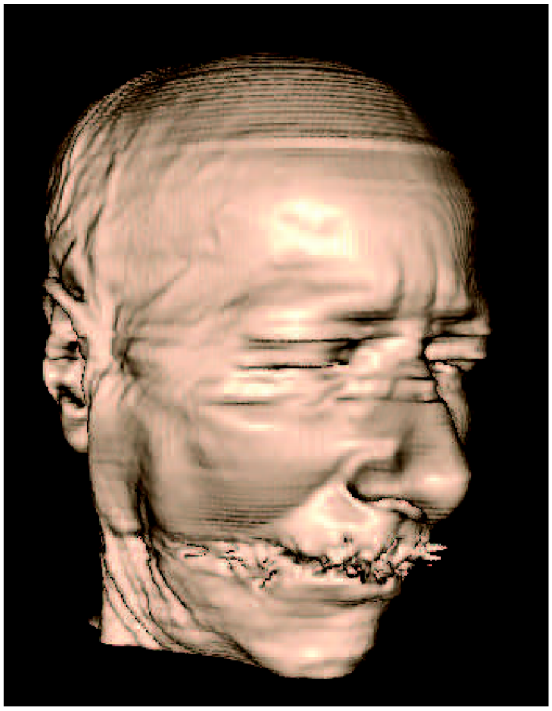
Obrázek 4.5: Noha, CT 256^3, SIP

Obrázek 4.6: Noha, CT 256^3, MIP



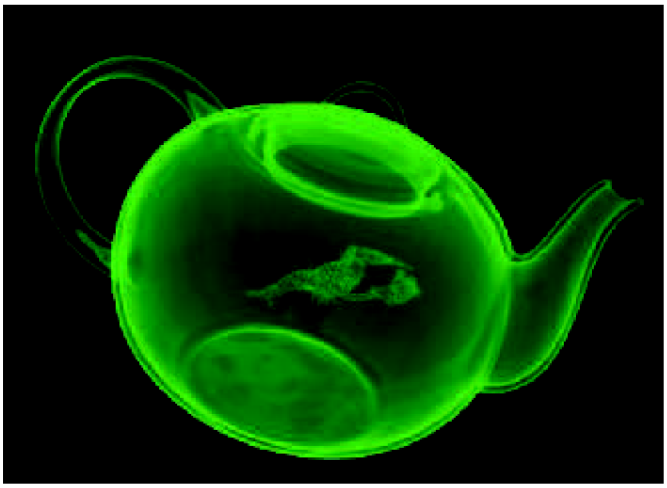
Obrázek 4.7: Hlava, CT 256x256x106, LUT

Obrázek 4.8: Hlava, CT 256x256x106, LUT



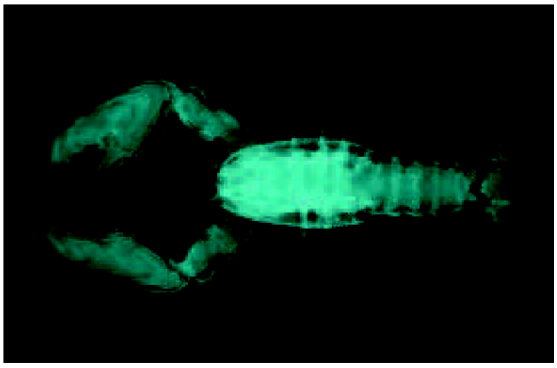
Obrázek 4.9:Hlava, CT 256x256x106, GRAD

Obrázek 4.10: Pánev, CT 512x512x174, LUT

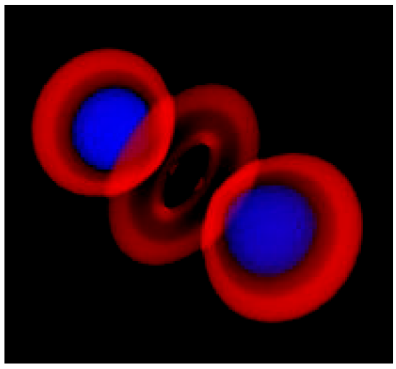


Obrázek 4.11: Konvice, CT 256x256x178, LUT.

Obrázek 4.12: Humr, CT 301x324x56, GRAD

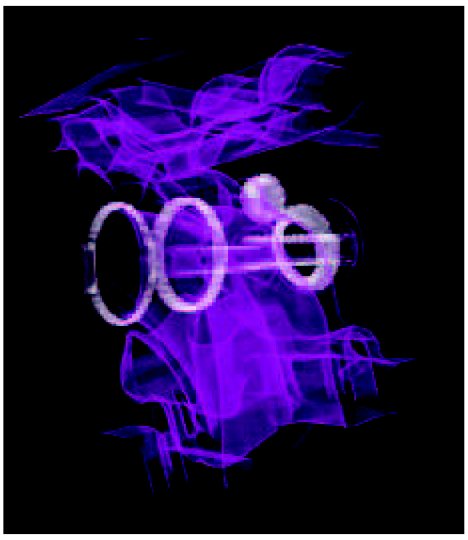


Obrázek 4.13:Humr, CT 301x324x56,LUT



Obrázek 4.14: Hydrogen atom, 64^3,LUT

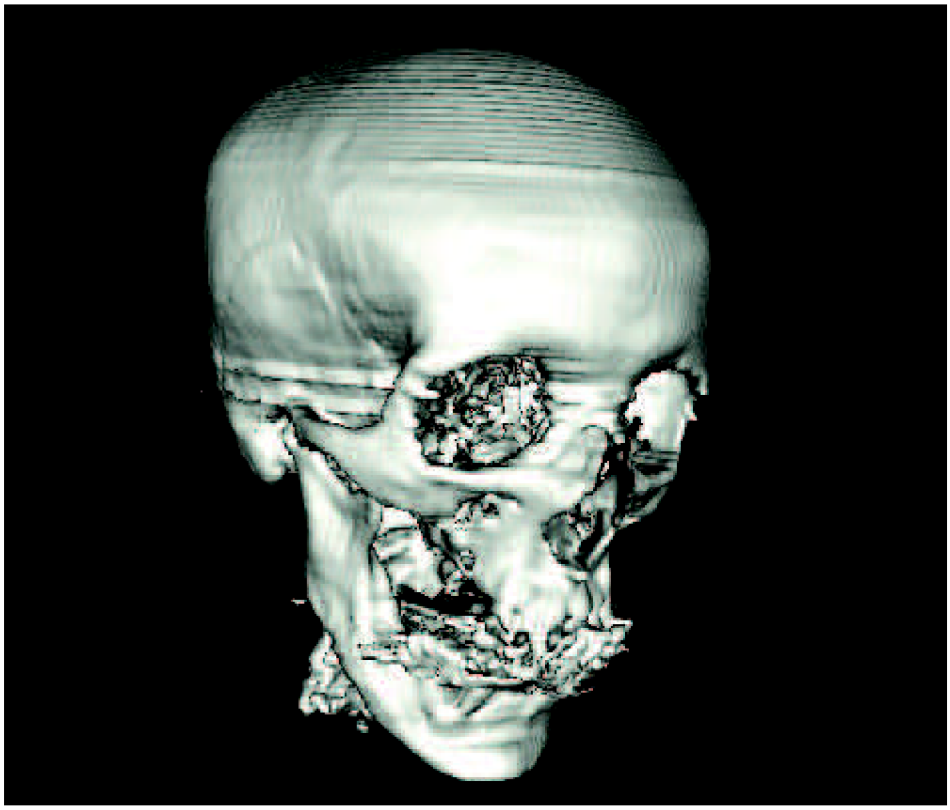
Obrázek 4.15: Plyšový medvídek, CT 128x128x62, GRAD



Obrázek 4.16: Blok motoru, CT 256x256x128, LUT

4.3 Kvalita zobrazování

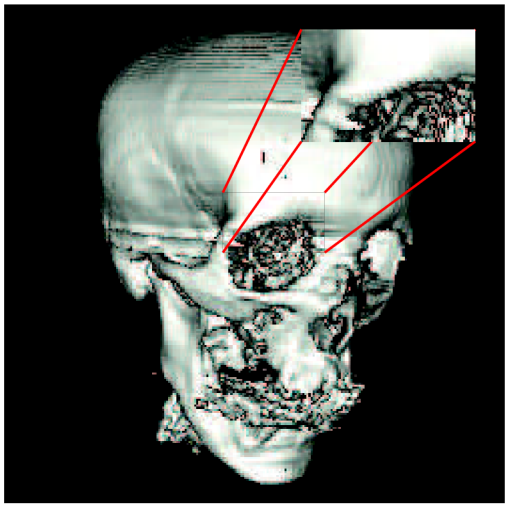
Aplikace poskytuje i možnost pořídit kvalitní výstupy, tentokrát už ovšem bez možnosti interaktivní práce. Pořízení jednoho snímku v závislosti na grafické kartě trvá i půl vteřiny. Kvalita se odvíjí od počtu vzorků na dráze paprsku. Příkladem může být obrázek 4.17. Použit byl opět testovací snímek hlavy v rozlišení 256x256x106, 8 bitů na vzorek, počet vzorků na dráze paprsku je 1024.



Obrázek 4.17: Lebka vyrenderovaná metodou gradientní stínování.

4.4 Zhodnocení implementace

Ač jsem se snažil při implementaci v co největší míře vyhnout artefaktů, ve výsledcích vizualizace se přesto objevily. V první řadě jsou to artefakty způsobené vzorkovací frekvencí, obrázek 4.18. Chceme-li v některých případech zachovat interaktivitu, není jiné možnosti než objemová data podvzorkovat. V druhém případě se artefakty objevily v souvislosti s výběrem voxelů na základě 8-bitových intenzit. Malý rozsah intenzit zapříčiní, že spolu s chtěnými jsou vybrané i nechtěné voxely. Na obrázku 4.19 jsem se snažil kostem přiřadit šedou barvu a stehenní tepně červenou.



Obrázek 4.18: Artefakt vzniklý podvzorkováním.

Obrázek 4.19: Artefakty způsobené malým rozsahem intenzit objemových dat.

5 Závěr

Cílem práce bylo navrhnout aplikaci s grafickým uživatelským rozhraním, která umožní interaktivní práci při vizualizaci objemových dat metodou ray casting. Aplikace obsahuje editor přenosových funkcí a čtyři vizualizační metody (MIP, SIP, LUT a Gradientní stínování). Při návrhu bylo přihlédnuto také k možnostem vizualizace na slabším hardwaru. V neposlední řadě může aplikace sloužit pro vytváření přenosových funkcí pro jiné vizualizační systémy.

Práce byla strukturovaná tak, aby si čtenář udělal přehled nad metodami používanými pro vizualizaci objemových dat. Snažil jsem se, aby čtenáři bylo po přečtení zhruba jasné, jak podobné implementace docílit vlastní silou. Z tohoto důvodu jsem také záměrně spojil kapitolu návrhu s implementací. Čtenář by tedy měl vidět, jak se návrh v programovém kódu projeví.

Oblasti práce týkající se jazyka GLSL, toolkitu Qt a OpenGL jsou popsány v nezbytné míře. Pro případ, že by čtenáři z výkladu nebylo něco jasné, jsem se snažil uvádět odkazy na literaturu, kde je možné potřebné informace dohledat.

Implementovaná aplikace pracuje pouze s 8-bitovými vzorky objemových dat. Mezi první vylepší by mělo patřit rozšíření až na 16-bitové. S touto úpravou souvisí i úprava editoru přenosové funkce v uživatelském rozhraní. Mezi další vylepšení by měla patřit nová metoda zmenšování velkých objemů, protože stávající není zcela optimální. Volume rendering je rychle se rozvíjející oblast počítačové grafiky a tak vylepšení pro implementovanou aplikace je velké množství. Patří sem například použití globálního osvětlení, vícefázový volume rendering a velkou výzvou je automatické generování přenosových funkcí.

Literatura

[1] Žara J., Beneš B., Sochor J., Felkel P.: Moderní počítačová grafika. 2. vyd. Praha, Computer press 2004, 612 s., ISBN 80-251-0454-0

[2] Kršek P.: Vizualizace a CAD [online]. cit. 13.12.2009. Dostupný z WWW: <https://www.fit.vutbr.cz/study/courses/VIZ/private/lecture/viz_slide_vizualizace_zaklady_print.pdf>

[3] Wikipedia contributors: Visualization (computer graphics) [online]. 13.11.2009 [cit. 13.12.2009]. Dostupný z WWW: <http://en.wikipedia.org/w/index.php?title=Visualization_%28computer_graphics%29&oldid=325699415>

[4] Roettger S.: The Volume Library [online]. 27.1.2006 [cit. 13.12.2009]. Dostupný z WWW: <http://www9.informatik.uni-erlangen.de/External/vollib/>

[5] Lacroute P.: Fast Volume Rendering Using A Shear-Warp Factorization Of The Viewing Transformation [online]. 9.1995 [cit. 16.12.2009]. Dostupný z WWW: <http://graphics.stanford.edu/papers/lacroute_thesis/lacroute_thesis.pdf>

[6] Wikipedia contributors: Volume ray casting [online]. 12.8.2009 [cit. 28.12.2009]. Dostupný z WWW:<http://en.wikipedia.org/w/index.php?title=Volume_ray_casting&oldid=307542954>

[7] Engel K.: Pre-Integrated Volume Rendering. Data [online]. 20.4.2001 [cit. 2.1.2010]. Dostupný z WWW:<http://wwwvis.informatik.uni-stuttgart.de/~engel/pre-integrated/data.html>

[8] Triers P.: GPU raycasting tutorial [online]. 4.2009 [cit. 3.1.2010]. Dostupný z WWW: <http://www.daimi.au.dk/~trier/?page_id=98>

[9] Krüger J., Westermann R.: Acceleration Techniques for GPU-based Volume Rendering [online]. 2003 [cit. 3.1.2010]. Dostupný z WWW:<http://portal.acm.org/citation.cfm?id=1081482>

[10] Jones R.: OpenGL FrameBuffer Object 101 [online]. 22.11.2006 [cit. 3.1.2010]. Dostupný z WWW:<http://www.gamedev.net/reference/articles/article2331.asp>

[11] Ramires A.: GLSL Tutorial [online]. cit. 3.1.2010. Dostupný z WWW:<http://www.lighthouse3d.com/opengl/glsl/>

[12] Ikits M., Magallon M.:The OpenGL Extension Wrangler Library [online]. 31.12.2009 [cit. 3.1.2010]. Dostupný z WWW:<http://glew.sourceforge.net/>

[13] Lorensen W., Cline H.: Marching cubes: A high resolution 3D surface construction algorithm [online]. 6. 1987 [cit. 3.1.2010]. Dostupný z WWW: <http://portal.acm.org/citation.cfm?id=37422>

[14] Herout A.: Zobrazování a zpracování objemových dat [online]. 24.11.2009 [cit. 3.1.2010]. Dostupný z WWW: <https://www.fit.vutbr.cz/study/courses/PGR/private/lect/PGR-Volumetric.pdf>

[15] Wikipedia contributors: Maximum intensity projection [online]. 29.4.2010 [cit. 19.5.2010]. Dostupný z WWW:<http://en.wikipedia.org/wiki/Maximum_intensity_projection>

[16] Engel K., Hadwiger M., Kniss J., Rezk-Salama C., Weiskopf D.: Real-Time Volume Graphics. Wellesley, A K Peters 2006, 497 s., ISBN 1-56881-266-3

[17] Engel K.: Advanced Volume Rendering Techniques [online]. 2003 [cit. 2. 5. 2010]. Dostupný z WWW: <http://www.vis.uni-stuttgart.de/vis03_tutorial/engel_advanced.pdf>

[18] Shreiner D., Woo M., Neider J., Davis T.: OpenGL: průvodce programátora. 1. vyd. Brno, Computer Press 2006. 679 s. ISBN 80-251-1275-6

[19] Wikipedia contributors: Blinn–Phong shading model [online]. 18. 5. 2010 [cit. 19. 5. 2010]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model>

[20] Meissner M.: Datasets [online]. 21. 3. 2005 [cit. 12. 4. 2010]. Dostupný z WWW: <http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/datasets.html>

[21] Institut für Visualisierung und Interaktive Systeme: OpenGL GLSL Debugger [online]. 16. 2. 2010 [cit. 8.5.2010]. Dostupný z WWW: <http://www.vis.uni-stuttgart.de/glsldevil/index.html>

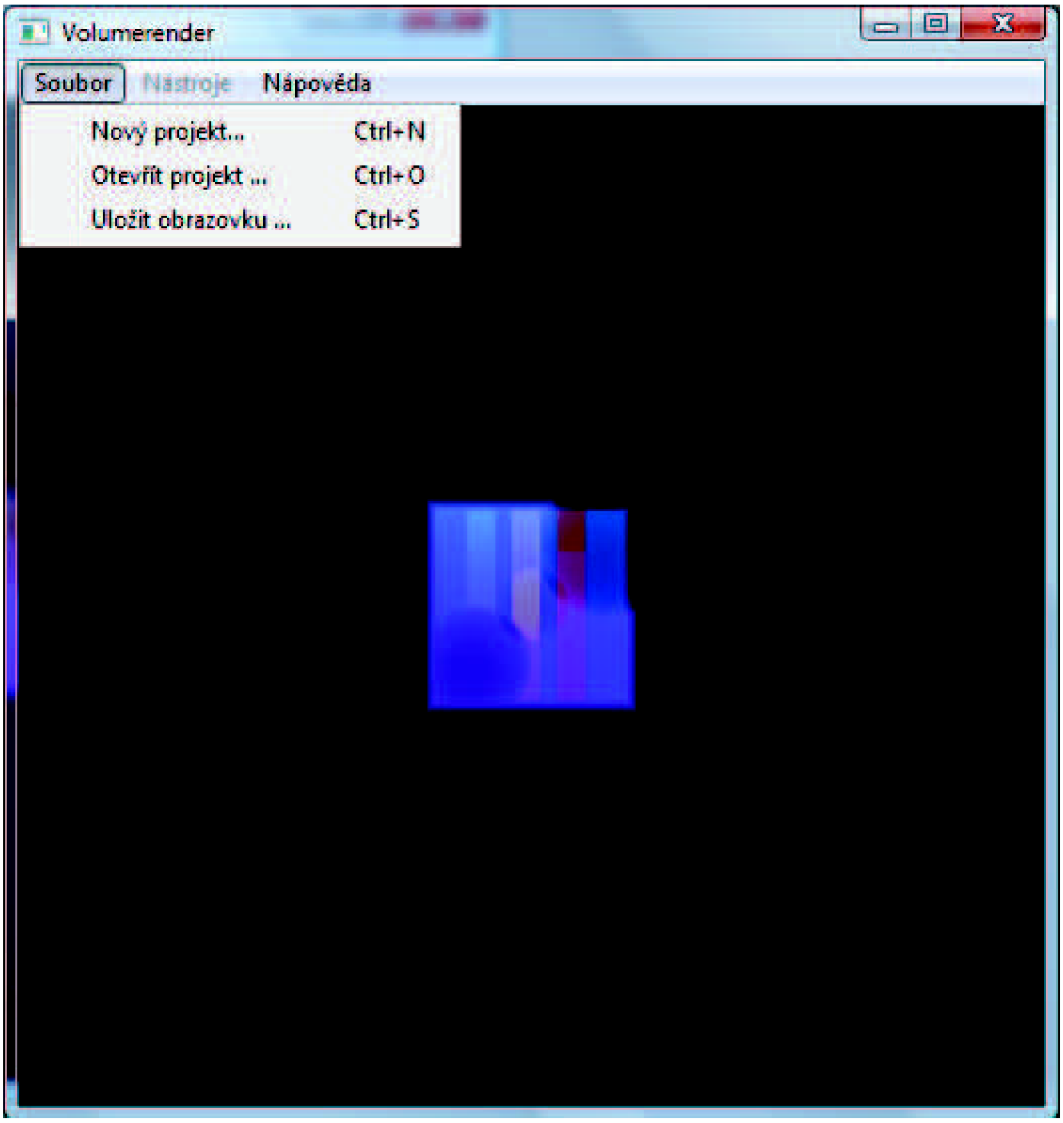
[22] Nokia Corporation: Qt Linguist Manual [online]. 2010 [cit. 28. 4. 2010]. Dostupný z WWW: <http://doc.qt.nokia.com/4.6/linguist-manual.html>
[23] Přispěvatelé Wikipedie: Trackball [online]. 4. 3. 2010 [cit. 10. 5. 2010]. Dostupný z WWW: <http://cs.wikipedia.org/w/index.php?title=Trackball&oldid=5036396>
[24] Lehenbauer D.: Rotating the Camera with the Mouse [online]. 15. 12. 2009 [cit. 16. 5. 2010]. Dostupný z WWW: <http://viewport3d.com/trackball.htm>
[25] Silicon Graphics Incorporated: Examples [online]. 1997 [cit. 21. 4. 2010]. Dostupný z WWW: <http://www.opengl.org/resources/code/samples/glut_examples/examples/examples.html>
[26] Kazík J.: Vizualizace objemových dat pomocí volume renderingu, diplomová práce, Brno, FIT VUT v Brně, 2009

Seznam příloh

Příloha 1. Uživatelský manuál.
Příloha 2. DVD se zdrojovými soubory aplikace, programovou dokumentací, textem diplomové práce a ukázkovými daty.

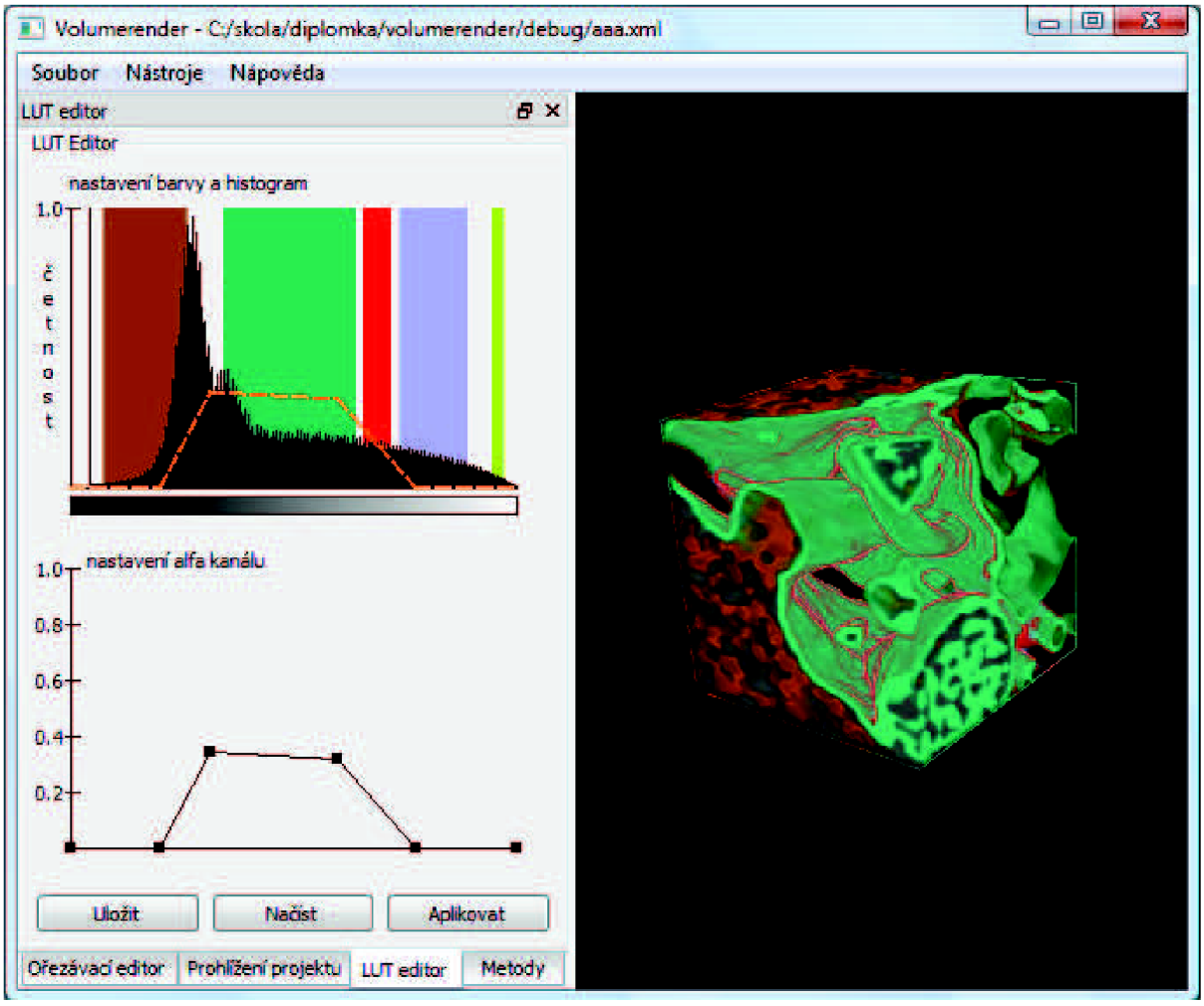
Příloha 1.: Uživatelský manuál

Po spuštění programu se spustí hlavní okno aplikace (obrázek 5.1). Značí, že počítač podporuje OpenGL 2.0 a vše je připraveno pro vizualizaci objemových dat. Z hlavní nabídky `Soubor` lze otevřít vytvořený projekt, nebo začít vytvářet nový. Každý projekt se vztahuje právě k jedněm objemovým datům. Položku menu `Uložit obrazovku` využijete hlavně při ukládání hotových vizualizací. Pomocí `nápovědy` lze zjistit, jak velkou 3D texturu váš systém podporuje. Tím je dáno, jak rozměrná objemová data lze v nezmenšené podobě vizualizovat. Rotace s objemovými daty lze provádět pomocí stisknutí a tažení levého tlačítka myši. Přiblížení je možné stisknutím a tažením pravého tlačítka.



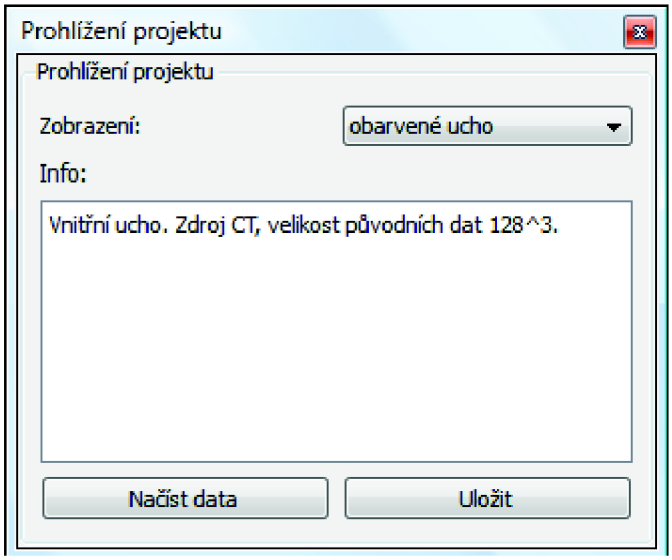
Obrázek 5.1: Hlavní okno aplikace.

Budeme postupovat cestou otevření již stávajícího projektu. Pokud dojde k úspěšnému otevření projektu, zobrazí se okno podobné obrázku 5.2. Na obrázku si všimněte záložek v levém spodním rohu. Postupně je projdu a vysvětlím jejich funkci.



Obrázek 5.2: Příklad stavu aplikace při editaci projektu.

Záložka prohlížení projektu (obrázek 5.3) slouží k editaci projektu na úrovni zobrazení. Tlačítkem `Načíst data` vložíme do projektu objemová data, která jsme si přichystali a existuje k nim `.dat` souboru, který je popisuje. Rolovací menu `Zobrazení:` nabízí již uložená zobrazení v projektu. Lze mezi nimi přepínat a prohlížet si tak již vytvořené vizualizace. Tlačítkem `Uložit` je možné k danému zobrazení vložit nebo přepsat stávající poznámku.



Obrázek 5.3: Karta prohlížení projektu vizualizace objemových dat.

Další záložku, kterou představím, je záložka metody (obrázek 5.4). Souží pro volbu zobrazovací metody a nastavení jejich parametrů. Metodu lze do projektu uložit a nebo přepsat právě editované zobrazení. Aktuální nastavení metody přejde v platnost tlačítkem `Aplikovat`.

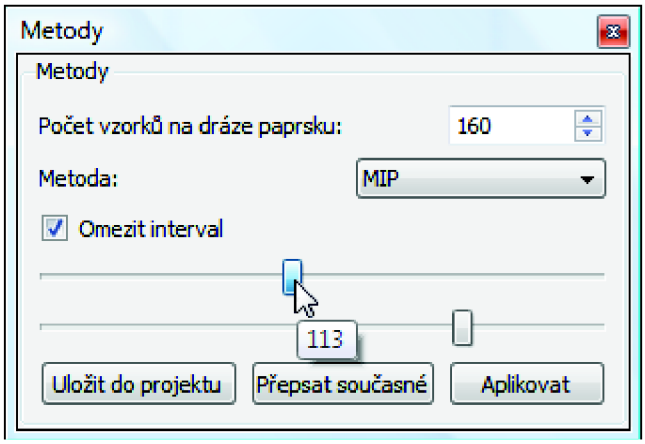
Počet vzorků na dráze paprsku ovlivňuje kvalitu zobrazení. Měl by být minimálně dvakrát vyšší než je maximální rozměr objemových dat.

Metoda MIP zobrazuje maximální hodnotu intenzity voxelu na dráze paprsku. Zkoumaný interval intenzit lze omezit táhly. Metoda SIP zobrazuje sumu intenzit voxelů na dráze paprsku. Interval intenzit lze opět omezit. K dispozici je i nastavení konstanty pro násobení výsledné sumy.

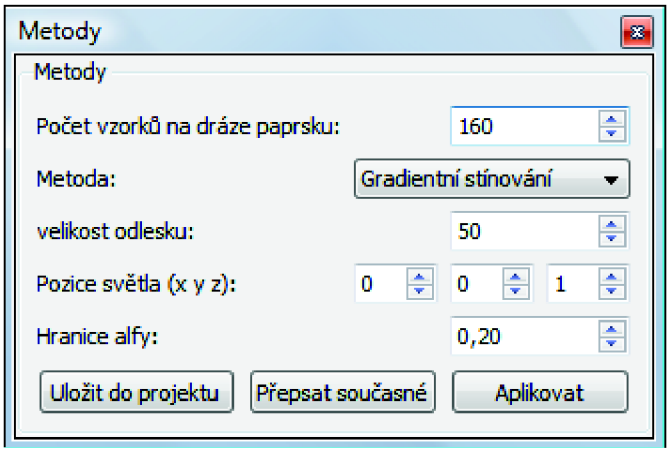
Metoda Gradientní stínování přidává parametry spojené s osvětlení povrchu, jako je lesklost materiálu a pozice světla.

Alfa průhlednost slouží k definování povrchů za pomocí přenosové funkce. Pokud tedy nastavíme například hranici alfy na 0,20, tak by se měl gradient počítat ve všech bodech, jejichž alfu jste v LUT editoru (viz dále, obrázek 5.7) určili vyšší než 0,20.

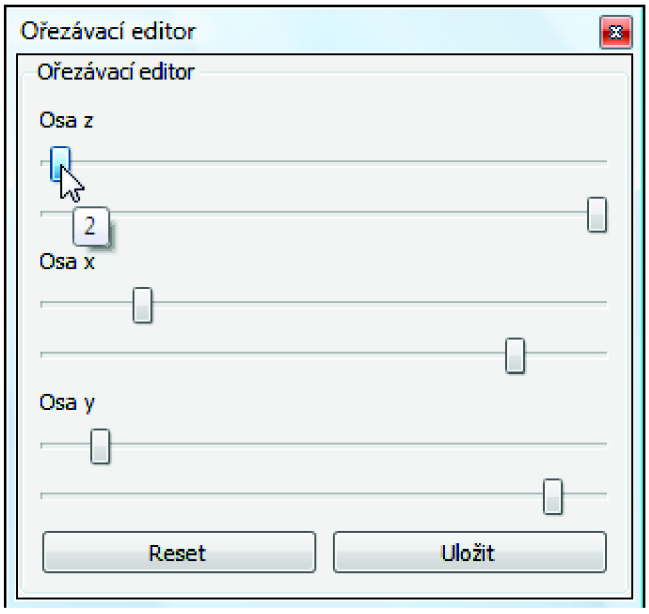
Než přejdeme k LUT editoru, zmíním kartu ořezávací editor (obrázek 5.6). Slouží k ořezání objemových dat mimo zájem pozorovatele. Vhodným ořezáním lze ušetři značné výpočetní prostředky. Aktuální nastavení lze k zobrazení uložit. Tlačítko `reset` slouží k obnovení základního nastavení.



Obrázek 5.4: Podoba karty metody pro metodu MIP.



Obrázek 5.5: Podoba karty metody pro metodu Gradientní stínování.

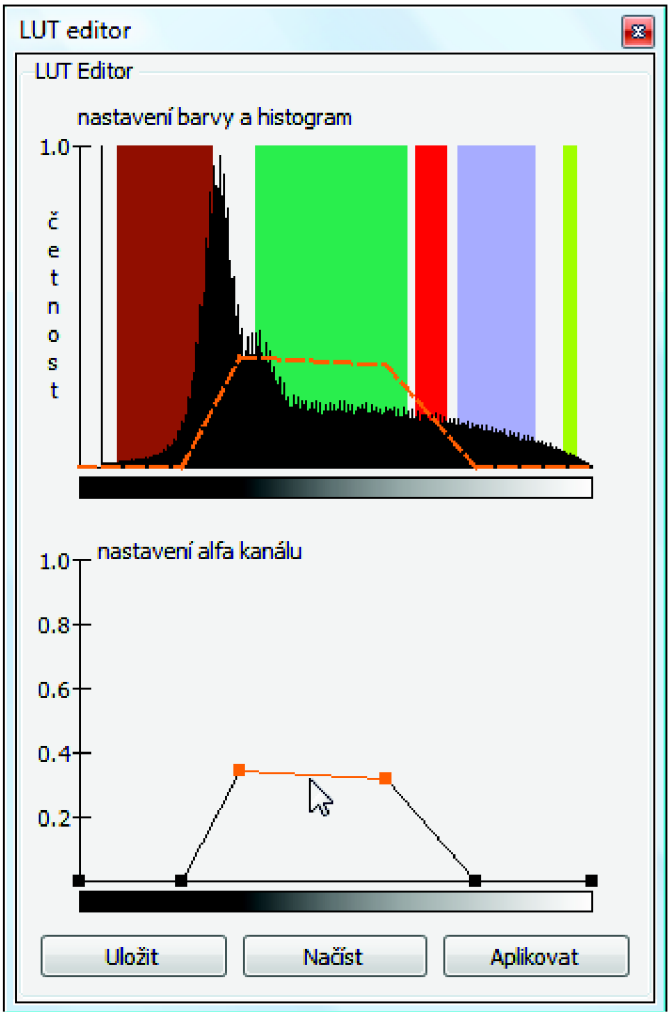


Obrázek 5.6: Karta ořezávacího editoru.

Karta LUT editoru se zobrazí po vybrání metody LUT nebo gradientní stínování z nabídky metod. Karta je tvořena dvěma komponentami. První zobrazuje histogram objemových dat a umožňuje jim přiřazovat barvu. Histogram (vykreslen černou barvou) udává počet výskytů voxelů dané intenzity vůči počtu voxelů s nejčastější intenzitou. Četnost 1.0 má tedy voxel, který můžeme v datech nalézt nejčastěji.

Barvu objemovým datům lze přiřadit označením oblasti histogramu. Dvojklikem na plochu s barvou je možná její editace pomocí standardního dialogu pro výběr barvy. Smazání obarveného region lze provést klávesou `delete` v momentě, kdy je kurzor myši nad danou oblastí.

Druhá komponenta na kartě slouží k editaci alfa kanálu přenosové funkce (na obrázku 5.7 dole). Čím jsou hodnoty nižší, tím je průhlednost voxelů dané intenzity větší. Lomenou čáru uživatel může upravovat táhnutím bodů zlomu nebo celé přímky. Mazání bodů je opět pomocí klávesy `delete` v okamžiku, kdy je kurzor nad bodem označující zlom. Přidávání bodů zlomu se provádí kliknutím myši na danou pozici. Vytvořenou předintegrovanou přenosovou funkci lze uložit nebo načíst ze samostatného souboru.



Obrázek 5.7: Karta pro vytváření přenosových funkcí.

Nyní mi už jen zbývá vám popřát mnoho hezkých vizualizací.