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

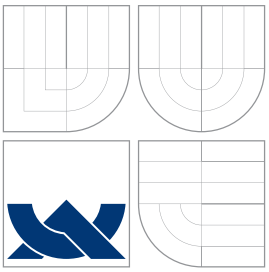
VIZUALIZACE ZNAČENÝCH BUNĚK MODELOVÉHO ORGANISMU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

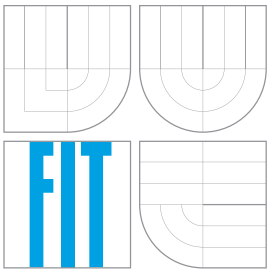
AUTOR PRÁCE
AUTHOR

Bc. RADEK KUBÍČEK

BRNO 2007



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 ZNAČENÝCH BUNĚK MODELOVÉHO ORGANISMU

VISUALIZATION OF MARKED CELLS OF A MODEL ORGANISM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK KUBÍČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ADAM HEROUT, Ph.D.

BRNO 2007

Zadání diplomové práce

Řešitel	**Kubíček Radek, Bc.**
Obor	Počítačová grafika a multimédia
Téma	**Vizualizace značených buněk modelového organismu**
Kategorie	Počítačová grafika

Pokyny:

1. Prostudujte problematiku zobrazování volumetrických dat v biologii a medicíně.
2. Seznamte se s problematikou studia modelového organismu „Caenorhabditis elegans“ a metodou značení pomocí GFP (fluoreskující protein).
3. Navrhněte algoritmy pro efektivní zobrazování značených částí organismu ve volumetrickém zobrazení.
4. Implementujte navržené algoritmy a integrujte je do nástroje pro zobrazování dat z dekonvolučního konfokálního mikroskopu.
5. Vyhodnoťte vlastnosti vyvinutých algoritmů na demonstračních datech pořízených mikroskopem.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek pro prezentování projektu.

Literatura:

- dle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1.-3., částečně bod 4.

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 paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí	**Adam Herout, Ing., PhD.**, UPGM FIT VUT
Datum zadání	28. února 2007
Datum odevzdání	22. května 2007

Licenční smlouva

Licenční smlouva v kompletním znění je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Výňatek z licenční smlouvy:

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.

2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.

3. Autor souhlasí se zvěřejněním díla v databázi přístupné v mezinárodní síti:

   ■ ihned po uzavření této smlouvy

   □ 1 rok po uzavření této smlouvy

   □ 3 roky po uzavření této smlouvy

   □ 5 let po uzavření této smlouvy

   □ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací).

4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Abstrakt

Tato diplomová práce, zabývající se zobrazením volumetrických (objemových) dat, má za úkol zobrazit a zvýraznit značené buňky modelových organismů v datech sejmutých konfokálním mikroskopem. Vstupní data tvoří jednolitý volumetrický celek složený z jednotlivých řezů, který je nutno vhodnou metodou zobrazit a poté identifikovat a zvýraznit buňky označené metodou GFP (*Green Fluorescent Protein*) nebo fluoreskováním chlorofylu.
Hlavním cílem této práce je nalézt pokud možno optimální efektivní metodu umožňující toto zvýraznění, pracující ideálně bez manuálního zásahu uživatele. Vzhledem ke struktuře dat je však tento cíl jen těžko splnitelný, proto postačí nalezení metody operující v manuálním režimu. Nakonec je třeba dosažené výsledky práce zaintegrovat do nástroje FluorCam pro zobrazování dat z dekonvolučního konfokálního mikroskopu.

Klíčová slova

volumetrická data, objemová data, zobrazení volumetrických dat, zobrazení objemových dat, volumetrické zobrazení, 3D textury, zobrazení biologických dat, Caenorhabditis elegans, GFP, Green Fluorescent Protein, FluorCam, GPU akcelerace, shader

Abstract

This master thesis is focused on volumetric data rendering and on highlighting and visualization of the selected cells of the model organisms. These data are captured by a confocal deconvolution microscope. Input data form one large volumetric block containing separate slices. This data block is rendered by an applicable method and then are identified and visualized the cells marked by the GFP (Green Fluorescent Protein) process or by chlorophyle fluorescency.
The principal aim of this work is to find out the preferably optimal effective method enabling this highlighting, most preferably working without a manual check. Due to the data structure, this ambition seems hardly realizable, so it suffices to find out a manual working method. The last step is to embed the results of this work into FluorCam application, the confocal deconvolution microscope data visualizer.

Keywords

volumetric data, volumetric data rendering, 3D textures, biological data rendering, GFP, Green Fluorescent Protein, Caenorhabditis elegans, FluorCam, GPU acceleration, shader

Citace

Radek Kubíček: Vizualizace značených buněk modelového organismu, diplomová práce, Brno, FIT VUT v Brně, 2007

Vizualizace značených buněk modelového organismu

Prohlášení

Čestně tímto prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Adama Herouta, PhD. a Ing. Martina Trtílka. Uvedl jsem též veškeré literární prameny a publikace, ze kterých jsem čerpal.

.....
Radek Kubíček
21. května 2007

Poděkování

Chtěl bych poděkovat svému školiteli Adamu Heroutovi za cenné rady, zadavateli práce Martinu Trtílkovi za biologické teoretické základy a poskytnutí materiálů a vstupních dat. Dále Dr. Christof-Rezk Salamovi za laskavé poskytnutí informací a hlavně své rodině a přítelkyni za jejich ohromnou trpělivost.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Na počítači je možné v současné době zobrazit téměř všechna data, která lze získat různými způsoby. Některá jsou zobrazitelná přímou renderovací technikou, na jiná se musí použít speciálních vizualizačních technik. Mezi tuto druhou skupinu dat se řadí hlavně ta z oblasti přírodních věd a medicíny. V jejich případě se totiž většinou jedná o data objemová, nařezaná na jednotlivé řezy a uložená nejčastěji v trojrozměrné prostorové mřížce. Data tohoto typu, i když to tak na první pohled nevypadá, jsou poslední dobou stále častěji využívána a některá vědecká odvětví by se bez nich téměř neobešla.

Zobrazení volumetrických, nebo také objemových či prostorových dat je účinnou a snadno dostupnou technikou, jak prozkoumat komplexní struktury takto uložených dat. Pokud by měla být srovnávána s klasickými technikami zobrazujícími 3D povrch, lze jako její hlavní výhodu uvést to, že dokáže pracovat s poloprůhlednými částmi, které odkrývají hlouběji zanořené struktury a poskytují tak více informací o vzájemných prostorových vztazích takto uložených dat. Na druhou stranu se ale dá považovat za její nevýhodu to, že tato vizualizační technika poloprůhledných dat potřebuje v drtivé většině případů ohromné množství vstupních dat. I proto je tento způsob zobrazení náročnější na grafický a výpočetní výkon a v neposlední řadě i na dostupnou paměť, převážně texturovací, umístěnou přímo na grafickém čipu. To už ale v dnešní době nečiní tak velký problém.

Velikost dat se postupným zdokonalováním snímacích zařízení neustále zvětšuje, protože pracují s vyšší přesností i snímací frekvencí, tudíž na více bitech a s větším počtem snímků za sekundu. Stejně tak však roste i výkon výpočetních a zobrazovacích zařízení, ať již procesorů, grafických karet nebo pamětí. Také požadavky na zobrazení volumetrických dat jsou stále vyšší, jedná se totiž o relativně jednoduchou metodu, která však dokáže zobrazit data jinými způsoby nezobrazitelná. Větší množství a objem dostupných dat také znamená vyšší poptávku po 3D vizualizaci. Z tohoto důvodu musejí být neustále objevovány nové metody, případně modifikovány a inovovány ty stávající tak, aby dosahovaly vyšší přesnosti a dokonalosti zobrazení, rychlosti a snadnější uživatelské manipulace.

Tato práce je zaměřena na specifickou část rozsáhlé oblasti vizualizace volumetrických dat, a to zvýraznění určitých částí vstupních dat dle zadaných kritérií. Jedná se především o nalezení a zvýraznění požadovaných oblastí ve vstupních datech. To může být automatické, na základě různých heuristických či detekčních metod, nebo plně manuální, kdy uživatel aplikace sám tyto oblasti identifikuje a bude záležet pouze na něm, jakým způsobem provede její zvýraznění. Takto nalezená oblast by poté měla být specifickým způsobem vizuálně rozlišena od okolí.

Jako vstupní data budou použity snímky buněčného pletiva rostlin se zvýrazněnými částmi pomocí chlorofylu, po dosažení určitého stádia implementace poté budou použity snímky buněk modelového organismu *Caenorhabditis elegans* obarveného metodou GFP. Oba typy dat jsou pořízené konfokálním dekonvolučním mikroskopem.

1.1 Cíle a požadavky

1.1

Tato část obsahuje shrnutí požadavků na cíle této práce, míru jejich splnění v současném stádiu a pravděpodobný odhad splnitelnosti těch zbývajících. Požadavky se ovšem s časem a postupným stavem implementace mění a rozšiřují tak požadovanou funkčnost metody.

Primárním cílem práce je vizualizace značených buněk v objemových datech modelového organismu. Preferencí však v rámci momentálních požadavků není ani tak rychlost zobrazení, jako spíše jeho vizuální kvalita. Což však neznamená, že na rychlost vizualizace i samotné interakce s uživatelem nebude kladen důraz. Pouze se tento optimalizační požadavek může začít projevovat až v některé z budoucích fází návrhu. Implementace by dále měla být provedena nejlépe v prostředí C++ Builder. V něm je totiž napsána i cílová aplikace FluorCam, jejíž součástí by se tato práce měla stát. Případně je možno vytvořit nezávislou dynamickou knihovnu, která bude použitelná v jakémkoli vývojovém prostředí, což se jeví jako schůdnější způsob.

Docela samozřejmým požadavkem je rozumná míra intuitivnosti pro uživatele. Dále také co nejnižší hardwarové požadavky aplikace, ovšem toto si částečně odporuje s požadavkem pro vizuální kvalitu výsledku. I tak ale bude kladen na nízké hardwarové nároky patřičný důraz, aby bylo možné výslednou aplikaci provozovat na větším množství konzumních grafických karet. Pokud by to však bylo nezbytně nutné, je možno práci koncipovat tak, aby využívala shaderů verze 3.0. I ta je totiž dnes již podporována velkým množstvím grafických karet.

1.2 Rozvržení textu

1.2

V kapitole 2 je popsán obecný princip vizualizace volumetrických dat, stručně nastíněny nejpoužívanější dostupné metody a oblast jejich použití. Ke každé z těchto metod jsou též uvedeny některé jejich výhody i nevýhody. Na konci této kapitoly je poté zhodnoceno, která z těchto metod by mohla nejlépe splňovat cíl této práce.

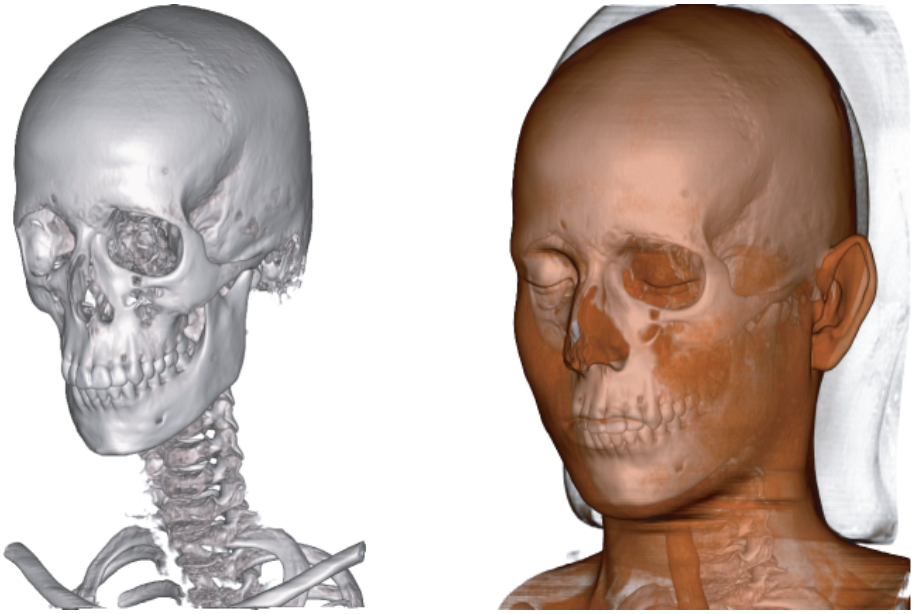
Další kapitola 3 obsahuje princip značení vstupních dat. Jsou zde popsány použité metody ke značení dat používaných v této práci. Dále zde jsou informace o modelovém organismu *Caenorhabditis elegans*, kdy bude ujasněno, o jaký organismus se jedná, čím je zvláštní a proč bude právě on později použit jako vstupní data této práce.

Druhou část dokumentu tvoří samotný návrh metody pro zobrazení volumetrických dat. Kapitola 4 popisuje princip renderování objemových dat, ale zatím pouze v takové podobě, jak byla načtena. Takto jsou data připravena k samotné vizualizaci, o které pojednává kapitola 5. V nejlepším případě by tato vizualizace mohla navazovat na některou z již exitujících a dobře popsaných technik a určitým způsobem ji modifikovat nebo vylepšit. Je však nutné zkombinovat více principů. V těchto dvou kapitolách jsou tedy popsány všechny použité techniky a jejich modifikace. Též zde jsou uvedena možná vylepšení, která byla během návrhu nalezena a mohla by být přínosná pro budoucí práci.

Poslední kapitola 6 shrnuje dosažené výsledky. Dále zde také je zhodnocen přínos této práce a míra splnění vstupních požadavků.

Kapitola 2

Zobrazování volumetrických dat



Obr. 2.1: Ukázka vizualizace volumetrických dat v medicíně [4]

Cílem této kapitoly je seznámení se s problematikou zobrazování volumetrických dat, principem činnosti jednotlivých metod a zhodnocení jejich vizuálních a výkonových vlastností.

Nejprve jsou v sekci 2.1 nastíněny jednotlivé principy z oblasti zobrazení objemových dat a popsána obecná klasifikace používaných metod. V další sekci 2.2 jsou popsány způsoby ukládání objemových dat v paměti a jejich získávání z ní.

Následující části dále rozebírají nejpoužívanější metody pro vizualizaci objemových dat. Jejich cílem je ukázat, jak jednotlivé metody pracují a čím se od sebe liší. V části 2.3 jsou popsány metody, které nejprve hledají povrch objemových dat a ten následně zobrazují klasickým způsobem, tedy vyrenderováním povrchových primitiv. Další sekce 2.4 je věnována přímému zobrazení objemových dat na průmětnu bez pomocného hledání jejich povrchu.

V poslední sekci 2.5 se nachází srovnání jednotlivých metod podle vhodnosti pro účely této práce a rozhodnutí, která metoda bude použita k její implementaci. Zvolená metoda bude podrobněji popsána v kapitole 4.

2.1 Obecné principy

2.1

Vizualizace volumetrických dat je způsob, respektive skupina technik, jak získat podstatné informace z bloku volumetrických dat, které bývají při zobrazení standardními metodami skryty.

Za volumetrická (objemová) data lze považovat všechna taková, která jsou uložena ve vícerozměrných skalárních nebo vektorových mřížkách. Tato data mohou být získána z různých zdrojů, např. medicinská data z CT (Computed Tomography), MRI (Magnetic Resonance Imaging) nebo PET (Positron Emission Tomography), biologická data získaná různými optickými snímači, např. konfokálním dekonvolučním mikroskopem, nebo data seismická. Typický příklad zobrazení objemových dat používaný v medicíně je na obrázku 2.1. Data mohou být generována i uměle. Všechna tato data jsou považována za skalární. Skutečná vektorová data, typicky čtyřrozměrná, jsou získána průmyslovými počítačovými simulacemi, např. CFD (Computational Fluid Dynamics), kde roli čtvrtého rozměru zastupuje časová složka.

Principem vizualizace volumetrických dat je nalezení zobrazení převádějící data z takové vícerozměrné mřížky na dvourozměrnou průmětnu. Od tradičního zobrazování objektů uložených pomocí povrchové reprezentace se tedy liší v mnoha aspektech. Reprezentací dat, jejich velikostí a způsobem uložení, ale hlavně konečnou fází vizualizace, kdy je třeba sáhnout po odlišné zobrazovací technice.

Ve vědecké a průmyslové vizualizaci se zobrazování objemu používá převážně pro získání lepšího náhledu na sledovaná data, např. jejich rozložení v prostoru, identfikaci významných hodnot nebo způsobu jejich shlukování v prostoru. Ve fotorealistické oblasti jde spíše o co nejvěrnější zobrazení přírodních fenoménů jako mlha, mraky nebo oheň. Přínosem některých metod je též věrné zobrazování vlastností průhledných a poloprůhledných ploch, např. rozptyl, lom a útlum světla.

Metody pro vizualizaci objemových dat se dají klasifikovat podle různých kritérií. Nejčastěji používaným kritériem je podle způsobu promítání, kdy se rozlišuje, jakým způsobem jsou objemová data promítnuta na průmětnu. Tomuto rozdělení se věnují části 2.3 a 2.4. Další klasifikací může být např. podle dominantního průchodu, kdy jsou rozlišovány metody orientované objemově nebo obrazově. Více o této klasifikaci lze najít v [8].

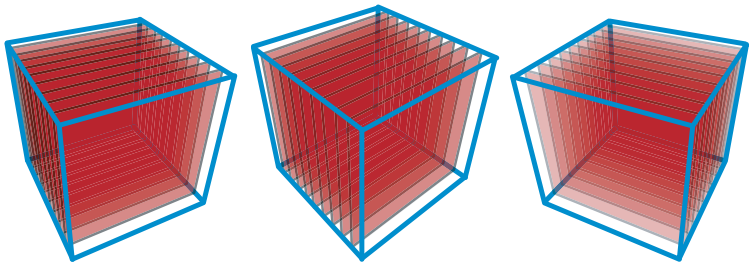
2.2 Uložení dat v paměti

2.2

Narozdíl od čistě softwarových řešení je potřeba v hardwarově akcelerovaných technikách co nejvíce přenést výpočetně drahé zpracování volumetrických dat na čip grafické karty. Toho lze docílit nejsnadněji tím způsobem, že jsou objemová data uložena do textury a ta je následně zaslána do paměti na grafické kartě. Zde jsou možné dva odlišné přístupy:

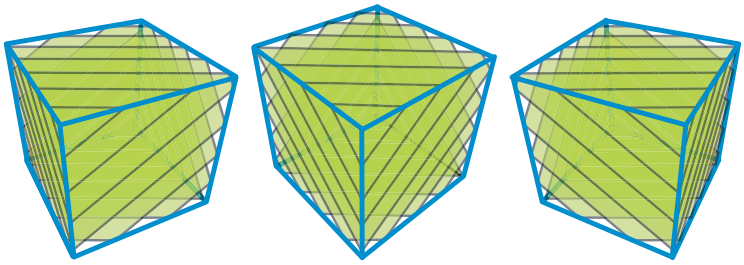
- **2D textury:** V tomto přístupu jsou uloženy jednotlivé řezy do sady 2D textur. Poskytuje tak vysoký výkon, je však nutné mít tyto řezy uloženy pro každou souřadnou osu zvlášť, takže je tato metoda velice náročná na velikost dostupné texturovací paměti. Proto je častěji doporučováno použití 3D textur, které už jsou dnes podporovány prakticky všemi dostupnými grafickými kartami. Grafické znázornění přístupu 2D textur

je na obrázku 2.2. Podle natočení objemu k pozorovateli je vždy vybrána sada textur nejvíce rovnoběžná s průmětnou. Ta je poté namapována na příslušná geometrická primitiva. Primitiva jsou renderována odzadu dopředu s využitím alpha blendingu. Nevýhodou totoho přístupu je také pouze bilineární interpolace.



Obr. 2.2: Přístup založen na 2D texturách [4]

- **3D textury:** V tomto případě jsou data uložena v paměti jako trojrozměrná textura. Grafické schéma tohoto přístupu ukazuje obrázek 2.3. Zde je pro vykreslení využito polygonů rovnoběžných s průmětnou, které jsou vykreslovány nejčastěji odzadu dopředu, popř. je lze vykreslovat i opačným směrem. Během renderování jsou pro každý polygon grafickým hardwarem trilineárně interpolovány hodnoty z 3D textury. Lze také simultánně vyhodnocovat skupinu paprsků vržených z průmětny, z čehož plyne díky vysokému výkonu rasterizace v grafickém čipu ohromné zrychlení. Způsob uložení dat jako 3D textura v paměti byl zvolen i k implementaci této práce z důvodu dostupnosti 3D texturovacích jednotek v současných grafických kartách i té nejnižší kategorie a také díky její nižší paměťové náročnosti. Navíc odpadá v každém snímku hledání nejvíce rovnoběžné sady textur.



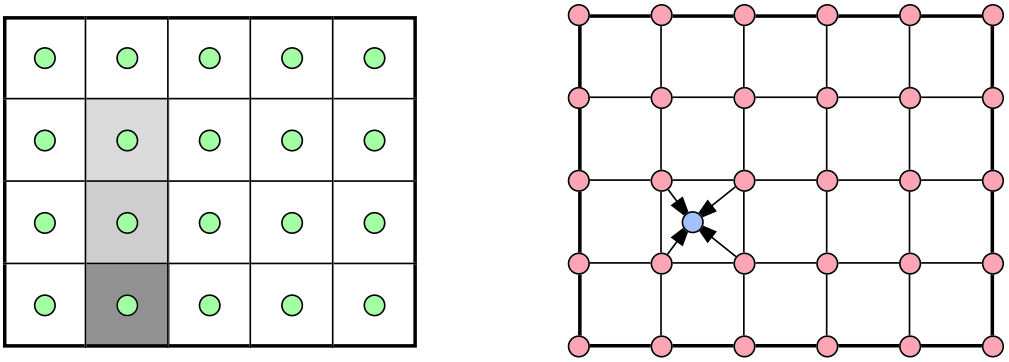
Obr. 2.3: Přístup založen na 3D texturách [4]

Dále je dána skalární spojitá funkce $f'(X) : \mathbb{R}^3 \rightarrow \mathbb{R}$ vracející hodnotu v libovolném bodě objemu X. Typicky je tato funkce implementována pomocí interpolace z diskrétních vstupních hodnot. Následují nejpoužívanější následující způsoby interpolace.

Prvním z nich je interpolace hodnotou nejbližšího souseda, kdy je pro daný bod X nalezen nejbližší vzorek ležící na mřížce a jeho hodnota je považována i za hodnotu v daném bodě X. V tomto případě je na objem nahlíženo jako na sadu voxelů, kde každý voxel má definovanou hodnotu a ta je konstantní přes celý objem voxelu. Tento způsob ilustruje obrázek 2.4(a), kde jsou naměřené hodnoty uprostřed voxelů a roztaženy přes celý jeho

objem. Tento způsob je velice rychlý a jednoduchý, bohužel však interpolace nejbližším sousedem vede k častým artefaktům v obraze, z nichž je nejznámější tzv. schodovitý efekt (*staircase effect*).

Další možností je považovat objem za sadu hodnot v uzlových bodech a při hledání hodnoty v bodě X ji interpolovat z hodnot uzlů v okolí tototo bodu. Nejčastěji používanou interpolací je trilineární, kdy je hodnota určována z osmi nejbližších uzlů. Lze samozřejmě použít i interpolace vyšších, případně nižších řádů, ale vzhledem k tomu, že je tato operace při zobrazování objemu jednou z nejčastějších, byla by „cena“ vyššího řádu interpolace již příliš vysoká. Grafické znázornění tohoto způsobu je na obrázku 2.4(b), kde jsou naměřené hodnoty uložené ve vrcholech a pro body ležící mezi nimi je jejich hodnota získána interpolací z vrcholových hodnot.



(a) Uprostřed voxelu (b) V uzlových bodech

Obr. 2.4: Způsoby rozložení naměřených hodnot v objemu [12]

2.3 Metody hledající povrch

2.3

Metody založené na hledání povrchové reprezentace objemových dat spočívají na nalezení takových míst v objemu, kde hodnoty nabývají požadovaného prahu, též nazývaného isoplocha. Tato místa se poté snaží aproximovat polygonální sítí a následně je zobrazit běžnými povrchovými zobrazovacími technikami.

Tyto metody nemají snahu modelovat objem jako celek ani napodobovat jeho optické vlastnosti. Pro zlepšení vizuálních vlastností lze ale využít např. stínování pomocí Phongova modelu, který umožňuje simulovat i odlesk světel na povrchu. K tomu je však obvykle potřeba implemetace příslušného optického modelu a výpočet normály v každém vrcholu nebo bodu nalezené polygonální sítě.

2.3.1 Napojování isočar

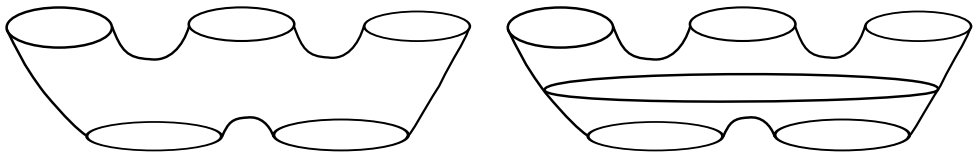
2.3.1

Metoda napojování isočar je založena na myšlence nalézt pro každý dvoudimenzionální průřez objemu množinu uzavřených isočar, které odpovídají zadané prahové hodnotě, a poté tyto isočáry napojovat řez po řezu a vytvářet tak isoplochu.

Tento přístup však skýtá několik problémů. V době, kdy byla tato metoda publikována, bylo problémem již samotné nalezení isočar v dvojrozměrném průřezu, zejména pokud byla

data silně zašuměná nebo měla nízký kontrast. Zde bylo obtížné nalézt isočáry tak, aby tvořily množinu uzavřených křivek. Často byl tedy nutný dodatečný manuální zásah operátora, aby opravil problematická místa [8].

Druhý problém vzniká ve fázi napojování isočar ze sousedních řezů. V některých případech nelze totiž jednoznačně rozhodnout, které části isočar mají být k sobě napojeny, např. jak to ilustruje obrázek 2.5. Vzniká tím chybný povrch s artefakty, což je ovšem nepřijatelné zejména v medicinských aplikacích. Tomuto problému lze zabránit hustějším vzorkováním, což ale přináší mnohem více isočar a tím i více dat a z toho plynoucí pomalejší a náročnější renderování. Nevýhodou je také poměrně komplikovaná implementace, kdy je nutno obsáhnout spoustu speciálních a problematických případů.



Obr. 2.5: Nejednoznačné napojení isočar [8]

2.3.2 Marching cubes

2.3.2

Algoritmus Marching Cubes narozdíl od hledání isočar generuje přímo polygoniální aproximaci isoplochy zadané definovaným prahem jako trojúhleníkovou síť ve třídimenzionálním prostoru. Jeho výhodou je snadná implementace, kde lze dobře využít i možností grafického hardwaru a urychlit tím výpočet. Vygenerované trojúhelníky mají též „vhodný“ tvar, kdy se maximálně podobají trojúhleníkům rovnostranným, bez dlouhých a protáhlých hran. Nevýhodou však je, že jich je ohromné množství a jsou často příliš malé, někdy i na sub-pixelové úrovni.

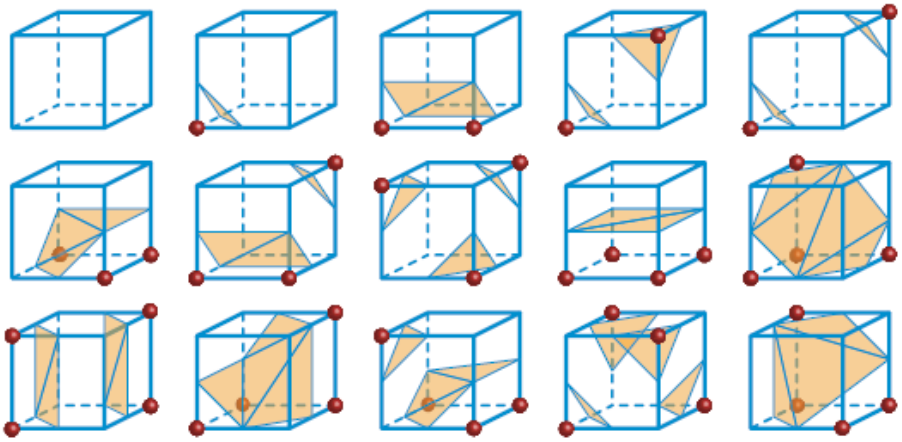
V tomto algoritmu je každá objemová buňka, kterou lze považovat za krychli skládající se z osmi sousedních voxelů, považována za specifickou hodnotu isoplochy. Potřebujeme tedy zjistit, jakým způsobem touto buňkou povrch prochází. Pro každý z osmi vrcholů jsou možné dva stavy: hodnota v něm je větší nebo rovna zadanému prahu (1) a vrchol tedy leží „uvnitř“ nebo na isoploše. Nebo je hodnota ostře menší než zadaný práh (0), a potom vrchol leží „vně“ isoplochy [8]. Z toho plyne, že může existovat $2^8 = 256$ konfigurací, jak může povrch tuto buňku protnout. Tento počet lze dále zredukovat symetriemi pomocí rotací nebo inverze na pouhých 15 konfigurací, které jsou ukázány na obrázku 2.6.

Bohužel bylo zjištěno, že tato zredukovaná sada variant občas generuje nekonzistentní povrchy obsahující díry v isoplochách, takže je často používáno plných 256 variant.

2.3.3 Zhodnocení

2.3.3

Lze říci, že tyto metody se obecně řadí mezi velmi rychlé, jakmile je povrch jednou nalezen a zkonstruován, protože ten je v datech nezávislý na pozici a natočení vůči pozorovateli. Nevýhodou těchto metod však mohou být drobné artefakty, nejčastěji falešné části povrchu a díry, nebo nepřesnosti při snaze o zachycení drobných detailů.



Obr. 2.6: Algoritmus Marching cubes [4]

Tyto metody jsou navíc vhodné pouze k vizualizaci objektů, u nichž je zobrazení pomocí isoploch víceméně přirozené. Pro amorfní objekty, jako mraky nebo mlha, nebo více zašuměná data je lze tedy obecně považovat za nevhodné.

Protože je cílem této práce vizulizace značených buněk, což jsou části ne nutně náležící povrchu zvolené isoplochy, a navíc se v této práci bude aplikovat princip přenosových funkcí modifikujících zobrazení dat podle požadavků uživatele, lze tyto metody klasifikovat pro její zpracování jako krajně nevhodné. Dokáží sice velmi dobře využít hardwarové akcelerace, ale většinou pouze pro vlastní renderování, a samotný průchod objemem a generování povrchové reprezentace je prováděno hlavním procesorem.

2.4 Přímé zobrazovací metody

Vzhledem k tomu, že byla k vypracování práce zvolena metoda z této oblasti, bude jí věnován větší prostor než metodám generujícím povrchovou reprezentaci. Přímé zobrazovací metody pracují naprosto odlišným způsobem než metody hledající povrch. Renderují výsledný obraz bez nutnosti generování polygonální reprezentace. Místo toho jsou data promítána rovnou na průmětnu. Hlavním důvodem, proč tyto metody vznikly, bylo odstranění některých defektů vznikajících při hledání povrchu a také umožnění práce s objemovými daty i mimo vědeckou oblast, např. v počítačové zábavě.

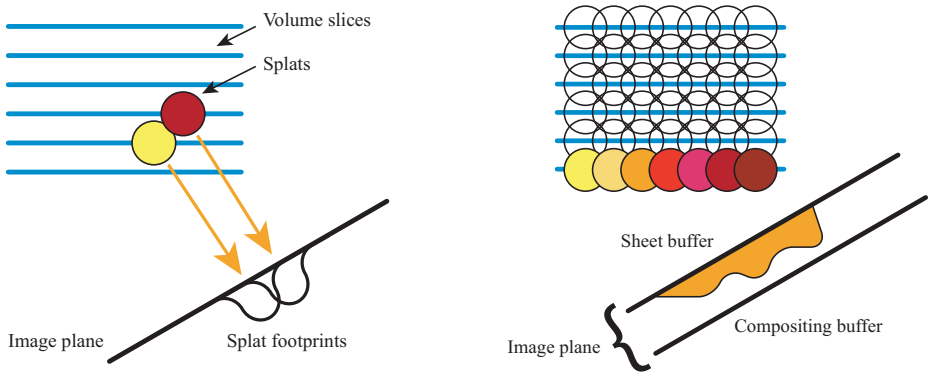
Metody přímého zobrazení objemu odstraňují použití polygonálních primitiv a snaží se objem zobrazit přímou projekcí na průmětnu. Takto lze odstranit spoustu problémů, zejména klasifikaci buněk ležících nebo neležících v isoploše a artefakty s tím spojené. Lze též zobrazovat i tzv. „amorfní“ objemy, kde není objem tělesa jasný na první pohled nebo jsou nevhodné pro vizualizaci. Jedná se převážně o přírodní fenomény jako mraky a oheň.

Bohužel se objevila spousta nových problémů, z nichž jsou některé stále velmi obtížně řešitelné. Jedním z hlavních problémů je možnost stínování zobrazeného objemu. Není totiž k dispozici povrch, který lze snadno stínovat a musí být tedy objeveny nové způsoby, jak přenést objemová data na zobrazitelné hodnoty jako je barva a průhlednost. K tomuto účelu je využíváno optických modelů, které jsou detailněji popsány v [8]. Tyto optické vlastnosti jsou dále mapovány na objemová data pomocí přenosových funkcí. To bude podrobněji popsáno v kapitole 5.

Dalším, neméně podstatným problémem je jejich výpočetní náročnost. V metodách hledajících povrch totiž stačilo pro zadanou hodnotu isoplochy projít objem pouze jednou, zaznamenat jeho povrch a poté již vizualizace probíhala bez opětovného procházení objemovými daty. Zde je však nutné při každém překreslení, hlavně při změně pozice pozorovatele, světelných podmínek nebo optických vlastností některé z hodnot objemu, aplikovat metodu na kompletní objem znovu. Je to dáno hlavně tím, že při přímém zobrazování je obraz závislý právě na pozici pozorovatele, světelných podmínkách a namapovaných optických vlastnostech. Jedním ze způsobů, jak tento problém řešit, může být použití metody faktorizace, která je popsána v části 2.4.2.

2.4.1 Metoda otisků (*Splatting*)

Metoda otisků je objektově řazeným přístupem. Hlavní myšlenkou této metody je promítnout každou buňku objemu na průmětnu a pomocí rekonstrukčního, typicky Gaussovského jádra, jinak také nazývaného otiskem buňky (*footprint*), určit její příspěvek do pixelů výsledného obrazu na průmětně. Otisk buňky je amplitudově vynásoben příslušnou hodnotou voxelu. Princip metody otisků ukazuje obrázek 2.7(a).



(a) Princip metody otisků (b) Kompozice buněk pomocí sheet bufferu

Obr. 2.7: Metoda otisků [4]

Hlavní výhodou této metody je to, že plně průhledné voxely, které se nijak nepodílí na výsledném obraze, mohou být z procesu zpracování velice rychle a snadno vypuštěny. Tímto je silně zredukováno množství dat, které je třeba zpracovat. Aby však byla zaručena správná kompozice obrazu, je třeba ho zpracovávat ve správném pořadí viditelnosti. K tomu je nejčastěji užívanou metodou tzv. *sheet buffer*, jak ilustruje obrázek 2.7(b). Více o této metodě se lze dozvědět v [4].

Tento algoritmus má však i své nevýhody. Hlavním problémem je určení tvaru otisku na průmětně. Většinou je k deformaci použita koule, která se však může promítnout jako kruh nebo elipsoid. Tvar otisku závisí na zvolené projekci a druhu mřížky, ve které jsou uložena objemová data, stejně jako na tom, jestli je použita perspektivní projekce. Dalším problémem je určení rekonstrukčního jádra, kdy složitost jádra odpovídá typicky nepřímo úměrně kvalitě zobrazení. Proto jádro často nelze integrovat analyticky, nebo to ani není z důvodu výpočetní složitosti možné. Více o těchto problémech se lze dočíst např. v [8].

2.4.2 Faktorizace (*Shear Warp Factorization*)

Od počátku volumetrického zobrazení byly snahy využít koherence buněk objemu či pixelů průmětny pro urychlení existujících metod. Problémy zde však způsobovala především perspektivní projekce, dlouhá doba nutná k předzpracování objemu pro vytvoření prostorově koherentních dat a jejich pohledová závislost.

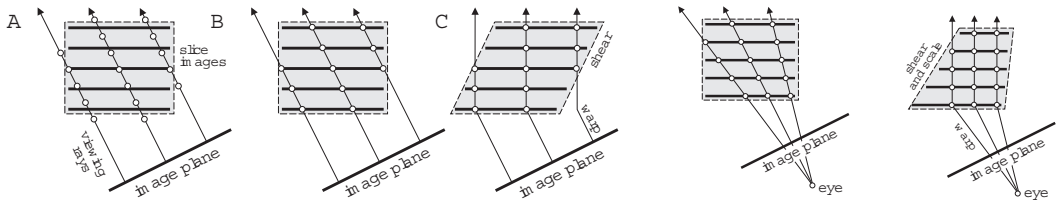
Metoda faktorizace spočívá v rozdělení (faktorizaci) pohledové matice na dvě nebo i více částí. Nejprve jsou objemová data promítnuta na dočasnou průmětnu a poté je obraz z této průmětny pomocí 2D transformací zdeformován na skutečnou průmětnu. Hlavní princip metody spočívá v tom, že dočasná průmětna je rovnoběžná s jednou z primárních os a pixely v ní jsou zarovnány s buňkami objemu, což umožňuje jeho efektivní promítnutí.

Výhodou této metody je využití prostorové koherence objemových dat při uchování krátké doby předzpracování a pohledové nezávislosti [8]. Tato metoda je stále považována za jednu z nejrychlejších čistě softwarových metod pro zobrazení objemových dat. Bohužel však výsledný obraz nedosahuje takových kvalit jako např. při použití metody vrhání paprsku, popsané v části 2.4.4.

Princip metody faktorizace

Princip této metody spočívá v transformaci objemových dat do pomocného souřadného systému, který je zvolen tak, aby se v něm objemová data snadno a rychle promítala na průmětnu. Základním předpokladem je fakt, že jsou objemová data vzorkována na rovnoběžné mřížce. Dále je zaveden pomocný „zkosený“ (*shear*) souřadný systém, tak, že jsou všechny promítací paprsky (*viewing rays*) rovnoběžné se třetí souřadnou osou.

Transformace dat do zkoseného souřadného systému je provedena následujícím způsobem. Je určen hlavní směr nejvíce rovnoběžný se směrem pohledu a uspořádány souřadnice tak, aby byl tento směr v pořadí jako třetí. Následně jsou objemová data zkosena v osách kolmých na tento směr a je provedeno promítnutí na dočasnou průmětnu. Pokud je použito perspektivního promítání, je třeba kromě zkosení objemových dat též příslušným způsobem změnit i měřítko. Tento postup je znázorněn na obrázku 2.8(a) pro rovnoběžné a na obrázku 2.8(b) pro perspektivní promítání.



(a) Rovnoběžné promítání (b) Perspektivní promítání

Obr. 2.8: Metoda faktorizace (zkosení) [1]

Podstatným krokem, kvůli kterému byl tento algoritmus navržen, je projekce na dočasnou průmětnu. Díky možnosti data libovolně vůči průmětně škálovat, zarovnat nebo natočit, je zde otevřen široký prostor pro mnoho urychlovacích technik, které by v obecném případě byly nepoužitelné.

Další podrobnosti o této metodě, včetně trasformačních matic a popisu algoritmu lze nalézt např. v [8], možné varianty urychlení jsou popsány např. v [4].

2.4.3 Metoda průřezů (*Slicing*)

Metoda průřezů vznikla hlavně jako snaha co nejlépe hardwarově urychlit zobrazování objemových dat. Jako nejvíce perspektivní se jevilo hadrwarově urychlované mapování textur, převážně 3D textur, kdy se jedná o trilineární interpolaci implementovanou přímo v hardwaru při čtení hodnot z 3D textury.

U této metody je typicky vytvořena sada polygonů, které vznikly ořezáním rovin rovnoběžných s průmětnou pomocí obálky objemu. Na tyto polygony je poté namapována textura, u níž záleží na reprezentaci dat v paměti, nejčastěji tedy 3D textura. Takto jsou namapována vlastní objemová data. Poté je na hardware samotném, aby provedl interpolaci z namapovaných hodnot v textuře.

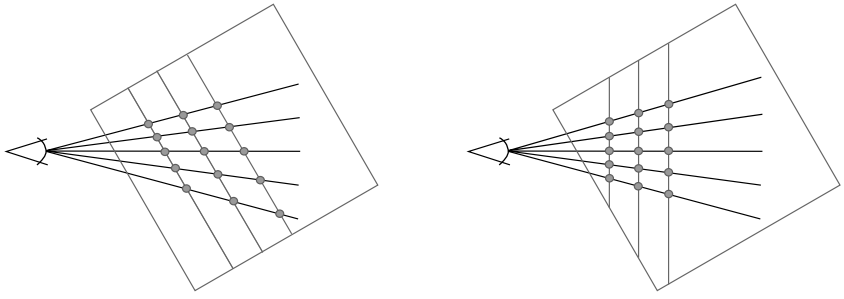
Jako výhodu této metody lze uvést fakt, že je implementačně poměrně snadná s ne příliš velkým množstvím potřebného kódu. Pro základní funkčnost většinou stačí nastavit v používané grafické knihovně příslušný zobrazovací řetězec a ořezávací roviny a hardware ve spolupráci s grafickou knihovnou provede většinu zbývajících operací. Další výhodou může být i to, že s pomocí této metody lze dosáhnout interaktivní rychlosti vizualizace. Také kvalita výsledného obrazu je poměrně dobrá. Tato metoda je též vhodná pro implementaci na GPU, takže se tím urychlí a zparalelizuje proces ořezávání a tvorby polygonů a také uvolní CPU pro další výpočty.

Nevýhodou je hlavně to, že i na většině dnešních grafických karet je omezená velikost dostupné texturovací paměti pro uložení 3D textury. Nelze tedy rychle zobrazovat velké množiny dat bez pomocných technik. Toto ale není přímo nevýhodou pouze této metody, týká se většiny zde uvedených technik.

Princip metody průřezů

Pro zobrazení objemových dat pomocí metody průřezů je třeba nalézt způsob, jak vzorkovat tato data podél pomyslných paprsků směřujících od pozorovatele skrze průmětnu do objemu. Pokud tyto vzorky leží v jedné rovině, lze tuto rovinu nahradit jediným polygonem, na kterém je provedeno vzorkování. V případě použití 3D textury tedy stačí pouze vygenerovat příslušné polygony uvnitř objemu a namapovat na ně souřadnice této textury. Hardware poté provede samotné správné navzorkování a příslušnou interpolaci hodnot.

Zde však vyvstává problém, jak tyto polygony vygenerovat. Jsou v podstatě dva používané způsoby. Buď je možné polygony orientovat v souřadném systému objemu, viz obrázek 2.9(a), nebo průmětny, jak je zobrazeno na obrázku 2.9(b).



(a) Souřadný systém objemu (b) Souřadný systém průmětny

Obr. 2.9: Používané způsoby orientace vzorkovacích polygonů [8]

Stejně jako při použití 2D textur v paměti, je nutné při práci v souřadném systému objemu nejprve určit osu nejvíce kolmou se směrem pohledu, např. analýzou pohledové a modelové matice. Polygony jsou poté generovány kolmo na tuto osu a ležící uvnitř objemu.

Při práci v souřadném systému průmětny lze polygony generovat přímo. Jen je nutné se postarat o to, aby byly převedeny do souřadného systému objemu a podle ohraničující obálky je oříznout. Nebo je lze spočítat rovnou tak, že se budou nacházet uvnitř objemu. V tom případě je zaručeno, že bude rasterizována jen potřebná část zobrazené scény a nebude tak plýtváno výkonem na rasterizaci fragmentů neobsahujících žádnou podstatnou informaci.

Posledním krokem této metody je volba způsobu, jak se budou navzorkované hodnoty na polygonech skládat. Pokud není nutností realistický výstup a fyzikální věrnost, stačí použít nějakou obecnou operaci. Jde-li ale o dostatečné zachování fyzikální věrnosti, bude nutné použít některý z osvětlovacích modelů, jak je popsáno např. v [8]. Je také možno použít některý ze sofistikovanějších operátorů nabízených jako OpenGL rozšíření.

2.4.4 Vrhání paprsku (*Raycasting*)

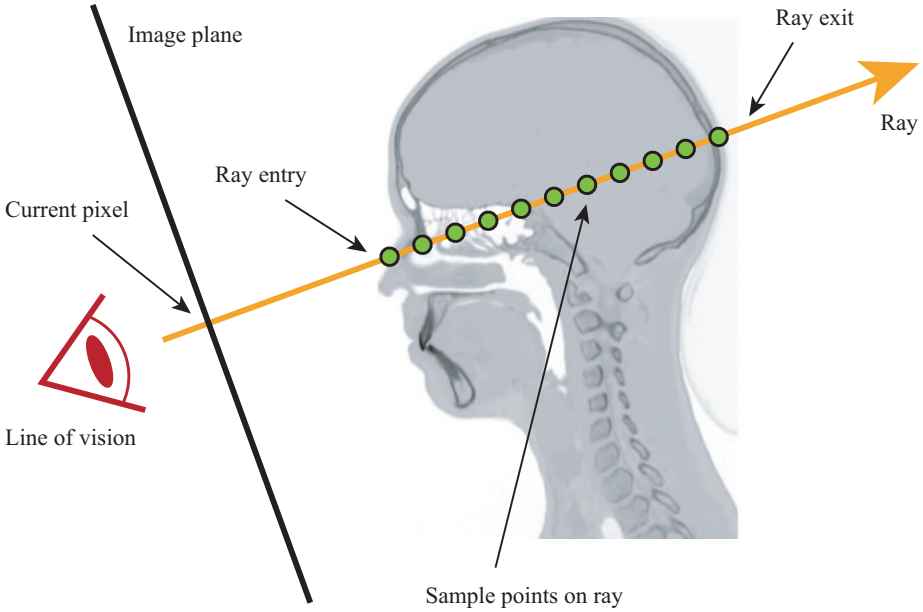
Metody založené na vrhání paprsku vychází z principu osvětlovacích modelů, proto se objevily spolu s prvními prácemi o použití teorie šíření záření v počítačové grafice. Jejich nesporno výhodou je schopnost generovat vysoce kvalitní výstup a velmi snadná implementace základních variant těchto algoritmů, ať již softwarově nebo s využitím hardwarové akcelerace. Nevýhodou však je vysoká výpočetní náročnost, která příslušné metody na dlouhou dobu činila nepoužitelnými pro interaktivní aplikace. Také při použití některé z pokročilejších variant metody vrhání paprsku je její implementace mnohem obtížnější. Na druhou stranu tato technika dokáže renderovat snad všechny typy vstupních dat a využít při tom libovolných dostupných prostředků.

Princip metody vrhání paprsku

Základem této metody a všech metod odvozených je obrazově orientovaný přístup, kdy je z pozice pozorovatele (*Line of vision*) vržen paprsek (*Ray*) přes každý bod průmětny (*Current pixel*) do objemu, tak jak je znázorněno na obrázku 2.10. Objem je rovnoměrně navzorkován podél tohoto paprsku a příslušný pixel je integrálně spočítán pomocí rovnice příslušného osvětlovacího modelu v pořadí od předního vzorku k zadnímu. To znamená, že je v místě každého vzorku hodnota barvy a průhlednosti spočítána pomocí příslušného osvětlovacího modelu.

Tento algoritmus vychází z faktu, že je k určení barvy pixelu na průmětně potřeba zjistit množství záření, které na tuto plochu dopadá. Toto záření je tedy sledováno podél parsku procházejícího od pozorovatele skrze objem tímto pixelem průmětny. Je určen průsečík tohoto paprsku s obálkou objemu v místě vstupu paprsku do objemu (*Ray entry*) a v místě jeho výstupu (*Ray exit*) a tato oblast paprsku ležící uvnitř objemu je poté navzorkována (*Sample points on ray*). Příspěvky z jednotlivých vzorků jsou zintegrovány do výsledné hodnoty, kterou paprsek přispěje do tohoto pixelu.

Způsobů, jak jednotlivé vzorky na paprsku skládat, je nepřeberné množství, jedná se totiž o vlastní teorii osvětlovacích modelů. Některé z nich jsou popsány v [8]. Také nemusí existovat přímá korespondence 1 : 1 mezi pixely průmětny a paprsky, ale lze vrhat více paprsků jedním pixelem stejně jako některé pixely vynechávat a hodnoty v nich dopočítávat interpolací z okolních pixelů. Takto lze dosáhnout kompromisu mezi rychlostí algoritmu a kvalitou zobrazení, stejně jako silně upřednostnit jednu vlastnost před druhou. Samozřejmě,



Obr. 2.10: Metoda vrhání paprsku [4]

že se s využitím jednoduchého osvětlovacího modelu nelze přiblížit realitě, ale je možné metodu postupně rozšiřovat tak, aby se fyzikální realitě dopadajícího zařízení maximálně podobala. Většinou ovšem za cenu rapidního zvýšení výpočetní náročnosti.

Vzhledem ke skutečnosti, že jsou objemová data ve skutečnosti skalární hodnoty uložené v mřížce, je potřeba z těchto skalárních hodnot nějakým způsobem určit jejich optické vlastnosti. K tomu se nejčastěji používá přenosová funkce (*transfer function*). Také u nich není dostupná normála povrchu v daném bodě objemu, takže by nebylo možné spočítat stínování. To lze vyřešit tak, že je jako normála povrchu použit gradient povrchu. K výpočtu gradientu lze použít více metod a detailněji se mu věnuje část 4.5.2.

2.5 Zhodnocení

Jak již bylo zmíněno v předešlých částech, metody hledající povrch jsou pro vypracování této práce silně nepoužitelné. Proto bylo voleno pouze mezi metodami přímého zobrazení. Z nich byla vypuštěna metoda faktorizace kvůli její nižší výsledné kvalitě výstupního obrazu, stejně jako metoda otisků kvůli závislosti na vhodně zvoleném rekonstrukčním jádru otisku. Tyto metody by navíc byly obtížněji implementovatelné, zejména pokud je jedním z cílů využití hardwarové akcelerace.

Ze zbývajících přístupů byla zvolena metoda průřezů, zejména protože je snadněji implementovatelná a vhodná pro data uložená ve 3D textuře. Také lze akcelerovat pomocí implementace na GPU, popř. snadno modifikovat podle nových požadavků. Tato metoda bude dále postupně rozšiřována o další přístupy, jako třeba fragment shadery pracující na obdobném obrazovém principu jako metoda vrhání paprsku a o přenosovou funkci určující optické vlastnosti dat. O návrhu a implementaci celkového renderovacího procesu jsou kapitolay 4 a 5.

Kapitola 3

Fluorescenční mikroskopie a modelový organismus

Tato kapitola má za úkol popsat princip fluorescenční mikroskopie, metodu použitou pro přípravu vstupních dat a dále vlastní modelový organismus. Jeho snímáním budou získávána vstupní data pro tuto práci po jejím dokončení. Přestože se bude jednat spíše o teoretický popis biologických a fyzikálních jevů, je tato kapitola nezbytná k porozumění procesu získávání vstupních dat a rozlišení buněk. Tomuto procesu se totiž musí výsledná metoda vizualizace přizpůsobit, jak bude později uvedeno v části 4.1.

Sekce 3.1 pojednává o principu fluorescenční mikroskopie, na čem je založena a k čemu se v současnosti používá. V části 3.2 je popsána metoda GFP (*Green Fluorescent Protein*), pomocí níž budou značeny buňky v používaných vstupních datech a jejichž vizuální odlišení bude úkolem této práce. Dále se v sekci 3.3 nachází popis modelového organismu *Caenorhabditis elegans* sloužícího k nasnímání vstupních dat.

V současné fázi jsou dostupná pouze data obsahující buňky rostlinného původu, nejčastěji listu, značené reakcí chlorofylu, ale po implementaci všech požadavků a zabudování do aplikace FluorCam budou používána data modelového organismu značeného právě metodou GFP. Momentálně totiž žádná data tohoto typu stále nejsou dostupná.

3.1 Fluorescenční mikroskopie

3.1

V této části je podrobněji popsán princip takzvané fluorescenční mikroskopie. Též zde je uvedeno, jak vzniká, na jakých fyzikálních a chemických procesech je založena a k čemu je v současné době využitelná.

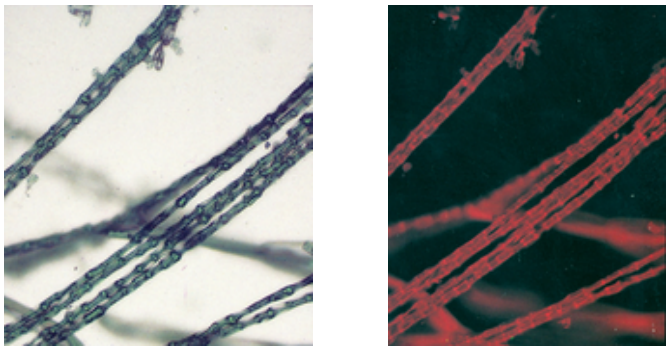
3.1.1 Luminiscence, fosforescence, fluorescence a fluorochrom

3.1.1

Zelené řasy se jeví v denním světle jako zelené, když jsou však v temnici osvíceny ultrafialovým světlem, zbarví se do červena. Je to proto, že chlorofyl v jejich buňkách pohlcuje ultrafialové světlo a vyzařuje světlo červené, které má delší vlnovou délku. Jevu, kdy látka vysílá do prostoru světlo, se říká luminiscence. Může k ní docházet při chemické reakci, např. pokud enzym luciferáza oxyduje luciferin u světlušky, potom se dá hovořit o chemiluminiscenci. Pokud k luminiscenci dochází po osvětlení zářením, jedná se o fluorescenci nebo fosforescenci. Jestliže světlo vystupuje z organismu důsledkem chemické reakce, celkově

se tomu říká bioluminiscence. Záření vyvolávající luminiscenci se nazývá excitační, záření vysílané látkou se nazývá emisní. Zatímco u fluorescence trvá vyzařování emisního světla krátkou dobu a po zhasnutí excitačního záření téměř okamžitě emise zaniká (asi za 10^{-8} sekundy), u fosforescence může k emisi docházet i dlouhou dobu po zhasnutí excitačního záření. Látka schopná fluorescence se nazývá fluorochrom.

Fyzikální podstata fluorescence a fosforescence spočívá ve vlastnostech elektronového obalu atomů v molekulách fluorochromu. Elektrony těchto látek jsou schopny absorbovat foton excitačního světla, čímž se zvýší jejich energie. Část této nově nabyté energie však elektron po chvíli vyzáří jako foton s nižší energií a tedy delší vlnovou délkou. Protože došlo ke ztrátě energie, je vlnová délka emisního světla vždy delší než vlnová délka světla excitačního, podle tzv. Stokeova pravidla. Fluorescence tedy zjednodušeně transformuje světlo z jedné vlnové délky na druhou. Jelikož vlnová délka určuje barvu světla, lze pozorovat u emitovaného světla posun k červené části spektra, jak ukazuje obrázek 3.1.



(a) Denní světlo (b) Ultrafialové světlo

Obr. 3.1: Zelené řasy osvícené různým světlem

3.1.2 Excitační a bariérový filtr

3.1.2

Aby se dalo dobře pozorovat emisní záření, jehož intenzita je vždy mnohem nižší než intenzita excitačního záření, používá se dvojice filtrů, jejichž princip je graficky znázorněn na obrázku 3.2.

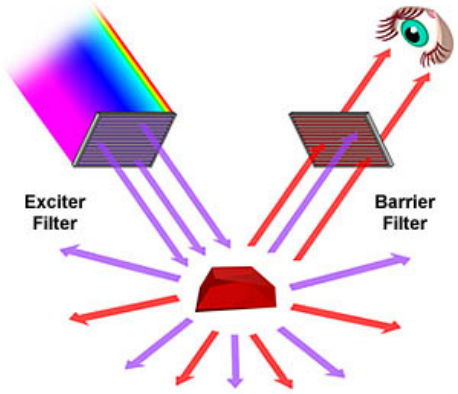
Excitační filtr propouští z barevného spektra pouze část potřebnou pro excitaci fluorescence a zabraňuje průchodu světla o stejné či podobné vlnové délce jako světlo emisní, které by vytvářelo nechtěné pozadí.

Bariérový filtr propouští pouze emisní část spektra a zabraňuje tak průchodu excitačnímu světlu. Excitační světlo se od emisního sice liší barvou, je však mnohem intenzivnější, takže v něm emisní světlo není lidským okem dobře rozpoznatelné.

3.1.3 Princip a využití fluorescence

3.1.3

Fluorescenční metody se používají především když je potřeba zviditelnit určité části a struktury v buňce. Některé fluorochromy (DAPI, ethidium bromid, Hoechst, propidium jodid) se samy o sobě váží na určité molekuly (např. na DNA) a lze je tedy použít na zviditelnění těchto molekul. Tento postup se nazývá *přímá fluorescence*.



Obr. 3.2: Excitační a bariérový filtr

Pro většinu struktur v buňce však nelze najít fluorochrom, který by se na ně specificky vázal. V takovém případě je použita metoda *imunofluorescence*. Jsou vypracovány postupy, s jejichž pomocí je možné vyrobit protilátku, která se specificky váže na téměř jakýkoli druh molekuly. Poté na ni lze kovalentně navázat fluorochrom a vyrobit tak fluoreskující molekulu specificky rozeznávající to, co je předmětem zájmu. V tomto případě se jedná o *přímou imunofluorescenci*.

Příprava tohoto kombinované konjugátu (molekuly) není úplně jednoduchá, a proto se často používá *nepřímá imunofluorescence*. U této metody je nejdříve připravena v určitém zvířecím druhu (např. v králíkovi) specifická protilátka proti použité molekule (primární protilátka) a nechá se navázat na molekulu nebo strukturu, kterou je cílem lokalizovat. Po odmytí přebytečné primární protilátky se na preparát přidá komerčně dodávaná protilátka konjugovaná s fluoresceinem specificky rozeznávající všechny protilátky daného zvířecího druhu. Ta se naváže na primární protilátku a zviditelní hledanou strukturu. Nevýhodou nepřímé fluorescence oproti přímé fluorescenci je nižší specifita.

Další metoda imunofluorescence využívá silné vazby biotinu a bazického glykoproteinu avidinu. Biotin lze snadno navázat na molekuly primárních protilátek, na které se pak místo sekundárních protilátek s fluorochromem váže fluorochromem značený avidin. Výsledkem je jasná a ostrá fluorescence.

Protože je k dispozici celé barevné spektrum fluorochromů, lze zviditelnit více různých struktur v téže buňce a sledovat tak jejich vzájemnou lokalizaci. Kromě uvedených postupů existují pochopitelně další metody, modifikace či kombinace těchto metod, jejich výčet je však již nad rámec tohoto textu. Více se lze o fluorescenci dozvědět např. v [5]. V další části budou popsány jednotlivé typy mikroskopů používané k pozorování fluorescence.

3.1.4 Fluorescenční mikroskop

3.1.4

Zdroj světla u fluorescenčního mikroskopu tvoří rtuťová výbojka. Podle konstrukce se fluorescenční mikroskopy dělí na dvě skupiny.

Transmisní fluorescenční mikroskop (*Transmition light fluorescence microscope*)

U tohoto typu mikroskopu prochází světlo excitačním filtrem a na preparát přichází zespodu jako u klasického světelného mikroskopu. Pro osvětlení preparátu se však nepoužívá

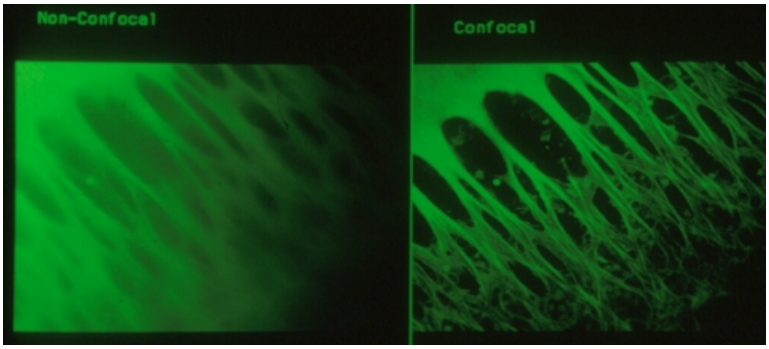
klasický kondenzor, ale kondenzor zástinový, který odráží světlo tak, že dopadá na preparát z boku. Procházející excitační světlo tak letí mimo objektiv a do objektivu se dostane pouze emitovaná fluorescence.

Epifluorescenční mikroskop (*Reflected light fluorescence microscope*)

U tohoto typu mikroskopu prochází excitační světlo objektivem, dopadá na preparát shora a emisní světlo se vrací zpět do objektivu. U tohoto mikroskopu je potřeba použít zvláštní typ zrcadla, které odráží excitační světlo do objektivu a propouští emisní světlo do okuláru. Zpočátku se používalo polopropustné zrcadlo, které polovinu světla propouští a polovinu odráží, bez ohledu na to, o jaké světlo jde. Pochopitelně, že u tohoto zrcadla docházelo k velkým ztrátám světla, a proto se začalo používat tzv. dichroické zrcadlo. To propouští a odráží světlo podle toho, jakou má vlnovou délku. Používá se tedy vždy takový typ zrcadla, který maximum excitačního světla odráží a maximum emisního světa propouští. Epifluorescenční typ mikroskopu je v současnosti více oblíbený než transmisní.

3.1.5 Konfokální mikroskop (*Confocal microscope*)

Konfokální mikroskopy jsou dalším typem mikroskopů používaným v rámci fluorescenční mikroskopie. Jejich výhoda oproti klasickým fluorescenčním mikroskopům spočívá především v kvalitnějším výsledném obrazu. Konfokální mikroskop totiž umožňuje odstranit z obrazu objektu šum, který vytváří světlo nebo fluorescence emitovaná z těch rovin vzorku, na které není zaostřena optika. Srovnání obrazu obou typů mikroskopů je ukázáno na obrázku 3.3.

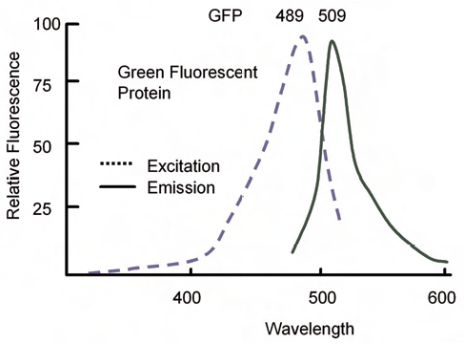


Obr. 3.3: Srovnání obrazu z fluorescenčního (vlevo) a konfokálního (vpravo) mikroskopu

Zdrojem světla je zpravidla laser, světlo prochází úzkou štěrbinou a je zaostřeno do jednoho bodu vzorku. Světlo emitované z tohoto bodu je pak snímáno detektorem. Aby dopadlo na detektor, musí opět projít úzkou šterbinou ležící v místě, kam objektiv zaměřuje světlo ze zaostřeného bodu objektu. Světlo emitované z osvětlených, ale nezaostřených bodů je fokusováno mimo stěrbinu a do detektoru nedopadá. Signál z detektoru je odeslán do počítače. Ten zároveň dostává informaci o souřadnicích snímaného bodu. Tímto způsobem je bod po bodu proskenován celý objekt v různých optických rovinách. Toto skenování je automatizováno a ovládáno řídícím počítačem. Z nashromážděných informací počítač sestaví celkový obraz. Nevýhodou konfokálního mikroskopu je především jeho cena, která se pohybuje v řádech milionů korun. Přesto se stává v poslední době stále běžnější součástí vybavení bohatších laboratoří a výzkumných ústavů.

3.2 Metoda GFP

GFP (*Green fluorescent protein – Zeleně fluoreskující protein*) je relativně novou biologickou detekční látkou, která má svůj základ v živé přírodě. Tento protein byl nalezen, a jeho genová sekvence získána, ze světelkující medůzy pohárovky (*Aequorea Victoria*). Ta obsahuje dvoufázový systém generující světlo vyžívající bioluminscenční protein *Aequorin*. Díky přítomnosti vápníku tento protein bioluminiscencí vyzařuje modré světlo, toto modré světlo proteinu je následně pohlceno GFP, a protože je tento protein fluoreskující, způsobí, že medúza zasvítí zeleně. GFP protein totiž obsahuje chromofor absorbující modré světlo a vyzařující zelené světlo. Excitační a emisní maxima proteinu GFP jsou vidět na obrázku 3.4.



Obr. 3.4: Excitační a emisní maxima pro GFP [3]

Roku 1994 byl úspěšně sekvencován a od té doby se používá v laboratořích po celém světě ke značení sledovaných buněk nebo celých částí zkoumaného organismu, jak *in vitro* (v umělých podmínkách) tak i *in vivo* (v živém organismu). V pozdější době byly též provedeny umělé modifikace tohoto proteinu a podařilo se tak získat celou řadu proteinů nových, vyznačujících se jinými absorbčními a emisními spektry, svítících tedy různými barvami a reagujících na jiné vlnové délky. Díky odlišnosti barevných spekter lze použít více těchto proteinů v rámci jednoho měření. Vedle zeleného fluorescenčního proteinu jsou tak dnes k dispozici také jeho žlutá a modrozelená varianta a nepříbuzný červený fluorescenční protein.

Jestliže je záměrem zjistit osud určitého genového produktu, v jaké tkáni, buněčném kompartmentu, části životního cyklu nebo fázi ontogenetického vývoje organismu je tento protein aktivní, lze použít právě GFP. Nejprve se připraví tzv. fúzní gen. Metodami genového inženýrství je spojen tento gen pro studovaný protein a gen GFP. Pokud se podaří nahradit původní gen organismu tímto fúzním genem, bude se při aktivaci překládat a vznikne funkční protein, který bude zároveň zeleně fluoreskovat. Takto lze kdykoli během života organismu sledovat, zda a kde je studovaný protein překládán. V buněčné biologii se používá pro studium určitého genového produktu, ke kterému se naváže a později se snadno lokalizuje pomocí právě pomocí fluorescence. Více o tomto proteinu, včetně detailnějšího popisu principu, je popsáno např. v [16, 20, 17] nebo v [3, 11]. Zde jsou popsány i jednotlivé modifikace tohoto proteinu, podrobnější spektrální grafy i sekvence aminokyselin.

Obr. 3.5: Dospělý jedinec *Caenorhabditis elegans* [19]

3.3 Modelový organismus Caenorhabditis elegans

Caenorhabditis elegans je v přírodě volně žijící, zhruba 1mm dlouhá škrkavka, obývající mírně slaná prostředí. Výzkum na molekulární a biologické bázi započal roku 1974 Sydney Brenner a od té doby je tento červ hojně vyžíván jako modelový organismus. Ve své době se jednalo o první vícebuněčný organismus, jehož genom byl kompletně sekvencován. Tento genom byl publikován roku 1998, přestože ještě obsahoval některá neprozkoumaná místa, převážně kvůli nevhodně pozicované DNA sekvenci. Poslední z těchto míst byla prozkoumána v říjnu 2002. Kompletní genomová sekvence je dlouhá zhruba 100 milionů bázových párů a obsahuje přibližně 20 tisíc genů.

Jako modelový organismus je používán z mnoha důvodů, včetně ekonomických a též z důvodu snadného udržení rozsáhlé populace v laboratorních podmínkách. Jedná se totiž o skutečný organismus obsahující fyziologické systémy, např. zažívací, nervový, svalový a reprodukční, které lze většinou nalézt pouze ve „vyšších“ organismech, jako jsou myši nebo lidé. Ovšem díky tomu, že je tento organismus velice malý, může snadno vyrůst populace o velkém množství jedinců i v petriho misce. Reprodukce také probíhá velmi rychle, jedná se o hermafroditní organismus, produkující spermie i vajíčka. Jedná se o adaptivní organismus, který může být relativně snadno uchován v chladu a poté opět zahřán na běžnou teplotu. Dokonce překvapil vědce tím, že přežil havárii raketoplánu Columbia v únoru 2003. Zjistilo se také, že tento organismus je dobře použitelný jako model pro závislost na nikotinu a že trpí stejnými symptomy jako lidé končící s kouřením.

Stejně jako ostatní živočichové, i tento organismus svůj život začíná jako vajíčko (zygota), které dále postupuje mytotickým dělením, čímž postupně vzniká dospělý jedinec. Díky tomu, že je tento červ celkově průhledný, je známá plná funkčnost jednotlivých buněk stejně jako časová linie jejich vzniku. Ukázalo se, že *C. elegans* je tak velmi vhodný pro studii buněčného rozlišování. Více o tomto organismu je dále popsáno např. v [18, 19].

Kapitola 4

Zobrazení vstupních dat

Pomocí fluorescenční mikroskopie byla tedy získána volumetrická sada dat a tu je nyní potřeba zobrazit. K tomu je třeba nalézt správný způsob. Protože tento zvolený způsob bude silně ovlivňovat jak rychlost zobrazení, tak samotnou výslednou kvalitu vizualizace značených buněk, jedná se o hlavní a nejrozsáhlejší částí této práce. Samotná vizualizace totiž spočívá pouze v nalezení správného mapování optických vlastnostní na vstupní data takovým způsobem, aby byly značené buňky co nejlépe odlišeny od svého okolí.

V této kapitole jsou shrnuty všechny metody a principy použité v současném stavu implementace ke zobrazení objemu. Většina z těchto metod je naimplementována plně, ale některé budou ještě pravděpodobně v budoucnu potřebovat určitá vylepšení. Zobrazení objemových dat již však nečiní žádné problémy žádnému podporovanému typu dat.

V následujících sekcích bude popsán kompletní proces zobrazení vstupních dat, od jejich načtení až po výsledné zobrazení. Výsledkem tohoto procesu jsou ale pouze zobrazená intenzitní data, z nichž uživatel pravděpodobně potřebné důležité informace žádným snadným způsobem nezjistí. Proto tedy bude potřebovat mapování optických vlastností na jednotlivé hodnoty, tedy samotný proces vizualizace značených částí. Tomu se dále věnuje kapitola 5.

Jako první je v sekci 4.1 popsán formát vstupních dat a jejich vlastnosti. Dále je v části 4.2 popsán proces načítání dat určených ke zobrazení, stejně jako problémy s tím spojené. Sekce 4.3 poté pojednává o hlavní použité metodě zobrazení dat, metodě řezů, a o detailech její implementace. Následující část 4.4 popisuje možnosti vypuštění nepotřebných částí objemu z vykreslovacího řetězce, pokud by implementace tuto schopnost vyžadovala, v části 4.5 se lze dočíst o principu použitého osvětlovacího modelu a metodách stínování a v sekci 4.6 je popsána výsledná fáze renderingu za použití fragment shaderu. Nakonec jsou v části 4.7 zmíněna vylepšení, která by v budoucnu mohla mít pro tuto práci patřičný přínos a mohla by se v ní tedy časem objevit. Poslední část 4.8 shrnuje dosavadní průběh návrhu a implementace tohoto komplexního zobrazovacího procesu.

4.1 Vstupní data

4.1

Před samotným zobrazením je třeba zpracovat příslušná vstupní data, aby bylo co zobrazovat. Ta jsou uložena v určitém formátu. Tento formát musí být rendererem podporován, aby mohl být načten. Použitý formát vstupních dat má své výhody i nevýhody a občas se mohou objevit probémy při jeho zpracování, které musí být vyřešeny, aby mohla být data správně načtena.

Vstupní data jsou momentálně dodávána v multiobrázkových komprimovaných souborech formátu `tiff`. Jedná se o jednotlivé řezy zkoumaného modelového organismu sejmuté konfokálním mikroskopem, který ovšem žádným způsobem neuchovává informace o detekovaných vlnových délkách. Data jsou tedy čistě intenzitní, s veškerou informací uloženou v jedné, nejčastěji 16 bitové intenzitní složce. Rozsah 16 bitů je dán tím, že mikroskop snímá ve 12 bitové hloubce, nelze tedy získaná data uložit beze ztráty informace do 8 bitových hodnot.

Již po počátečním nahlédnutí do některých vzorků těchto dat je vidět, že jsou v určitých místech, převážně na okrajových řezech, zašuměná. Intenzita a množství šumu ovšem závisí na typu modelového organismu stejně jako na použité metodě značení. Tento šum však může způsobovat problémy některým způsobům vizualizace, proto na něj bude potřeba dbát ohled při návrhu samotné metody.

Také bude nutné přizpůsobit použité algoritmy a metody velikosti těchto dat, protože se jedná o řádově desítky megabajtů. Pokud je v takových datech potřeba uložit i gradient pro výpočet stínování, je nutné počítat se čtyřnásobkem jejich velikosti, kvůli převodu z textury obsahující pouze intenzitní složku do RGBI textury, kde se do RGB složek uloží gradient do I složky vlastní hodnota vzorku. Data jsou též často o rozměrech přesahujících maximální OpenGL podporovaný rozměr 3D textur, bude tedy nutné se vypořádat i s tímto problémem.

Se všemi těmito skutečnostmi muselo být při návrhu metody počítáno a podle nich samotný návrh přizpůsobit. Dále bylo třeba naimplementovat načítání dat z multitiff souborů. Toho je dosaženo s využitím volně šiřitelné knihovny `libtiff` pro práci s těmito typy souborů. Tato knihovna je šířena pod licencí „as-is“, lze ji tedy bez problémů využít i v komerčních aplikacích. Vzhledem k její ne příliš rozsáhlé dokumentaci se vyskytlo v implementaci několik drobných chyb, které se ovšem ve výsledku jevily jako velice závažné a jejich hledání a odstraňování se protáhlo na celou dobu implementace.

4.2 Načtení a zpracování dat

Prvním úkolem, který je potřeba provést, je načtení dat a jejich zpracování. Aplikace podporuje načítání dat ze souborů typu `tiff`, protože jsou v nich momentálně dodávána všechna data, dále data mohou být uložena také v „surové“ formě v `raw` souboru s jejich popisem umístěným v `dat` souboru. Později bude pravděpodobně potřeba doimplementovat práci s paměťovými strukturami, pomocí nichž bude možno komunikovat s aplikací FluorCam.

Načtená data jsou dále zpracována. Je zjištěn jejich histogram, údaje o rozměrech a rozsah jejich hodnot. Nakonec jsou data uložena do 3D textury, která je odeslána do texturovací paměti. V tomto místě se ovšem může vyskytnou již zmíněný problém, kdy rozměr dat přesahuje maximální podporovaný rozměr 3D textury. Na referenčním grafickém adaptéru NVIDIA GeForce 6600 Go je to 512 hodnot v každém směru. Tento problém lze řešit více způsoby, např.:

- Zmenšením dat na potřebný rozměr externí aplikací před načtením.
- Zmenšením dat na potřebný rozměr přímo v aplikaci po načtení.
- Rozdělením načtených dat do více 3D textur o menší velikosti.

Prvním způsobem je problém vyřešen dokonale, bohužel se u něj objevují problémy nové. Jako nejmarkantnější by mohlo být, že vyžaduje aktivitu uživatele navíc, což pro některé méně znalé a zkušené uživatele může být problematické nebo obtěžující. Dalším problémem

je nalezení aplikace, která tuto změnu umožní. Pravděpodobně totiž neexistuje příliš mnoho aplikací schopných práce s multitiff soubory. Zde by mohla být zmíněna např. aplikace ImageJ, která tento formát plně podporuje. Výhodou tohoto způsobu může být alespoň fakt, že data potřebují méně paměti a práce s nimi by tedy měla být rychlejší.

Druhý způsob spočívá v načtení celého objemu a jeho zmenšení přímo v rámci procesu načítání. To může být uskutečněno více způsoby, z nichž se jako nejvhodnější jeví tyto:

- **Oříznutí dat na požadovanou velikost.** Ze vstupních dat bude jednoduše oříznuta oblast přesahující maximální povolený rozměr textury. Je též snadné naimplementovat modifikaci, kdy si uživatel může zvolit offset, od kterého bude oříznutí provedeno, aby tak mohl zahrnout ty podstatnější části dat.
- **Zmenšení rozměrů pomocí interpolace nebo podvzorkování hodnot.** Objem je zmenšen na požadovanou velikost pomocí interpolace, tj. každý voxel nového objemu je interpolován z voxelů objemu původního, nejčastěji lineárně nebo bilineárně. Další možností je místo interpolace použít prosté podvzorkování, kdy jsou vybírány jen vzorky odpovídající nové vzorkovací frekvenci. Tu lze získat pomocí vztahu

$$f_n(d) = N(d)/T_{max} \tag{4.1}$$

kde $f_n(d)$ odpovídá nové vzorkovací frekvenci ve směru d, $N(d)$ rozměru vstupních dat ve směru d a T_{max} maximálnímu podporovanému rozměru 3D textury, zjištěného pomocí dotazu OpenGL. Tento způsob je sice nejsnáze implementovatelný, ale v datech se mohou objevit nepříjemné artefakty. Protože ale data nejsou mnohonásobkem maximální povolené velikosti, pravděpodobně se tyto artefakty neobjeví, vzorkování totiž bude probíhat s krokem ležícím někde mezi 1 a 2 vzorky.

Nakonec lze uvolnit původní objemová data z paměti, protože už nad nimi nebude potřeba provádět žádné operace.

Poslední způsob spočívá v tom, že jsou do paměti načtena celá objemová data a rozdělena do více 3D textur, podle jejich rozměru v jednotlivých osách. Není tedy nutné provádět operaci zmenšení, ale na druhou stranu je třeba renderovat větší množství textur, což ve výsledku algoritmus znatelně zpomalí. Dalším problémem zde je, že pokud objemové textury obsahují jen určitou část vstupních dat, po jejím vyrenderování je mezi dvěma sousedními bloky vidět znatelná hranice. To je dáno použitou interpolací dat v textuře. Toto lze vyřešit tak, že textura obsahuje o jednu řadu vertexů okolo hranice více. Zde se však objevuje problém s rozměry 3D textur v OpenGL, kdy při některých rozměrech OpenGL knihovna ohlásí kritickou chybu a aplikace je ukončena. Musely by se tedy ošetřit všechny tyto možné varianty, navíc nelze s jistotou říct, jak by se tento způsob choval na jiných grafických adaptérech.

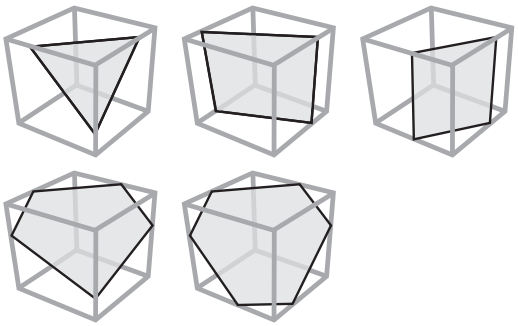
Jistě existují i jiné způsoby řešení, ale tyto uvedené jsou pravděpodobně nejvhodnější možnosti, jak se s daným problémem vypořádat. Jako nejlepší pro tuto práci se jeví naimplementování podvzorkování a interpolace. Uživatel si potom bude moci zvolit, který způsob má být po načtení dat, jejichž rozměry přesahují ty maximálně povolené, aplikován.

4.3 Metoda řezů

Nyní již jsou data načtena, prvotně zpracována a uložena ve 3D textuře. Další fází je tedy jejich zobrazení. K tomu byla zvolena metoda řezů, jejíž základní princip je popsaný v části

2.4.3. Tato metoda byla dále rozšířena o implementaci na GPU pomocí vertex shaderů, čímž bylo dosaženo mírného zvýšení výkonu, ale hlavně nezávislosti na CPU. Také se tím částečně vyrovnala zátěž mezi vertexovými a fragmentovými jednotkami. Toto řešení pomocí vertex shaderů je založené na práci [14]. Ve zbývající části této sekce bude popsán pricip, jak tato metoda pracuje, stejně jako detaily implementace na vertexové jednotce GPU.

4.3.1 Výpočet průsečíků roviny řezu a obalového tělesa



Obr. 4.1: Varianty průniku krychle s rovinou [14]

Výsledkem průniku roviny řezu a objemové obálky tělesa je vždy polygon se třemi až šesti vrcholy, jak je ukázáno na obrázku 4.1. Princip naimplementované metody je založen na výpočtu těchto průnikových polygonů čistě uvnitř vertex procesoru. Protože však aktuální verze vertex shaderů nemůže měnit počet vertexů, je třeba pro každý generovaný polygon odeslat plný počet 6 vrcholů a stejný počet též vystoupí z vertex programu. To je provedeno pomocí předpřipraveného display listu. Pokud výsledný polygon obsahuje méně než 6 vertexů, vertex program některé z nich zduplikuje tím, že mezi nimi vytvoří hranu o nulové délce.

Pokud je rovina řezu zadaná v Hessově normální formě,

$$\langle \overrightarrow{n}_P \circ \overrightarrow{x} \rangle = d \tag{4.2}$$

kde $\overrightarrow{n}_P$ je normálový vektor roviny a d typicky vzdálenost roviny od počátku, je výpočet průniku hrany obálky objemu a roviny snadný. Pro roviny rovnoběžné s průmětnou lze totiž nahradit normálový vektor $\overrightarrow{n}_P$ vektorem pohledovým. Hranu mezi sousedními vrcholy obálky V_i a V_j lze popsat jako

$$\begin{aligned} E_{i\to j} : X(\lambda) &= V_i + \lambda(V_j - V_i) \\ &= V_i + \lambda \overrightarrow{e}_{i\to j} \text{ , kde } \lambda \in [0,1] \end{aligned} \tag{4.3}$$

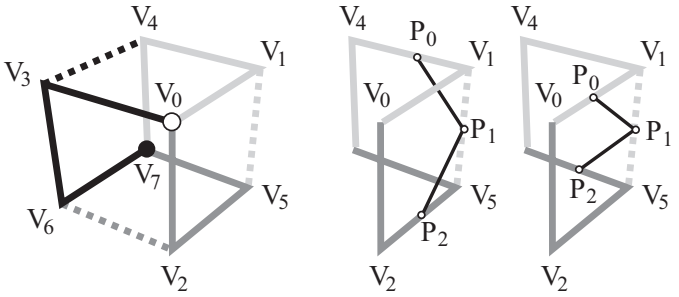
Vektor $\overrightarrow{e}_{i\to j}$ je bezrozměrný. Průsečík roviny a úsečky $E_{i\to j}$ lze pak spočítat pomocí vztahu

$$\lambda = \frac{d - \langle \overrightarrow{n}_P \circ V_i \rangle}{\langle \overrightarrow{n}_P \circ \overrightarrow{e}_{i\to j} \rangle}. \tag{4.4}$$

Jmenovatel v rovnici 4.4 je roven nule pouze v případě hrany rovnoběžné s rovinou řezu. V tomto případě je možno průsečík s klidem ignorovat. Za platný průsečík lze označit pouze ten, kdy λ leží v rozsahu $[0, 1]$. Jinak rovina přímku danou vektorem hrany sice protíná, ale mimo hranu tělesa. Tento případ je tedy nepodstatný.

Nyní již je možné spočítat průsečíky řezu s objemem tak, aby to bylo implementovatelné na GPU, ale je třeba se ještě vypořádat s hlavním problémem, který se během výpočtu ve vertexové jednotce objeví. Tím je nutnost správně seřadit nalezené průsečíky tak, aby tvořily validní polygon. Jinak by rasterizace ve většině případů nemusela dopadnout podle očekávání. K vyřešení tohoto problému je potřeba pochopit, jak algoritmus řezů funguje. Všem vrcholům obalového tělesa je tedy přiřazen identifikační index a toto značení bude použito dále v tomto popisu.

Existuje takový vrchol V_0, který je nejblíže ke kameře. Dále lze předpokládat, že v protilehlém rohu na opačném konci tělesové úhlopříčky, tedy nejdále od kamery, leží vrchol označený jako V_7, jak ukazuje obrázek 4.2. Jestliže vrchol V_0 je přední a V_7 zadní, existují mezi nimi tři vzájemně nezávislé cesty, na obrázku označené souvislou čarou různými odstíny šedé. *Nezávislé* je zde použito ve významu, že kromě počátečního a koncového nesdílí žádný jiný vrchol. Každá cesta je tedy složena z dané posloupnosti tří vrcholů označených jako $\{E_1, E_2, E_3\}$, tedy např. pro černě označenou cestu $E_1 = E_{0\to3}$, $E_2 = E_{3\to6}$ a $E_3 = E_{6\to7}$. Pro libovolný přední vrchol lze určit tyto tři cesty za předpokladu, že vektory mapované na hrany E_1, E_2 a E_3 tvoří pravotočivý systém.



Obr. 4.2: Sekvence očíslování vrcholů obalového tělesa objemu [14]

Nyní jsou definovány rovnice pro výpočet průsečíku, označené vrcholy krychle a zbývá už jen samotný postup vygenerování polygonu řezu. Lze si představit, že je rovina rovnoběžná s průmětnou posouvána ve směru od předního vrcholu k zadnímu. Vrchol, se kterým tato rovina bude mít kontakt jako první, je V_0. Až do této doby nemohl být zjištěn žádný platný průsečík. Jako poslední bude rovina interagovat s vrcholem V_7. Také od této doby nebude existovat žádný platný průsečík. Zde je důležité poznamenat, že jakákoliv rovina protínající objem, má právě jeden platný průsečík s každou ze tří hlavních cest. To znamená, že se dají snadno zjistit tři ze šesti možných průsečíků P_i ověřením průniku se sekvencí hran cest, a to.

P_0 = průnik s $E_{0\to1}$ nebo $E_{1\to4}$ nebo $E_{4\to7}$
P_2 = průnik s $E_{0\to2}$ nebo $E_{2\to5}$ nebo $E_{5\to7}$
P_4 = průnik s $E_{0\to3}$ nebo $E_{3\to6}$ nebo $E_{6\to7}$

Nyní již tedy existují tři nalezené vrcholy polygonu, ale stále zbývá zjistit ty zbývající, protíná-li rovina těleso ve více bodech. Každý z těchto bodů může ležet na některé ze

zbývajících hran nenáležejících do hlavních cest. Ty jsou na obrázku 4.2 označeny tečkovanou čarou.

Jako první se bude hledat průsečík s hranou $E_{1\to5}$, na obrázku 4.2 označenou tečkovaně. Pokud existuje platný bod průniku s touto hranou, musí být vložen mezi nalezené průsečíky se světle a středně šedou cestou, aby byla zachována konzistence výsledného polygonu. To je znázorněno na obrázku 4.2 vpravo. Pokud rovina tuto hranu neprotíná, je nastaven bod P_1 na některý z bodů P_0 nebo P_2, což jsou průsečíky roviny se světle a středně šedou cestou. V tomto případě bude uvažováno vždy přiřazení bodu s nižším indexem. Analogicky se vyřeší průsečíky se zbývajícími hranami a jsou tak získány zbývající tři body průniku s hranami ležícími mimo hlavní cesty:

$P_1 =$ průnik s $E_{1\to5}$, jinak P_0
$P_3 =$ průnik s $E_{2\to6}$, jinak P_2
$P_5 =$ průnik s $E_{3\to4}$, jinak P_4

Nyní je tedy známo všech šest průsečíků aktuální roviny řezu a obálky objemu. Ty jsou seřazeny ve správném pořadí, takže tvoří platný polygon. Jako poslední zbývá ověřit, jestli lze stejný postup použít i v případě, že je přední hrana nebo dokonce celá přední stěna rovnoběžná s průmětnou. V tomto případě pouze stačí vybrat a označit jeden z předních vrcholů jako V_0 a protější zadní vrchol jako V_7 a potom lze algoritmus analogicky aplikovat i na tento případ. Platí totiž, že je průsečík roviny a hrany rovnoběžné s průmětnou ignorován, existuje jich totiž nekonečně mnoho.

4.3.2 Implementace na GPU

4.3.2

Implementace popsaného algoritmu byla provedena pomocí vertex programu s využitím jazyka Cg. Tento program byl navržen k tvorbě vysokého počtu rovnoběžných řezů, kdy mezi každými dvěma je stejně velká vzdálenost. Dalším aspektem návrhu byl co nejmenší počet přenášených dat do grafické karty při každém snímku a minimalizace stavových změn.

```
1 glBegin(GL_POLYGON);  // GL_TRIANGLE_FAN
2   glVertex2i(0,0);
3   glVertex2i(1,0);
4   glVertex2i(2,0);
5   glVertex2i(3,0);
6   glVertex2i(4,0);
7   glVertex2i(5,0);
8 glEnd();
```

Výpis 4.1: Display list pro jeden řez

Pro každý řez je do vertex shaderu posláno šest vrcholů, uložených v display listu. Ten je ukázán na výpisu 4.1. Pořadové číslo řezu je také odesláno do vertex programu jako další parametr, popř. lze k odeslání využít např. texturovací souřadnici. Tento způsob byl upřednostněn před použitím `Vertex Buffer Object (VBO)` z ryze praktického důvodu. Pokud je totiž generován velký počet řezů, interakce se zobrazeným objemem může být příliš pomalá. Proto má uživatel možnost volby, jestli bude během interakce s objemem redukován počet generovaných řezů, stejně jako možnost nastavení tohoto koeficientu redukce. Tím bude vytvářeno a rasterizováno méně polygonů a interakce se tak stane rychlejší. Pokud by bylo použito VBO, lze sice přímo uložit pořadové číslo řezu na pozici nevyužitého indexu, ale bylo by příliš složité vybírat ty správné indexy z pole vrcholů během redukovaného

zobrazení. Tím by v důsledku pravděpodobně bylo zobrazení s využitím VBO pomalejší, než pomocí display listu. Navíc je index řezu konstatní pro všech šest vrcholů a OpenGL nepodporuje primitiva obsahující pouze jednu souřadnici, proto by i s použitím VBO musel být index roviny řezu zduplikován do všech vrcholů roviny, tudíž odeslání jednoho parametru do vertex programu navíc žádné znatelné snížení výkonu nezpůsobí. Při použití display listu je sice posíláno více příkazů `glBegin()` a `glEnd()`, ale na druhou stranu to při rozumném počtu řezů nebude rapidní ztráta výkonu a nejsou potřeba příslušná OpenGL rozšíření.

Současná implementace využívá pouze jedné 3D textury, tudíž není třeba používat pokročilých geometrických translací a podobných operací, aby byla obálka objemu nasměrována na správnou pozici. I v případě, že by bylo nutné naimplementovat rozložení rozsáhlých dat do více 3D textur, by pravděpodobně příslušné translace a změny měřítka obstarávala samotná aplikace. Také je možné rozšířit vertex program o možnost generování řezů s různě velkými nebo umístěnými tělesy, pokud by to bylo potřeba, za cenu mírného zvýšení počtu stavových změn.

V aktuální implemetaci, která je vidět na výpisu 4.2, každý řezaný objem obsahuje konstatní pole vrcholů a translační vektor určující pozici aktuálně renderované části objemu v prostoru `vecTranslate`. Vrcholy jsou nainicializovány do pole `pVertices` (řádek 2) ihned po vytvoření vertex programu a od tohoto momentu se již nemění. V rámci každého renderovaného snímku jsou do vertex programu zaslány následující proměnné. Pohledová a projekční transformační matice `matModelViewProj`, podle níž je nakonec vypočítána výsledná pozice vrcholu, index předního vrcholu – předního ve smyslu směru generování řezů, tedy v tomto případě zadního, protože se řezy generují odzadu dopředu – `nFrontIndex`. Protože i v případě použití více objemových těles jsou všechna natočená stejným směrem, zůstává tento index neměnný v rámci jednoho snímku. Dále je poslán vektor `vecView`, který je použit jako normálový vektor $\overrightarrow{n}_P$ roviny, a vzdálenost mezi sousedními řezy `fPlaneIncr`. Na řádku 24 je poté spočtena vzdálenost d v rovnici roviny 4.2.

Konstatní pole vrcholů `pSequence` obsahuje permutace indexů vrcholů s ohledem na daný index předního vrcholu, a z něj jsou na řádcích 29 a 30 získány ohraničující vrcholy právě testované hrany. Jak bylo zmíněno v předešlé části, musí být na průsečík otestovány příslušné hrany ve správném pořadí, v závislosti na indexu aktuálně počítaného průsečíku.

Aby mohly být zjištěny průsečíky P_1, P_3 a P_5, musí se nejprve otestovat průsečík s „tečkovanými“ hranami (viz obrázek 4.2) a pokud nebude průsečík ani s jednou z těchto hran nalezen, je nakonec otestována odpovídající cesta (na obrázku 4.2 souvislou čarou). Z toho plyne, že je nutno otestovat na průnik maximálně čtyři hrany. Příslušný `for` cyklus je na řádku 27. Pokud je testován průsečík P_0, P_2 nebo P_4, musí být testovány pouze tři hrany, náležející příslušné cestě. Pro tento případ cyklus obsahuje příkaz `break`, který ukončí iteraci, pokud je průsečík nalezen po nejvýše třech iteracích.

V konstantních polích vrcholů `pVerts1` a `pVerts2` jsou uloženy počáteční a koncové vrcholy hran, které musí být postupně otestovány na průnik. Jejich indexování je provedeno pomocí pořadového čísla vrcholu, uloženého v `pos.x` (první souřadnici vrcholu odeslané do vertex shaderu, viz display list) v kombinaci s aktuálním číslem iterace e, jak je uvedeno na řádcích 29 a 30. Zde jsou zjištěny správné indexy vrcholů testované hrany. Samotné vektory vrcholů jsou poté na řádcích 32 a 33 načteny z konstatního pole vrcholů `pVertices`. Na následujících řádcích je spočítán počátek a směrový vektor hrany. Zde je bráno v úvahu i posunutí objemu specifikované vektorem `vecTranslate`.

Na řádku 37 je dále spočítán jmenovatel z rovnice 4.4. Pokud je jmenovatel roven 0, z předchozí části nebo definice skalárního vektoru je známo, že rovina řezu a testovaná hrana jsou rovnoběžné. V tom případě je na řádcích 38 a 39 nastaven parametr λ jako -1,

```
1  uniform int         pSequence[64];
2  uniform float3      pVertices[8];
3  uniform int         pVerts1[24];
4  uniform int         pVerts2[24];
5
6  void main(
7    in       int2       pos    : POSITION,
8    in       float3     color  : COLOR,
9    // updated per cube
10   uniform  float3     vecTranslate,
11   // updated per frame
12   uniform  int        nFrontIndex,
13   uniform  float      fPlaneStart,
14   uniform  float      fPlaneIncr,
15   uniform  float      fSliceNum,
16   uniform  float3     vecView,
17   uniform  float4x4   matModelViewProj,
18   // output variables
19   out      float4     outVertex   : POSITION,
20   out      half3      outTexCoord : TEXCOORD0,
21   out      half3      outColor    : COLOR
22 )
23 {
24   float PlaneDist = fPlaneStart + (fSliceNum * fPlaneIncr);
25   float3 Position;
26
27   for (int e = 0; e < 4; e++)
28   {
29     int Idx1 = pSequence[int(nFrontIndex * 8 + pVert1[pos.x*4 + e])];
30     int Idx2 = pSequence[int(nFrontIndex * 8 + pVert2[pos.x*4 + e])];
31
32     float3 vecV1 = pVertices[Idx1];
33     float3 vecV2 = pVertices[Idx2];
34     float3 vecStart = vecV1 + vecTranslate;
35     float3 vecDir   = vecV2 - vecV1;
36
37     float Denom = dot(vecDir, vecView);
38     float Lambda = (Denom != 0.0) ?
39                    (PlaneDist - dot(vecStart, vecView))/Denom : -1.0;
40
41     if ((Lambda >= ZERO) && (Lambda <= ONE))
42     {
43       Position = vecStart + (Lambda * vecDir);
44       break;
45     }
46   }
47   outVertex = mul(matModelViewProj, float4(Position, 1.0));
48   outTexCoord = 0.5 * (Position) + 0.5;
49   outColor = color;
50   return;
51 }
```

Výpis 4.2: Vertex program pro výpočet průniku roviny řezu a obálky objemu

aby bylo zaručeno, že nebude považován za validní. Je-li jmenovatel nenulový, je hodnota λ vypočítána podle rovnice 4.4. Na řádku 41 se testuje hodnota λ, jestli leží v intervalu $[0, 1]$. Vyhoví-li tomuto testu, byl nalezen platný průsečík, je zjištěna jeho přesná pozice na hraně a lze ukončit cyklus.

Nakonec je na řádku 47 transformována pozice průsečíku pomocí transformační matice a získána tak přesná pozice v prostoru. Řádek 48 už pouze převádí získanou pozici vektoru z intervalu $[-1, 1]$ jednoduchou změnou měřítka na interval $[0, 1]$ používaný texturovacími jednotkami.

Nepochybně by tato metoda mohla být použita i pro výpočet průsečíku s několika rovnoběžnými rovinami řezu s využitím funkce, která by stihla grafickou pipeline plnit předdefinovaným bufferem vrcholů. Stejně tak by tato funkce mohla být použita pro výpočet řezů rovinami nerovnoběžnými s průmětnou.

4.4 Eliminace prázdných oblastí

Protože se ve vstupních datech může vyskytovat velké množství nevyužitého prostoru nebo oblastí obsahující data mimo aktuálně zobrazovaný rozsah, je vhodné naimplementovat některou z metod na detekci a odstranění tohoto prázdného prostroru. Tím lze ušetřit výpočetní výkon GPU a dosáhnout tak více snímků za vteřinu.

Jako nejvhodnější se zde jeví využití oktalových stromů (*octree*), kdy je v každé další úrovni objemový blok rozdělen na osm stejně velkých uzlů. Každý uzel uchovává informaci o minimální a maximální hodnotě dat náležejících jemu nebo jeho poduzlům. Poté lze snadno omezit rozsah zobrazovaných hodnot pomocí přenosové funkce (více informací viz kapitola 5) nebo specifikací minimální a maximální zobrazené hodnoty. Všechny uzly, které leží mimo rozsah zobrazených hodnot nebo obsahují pouze plně transparentní voxely, potom mohou být jednoduše vypuštěny z renderovacího řetězce.

Prvotní implementace práce tyto oktalové stromy obsahovala. Protože však testovací data byla mírně odlišného charakteru než vstupní data, nejevila se tato metoda jako přínosná pro zobrazování tohoto typu dat. A to z toho důvodu, že testovací data obsahovala nepříliš rozdílné hodnoty po celém povrchu řezu, proto se ořezání volného prostoru pomocí oktalového stromu nemohlo uskutečnit. Aby měly tyto stromy přínos a ušetřily tak renderování nepotřebných dat, musely by být rozděleny na mnohem více úrovní než je tomu u jiných typů dat, např. získaných pomocí CT.

Implementace těchto oktalových stromů byla jednoduchá. Každý uzel uchovával údaje o minimální a maximální hodnotě dat, která tento uzel pokrýval a tyto hodnoty byly dále propagovány do všech vyšších úrovní. Pokud se poté vyskytl požadavek na vyrenderování dat pouze z určitého rozsahu, postupně se procházel celý strom a pokud některý z uzlů tomuto rozsahu vyhovoval, algoritmus se zanořil do další úrovně tohoto uzlu a takto rekurzivně pokračoval až do doby, než byl celý strom otestován. Každý uzel též obsahoval informaci o posunutí jeho počátku v rámci celého objemu, což reprezentuje vektor `vecTranslate` odesílaný do vertex programu. Na základě tohoto vektoru byl poté celý uzel umístěn na patřičnou pozici a správně vyrenderován.

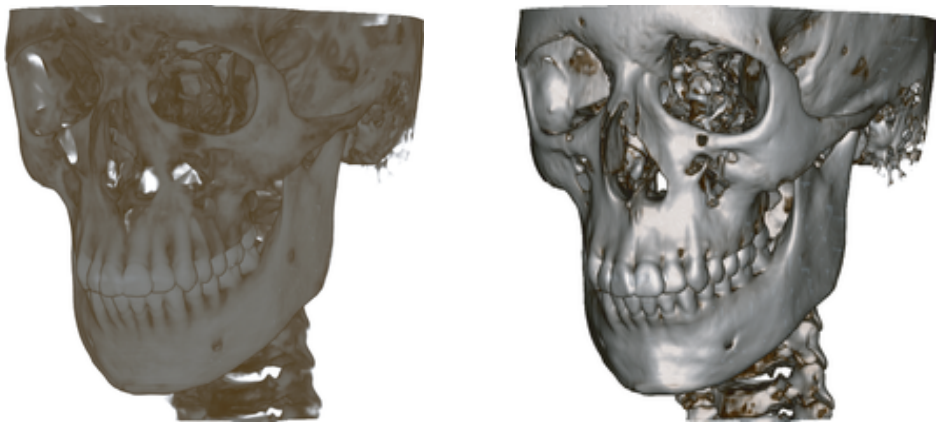
Protože nově získaná testovací data jsou lehce odlišného charakteru a nejsou nutně rozprostřena přes celý řez, pravděpodobně by použití oktalového stromu mělo vyšší přínos pro zobrazení tohoto typu dat a bude znovu zahrnuto.

4.5 Osvětlení a stínování

I když jsou ve výsledcích vizualizace již docela obstojně identifikovatelné jednotlivé části dat, stínováním lze přinejmenším vylepšit celkovou vizuální kvalitu výstupu a také umožňuje detailnější představu o prostorovém rozložení dat.

Osvětlení a stínování ve volumetrickém zobrazení využívá stejných osvětlovacích modelů a stínovacích technik jako v klasickém polygonálním zobrazení. Cílem těchto technik je vylepšení vizuálního vjemu renderovaných objektů, hlavně zdůraznění jejich tvarů a struktury. Toho je dosaženo simulací světelných efektů interagujících s povrchem objektu. Stínování

je tedy uskutečněno pomocí světel, kdy je v každém bodě povrchu znám normálový vektor a je na něj aplikován osvětlovací model. Porovnání výsledku bez použití stínování a s použitím stínování je na obrázku 4.3. Jsou rozlišovány dva různé osvětlovací modely: globální osvětlení a lokální osvětlení.



(a) Bez použití stínování (b) S použitím stínování

Obr. 4.3: Porovnání volumetrického zobrazení. CT snímek lidské hlavy (512x512x333, 16 bit) [4]

Příspěvek světelného paprsku směřujícího od světelného zdroje, který je odražen objektem, je nazýván *lokálním osvětlovacím modelem*. V tomto modelu je stínování jakéhokoliv objektu scény nezávislé na stínování každého dalšího objektu. Oproti tomu se v *globálním osvětlovacím modelu* přičítá odražené světlo od ostatních objektů k lokálnímu osvětlovacímu modelu aktuálního objektu. Tento model je přesnější, fyzikálně správnější a produkuje realističtěji vypadající výsledky. Je však také výpočetně mnohem náročnější.

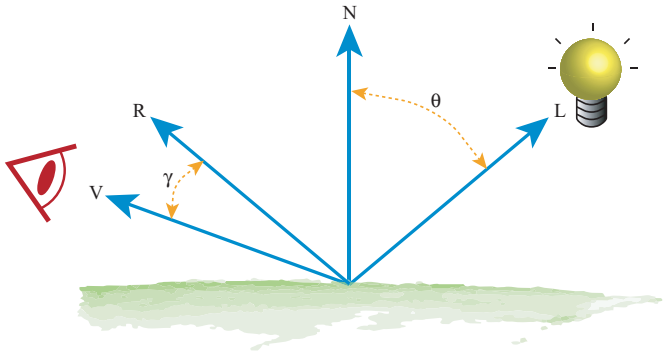
4.5.1 Osvětlovací model

4.5.1

Ve volumetrickém zobrazení se právě díky nižší výpočetní náročnosti používá lokální osvětlovací model, nejčastěji Phongův osvětlovací model. Tento model je navržen tak, aby byl výpočet osvětlení co nejrychlejší, ale současně, aby osvětlení výsledné scény působilo přirozeně. Výsledkem obou těchto snah je samozřejmě určitý kompromisní model, který však poskytuje pozoruhodně kvalitní výsledky. Přestože tedy postrádá přílišnou fyzikální realističnost, velice často se používá z důvodu efektivního výpočtu. Je vhodný zejména pro renderování obrazu s využitím fragment shaderů, protože je takto zpracován opravdu každý fragment podílející se na výsledné scéně a lze v něm tedy rychle spočítat osvětlení s podporou GPU akcelerovaných vektorových výpočtů. Phongův osvětlovací model je popsán např. v [4].

Phongův osvětlovací model potřebuje k výpočtu pouze aktuální bod a pozici všech světelných zdrojů a světlo se v něm na povrchu tělesa rozkládá do tří světelných složek: složky (ambient light) okolního světla, difúzní složky (diffuse light) a odlesků (specular light).

Složka okolního (ambient) světla udává intenzitu světla přicházejícího na povrch objektu ze všech směrů poté, co se mnohokrát ve scéně odrazilo od různých povrchů. Zjednodušeně tedy reprezentuje přirozené osvětlení scény. Phongův osvětlovací model předpokládá konstantní intenzitu ambientní složky v rámci celé scény. Každý povrch



Obr. 4.4: Parametry Phongova osvětlovacího modelu [4]

má daný ambientní koeficient, který závisí na fyzikálních vlastnostech. Ten udává, jaká část světla je od povrchu odražena. Intenzita odraženého světla, tj.

$$I_{amb} = I_a \cdot \kappa_{amb} \tag{4.5}$$

kde I_a je konstantní intenzita ambientního světla ve scéně a κ_{amb} je ambientní koeficient povrchu, je nezávislá na vzájemné poloze zdroje, tělesa i pozorovatele.

Složka difuzního (rozptýleného) světla udává intenzitu části světla dopadající na těleso z jednoho světelného zdroje (ať už bodového, či směrového) a odraženého rovnoměrně do všech směrů. Intenzita odraženého světla, tj.

$$I_{diff} = \begin{cases} I_i \cdot \kappa_{diff} \cdot \cos(\Theta), & |\Theta| < 90° \\ 0 & jindy \end{cases} \tag{4.6}$$

kde I_i je intenzita dopadajícího světla, κ_{diff} je koeficient materiálu pro difuzní odraz a Θ je úhel mezi normálou povrchu a vektorem dopadu světla, jak ukazuje obrázek 4.4. Je závislá pouze na vzájemné poloze normály stěny tělesa N a vektoru dopadu světla L. Nezávisí však na pozici pozorovatele, protože je světlo odraženo do všech směrů rovnoměrně.

Odlesky (specular) jsou část světla dopadajícího na těleso z jednoho světelného zdroje a odražené převážně v jednom směru dodržující zákon odrazu světelných paprsků. Objekt se tak chová jako zrcadlová plocha. Protože se ale žádné těleso se nechová jako ideální zrcadlo, i zde jde minimum odraženého světla do okolí. Nicméně i přesto intenzita takto odraženého světla závisí na vzájemné poloze světelného zdroje, povrchu tělesa i pozici pozorovatele. Intenzita odlesků je určena vztahem

$$I_{spec} = \begin{cases} I_i \cdot \kappa_{spec} \cdot \cos^m(\gamma), & |\gamma| < 90° \\ 0 & jindy \end{cases} \tag{4.7}$$

kde I_i je intenzita dopadajícího světla, κ_{spec} je koeficient odlesku materiálu a γ úhel mezi úhlem odrazu R a vektorem pohledu V, jak je uvedeno na obrázku 4.4. Parametr

m je tzv. míra lesklosti tělesa. Difúzní těleso bude mít exponent nulový nebo velmi blízký nule, vysoce lesklé těleso zde naopak může mít velmi vysokou hodnotu. Čím vyšší je hodnota tohoto parametru exponentu, tím jsou odlesky na tělese plošně menší, ale o to intenzivnější.

Celkový vztah pro výpočet Phongova osvětlovacího modelu je potom získán pomocí jednoduchého sečtení všech tří odražených složek, tj.

$$I = I_{amb} + I_{diff} + I_{spec} \tag{4.8}$$

4.5.2 Výpočet normálového vektoru

Vzhledem ke skutečnosti, že vstupní objemová data jsou čistě skalární hodnoty, musí být vyřešen problém získání a následného uložení normálového vektoru povrchu pro každý bod objemu. Bez něj totiž nelze vypočítat rozptýlenou a odraženou složku světla.

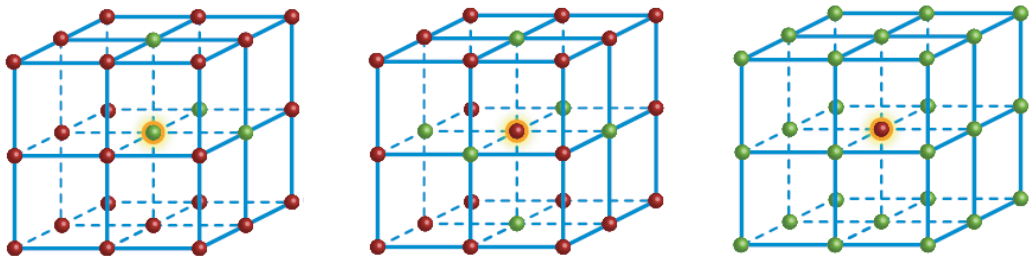
Objemová data jsou obvykle získána snímáním spojitých objektů a po ukončení snímání neexistuje žádná znalost o normálovém vektoru. Proto musí být normála povrchu nalezena prozkoumáním okolí daného voxelu. Kvalita výsledného obrazu silně závisí na metodě provádějící výpočet normály. Místo normálového vektoru pro libovolný bod objemových dat lze použít gradient v tomto bodě. Gradientem se rozumí parciální derivace prvního řádu skalární hodnoty $I(x, y, z)$ definovaná jako

$$\nabla I = (I_x, I_y, I_z) = (\frac{\delta}{\delta x} I, \frac{\delta}{\delta y} I, \frac{\delta}{\delta z} I). \tag{4.9}$$

a velikost tohoto vektoru lze pak získat pomocí vztahu

$$\|\nabla I\| = \sqrt{I_x^2 + I_y^2 + I_z^2}. \tag{4.10}$$

Nejčastěji používané metody pro výpočet gradientu jsou vidět na obrázku 4.5:



(a) Intermediate differece gradient

(b) Central difference gradient

(c) Neumann gradient

Obr. 4.5: Metody pro výpočet gradientu. Zelené body označují použité vzorky. [4]

Intermediate difference gradient přijímá jako vstup čtyři sousední voxely (viz obrázek 4.5(a)). Gradient ∇ daného voxelu V na pozici (x, y, z) se zjistí jako:

$$\begin{aligned} \nabla_x &= V_{x+1,y,z} - V_{x,y,z} \\ \nabla_y &= V_{x,y+1,z} - V_{x,y,z} \\ \nabla_z &= V_{x,y,z+1} - V_{x,y,z} \end{aligned} \tag{4.11}$$

Central difference gradient přijímá jako vstup šest sousedních voxelů (viz obrázek 4.5(b)). Gradient ∇ daného voxelu V na pozici (x, y, z) se zjistí jako:

$$\begin{aligned} \nabla_x &= V_{x+1,y,z} - V_{x-1,y,z} \\ \nabla_y &= V_{x,y+1,z} - V_{x,y-1,z} \\ \nabla_z &= V_{x,y,z+1} - V_{x,y,z-1} \end{aligned} \tag{4.12}$$

Neumann gradient přijímá jako vstup 26 sousedních voxelů (viz obrázek 4.5(c)). Obecně je tato metoda spíše teoretickým rozhraní založeném na lineární regresi. Více o této metodě je napsáno např. v [4].

Jak testy provedené v [4] ukázaly, výhodou operátoru Intermediate difference je, že dokáže detekovat detaily o vysoké frekvenci. Bohužel však také vede k méně hezkým výsledkům, pokud jsou renderována silně zašuměná data. Naopak operátor Central difference tyto vysokofrekvenční detaily odfiltruje, může tak být použit jako filtr typu dolní propusť. Avšak tento operátor může někdy také vynechat příliš úzké struktury, které mohou být ve výsledku podstatné. Stejně jako Intermediate difference operátor, ani tento není izotropní, což může způsobovat problémy v případě, kdy by měla být přidělována průhlednost voxelům na základě velikosti gradientního vektoru. To ovšem v této práci příliš nevadí.

4.5.3 Použití gradientu

4.5.3

Nyní je známý způsob, jak získat pro každou buňku objemu její normálový vektor. Jak již ale bylo zmíněno, objemová data jsou čistě intenzitní. Jak k nim tedy gradient přiřadit? Jako nejvhodnější se jeví dva různé způsoby:

- Předpočítat a uložit do textury společně s objemovými daty
- Počítat během rasterizace ve fragment shaderu

První způsob je rychlejší, výpočet gradientu musí proběhnout pouze jednou a v procesu rasterizace jsou již známé hodnoty gradientu pro všechny buňky objemu a jsou tak získány společně s její hodnotou pomocí jednoho přístupu do 3D textury. Jeho nevýhodou je ovšem paměťová náročnost. Pokud jsou totiž data čistě intenzitní, 3D textura obsahuje pouze jednu barevnou složku. Po výpočtu gradientu se ovšem rozšíří o další tři složky, pro každou osu jednu. Vznikne tak textura o čtyřnásobné velikosti. Proto není příliš doporučeno tento způsob používat na systémech s malým množstvím operační paměti.

Oproti tomu druhý způsob zachovává původní paměťovou náročnost samotných dat, na druhou stranu je ale mnohem pomalejší. Gradient je totiž počítán za běhu rasterizačního procesu, kdy se pro každý zpracovávaný fragment provede příslušný počet přístupu do 3D

textury, výpočet gradientního vektoru a následně samotného stínování. Tento proces je proveden při každém překreslení scény a podle použité metody pro výpočet gradientu ovlivňuje rychlostně celý proces renderování.

V této práci je naimplementováno použití metody Central difference a Intermediate difference a uživatel si mezi nimi může přepínat podle svého uvážení a jejich vhodnosti.

4.6 Zobrazení pomocí fragment shaderu

Konečným procesem zobrazení je zpracování vygenerovaných a rasterizovaných fragmentů. To může být provedeno pomocí fixního zobrazovacího řetězce nebo s využitím programovatelného GPU zobrazovacího řetězce. Pro tuto práci bylo zvoleno zobrazení pomocí fragment shaderů, protože je toto řešení vysoce modifikovatelné a přizpůsobitelné konkrétním požadavkům. Dále lze také pomocí programovatelného řetězce provádět efekty, kterých by pomocí fixní rasterizační jednotky pravděpodobně nebylo možné docílit.

Pro implementaci fragment (pixel) shaderů padla volba na jazyk Cg, stejně jako příslušný překladač a běhové prostředí. Ve stejném jazyce je napsán i vertex program provádějící výpočet polygonů řezu popsaný v 4.3. Volba tohoto jazyka však nemusí být konečná. Pokud se prokáže, že některý z existujících nebo budoucích jazyků pro shadery, např. GLSL, by mohl být pro implementaci vhodnější, nemělo by činit velké problémy vytvořené shadery do tohoto jazyka přepsat.

V následujících částech budou popsány všechny naimplementované varianty fragment shaderů, u každé z nich uvedeny i jeho požadavky na minimální verzi fragment profilu, stejně jako výsledky po kompilaci a jeho případná vylepšení.

4.6.1 Jádro fragment shaderu

V rendereru je během vykreslování jednoho snímku vypočítána a změněna spousta hodnot, které mají ve většině případů vliv na výsledné zobrazení. Všechny tyto parametry, stejně jako identifikátory textur v OpenGL stroji, musí být odeslány do použitého fragment programu, který s nimi poté nějakým způsobem pracuje, ať již v rámci prostého získání hodnoty objemu a mapování optických hodnot, nebo ve výpočtu osvětlení, popř. gradientu. Aby nemusely být tyto parametry specifikovány v každém z naimplementovaných fragment shaderů, byl vytvořen standardní hlavičkový soubor, kde jsou všechny uvedeny. Tento soubor je poté pomocí direktivy `#include „fragGlobals.cgh“` vložen do všech vytvořených fragment shaderů. Ve výpisu 4.3 jsou všechny tyto parametry uvedeny. Renderer je posílá do GPU během renderování nového snímku vždy všechny, záleží však na daném fragment programu, které z nich využije.

V parametru `texVolume` je uložen OpenGL identifikátor textury obsahující vlastní objemová data, v `texColorTable` a `texPreintTable` identifikátory textur s přenosovými funkcemi. Následující tři parametry `texSlicesX`, `texSlicesY` a `texSlicesZ` obsahují identifikátory příspěvkových textur v jednotlivých osách objemu, viz kapitola 5. Parametr `vecView` obsahuje pohledový vektor, `vecLight` vektor pozice světelného zdroje a `vecDistance` vzdálenost dvou sousedních voxelů na každé ose objemu. Všechny tyto parametry mohou být bez problémů použity v těle fragment programu.

Následující čtyři parametry specifikují vzdálenost k dalšímu řezu v pořadí, tedy renderovanému v předchozím kroku (`vecNextPlane`), rozsah hodnot, které jsou obsaženy v objemu (`nDataMinMax`) a použitý typ přenosové funkce (`nTableType`) a parametr `nGradientType` na řádku 16 obsahuje uživatelem zvolenou metodu pro výpočet gradientu. Nemělo by s nimi

```
1  uniform sampler3D   texVolume;
2  uniform sampler1D   texColorTable;
3  uniform sampler2D   texPreintTable;
4  uniform sampler1D   texSlicesX;
5  uniform sampler1D   texSlicesY;
6  uniform sampler1D   texSlicesZ;
7
8  uniform half3       vecView;
9  uniform half3       vecLight;
10 uniform half3       vecDistance;
11 uniform half3       vecNextPlane;
12
13 uniform float2      nDataMinMax;
14 uniform int         nTableType;
15
16 uniform int         nGradientType;
17
18 #define COEFF_DISCARD  (1e-10)
19 #define SHININESS      (10)
```

Výpis 4.3: Parametry fragment shaderů

být manipulováno přímo v těle fragment shaderu, ale jen pomocí vytvořených funkcí, jejichž přesnější popis je uveden ve zbytku této části. Na řádcích 18 a 19 jsou dále definice konstant pro minimální zobrazitelnou hodnotu fragmentu a míru lesklosti použitou ve výpočtu osvětlení. Tyto konstanty mohou být později též nahrazeny externě modifikovatelnými parametry, pravděpodobně to však nebude nutné.

Protože jsou data uložena v 16 bitové textuře, ale vlastní hodnoty mají většinou rozsah okolo 12 bitů, je tak využito jen minimum hodnot z celkového rozsahu 3D textury. To však může činit problémy při specifikaci přenosové funkce, kdy je vytvořena textura hodnot, které jsou poté mapovány na vlastní datové složky. Bylo by totiž třeba nastavovat jen určitou malou část této textury a zbytek by zůstal nevyužitý. Nastavení by tak bylo mnohem méně přesné a mohly by snáze vznikat vysoké frekvence. Uživatel také může chtít specifikovat míru, jakou se jednotlivé řezy objemu na všech osách budou podílet na výsledném zobrazení. Pokud je fragment určen jako nepodstatný, je třeba mít možnost jej nějakým způsobem ignorovat.

Nebo lze používat určitou sekvenci příkazů ve více shaderech a vznikal by tak zbytečně duplicitní kód. Z tohoto důvodu bylo vytvořeno několik funkcí, tvořících jádro fragment shaderů. Tyto funkce zjednodušují rozhodování, které fragmenty se podílejí na výsledném obraze a které mohou být bez problémů z renderovacího řetězce vypuštěny, stejně jako implementaci některých používaných technik.

Funkce `GetSlices()`

Nejprve byla vytvořena funkce `GetSlices()` uvedená ve výpisu 4.4, vracející míru celkového příspěvku daného fragmentu do výsledného obrazu. Tato funkce pouze získá příslušné hodnoty z příspěvkových textur, jejichž souřadnice jsou určeny jednotlivými složkami parametru `texCoord`. Do něj se předávají texturovací souřadnice vytvořené vertex programem. Tyto hodnoty poté mezi sebou vynásobí a jejím výstupem je celkový koeficient příspěvku. Tato funkce by měla být použita v každém fragment programu jako první. Tím se snadno ze zpracování vypustí fragmenty, u kterých sám uživatel rozhodl, že je nechce zobrazovat.

```
1  float GetSlices(in half3 texCoord)
2  {
3    half4 slicesX = tex1D(texSlicesX, texCoord.x);
4    half4 slicesY = tex1D(texSlicesY, texCoord.y);
5    half4 slicesZ = tex1D(texSlicesZ, texCoord.z);
6
7    return float(slicesX.r * slicesY.g * slicesZ.b);
8  }
```

Výpis 4.4: Funkce GetSlices() pro výpočet celkového příspěvku fragmentu

Funkce `GetTableValue()`

Patrně nejdůležitější funkcí, zajišťující získání správných hodnot optických vlastností z přenosové textury, je `GetTableValue()` uvedená na výpisu 4.5. Ta provádí interpolaci načtené hodnoty z 3D textury do rozsahu $[0.0, 1.0]$ použitém jako texturovací souřadnice přenosové textury. Tato interpolace je nutná právě z dříve zmíněného důvodu omezeného rozsahu vstupních dat. Do parametru `nDataMinMax` je po načtení nových vstupních dat odeslán rozsah jejich hodnot a poté pro každý renderovaný fragment interpolováno umístění jeho hodnoty do tomto rozsahu. Tím je umožněna specifikace přenosové funkce v rámci celého rozsahu vstupních dat mnohem přesněji, než kdyby byla namapována na všechny hodnoty 16 bitového objemu. Lze pak využít celý rozsah přenosové textury a záleží pouze na implementaci, jak velká tato textura bude. Tato interpolace je provedena pomocí vnitřní funkce Cg jazyka `smoothstep()`, jak je ukázáno na řádku 3.

```
1  half4 GetTableValue(in half value, in half3 texCoord)
2  {
3      half index1 = smoothstep(nDataMinMax.x, nDataMinMax.y, value);
4      // color table
5      if (nTableType == 0)
6      {
7          return tex1D(texColorTable, index1);
8      }
9      // preint table
10     else if (nTableType == 1)
11     {
12         half4 value1 = tex3D(texVolume, texCoord + vecNextPlane);
13         half index2 = smoothstep(nDataMinMax.x, nDataMinMax.y, value1.a);
14         return tex2D(texPreintTable, half2(index1, index2));
15     }
16 }
```

Výpis 4.5: Funkce GetTableValue() pro výběr hodnot z přenosové textury

Funkce `GetTableValue()` se ovšem také dokáže podle nastavených parametrů rozhodnout, kterou přenosovou texturu má použít a podle toho vrací správnou hodnotu. Proto má dva vstupní parametry. Jedním je zjištěná hodnota objemu právě zpracovávaného fragmentu, která je interpolována. Ta je použita v případě 1D i 2D přenosové funkce. Druhým parametrem jsou kompletní texturovací souřadnice (a tedy i pozice v objemu) daného fragmentu. Pokud je použita preintegrace, tedy 2D přenosová funkce, je pomocí těchto souřadnic zjištěna hodnota fragmentu z předchozího řezu a poté společně s aktuální hodnotou tvoří texturovací souřadnici do pre-integrační textury. To je ukázáno na řádcích $12 - 14$. Návratovou hodnotou této funkce jsou tak v každém případě správné optické vlastnosti daného fragmentu.

Funkce `CalculateShading()`

Protože je osvětlení použito ve více fragment shaderech, byla též naimplementována funkce uvedená ve výpisu 4.6 zajišťující výpočet potřebných světelných složek. Tato funkce přijímá jako vstupní parametr normálový vektor `normal` a do výstupních parametrů `diffuse` a `specular` ukládá hodnotu difúzní a odleskové složky světelného modelu.

```
1  void CalculateShading(
2    in    half3   normal,
3    out   half    diffuse,
4    out   half    specular)
5  {
6    diffuse = abs(dot(vecLight, normal));
7    half3 vecHalf = normalize(vecLight + vecView);
8    specular = pow(max(dot(vecHalf, normal), 0), SHININESS);
9    if (diffuse < 0) { specular = 0; }
10   return;
11 }
```

Výpis 4.6: Funkce CalculateShading() pro výpočet osvětlení

Funkce `CalculateGradient()`

Tato funkce slouží k výpočtu gradientu a jejím výstupem je difúzní a odlesková složka světelného modelu. Podle specifikovaného typu gradientu se rozhodne, kterou funkce pro výpočet gradientu použije. Po jejím dokončení je získaná hodnota gradientu předána jako normálový vektor do funkce `CalculateShading()`, jejímž výstupem jsou již zmíněné složky světelného modelu. Jednotlivé funkce pro výpočet gradientu jsou uvedeny níže.

```
1  void CalculateGradient(
2    in    half4   value,
3    in    half3   texCoord,
4    out   half    diffuse,
5    out   half    specular)
6  {
7    half3 gradient = 0.0.xxx;
8    if (nGradientType == 0)
9    { CalculateGradientIntermediateDifference(value, texCoord, gradient); }
10   if (nGradientType == 1)
11   { CalculateGradientCentralDifference(value, texCoord, gradient); }
12   gradient = normalize(gradient);
13   CalculateShading(gradient, diffuse, specular);
14 }
```

Výpis 4.7: Funkce CalculateGradient() pro výpočet gradientu

Funkce `CalculateGradientIntermediateDifference()`

Tato funkce počítá gradient metodou Intermediate Difference. Obecně lze gradient spočítat dvěma způsoby. Buď tak, že je vypočten přímo z hodnot příslušných voxelů, nebo že jsou nejdříve zjištěny optické vlastnosti všech příslušných voxelů a je spočítán až z těchto hodnot. Současná implementace používá první způsob výpočtu z hodnot buněk. Důvod, proč byl zvolen právě tento způsob, je hlavně ten, že při použití optických vlastností je uskutečněno stejné množství přístupů do přenosové textury jako do objemové. S použitím hodnot je ovšem přístup do přenosové textury pouze jeden, a to pro aktuální fragment. Navíc gradient

založený na optických vlastnostech závisí na uživatelském nastavení těchto vlastností a ne na datových hodnotách. Zdrojový kód této funkce je vidět na výpisu 4.8.

```
1  void CalculateGradientIntermediateDifference(
2    in    half4   value,
3    in    half3   texCoord,
4    out   half3   gradient)
5  {
6    half4 valueX = tex3D(texVolume, texCoord+half3(vecDistance.x, 0.0, 0.0));
7    half4 valueY = tex3D(texVolume, texCoord+half3(0.0, vecDistance.y, 0.0));
8    half4 valueZ = tex3D(texVolume, texCoord+half3(0.0, 0.0, vecDistance.z));
9
10   gradient.x = valueX-value;
11   gradient.y = valueY-value;
12   gradient.z = valueZ-value;
13 }
```

Výpis 4.8: Funkce CalculateGradientIntermediateDifference()

Funkce `CalculateGradientCentralDifference()`

Tato funkce počítá gradient metodou Central Difference. Využívá stejného principu jako metoda Intermediate Difference popsaném výše. Tělo funkce zobrazuje výpis 4.9.

```
1  void CalculateGradientCentralDifference(
2    in    half4   value,
3    in    half3   texCoord,
4    out   half3   gradient)
5  {
6    half4 valueX1 = tex3D(texVolume, texCoord-half3(vecDistance.x, 0.0, 0.0));
7    half4 valueX2 = tex3D(texVolume, texCoord+half3(vecDistance.x, 0.0, 0.0));
8    half4 valueY1 = tex3D(texVolume, texCoord-half3(0.0, vecDistance.y, 0.0));
9    half4 valueY2 = tex3D(texVolume, texCoord+half3(0.0, vecDistance.y, 0.0));
10   half4 valueZ1 = tex3D(texVolume, texCoord-half3(0.0, 0.0, vecDistance.z));
11   half4 valueZ2 = tex3D(texVolume, texCoord+half3(0.0, 0.0, vecDistance.z));
12
13   gradient.x = valueX2-valueX1;
14   gradient.y = valueY2-valueY1;
15   gradient.z = valueZ2-valueZ1;
16 }
```

Výpis 4.9: Funkce CalculateGradientCentralDifference()

Pokud by bylo potřeba použít nějaký další parametr nebo funkci ve více fragment programech, je tento hlavičkový soubor možností, jak toho může být snadno dosaženo. Takto je usnadněna implementace dalších fragment shaderů, aniž by uživatel musel zkoumat, jaké parametry jsou předávány z rendereru a jak používat textury vizualizačních funkcí.

4.6.2 Základní fragment shader (*Basic*)

Tento fragment shader vykonává nejjednodušší akci, jaká by měla být implementována také ve všech ostatních pokročilejších fragment programech. Teprve na ni se budou nabalovat další rozšiřující funkce. Tento shader tedy slouží také jako šablona pro ostatní renderovací fragment shadery. Činnost tohoto základního fragment programu uvedeného na výpisu 4.10 je popsána v následujícím odstavci.

Nejprve je zjištěn celkový příspěvek daného fragmentu (viz 5.1.4) ve výsledném obraze, jak je vidět na řádku 5. Je-li tento koeficient menší než specifikovaná hodnota (řádek 6), daný

fragment obraz neovlivňuje a může být tudíž z renderovacího procesu vypuštěn pomocí direktivy `discard`, jak je vidět na řádku 7. Jestliže fragment ovlivňuje určitou měrou výsledek renderování, dalším krokem je přečtena hodnota objemu nacházející se na pozici `texCoord0`. To je provedeno na řádku 9 pomocí interní funkce Cg jazyka `tex3D()`. Na základě této hodnoty jsou poté získány příslušné optické vlastnosti voláním funkce `GetTableValue()` na řádku 10. Nakonec je tato hodnota barvy a průhlednosti vynásobena příspěvkovým koeficientem a poslána na další zpracování dále do renderovacího řetězce. Alternativním řešením by mohlo být vynásobit tímto koeficientem pouze průhlednost, její hodnota totiž současně ovlivňuje i výslednou barvu fragmentu.

Rozšiřující kód pokročilejších fragmentů, který bude uváděn v následujících částech, je poté vložen mezi řádky 10 a 12.

```
1  half4 main(
2    half3 texCoord0 : TEXCOORD0,
3    half4 color : COLOR) : COLOR
4  {
5    float coeff = GetSlices(texCoord0);
6    if (coeff < COEFF_DISCARD)
7    { discard; }
8
9    half4 value = tex3D(texVolume, texCoord0);
10   half4 outColor = GetTableValue(value.a, texCoord0);
11
12   return (outColor * coeff);
13 }
```

Výpis 4.10: Základní fragment shader (Basic)

Tabulka 4.1 ukazuje srovnání základního fragment shaderu v rámci dostupných fragment profilů s ohledem na potřebný počet instrukcí GPU jednotky a využití dostupných pomocných registrů. Podobná tabulka bude též uvedena i u všech ostatních popsaných fragment programů.

Fragment profil	FP20	ARBFP1	FP30	FP40
Počet instrukcí	N/A	35	32	32
Počet registrů (R/H)	N/A	3/-	1/3	2/3

Tabulka 4.1: Srovnání fragment programu Basic při použití různých fragment profilů

4.6.3 Fragment shader s výpočtem stínování (*Shading*)

4.6.3

Program Shading slouží ke zobrazení objemových dat s využitím stínování a předpočítaného gradientu uloženého ve 3D textuře. Její tělo je uvedeno ve výpisu 4.11. Na řádku 2 je nejprve převedena uložená hodnota gradientu na normálový vektor v rozsahu hodnot $[-1.0, 1.0]$ a řádek 3 volá funkci `CalculateShading()` pro výpočet potřebných složek světelného modelu. Ty jsou poté aplikovány na optické vlastnosti a tvoří výslednou barvu, jak ukazují řádky $5-6$.

Tabulka 4.2 ukazuje srovnání tohoto fragment shaderu v rámci dostupných fragment profilů. Jak je vidět, ve srovnání na podle počtu instrukcí se základním fragment shaderem a fragment programem Gradient se tento shader pohybuje někde mezi nimi, s téměř polovičním počtem instrukcí než program Gradient. To je ovšem vykompenzováno již zmíněnou

```
1 half diffuse, specular;
2 half3 vecNormal = 2.0*(value.xyz) - 1.0.xxx;
3 CalculateShading(vecNormal, diffuse, specular);
4
5 outColor.xyz *= diffuse;
6 outColor.xyz += specular;
```

Výpis 4.11: Fragment shader s výpočtem stínování (Shading)

vyšší spotřebou paměti pro vytvoření gradientní 3D textury. Je ovšem rychlejší než výpočet gradientu přímo v rámci fragment shaderu.

Fragment profil	FP20	ARBFP1	FP30	FP40
Počet instrukcí	N/A	48	45	44
Počet registrů (R/H)	N/A	4/-	2/3	2/3

Tabulka 4.2: Srovnání fragment programu Shading při použití různých fragment profilů

4.6.4 Fragment shader s výpočtem gradientu (*Gradient*)

4.6.4

Fragment shader Gradient je výpočetně nejnáročnější, hlavně kvůli vysokému množství přístupů do 3D textury. Pro výpočet gradientu a složek světelného modelu je použita naimplementovaná funkce `CalculateGradient()`, která provede veškerou potřebnou činnost a vlastní fragment shader již jen použije získané světelné složky k vytvoření výsledné barvy fragmentu. Kód vlastního fragment programu zobrazuje výpis 4.12.

```
1 half diffuse, specular;
2 CalculateGradient(value, texCoord0, diffuse, specular);
3
4 outColor.xyz *= diffuse;
5 outColor.xyz += specular.xxx;
```

Výpis 4.12: Fragment shader s výpočtem gradientu (Gradient)

Tabulka 4.3 ukazuje srovnání tohoto fragment shaderu v rámci dostupných fragment profilů. Jak je vidět, tento fragment shader ke svému běhu potřebuje více než dvojnásobný počet instrukcí oproti shaderu základnímu, což se ovšem také patřičně promítne do výsledné rychlosti zobrazení.

Fragment profil	FP20	ARBFP1	FP30	FP40
Počet instrukcí	N/A	80	75	71
Počet registrů (R/H)	N/A	7/-	1/6	2/6

Tabulka 4.3: Srovnání fragment programu Gradient při použití různých fragment profilů

4.6.5 Řezový fragment shader (*Wire*)

4.6.5

Tento fragment shader má pouze jediný úkol, zobrazit drátěný model jednotlivých řezů. Slouží tak spíše pro představu o hustotě a pozici řezů, popř. k ladicím účelům. Po aktivaci

zobrazí každý fragment uričtou barvou, jak je vidět ve výpisu 4.13 na řádku 5. Současně s tím je přepnuto zobrazení scény pomocí `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)` na drátěný model, aby tak byly vyrenderovány pouze obrysy řezových polygonů. Jedná se pouze o pomocný fragment program, který je jako jediný použitelný současně, resp. ihned po dokončení renderování objemu, s některým z hlavních renderovacích shaderů.

```
1 half4 main(
2   half3 texCoord0 : TEXCOORD0,
3   half4 color : COLOR) : COLOR
4 {
5   return half4(0.73, 0.73, 1.0, 0.5);
6 }
```

Výpis 4.13: Fragment shader zobrazující řezové polygony (Wire)

Hardwarové nároky tohoto programu jsou minimální. Je funkční i v rámci fragment profilu FP20. Tabulka 4.4 nabízí srovnání počtu instrukcí a využitých registrů při použití různých fragment profilů.

Fragment profil	FP20	ARBFP1	FP30	FP40
Počet instrukcí	0	1	1	1
Počet registrů (R/H)	-/-	0/-	0/0	0/0

Tabulka 4.4: Srovnání fragment programu Wire při použití různých fragment profilů

4.6.6 Shrnutí

Vzhledem k vlastnostem dat popsaným výše muselo být vytvořeno jádro fragment shaderů, skládající se z několika parametrů a funkcí. Ty tak mohou být obsaženy v každém vytvořeném fragment shaderu. Ten by měl být založen na fragment programu Basic, jinak nemusí renderer fungovat přesně podle očekávání. Dále bylo úspěšně naimplementováno několik fragment programů potřebných nebo užitečných pro tuto práci. Tyto programy vykonávají každý svou pevně přiřazenou akci a v současnosti je nelze mezi sebou kombinovat nebo sekvenčně sestavovat. Jedinou výjimku tvoří fragment shader Wire zobrazující drátěný model jednotlivých řezů.

Nemělo by být obtížné naimplementovat nový shader vykonávající požadovanou akci. Z rendereru jsou totiž posílány veškeré potřebné parametry a informace. Pokud by ovšem měla být uživateli dána možnost implementace vlastních fragment programů, bude potřeba naimplementovat inteligentní správu těchto programů uvnitř rendereru, aby mohly být nové shadery snadno rozpoznány a připraveny k použití.

Protože vnitřní funkce Cg jazyka `smoothstep()`, potřebná k mapování objemových hodnot na pozici v přenosové textuře, vyžaduje minimálně fragment profil FP30, je zřejmé, že žádný z fragment shaderů implementujících renderování objemových dat nebude fungovat na grafických kartách nepodporujících žádný z vyšších fragment profilů. FP20 je už však stejně silně zastaralý profil, takže by to v důsledku nemělo vadit.

4.7 Možná vylepšení a modifikace

Během návrhu zobrazovací metody bylo objeveno několik vylepšení a modifikací. Tyto modifikace mohou posloužit k vylepšení kvality zobrazení, zrychlení vizualizace nebo např. přidávají novou funkčnost umožňující lepší interakci s uživatelem a budou uvedeny v této části. U každé z nich bude popsán její princip, oblast vylepšení a momentální stav implementace.

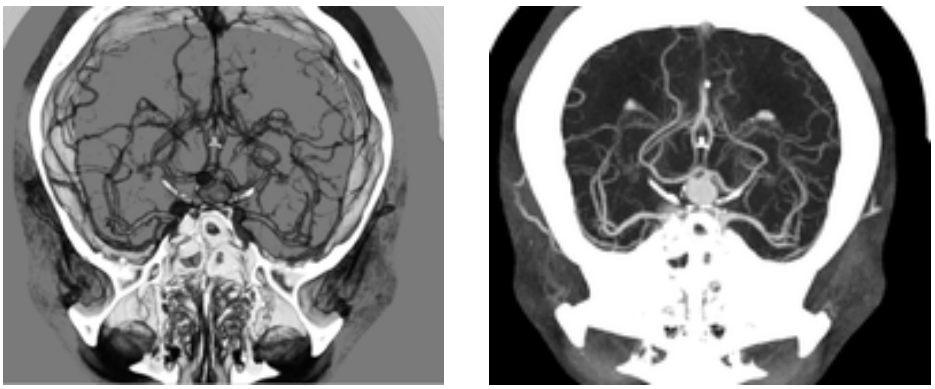
Budou zde však uvedena pouze rozšíření týkající se samotného renderovacího procesu. Modifikace vlastní metody pro vizualizaci značených částí jsou popsány až v části 5.2.

4.7.1 Podpora méně výkonných grafických karet

Kvůli vysokému počtu konstant přesouvaných do registrů GPU po inicializaci vertex programu, jedná se o seřazená pole indexů a jednotlivých vrcholů, nemůže být tato metoda použita na grafických kartách podporujících pouze vertex profil VP20. Pokud by bylo cílem zmíněný algoritmus použít i na grafické kartě s takto nízkou verzí profilu, bylo by nutné tento algoritmus naimplementovat také s využitím klasického CPU přístupu a pomocí přepínače, detekujícího schopnosti použité grafické karty přepínat mezi výpočtem na GPU nebo CPU. Toto by mohlo být jedním z rozšíření použitých v budoucí verzi, protože grafických karet s podporou maximálně VP20 je v současnosti stále veliké množství. Na druhou stranu je třeba zvážit, jestli by se kvůli ceně implementace vyplatilo něco takového zahrnout do požadavků na rozšíření. Bylo by totiž třeba počítat také s nižšími fragment profily, které tyto karty samozřejmě obsahují, a tak by bylo znemožněno použití pokročilých fragmentových technik. Cena této modifikace by tak v konečném souhrnu nejspíše vyšla mnohem vyšší než pořízení výkonnější grafické karty. Některé z podporovaných grafických karet jsou uvedeny v kapitole 6.

4.7.2 Projekce maximální intenzity

Projekce maximální intenzity (*Maximum intensity projection – MIP*) je variantou přímého zobrazení objemu, kde je však místo skládání vlastností optického modelu podél paprsku vybrána pouze maximální hodnota. Ta je poté promítnuta na průmětnu nebo použita k určení barvy a průhlednosti daného pixelu.



(a) Přímé zobrazení (b) MIP

Obr. 4.6: Srovnání přímého zobrazení a projekce maximální intenzity [1]

Důležitou oblastí, kde se tato metoda velice často využívá, je zobrazování dat získaných pomocí magnetické resonance (MRI). Tato data totiž obvykle trpí obrovským množstvím šumu, takže je obtížné v nich nalézt nějakou významnou isoplochu nebo určit přenosovou funkci s požadovanou přesností. Proto může být MIP v jejich iterpretaci užitečná. V těchto datech totiž většinou mívají pevné struktury vyšší hodnoty než jejich okolí a mohou tedy být touto metodou snadno vizualizovány, jak ilustruje obrázek 4.6. Na vstupní data používaná v této práci pravděpodobně příliš velký vliv mít nebude, ale bylo by možné ji časem propojit s některou z pokročilejších metod do dvouprůchodového zobrazení, kdy budou v prvním kroku nejprve získány oblasti s hodnotou vysokou nebo blížící se nastavenému prahu a ty poté vyrenderovány pomocí další metody schopné pracovat s přenosovými funkcemi, popř. gradientem kvůli stínování.

V grafickém hardwaru je MIP velice snadno implementovatelná pomocí operátoru maximum místo klasického alpha blendingu během míchání výsledného obrazu do frame bufferu. Příslušné OpenGL rozšíření se jmenuje `GL_EXT_blend_minmax` a je podporováno snad všemi současnými i staršími grafickými kartami.

Projekce maximální intenzity v aktuální fázi zahrnuta zatím není, ovšem neměl by být příliš problém ji naimplementovat, pokud by se ukázala jako užitečná. Jejím možným vylepšením by dále ještě mohlo být určení výsledné barvy pixelu podle přenosové funkce nebo již zmíněné použití k předzpracování vstupních dat pro některou z pokročilejších technik.

4.7.3 Zobrazení isoploch

4.7.3

Zobrazení isoploch může být považováno za metodu ležící zhruba mezi metodami přímého zobrazení a hledání povrchu. Jedná se totiž o způsob, kdy je hledán povrch o požadované hodnotě, ale pouze tyto voxely (nebo i voxely s vyšší nebo nižší hodnotou) jsou poté vizualizovány.

Na počátku je zvolena hodnota hledané izoplochy a poté jsou zobrazeny všechny voxely s hodnotou vyšší nebo rovnou této zadané. Tato technika se tedy dá přirovnat k funkci `glAlphaTest` s parametrem `GL_GEQUAL` v OpenGL. Stejně tak lze použít i zbývající operátory funkce `glAlphaTest` podle toho, jaká data se mají zobrazovat. Zobrazení izoploch by mohlo být jednoduše naimplementovatelné pomocí této vestavěné funkce OpenGL, bohužel je kvůli mapování hodnot na objemová data ztracena alfa složka barvy, podle níž se tato funkce řídí. Jedinou možností se tak nejspíše jeví její umístění do hardwarového shaderu. Její hlavní výhodou je to, že omezuje počet fragmentů nutných ke zpracování. Bohužel je alfa test proveden až po rasterizaci, takže musí být ověřeny všechny fragmenty. Nevýhodou samozřejmě bude její nedokonalost zobrazení, kdy je většinou třeba ještě propojit výsledky této metody s nějakou pokročilejší metodou pro vizualizaci výsledných hodnot pomocí jejich optických vlastností.

Stejně jako lze předpočítat integrální sumu všech příspěvků optických vlastností do přenosové funkce, je možné tuto sumu předpočítat i pro metodu zobrazení isoploch. Více se o této problematice lze dočíst např. v [1]. Problémem ovšem je, že není tato metoda příliš dobře dokumentovaná a pravděpodobně by její implementace zabrala spoustu času kvůli různým experimentům.

Implementace zobrazení isoploch zatím není implementována. Jedná se totiž o specifičtější techniku. I ta ovšem může nalézt své opodstatnění, pokud by měla důležitá data shodnou nebo velice blízkou hodnotu. Vhledem k charakteru vstupních dat by tato metoda své místo v implementaci nalézt mohla i v nějaké pokročilejší formě, protože značené části

jsou od okolí většinou odlišeny rozdílnými hodnotami a samotné značené buňky mají často hodnoty relativně blízké.

4.7.4 Vyrovnání jasu

Tuto modifikaci lze zařadit spíše do kategorie post-processingu. Jedná se o to, že se v případě, kdy je použitý základní fragment shader a není tedy aplikováno osvětlení, vyrenderovaný objem může jevit jako příliš tmavý. Ekvalizace jasu tak může pomoci zesvětlit výsledného obrazu tak, že nalezne pixel s nejvyšší intenzitou, tomu přiřadí jasovou hodnotu 1.0 a podle ní dále roznásobí všechny ostatní pixely.

Pravděpodobně jediným způsobem, jak toho dosáhnout, pokud jsou použity fragment shadery, je již zmíněným post-processingem. Objemová data jsou renderována do framebufferu, ten je poté načten jako textura, popř. jako pole pixelů v paměti a operace ekvalizace probíhá nad ním. Upravený framebuffer se nakonec odešle zpět na průmětnu a renderovací proces může pokračovat vykreslováním obalového tělesa, editoru funkcí atd.

Nevýhodou této modifikace jistě bude zvýšená výpočetní náročnost, která se projeví hlavně v rámci procesu ekvalizace. Ten totiž bude muset probíhat buď kompletně na CPU, nebo se pomocí CPU zjistí potřebná maximální jasová hodnota, získaný framebuffer se vyrenderuje jako textura a pomocí speciálního fragment shaderu bude ekvalizována jasová složka pro všechny pixely této textury. Řešení pomocí GPU by pravděpodobně nemuselo být tak výpočetně náročné jako s využitím čistě softwarového řešení.

Modifikace tohoto typu se jeví jako užitečná hlavně v případě, kdy je přenosová funkce specifikována především v nižší hladině hodnot. Potom jsou jasové složky příliš nízké a lidské oko může mít problémy s rozpoznáváním důležitých detailů, ale po zvýšení hodnot optických vlastností se tyto detaily mohou skrýt za méně průhledné části dat nebo může být jas naopak příliš vysoký. Proto by mohlo být vhodné toto rozšíření v některé z příštích verzí naimplementovat, pravděpodobně verzi s využitím GPU akcelerace.

4.8 Zhodnocení

Nyní je tedy dostupný renderer, který dokáže zobrazovat libovolná volumetrická data ne příliš složitými, ale dostatečně mocnými a vizuálně kvalitními technikami. Tento renderer je schopný se vypořádat s daty celočíselného typu o rozsahu 8 nebo 16 bitů načtenými z různých zdrojů. Kvůli použité metodě výpočtu průsečíku řezu s objemem implementované na GPU je potřeba grafická karta s podporou vertex profilu vyššího než VP20 a fragment profilu vyššího nežř FP20. Což však v současné době splňují i karty ze segmentu těch levnějších a konzumních.

Také bylo objeveno několik metod, které by mohly mít pro aplikaci přínos, stejně jako bylo otestováno několik různých technik u nichž se zjistilo, že nejsou pro tuto práci příliš vhodné. Není však jisté, které z těchto modifikací se v dalších verzích objeví.

Kapitola 5

Vizualizace značených buněk

Metoda vizualizace značených částí objemových dat je založená na jednoduchém principu mapování optických hodnot na intenzitní hodnoty vyskytující se v objemových datech. Díky tomuto mapování lze rozlišit jednotlivé části dat podle jejich barevnosti a průhlednosti.

V části 5.1 se nachází úvod do problematiky vizualizace a mapování hodnot a jsou zde popsány použité metody, pomocí nichž lze specifikovat mapovací funkci nebo usnadnit zobrazení některých částí dat. Další část 5.2 popisuje možné modifikace, které mohou uživateli usnadnit výběr vhodné přenosové funkce, interakci uživatele s aplikací nebo mohou pomoci k lepším výsledkům. Na závěr této kapitoly je v sekci 5.3 shrnut přínos použitých metod pro tuto práci a zhodnoceny výsledky kompletního renderovacího procesu.

5.1 Použité metody

5.1

Protože objemová data jsou jen skalárními hodnotami, takže z nich není možné přímo určit jejich optické vlastnosti, je nutné dát uživateli možnost jednotlivým hodnotám objemu tyto vlastnosti vhodným způsobem přiřadit. Bohužel tento proces přiřazení nepatří mezi příliš snadné, uživatel je totiž nucen zkoušet různé varianty, což může být časově náročné. Nejjednodušším způsobem, jak specifikovat optické vlastnosti, by mohlo být vytvoření statické barevné palety, která by se jednoduše odeslala do grafického adaptéru. Uživatel by ovšem raději měl možnost specifikovat tyto hodnoty podle svého uvážení, aby mohl nalézt nejvhodnější variantu. Aby byla tato činnost co nejintuitivnější, bylo prozkoumáno a aplikováno několik metod, pomocí nichž lze tento proces zjednodušit a uživatelsky zpříjemnit.

5.1.1 Editor přenosové funkce

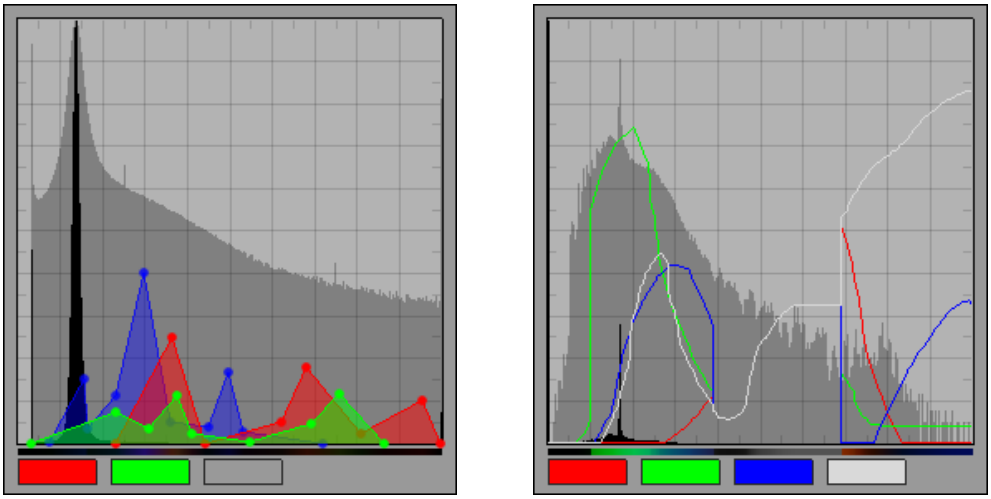
5.1.1

Přenosová funkce patří do oblasti klasifikace dat, kdy je jejím hlavním úkolem přiřazení optické vlastnosti silně abstraktním skalárním datovým hodnotám objemu. Právě tyto optické vlastnosti jsou poté použity při renderingu výsledného obrazu. Existuje více typů přenosové funkce z nichž lze zvolit ten nejvhodnější pro potřeby cílové aplikace.

K usnadnění jejího nastavení slouží editor přenosové funkce. Pomocí něj lze pro každou hodnotu objemu určit její optické vlastnosti. Pro snadnější orientaci v objemových datech editor též často obsahuje histogram vstupních dat, někdy i s jeho variantou v logaritmickém měřítku. Podle něj lze v některých případech částečně určit vhodný průběh funkce.

Samotné nastavení může být poté provedeno pomocí křivek jednotlivých barevných kanálů nebo průhlednosti, popř. pomocí polygonů, s nimiž se lépe manipuluje, ale nejsou tak přesné.

V této práci jsou použity oba tyto přístupy a uživatel se mezi nimi v průběhu nastavování může snadno přpínat. Jsou však na sobě vzájemně nezávislé a každý z nich generuje svou vlastní přenosovou texturu, viz 5.1.2. Editor též zobrazuje histogram dat v lineárním i logaritmickém měřítku pro snadnější identifikaci požadovaných oblastí. Ukázka jednotlivých možností nastavení se nachází na obrázku 5.1, kde je na obrázku 5.1(a) ukázáno nastavení pomocí polygonu, zatímco obrázek 5.1(b) zobrazuje nastavení pomocí křivek. Ty obsahují vlastní kanál pro průhlednost, zatímco u polygonů je určena z jejich hodnot.



(a) Polygonální editor (b) Křivkový editor

Obr. 5.1: Editor přenosové funkce

U polygonů ovšem nemusí stačit, a v drtivé většině případů ani nestačí, výchozí počet uzlových bodů. V editoru tedy byla naimplementována možnost přidání nebo odebrání uzlu polygonu. Maximální počet těchto uzlů je v současné fázi omezen na 100, tento počet lze ale velice snadno změnit. Navíc, pokud by bylo potřeba přesněji určit více lokálních maxim nebo minim, se jako vhodnější jeví použití křivkové funkce namísto polygonální.

Po načtení dat je velikost přenosové funkce automaticky přizpůsobena rozsahu datových hodnot. Tím je dostupná možnost nastavení vlastnosti libovolné hodnotě objemu. Protože však rozsah objemových hodnot může být větší než šířka editoru, bylo třeba naimplementovat možnost změny aktuálně zobrazené oblasti přenosové funkce. Takto lze určit přibližnou funkci pro celý rozsah a poté postupně přesně specifikovat optické vlastnosti pro jednotlivé části.

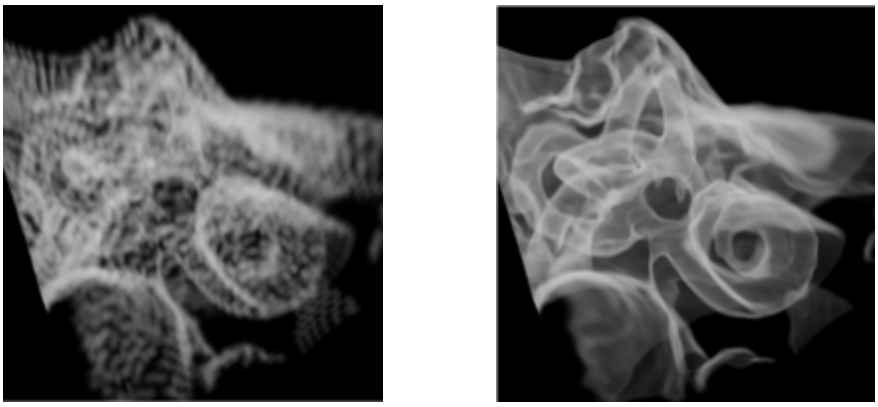
5.1.2 Přenosová textura

5.1.2

Nyní je dostupný dostatečně silný nástroj na specifikaci optických vlastností objemu. Vyvstává však problém, jakým způsobem tyto vlastnosti aplikovat na odpovídající data. Protože je pro komunikaci aplikace s GPU jednotkou nejvhodnější textura, budou nastavené hodnoty vhodným způsobem převedeny do textury. To je provedeno po každé změně některého z kanálů, aby uživatel mohl interaktivně kontrolovat vhodnost nastavených hodnot. Tuto texturu je možné také pro kontrolu zobrazit v hlavním okně rendereru. Přenosová tex-

tura je také někdy nazývána závislou, protože je nejprve získána hodnota objemu a teprve tato hodnota tvoří její texturovací souřadnici.

Vždyť přenosová funkce by ale také mohla být uložena přímo v objemových datech. Existují však minimálně dva důvody, proč tomu tak není. Je vysoce neefektivní aktualizovat kompletní objemová data při každé změně přenosové funkce, navíc jsou-li tyto změny velice časté. Je proto vhodnější načítat menší lookup texturu než celá objemová data. Také by tím vznikaly rušivé artefakty, protože jsou optické vlastnosti uloženy před samotnou vizualizací, tedy i před interpolací hodnot, které jsou poté skutečně renderovány. Je totiž mnohem vhodnější získat optické vlastnosti až pro přesné nainterpolované hodnoty. Proces, kdy jsou přiřazeny optické vlastnosti ještě před interpolací, se nazývá pre-klasifikace (*pre-classification*), jestliže jsou optické hodnoty namapovány až na interpolovaná data, jedná se o post-klasifikaci (*post-classification*). Rozdíl mezi pre-klasifikací a post-klasifikací je ukázán na obrázku 5.2. Zde je také jasně vidět, že post-klasifikace vykazuje mnohem lepší výsledky, proto je použita i v této práci.



(a) Pre-klasifikace (b) Post-klasifikace

Obr. 5.2: Srovnání pre-klasifikace a post-klasifikace

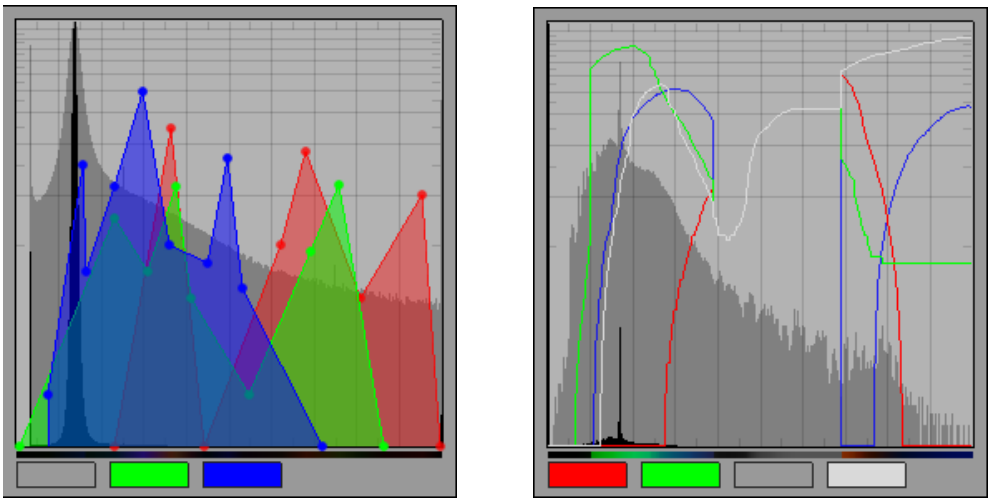
Jsou-li data o rozsahu 8 bitů, je jednoduše pro každou z těchto hodnot získána výsledná barva složením hodnot jednotlivých kanálů na příslušné pozici. Vznikne tak jednorozměrná přenosová textura o velikosti 256 hodnot. Ta je následně v každém renderovaném snímku poslána do grafického adaptéru a v rámci rasterizace jsou z ní získávány optické vlastnosti každého renderovaného fragmentu. Je-li použita polygonální přenosová funkce, složka průhlednosti je spočtena z poměru největší nastavené hodnoty na této pozici vůči maximální možné hodnotě. Pomocí polygonů lze tak lépe specifiovat průhledné oblasti.

Pokud je ovšem rozsah dat vyšší než 256 hodnot, nelze vždy vytvořit takovou texturu, aby pokryla celý rozsah. Na všech grafických adaptérech je totiž omezena maximální podporovaná velikost textury. Tento problém lze nejlépe vyřešit tak, že v případě klasické 1D textury obsahující barevnou paletu se bude její velikost měnit v závislosti na rozsahu dat, a to na 256, 512 nebo 1024 hodnot. U těchto rozměrů je zaručena podpora takových textur grafickým adaptérem a současně je zajištěno, že intervaly mezi jednotlivými vzorky nebudou příliš velké a textura tedy bude přesnější. Vzhledem k potřebným požadavkům na grafický adaptér pro tuto práci by bylo možné použít i texturu o velikosti 2048, protože podporu pro tento typ textur mají snad všechny grafické karty podporující shader profily 3.0. U 2D pre-integrační textury už by změna rozměru měla dopad na rychlost a paměťovou náročnost

aplikace. Pokud by byla zvětšena na 512x512, alokovala by v paměti zhruba 1MB, ovšem výpočet by trval mnohem déle a uživatel by tak přišel o interaktivní odezvu editoru. Při velikosti 1024x1024 by obsadila cca 4MB paměti, ale délka výpočtu by již byla neúnosná a pohybovala se v řádech minut.

5.1.3 Logaritmické měřítko

Jak bylo v [13] zjištěno, ve většině objemových dat více než 30% dat od okraje velice silně ovlivňuje výsledné zobrazení, ale přitom se jedná o hodnoty s viditelností okolo 0.001 (0.1%), tedy velmi blízké 0. I v případě, že by editor přenosové funkce byl roztažen přes celou obrazovku, musel by uživatel specifikovat přenosovou funkci s přesností často i na méně než jeden pixel, aby mohl určit takto nízkou hodnotu, pokud je použito lineární měřítko. Naopak, hodnoty nad 0.05 (5%) jsou ve většině případů mapovány jako téměř neprůhledné, což vede ke skutečnosti, že je 95% prostoru editoru přenosové funkce nevyužito.



(a) Polygonální funkce (b) Křivková funkce

Obr. 5.3: Logaritmické měřítko editoru přenosové funkce

Pokud je však v editoru přenosové funkce použita logaritmická stupnice namísto lineární, jsou oblasti hodnot lépe rozloženy a uživatel tak dostává možnost přesněji specifikovat přenosovou funkci pro téměř průhledné oblasti, nacházející se ve spodní hranici nastavovaných hodnot. Logaritmické měřítko pro jednotlivé typy přenosové funkce je zobrazeno na obrázku 5.3. Převod mezi lineární a logaritmickou stupnicí je založen na vztazích uvedených v [13].

Lze vytvořit bijektivní mapování mezi lineární a logaritmickou stupnicí mapující vzájemně mezi intervaly $[0.0, 1.0]$. Pokud je α hodnotou na lineární stupnici, její logaritmický ekvivalent α' lze vypočítat pomocí vztahu

$$\alpha' = 1.0 - \frac{1}{-a} ln((1 - e^{-a})\alpha + e^{-a}) \tag{5.1}$$

kde $a = max(ln(d), 1)$ a d je maximální nastavitelná hodnota. Zpětné mapování z logaritmického měřítka na lineární, což je potřeba během interaktivní editace přenosové funkce, je provedeno zpětnou transformací pomocí vztahu

$$\alpha = \frac{e^{-a(1-\alpha')} - e^{-a}}{1 - e^{-a}} \tag{5.2}$$

Hodnoty α a α' musí být do těchto vztahů zadány v intervalu $[0.0, 1.0]$. Proto byla vytvořena makra obalující rovnice 5.1 a 5.2, do nichž je možné zadat přímou hodnotu přenosové funkce stejně jako tuto hodnotu již převedenou do uvedeného intervalu.

Logaritmické měřítko je užitečné hlavně v případech, kdy je třeba specifikovat průhledné plochy, ale tak, že jsou stále alespoň minimálně viditelné. Takto lze totiž získat lepší představu o prostorových vztazích v datech. Protože se však nastavení takto nízkých hodnot odehrává v rámci několika málo pixelů, lze si tak pomocí logaritmického měřítka dolní spektrum hodnot roztáhnout.

Jediným dosavadním omezením logaritmické stupnice jsou celočíselné hodnoty. Takto totiž uživatel stejně nemá možnost specifikovat hodnoty ležící „někde mezi“. Pro použití čísel s plovoucí desetinnou čárkou by bylo třeba použít OpenGL rozšíření umožňující pracovat s texturami typu float. V současné fázi se tedy spíše jedná o umožnění snadnější manipulace se spodními hodnotami než o opravdovou logaritmickou stupnici.

5.1.4 Příspěvková funkce

5.1.4

Jak bylo zmíněno v části 4.1, občas se ve vtupních datech vyskytnou zašuměné nebo nepodstatné oblasti, které pouze zpomalují proces vizualizace nebo dokonce degradují výsledek svými optickými vlastnostmi. Proto je vhodné tyto oblasti z renderovacího řetězce vypustit. Toho lze dosáhnout pomocí navržené příspěvkové funkce.

Ta je založena na editoru přenosové funkce a obsahuje tři kanály, pro každou dimenzi jeden. Pomocí těchto kanálů lze specifikovat příspěvek jednotlivých částí objemu podél všech os. Pokud se tedy uživatel rozhodne, že nechce zobrazit 20% objemu od obou okrajů podél osy Z, může toho lehce dosáhnout právě pomocí příspěvkové funkce.

Výstupem funkce jsou tři jednorozměrné textury, pro každý směr jedna. Tyto textury jsou poté použity při výpočtu celkového příspěvkového koeficientu každého fragmentu tak, jak bylo popsáno v části 4.6.1. Tyto textury mají vždy 256 položek, protože je tato velikost postačující a stejný rozměr má také editor funkce. Maximální povolený rozměr 3D textury je nejčastěji 512 hodnot, takže bude použita jedna hodnota nejvýše pro 2 voxely objemu podél jedné souřadné osy, což dává dostatečnou přesnost specifikace.

Pomocí této jednoduché funkce lze velkou měrou ovlivnit výsledný obraz tím, že umožňuje snadným způsobem odfiltrovat nepotřebné nebo rušivé části objemu, stejně jako poskytuje jednoduchý nástroj, jak ořezat objem podél souřadných os.

5.2 Vylepšení a modifikace

5.2

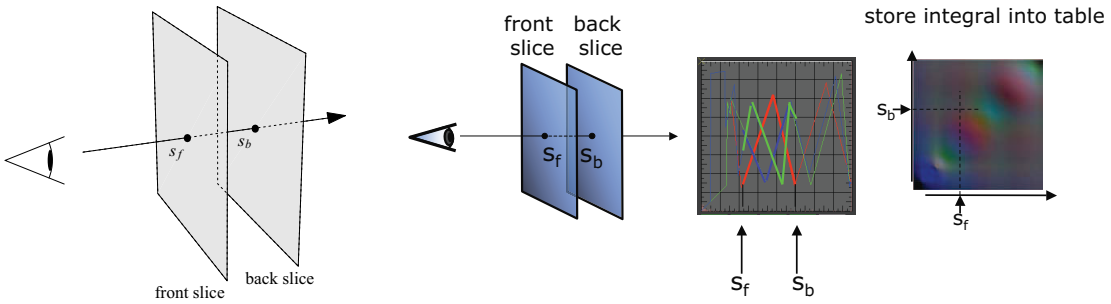
Během návrhu vizualizační metody bylo nalezeno i několik vylepšení týkajících se procesu vizualizace. Některá z nich se již nachází v počáteční fázi návrhu, ale zatím stále nejsou plně implementována. Výčet těchto modifikací samozřejmě není úplný, není vyloučeno, že existuje i jiná metoda vykazující kvalitní výsledky vhodná pro použití v této práci. Zde jsou uvedena pouze dvě rozšíření. Ta se však jeví jako vysoce užitečná, bohužel také implementačně dost náročná.

5.2.1 Pre-integrace přenosové funkce

Jak již bylo zmíněno v předchozí části, existují dvě metody klasifikace dat. Pre-klasifikace a post-klasifikace. Obě mají své výhody i nevýhody. Výhodou pre-klasifikace je hlavně fakt, že se dokáže vypořádat s vysokými frekvencemi specifikovanými v rámci přenosové funkce. Nevýhodou naopak interpolace barev a rozmazanost. Výhodou post-klasifikace je mnohem lepší vizuální kvalita, ovšem za cenu projevování vysokých frekvencí přenosové funkce.

Co ovšem dělat v případě, že je důležitá informace umístěna přesně mezi dvěma sousedními řezy. Intuitivním řešením je zvýšení počtu řezů, čímž se patřičně zahustí vzorkování. Tím ovšem také rapidně klesne výkonnost celého renderovacího systému a začnou se objevovat nepřesnosti a „přepálení“ některých míst, na nichž se míchá mnoho optických vlastností dohromady.

Možným řešením tohoto problému, jak zahrnout i voxely nacházející se mezi dvěma sousedními řezy do výsledného obrazu, může být pre-integrace. V rámci zobrazení řezů jsou totiž vždy specifikovány dvě hodnoty objemu ležící ve směru pohledu přímo za sebou. Ty jsou poté složeny do výsledné barvy pixelu. Tato metoda ovšem nepočítá s tím, že leží další hodnoty i mezi nimi. Metoda pre-integrace tedy oblast mezi sousedními řezy považuje za tzv. „desku“ (*slab*), kde aktuální (přední) řez je přední stranou a předchozí řez stranou zadní, viz obrázek 5.4(a). V této oblasti jsou zintegrovány všechny optických hodnot od přední po zadní stěnu a výsledná hodnota je uložena do 2D přenosové textury, jak je ukázáno na obrázku 5.4(b). Skalární hodnoty získané na předním řezu jsou značeny jako s_f, vzorky na zadním řezu jako s_b.



(a) Slab mezi sousedními řezy

(b) Princip pre-integrace

Obr. 5.4: Metoda pre-integrace [1]

Pre-integrační textura přenosové funkce by se tak dala přirovnat k integraci optických vlastností podél paprsku používané metodou vrhání paprsku, kdy ale nejsou integrovány navzorkované hodnoty podél celého paprsku, ale jen dva vzorky – počáteční a koncový. Proto je pro každé dvě vzdálenosti postupně předpočítána integrace optických vlastností a uložena do pre-integrační textury. Odtud také název této metody. Tato textura je poté zaslána do texturovací paměti a za pomoci shaderu je z ní v poslední fázi zobrazení přečtena hodnota barvy a průhlednosti pro právě zpracovávaný fragment. K tomu je však potřeba dvě texturovací souřadnice, kdy jednou je pozice aktuálně zpracovávaného fragmentu a druhá je zjištěna z texturovacích souřadnic fragmentu, jak je uvedeno v části 4.6.1.

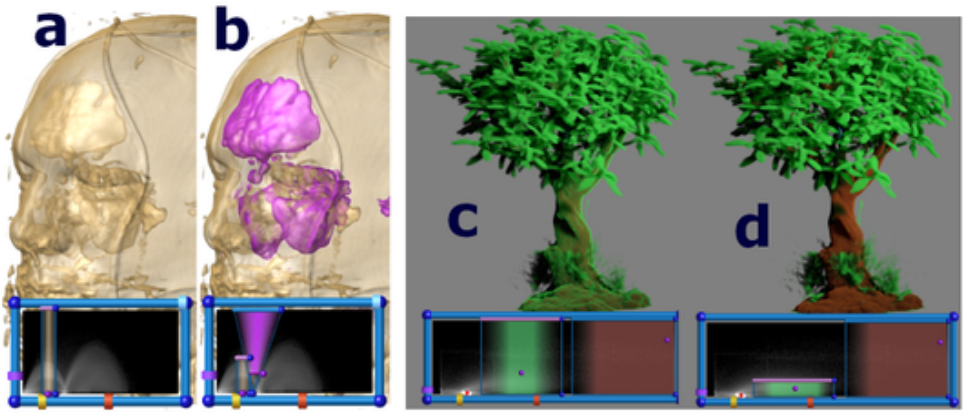
Výpočetní procedura je však výpočetně a časově náročná i při použití tabulky o rozměrech 256x256 hodnot. Při implementaci pre-integrace v rámci této práce by tedy pravděpodobně bylo třeba sáhnout i po některé z jejích akceleračních metod. Ta sice není během

interakce tak přesná jako plná pre-integrace, ale její výpočet probíhá v realném čase. Po dokončení editace přenosové funkce může být poté pre-integrační textura aktualizována spuštěním kompletního výpočtu. Protože je tato oblast v poslední době velice zkoumanou, věnuje se jí spousta prací, jako např. [1, 2, 15, 7]. Kompletní proces pre-integrace, včetně potřebné teorie a matematických vztahů, je dobře popsán v [1].

5.2.2 Vícerozměrná přenosová funkce

Vzhledem k tomu, že je vícerozměrná funkce speciální komplexní technikou vizualizace určitých částí volumetrických dat, bude v této části zmíněna jen velmi stručně. Více podrobností lze získat např. v [1].

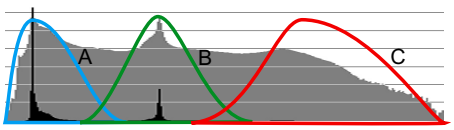
Objemová skalární hodnota nemusí být jedinou veličinou určující rozdílnost jednotlivých vnitřních součástí objemových dat v rámci přenosové funkce. Lze mít v každém vzorku více hodnot tvořících jednotlivé osy vícerozměrné přenosové funkce. Ta je poté naimplementována pomocí čtení ze závislých textur. Jsou-li v každém vzorku uloženy dvě nezávislé hodnoty, přenosová funkce může být poté uložena jako 2D textura v paměti grafického adaptéru. Připojením gradientu k objemovým datům lze poté snáze identifikovat hranice různých součástí objemových dat a také tyto součásti samotné. Na obrázku 5.5 c) a d) je zobrazen způsob separace listů od kmenu v datech bonsaje.



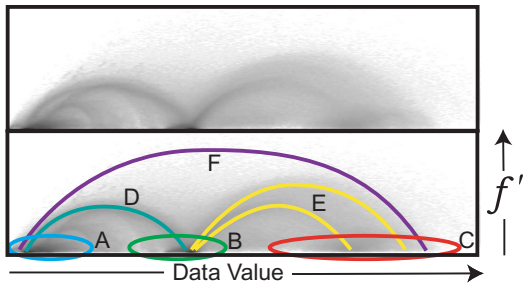
Obr. 5.5: Srovnání 1D (a,c) a 2D (b,d) přenosové funkce [1]

Editor přenosové funkce s jednoduchou funkčností může uživateli umožnit přímý přístup ke všem optickým vlastnostem pomocí sady kontrolních bodů postupně vytvářejících šikmé plochy (rampy) lineárního nebo vyššího řádu. Tento přístup bohužel velice často vede k dlouhotrvajícímu procesu metodou pokus omyl. Přidání další dimenze přirozeně vede k dalšímu zkomplikování uživatelského rozhraní.

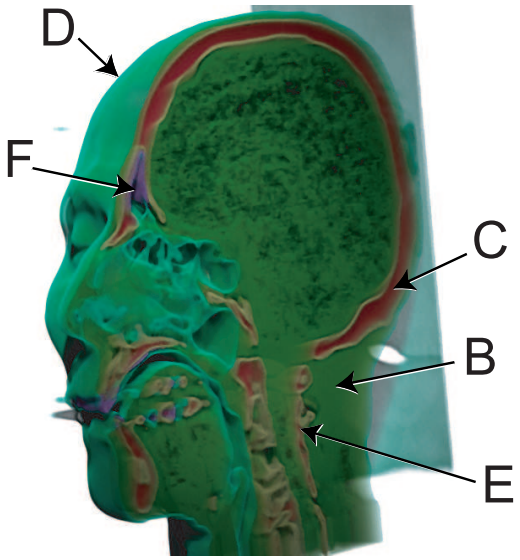
Efektivita editoru přenosové funkce ale může být zvýšena zvýrazněnými charakteristikami dat, které vedou uživatele ke snadnějšímu nastavení správné přenosové funkce. Toho je většinou dosaženo pomocí vícerozměrné globální nebo lokální klasifikační funkce aplikované na příslušnou část vstupních dat a jejíž výstupní charakteristika klasifikace dat je poté zobrazena v rámci editoru přenosové funkce. Po aplikaci všech těchto funkcí budou segmentovány nebo jiným způsobem odlišeny jednotlivé oblasti dat a uživatel tak má usnadněnou specifikaci správné funkce. Ukázka použití těchto klasifikačních charakteristik je na obrázku 5.6.



(a) 1D histogram. Černá oblast značí počet výskytů hodnot v datech, šedá oblast logaritmické měřítko. Barevné oblasti (A,B,C) identifikují základní části dat. Je použita pouze samotná hodnota vzorku.



(b) Spojený 2D histogram v logaritmickém měřítku. Na spodním obrázku je vidět pozice jednotlivých částí (A,B,C) a jejich hranice (D,E,F). Je použita hodnota vzorku a velikost jeho gradientního vektoru.



(c) Všechny označené části kromě vzduchu (A) a jejich hranice nalezené výše za použití 2D přenosové funkce.

Obr. 5.6: Rozlišení nalezených částí dat a jejich hranic. Vzduch (A), měkké tkáně (B) a kosti (C) mohou být identifikovány již pomocí 1D přenosové funkce (a). 1D funkce nicméně nezachytí kombinace těchto částí jako hranice vzduchu a tkání (D), napojení tkání a kostí (E) a hranici kosti se vzduchem (F). To je zobrazeno v (b) a (c). [1]

Další informace o vícerozměrných přenosových funkcích, jejich konstrukci a různých typech lze nalézt v [1]. Tato metoda by jistě byla této práci velice prospěšná, bohužel by tak ale muselo být kompletně reimplementováno rozhraní přenosových funkcí. Vícerozměrné přenosové funkce jsou totiž založeny na obslužných ovládacích prvcích, pomocí nichž lze měnit pozici, šířku a optické vlastnosti. Tyto prvky často využívají předdefinovaných specifických typů klasifikace jako Gaussův elipsoid, inverzní trojúhelník nebo lineární šikmé plochy. Dále by pravděpodobně bylo třeba mít v aplikaci kompletní přehled o všech nastavených oblastech a mít možnost určení např. celkové barvy průhlednosti pro každou z nich.

5.3 Shrnutí a zhodnocení

I když je proces transformace datových hodnot na optické vlastnosti jednoduše implementovatelný jako vyhledávací tabulka (*lookup table*), manuální specifikace přenosové funkce často bývá obtížným a časově náročným procesem. Nezkušený uživatel většinou také nedokáže odhadnout vhodný průběh z histogramu, a proto stráví spoustu času různými variantami nastavení. Pokud je ovšem přenosová funkce specifikována vhodně, výsledek je vizuálně vysoce kvalitní.

Přínos použitých metod pro tuto práci je vysoký, zvláště pokud se uživatel lépe seznámí s editorem přenosové funkce a naučí se jej ovládat a odhadovat vhodný průběh funkce. Ve většině případů stačí v histogramu nalézt některá z lokálních minim a maxim a podle nich orientovat přenosovou funkci. I bez těchto znalostí lze ale v relativně krátké době specifikovat funkci, která dává alespoň orientační výsledky.

Z provedených pokusů vyplynulo, že nejlépe lze přenosovou funkci specifikovat iterativním procesem. Po načtení dat je důležité získat rychlý přehled o jejich vnitřní struktuře. Ve většině případů k tomu poslouží přednastavená přenosová funkce. Po přiřazení vyšší neprůhlednosti oblastem s větším gradientem a proměnné barevnosti datovým hodnotám je vizualizována většina podstatných informací. Od tohoto stavu se lze různými změnami specifikované funkce dobrat poměrně rychle k uspokojujícímu výsledku. Dalším postupným zkoumáním, specifikací a upřesňováním může uživatel nastavit přenosovou funkci produkující výsledky vysoké kvality.

Kapitola 6

Závěr

Tato práce prezentuje principy a různé metody vizualizace objemových dat a popisuje návrh metody pro vizuální odlišení označených částí objemových dat. Práce bude nejčastěji využívat vstupních dat z určité množiny objemových dat, kdy se jedná převážně o biologické snímky sejmuté konfokálním mikroskopem. Nicméně vytvořený renderer je schopen vizualizovat libovolný typ objemových dat a poskytuje výsledky, které lze snadno aplikovat v praxi. Podrobněji lze výsledky této práce shrnout do následujících bodů:

- Nejprve je podán ucelenější úvod do problematiky objemových dat, způsobů jejich zobrazování, různých metod a jejich vlastností. Důraz je kladen na obecné principy těchto metod, jejich výhod i nevýhod a nástin toho, v čem se jednotlivé metody liší. Také je v této části zmíněna i problematika reprezentace objemových dat v paměti a navržen způsob řešení.

- Dále následuje stručné seznámení s fluorescenční mikroskopií, která je použita ke snímání vstupních dat. Také je v této části stručně prezentován princip metody značení GFP. Ta bude v pozdějších fázích implementace použita k vlastnímu značení buněk ve snímaných datech, což je pro tuto práci důležitou procedurou, i když je prováděna ještě před samotným získáním dat. Nakonec je zmínka o modelovém organismu *Caenorhabditis elegans*, který bude sloužit jako organismus pro značení buněk metodou GFP.

- Stěžejní částí celé práce je poté část popisující návrh algoritmu. Protože objemová data nejčastěji uchovávají důležité informace někde uvnitř a ne přímo na svém povrchu, je nutná jejich vizualizace odlišným způsobem. Cílem této práce bylo zobrazení tohoto objemu a umožnění vizualizace některých částí těchto dat tak, aby byly odlišeny od svého okolí. Proto byl také navržen algortimus, jak v objemových datech identifikovat a poté vizuálně zvýraznit patřičné označené oblasti těchto dat. Toho je dosaženo s využitím standardních i modifikovaných algoritmů pro vizualizaci objemových dat. Bohužel musí být tato procedura provedena prozatím pouze manuálně. Byly též nastíněny i některé pomocné metody vylepšující vizuální stránku výsledného obrazu.

  Tento algoritmus bude postupně vylepšován tak, aby co nejlépe splňoval současné i následující požadavky této práce. Nakonec by měl být naimplementován a zakomponován do aplikace FluorCam, zobrazující data získaná dekonvolučním konfokálním

mikroskopem. Některá z možných nebo plánovaných vylepšení jsou detailněji popsaná v částech 4.7 a 5.2.

Po dovedení implementace k aktuálnímu stavu se objevily další požadavky, které by mohly být pro vizualizaci buněk užitečné a k praktickému užívání i potřebné. Tyto požadavky by se daly shrnout do následujících bodů:

- **Informace o skutečném měřítku** – uživatel by měl být informován o skutečných rozměrech dat, stejně jako o pozici určité oblasti v datech. Také by mohly být jednotlivé řezy od sebe vzdáleny proměnliovou vzdáleností, proto by bylo vhodné mít informace i o těchto vzdálenostech a přizpůsobit tomu grafický výstup.
- **Předdefinovaná banka funkcí** – protože je proces nalezení správné přenosové funkce obtížný a zdlouhavý, měl by mít uživatel k dispozici několik předspecifikovaných přenosových funkcí pro data s různými charakteristikami. Poté by mohl po výběru nejvhodnější funkce snadněji upravit přenosovou funkci podle svých potřeb.
- **Časová dimenze** – výsledná aplikace by měla sloužit hlavně ke zkoumání buněčných procesů v organismech. Ty se projevují nejčastěji tak, že zkoumané buňky v čase „cestují“ organismem nebo se mění jejich intenzita vyzařování, o čemž pojednává část 3.1. Toto lze nejsnadněji zkoumat tak, že je organismus nasnímán a vytvořena volumetrická data v určitých časových intervalech. Proto by mělo být umožněno po nastavení správné přenosové funkce postupné načítání po sobě následujících vzorků objemových dat. Takto lze snadno pozorovat časově proměnlivé procesy.
- **Uložení výsledku do obrázku** – aby bylo s výsledky možno pracovat i po dokončení vizualizace, je vhodné uložit zobrazený objem do obrázku. Poté lze tyto obrázky různě spojovat, popř. vytvářet animace zobrazující již zmíněné časově závislé procesy. Tento požadavek by měl být snadno implemetovatelný a v další verzi se jistě objeví.

V příloze A lze nalézt některé informativní obrázky, které nebyly umístěny do hlavního textu, demonstrační ukázky výsledků jsou umístěny v příloze B a v příloze C se dále nachází stručný uživatelský manuál k přiložené aplikaci. Vzhledem k tomu, že záměrem práce bylo vytvoření knihovny obalující renderer volumetrických dat, která by potom mohla být snadno použita v aplikaci FluorCam, je přiložená aplikace spíše experimentálního rázu sloužící k testovacím účelům. Nicméně je funkční a vykazuje dobré výsledky. Naimplementovaná práce je již připravena k zakomponování a testovacímu provozu s aplikací FluorCam. Po tomto testování bude rozhodnuto, jestli se práce dále bude soustředit spíše na vytvoření samostatné knihovny, popř. komponenty nebo na vytvoření celé aplikaci, která bude z nástroje FluorCam volána externě.

Výsledná knihovna, potažmo celá demonstrační aplikace, je spustitelná na operačních systémech Microsoft Windows, testována byla na Microsoft Windows XP Professional SP2. Jako implementační prostředí je použit Borland C++ Builder. V budoucích verzích ale bude pravděpodobně přepsána do takové podoby, aby nemusela záviset na žádné vývojové platformě a byla tak použitelná i v aplikacích napsaných mimo C++ Builder a nevyužívajících jeho komponent a knihoven. Port na operační systém Linux prozatím není v plánu, protože cílová aplikace FluorCam je tvořena pouze pro prostředí Microsoft Windows. Ani ten by ovšem s největší pravděpodobností neměl být obtížný, pouze by bylo třeba se vypořádat s tvorbou a obsluhou kontextu cílové OpenGL oblasti, což je v současnosti téměř jediná platformově závislá část kódu.

6 Závěr

Renderer byl primárně vyvíjen a testován na grafické kartě řady NVIDIA GeForce 6600. Dále byl testován na grafických kartách NVIDIA řady 8xxx, a NVIDIA GeForce 6100 Go. Na nich je zaručena plná funkčnost. Bohužel kvůli nedostatku hardwaru nemohla být tato práce otestována na jiných grafických adaptérech, není tedy plně zaručeno její korektní chování, obzvláště na kartách jiných výrobců. Vzhledem k tomu, že využívá pouze schválených a standardizovaných OpenGL rozšíření a univerzální jazyk Cg, nemělo by pravděpodobně činit problém ani spuštění na grafickém adaptéru jiného výrobce. Ten ovšem samozřejmě musí obsahovat podporu pro potřebné verze vertex a fragment profilů a použitá OpenGL rozšíření.

Práce byla též prezentována na studentské konferenci počítačové grafiky Central European Seminar on Computer Graphics (CESCG) 2007 a ve školním kole studentské soutěže a konference EEICT 2007. Zde se umístila na třetím místě v kategorii Kybernetika, grafika a multimédia.

Literatura

[1] Engel, K., Hadwiger, M., Kniss, J. M., aj.: *High-Quality Volume Graphics on Comsumer PC Hardware. SigGraph Course Notes 42*, 2002, 122 s.

[2] Engel, K., Kraus, M., Ertl, T.: *High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. Visualization and Interactive Systems Group, University of Stuttgart, Germany*, 2001, 9 s.

[3] Enten, J., Yee, B.: *Green Fluorescent Protein (GFP). Beckman Coulter, Inc., Miami, FL*, 2005.

[4] Grimm, S.: *Real-Time Mono- and Multi-Volume Rendering of Large Medical Datasets on Standard PC Hardware*. Dizertační práce, Technischen Universität Wien, Fakultät für Informatik, 2005, 140 s. Vedoucí dizertační práce Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller.

[5] Hamashima, Y.: *The Use of the Olympus Fluorescence Microscope*. Olympus Optical Company, Tokyo, Japan, 1982, 56 s.

[6] Hart, E.: *3D Textures and Pixel Shaders. ATI Research*, 2004, 13 s.

[7] Ma, K.-L.: *An Efficient Pre-Integrated Volume Rendering Algorithm. Department of Computer Science, University of California at Davis*, 2006, 52 s.

[8] Maršálek, L.: *Osvětlení na GPU*. 2005, 90 s. Vedoucí diplomové práce RNDr. Josef Pelikán.

[9] NVIDIA: *Cg Reference Manual*. 2005, 492 s., dostupné z WWW: `http://developer.nvidia.com/Cg`.

[10] NVIDIA: *Cg Toolkit User's Manual, A Developer's Guide to Programmable Graphics*. 2005, 356 s., dostupné z WWW: `http://developer.nvidia.com/Cg`.

[11] O'Carroll, C.: *Green Fluorescent Protein. B/MB senior seminar*, 2006, 21 s.

[12] Pelikán, J.: *Zobrazování objemových dat. KSVI MFF UK Praha*, 1996-2001, skripta k přednášce Počítačová grafika III, dostupné z WWW: `http://cgg.ms.mff.cuni.cz/~pepca`.

[13] Potts, S., Möller, T.: *Transfer Functions on a Logarithmic Scale for Volume Rendering. School of Computing Science, Simon Fraser University*, 2004, 7 s.

[14] Rezk-Salama, C., Kolb, A.: *A Vertex Program for Efficient Box-Plane Intersection. Computer Graphics and Multimedia Systems Group University of Siegen, Germany*, 2005, 9 s.

[15] Roettger, S., Ertl, T.: *A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. Visualization and Interactive Systems Group, University of Stuttgart, Germany*, 2002, 6 s.

[16] WWW: *Green Fluorescent Protein - GFP.* `http://www.conncoll.edu/ccacad/zimmer/GFP-ww/GFP-1.htm`.

[17] WWW: *The Molecular Structure of Green Fluorescent Protein.* `http://www-bioc.rice.edu/Bioch/Phillips/Papers/gfpbio.html`.

[18] WWW: *Caenorhabditis elegans.* `http://users.rcn.com/jkimball.ma.ultranet/BiologyPages/C/Caen.elegans.html`, 2003.

[19] WWW: *Caenorhabditis elegans.* `http://en.wikipedia.org/wiki/Caenorhabditis_elegans`, 2006.

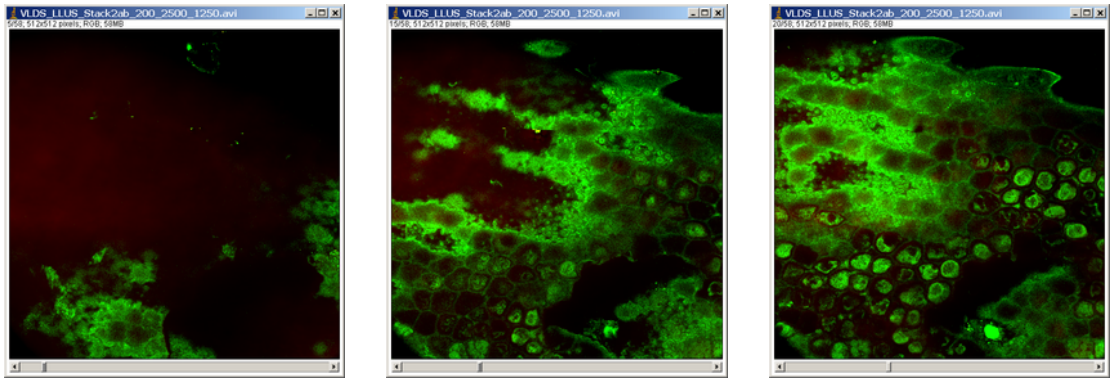
[20] WWW: *Green Fluorescent Protein - GFP.* `http://en.wikipedia.org/wiki/Green_fluorescent_protein`, 2007.

[21] Zhang, Q., Eagleson, R., Peters, T. M.: *GPU-Based Real-Time Beating Heart Volume Rendering Using Dynamic 3D Texture Binding. First Canadian Student Conference on Biomedical Computing Paper*, 2006, 6 s.

Všechny zde uvedené URL zdrojů byly na příslušných internetových adresách dostupné v květnu 2007.

Dodatek A

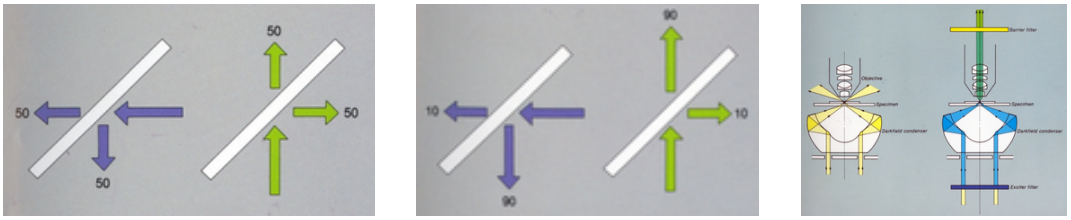
Obrázky



(a) Řez 5: ref-0μm (b) Řez 15: ref-5μm (c) Řez 20: ref-7.5μm

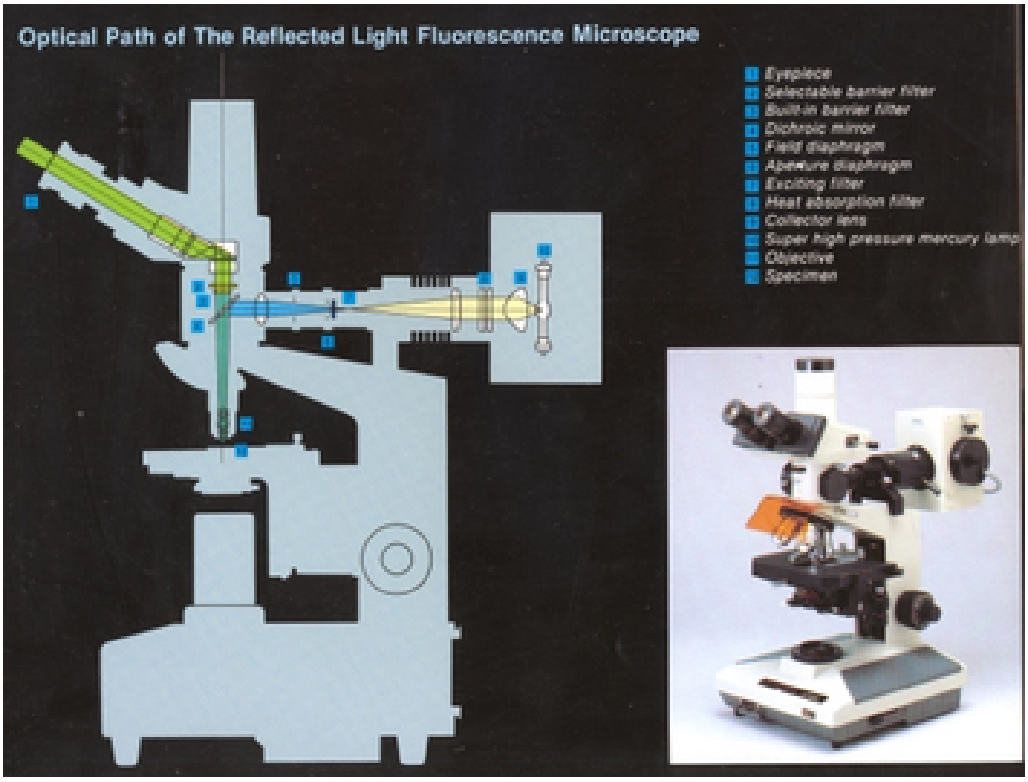
(d) Řez 29: ref-12μm (e) Řez 33: ref-14μm (f) Řez 41: ref-18μm

Obr. A.1: Ukázka řezů vstupních dat. Data jsou stále intenzitní, jen je na nich namapována zelená barva pro lepší vzhled.

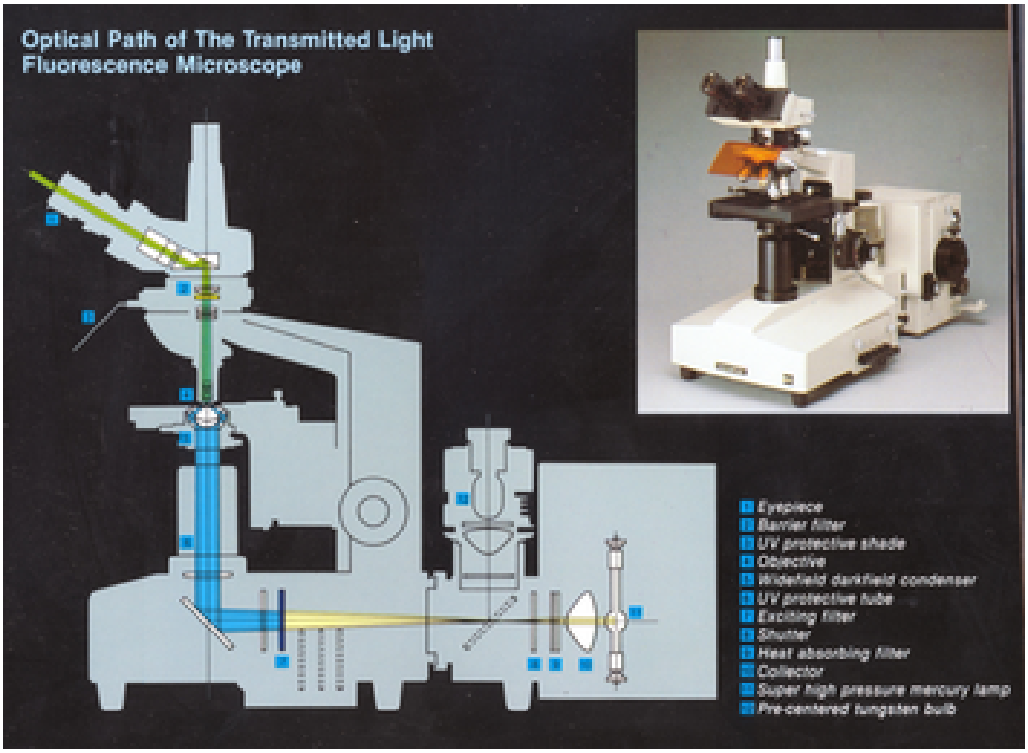


(a) Polopropustné zrcadlo (b) Dichroické zrcadlo (c) Zástinový kondenzor

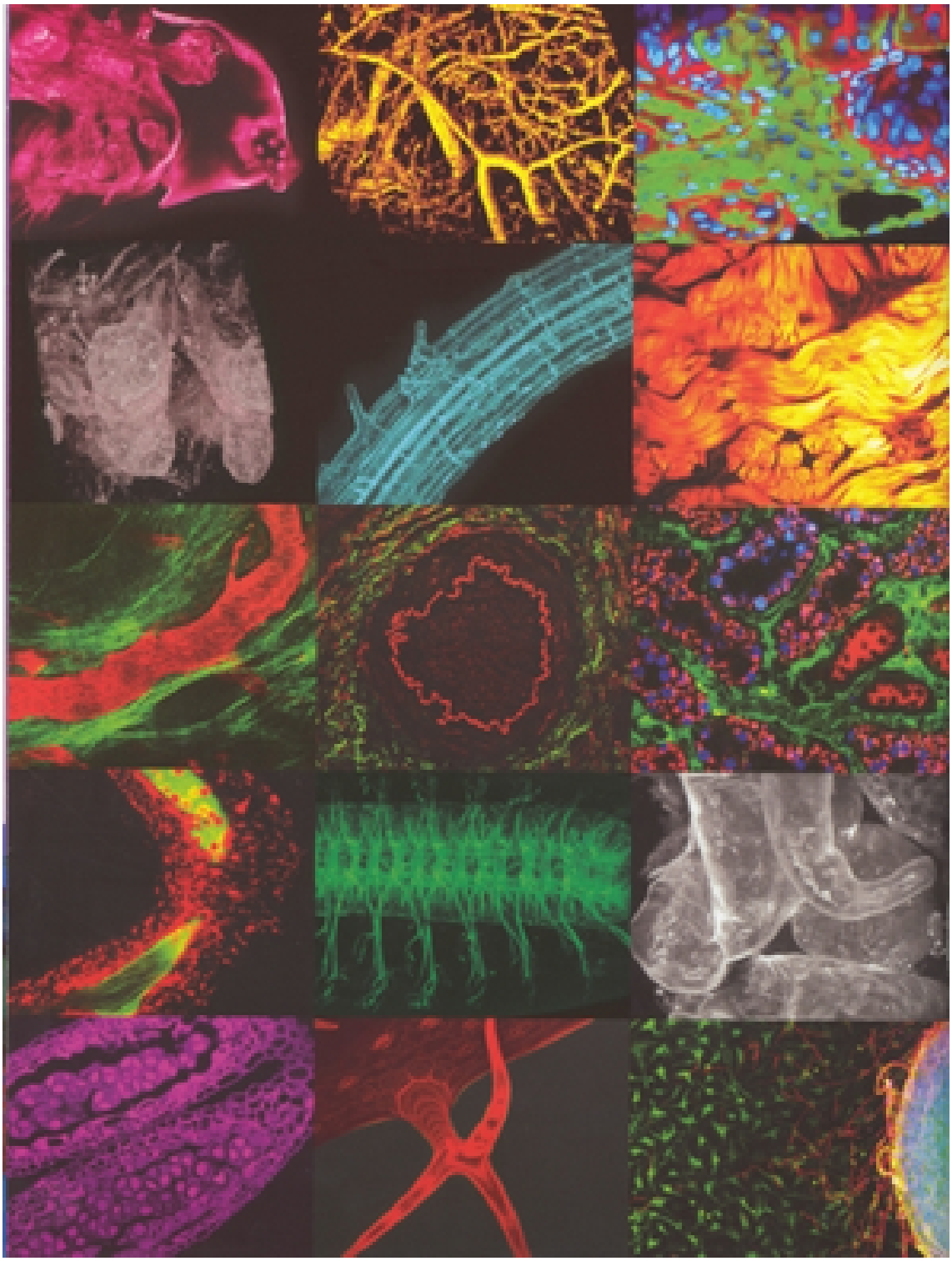
Obr. A.2: Srovnání polopropustného a dichroického zrcadla, zástinový kondenzor.



Obr. A.3: Schéma epifluorescenčního mikroskopu



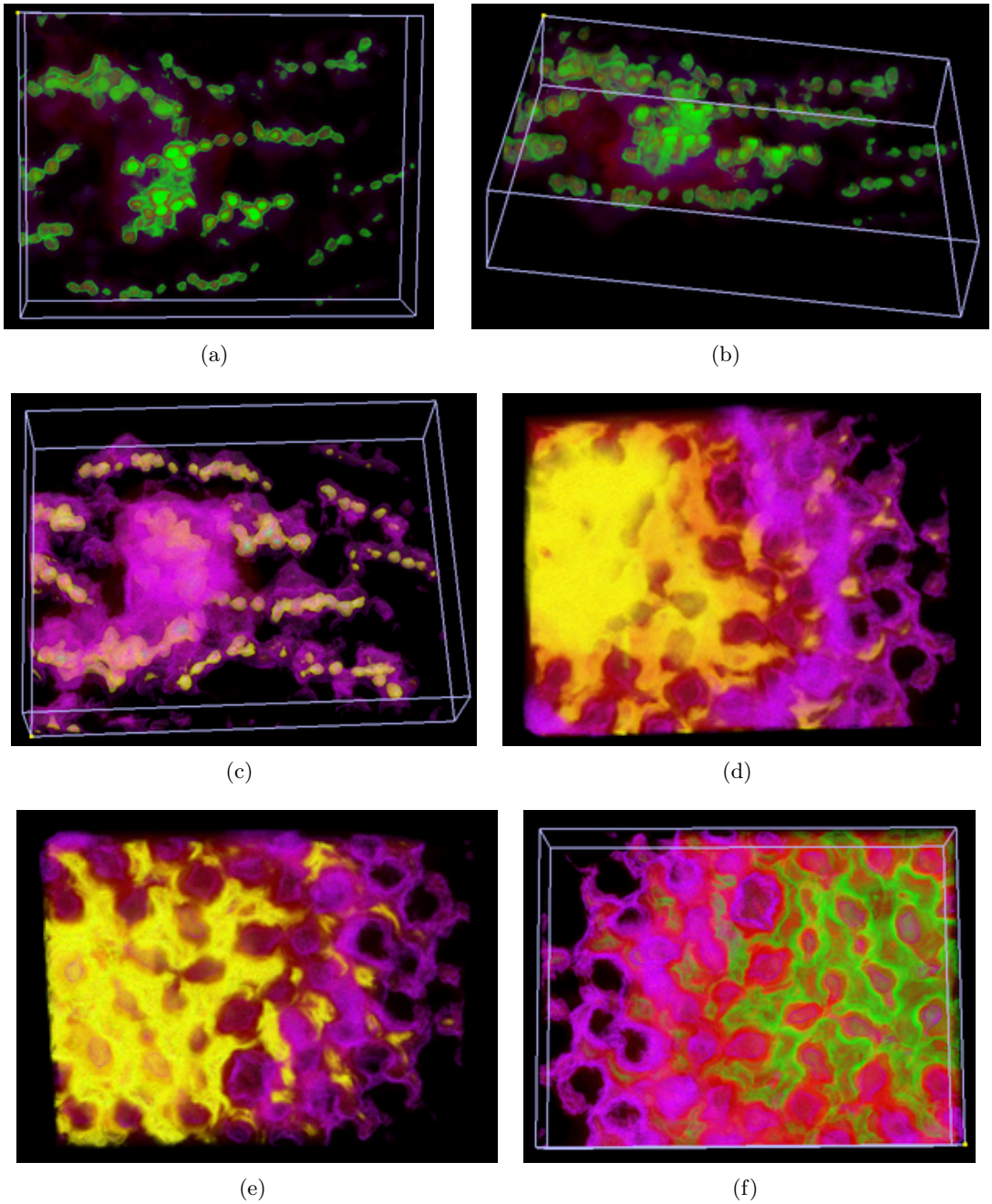
Obr. A.4: Schéma transmisního fluorescenčního mikroskopu



Obr. A.5: Ukázky obrazu z konfokálního mikroskopu

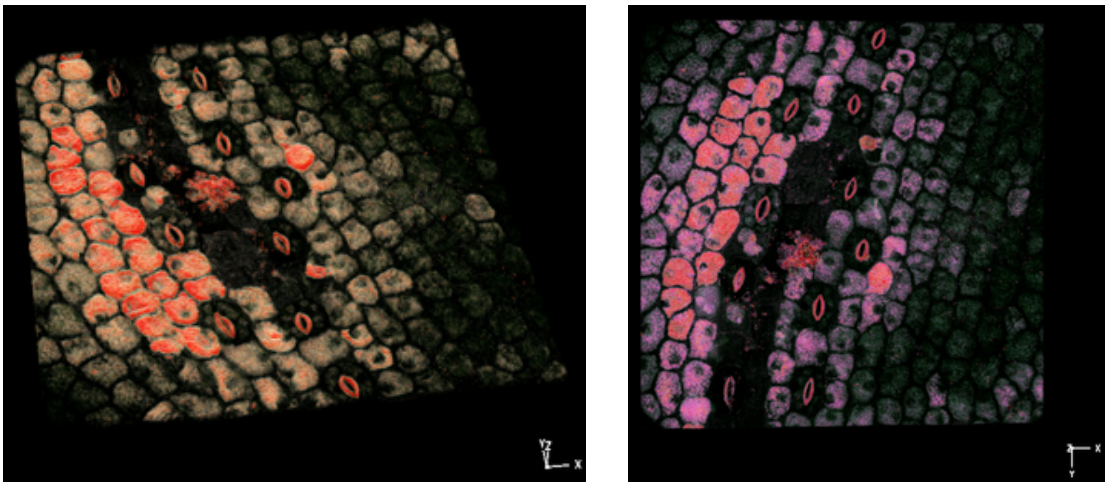
Dodatek B

Demonstrační ukázky výsledků

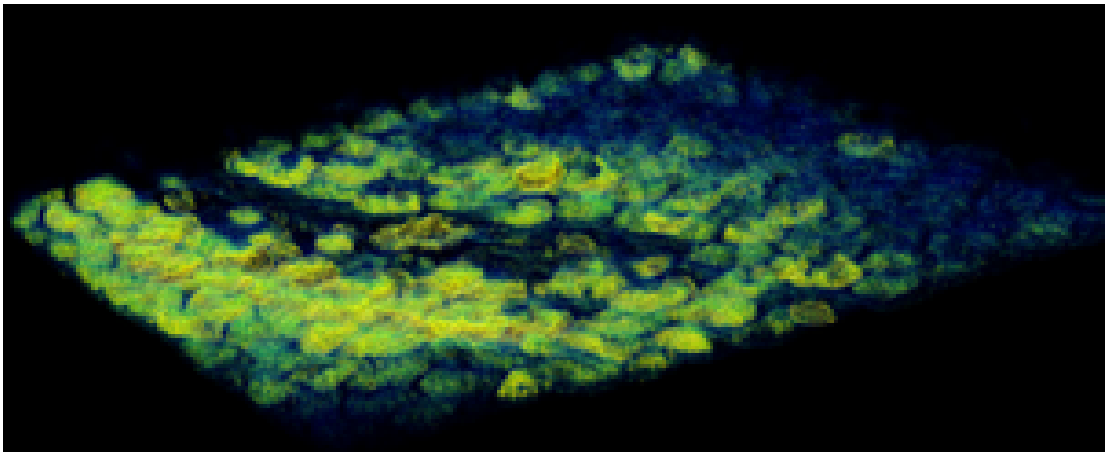


(a) (b) (c) (d) (e) (f)

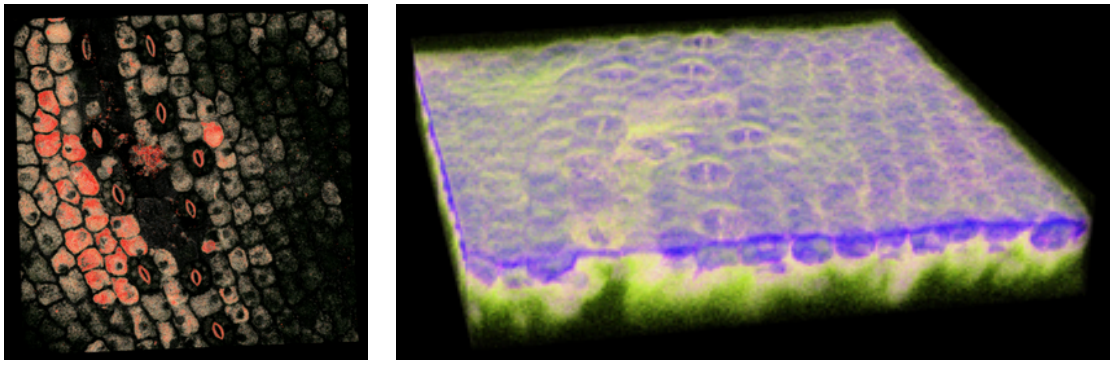
Obr. B.1: Rostlinné buňky značené chlorofylem



(a) (b)



(c)

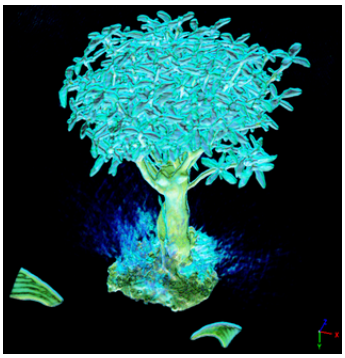


(d) (e)

Obr. B.2: Rostlinné buňky značené chlorofylem

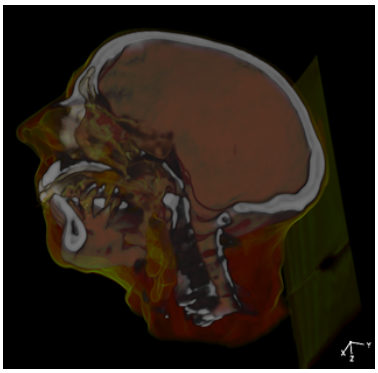
(a) Bonsaj. 256x256x128x8b.

(b) Bonsaj. 256x256x128x8b.

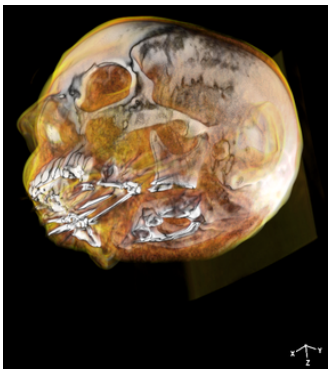


(c) Bonsaj. 256x256x128x8b. Gradient aktivní.

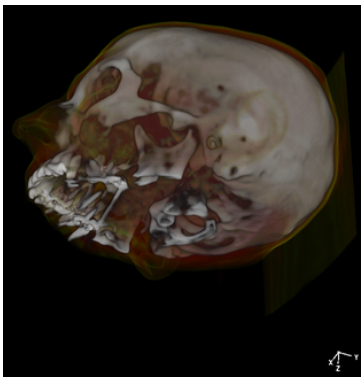
(d) CT lidské hlavy. 256x256x225x8b.



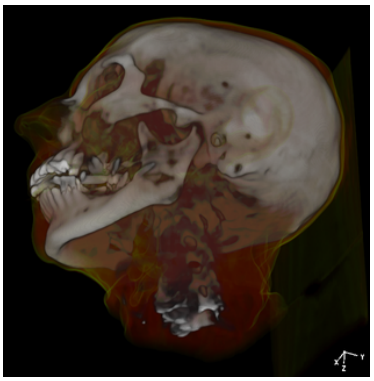
(e) CT lidské hlavy. 256x256x225x8b. Ořezáno podle osy X.



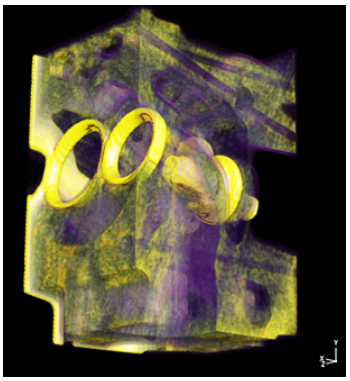
(f) CT lidské hlavy. 256x256x225x8b. Ořezáno podle osy Z. Gradient aktivní.



(g) CT lidské hlavy. 256x256x225x8b. Ořezáno podle osy Z.



(h) CT lidské hlavy. 256x256x225x8b.



(i) Motor. 256x256x225x8b.

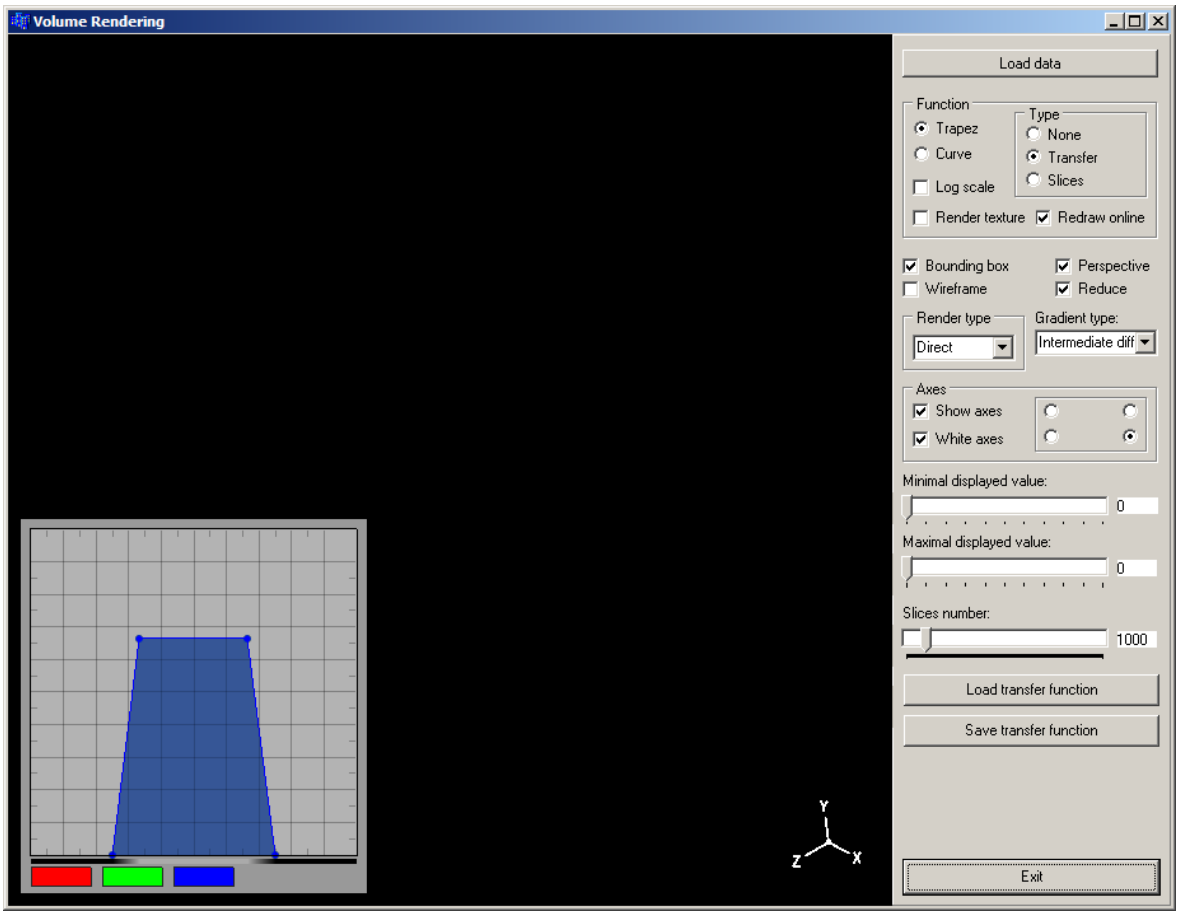
Obr. B.3: Ukázky snímků dalších typů dat. Jedná se o `raw` data s popisovým `dat` souborem.

Dodatek C

Uživatelská příručka

Protože je cílem práce vytvoření spíše renderovací knihovny než kompletní aplikace, je aplikace dodávaná s touto prací hlavně experimentálního rázu a sloužila převážně k průběžnému testování rendereru. Není tedy vyloučeno, že obsahuje některé chyby. Během testování ale byla většina těch závažných opravena, proto je aplikace pro testování dostatečně funkční a použitelná. Renderer má všechny své parametry plně nastavitelné, ale kvůli jednoduchosti aplikace nejsou všechna tato nastavení zahrnuta. Obsahuje jen to nejnutnější k úspěšné specifikaci vizualizační funkce.

Aplikace po spuštění vypadá zhruba jako na obrázku C.1. Prvním krokem by tedy mělo být načtení dat. To lze provést pomocí tlačítka `Load data` nacházejícího se v pravém panelu nahoře. Po nalezení požadovaného souboru a jeho potvrzení se provedou veškeré potřebné akce a obsah dat je vyrenderován do černé oblasti vlevo. Pokud nebyla předem specifikována přenosová a příspěvková funkce, použijí se implicitně přednastavené hodnoty.



Obr. C.1: Vzhled aplikace krátce po spuštění. Nejsou načtena žádná data.

Nyní je již aplikace připravena k nastavení správné přenosové funkce a ostatních parametrů rendereru. Následuje popis jednotlivých ovládacích prvků tak, jak jdou v pravém panelu za sebou.

Tlačítko `Load data` slouží k výběru bloku dat, který má být vizualizován. V oddíle `Function` se nacházejí následující ovládací prvky:

- `Type` – výběr typu funkce, se kterou se momentálně bude pracovat. Na výběr je přenosová funkce `Transfer`, příspěvková funkce pro jednotlivé řezy `Slices` nebo žádná funkce aktivní `None`. Po aktivaci některé z funkcí se zobrazí její editor v levém dolním rohu aplikace, jak je zobrazeno na obrázku C.1, kde je aktivována přenosová funkce.
- `Trapez` – právě aktivní funkce bude pracovat v polygonálním režimu.
- `Curve` – právě aktivní funkce bude pracovat v křivkovém režimu.
- `Log scale` – přepnutí lineární/logaritmické stupnice pro aktuálně zvolenou funkci. Změna se ihned projeví v editoru funkce.
- `Render texture` – přepnutí zobrazení funkční textury. Pokud je aktivní, jsou napravo od editoru funkce zobrazeny všechny textury, které jsou od aktuální funkce odesílány do fragment shaderu. To je ukázáno na obrázku C.2.
- `Redraw online` – je-li zaškrtnuto, veškeré změny provedé v editoru jsou ihned aplikovány a promítnuty do výsledného obrazu. Pokud aktivní není, provedené změny jsou aplikovány až po dokončení editace. Toto může být vhodné zejména při použití pre-integrace a 2D přenosových textur, kdy trvá neakcelerovaný výpočet dlouhou dobu. Zatím tedy nemá tento přepínač příliš velké využití, protože pre-integraci v současné fázi nelze použít.

Přepínače nacházející se pod oddílem `Function` modifikují výsledné zobrazení. `Bounding box` mění viditelnost obálky objemu. Po aktivaci `Wireframe` se spustí fragment program Wire a jsou tak zobrazeny jednotlivé řezy. Pomocí `Perspective` lze přepínat mezi ortogonálním a perspektivním promítáním a přepínací tlačítko `Reduce` pomáhá při interaktivním pohybu s objemem. Pokud je aktivováno a s objemem je nějakým způsobem manipulováno, zredukuje se počet generovaných řezů podle redukčního koeficientu. Ten je přednastaven na 2.0 a je plně nastavitelný. Bohužel aplikace momentálně neobsahuje ovládací prvek, kterým by toho mohlo být dosaženo.

Ve výběrovém poli `Render type` lze zvolit použitý fragment shader. Momentálně dostupné hodnoty jsou `Direct`, aktivující fragment shader Basic, dále lze zvolit `Shading` aktivující program Shading a `Gradient`, kdy je použit fragment program Gradient. Pro jeho nastavení slouží další výběrový prvek `Gradient type`, kde lze zvolit metodu pro výpočet gradientu. Nyní jsou dostupné možnosti `Intermediate difference` a `Central difference`, aktivující stejnojmenné metody pro výpočet gradientu.

Další oddíl `Axes` umožňuje nastavení osového kříže zobrazeného v zobrazovací oblasti. Pomocí `Show axes` se přepíná jeho zobrazení, `White axes` nastavuje jeho barvu. Pokud je zaškrtnuto, jsou všechny osy zobrazeny bílou barvou. To však při nevhodném natočení ve směch směrech může být nepřehledné. Proto pokud je tento přepínač neaktivní, je zobrazena každá osa jinou barvou. Osa X červeně, Y zeleně a Z modře. Čtyřmi přepínači lze nastavit pozici, na jaké se osový kříž zobrazí, pokud nevyhovuje výchozí vpravo dole.

Dva posuvníky označené jako `Minimal displayed value` a `Maximal displayed value` určují rozsah hodnot zobrazených v editoru přenosové funkce. Pokud jsou načtena data o rozsahu 8 bitů, nelze tento rozsah modifikovat. Ten musí být totiž vždy minimálně 255 hodnot. Jsou-li načtena 16 bitová data, je minimální a maximální hodnota těchto posuvníků

nastavena podle hodnot uložených v datech. Poté lze pomocí posuvníků určit rozsah pro přesnější specifikaci přenosové funkce.

Následující posuvník `Slices number` ovlivňuje počet generovaných řezů. Pokud je tento počet příliš velký, renderování je pomalé a objevuje se přepalování některých oblastí, kde se skládá velké množství optických hodnot. Je-li naopak počet řezů příliš malý, výsledný obraz je nekvalitní a chudý na drobné detaily. Pomocí tohoto posuvníku tak lze nalézt optimální počet řezů pro aktuálně zobrazovaná data.

Tlačítka `Load transfer function` a `Save transfer function` slouží k nahrání nebo uložení přenosové funkce do souboru. Při nahrávání je automaticky rozpoznán uložený typ funkce. V případě polygonální funkce jsou nahrány pouze řídící body ležící v aktuálním datovém rozsahu. V případě křivkové funkce jsou změněny hodnoty křivek tak, že pokud je rozsah dat větší než uložený rozsah, změní se jen příslušná část. Je-li uložený rozsah křivek větší než maximální datová hodnota, změní se celá křivka a zbytek uložených hodnot bude přeskočen. Ukládání je aktivní vždy pro právě zvolený typ přenosové funkce. Pro uložení obou typů je tedy nutné ho před každým uložením změnit a ukládat celkem dvakrát. Tlačítkem `Exit` je aplikace ukončena.

C.1 Ovládání renderovací oblasti

C.1

V oblasti rendereru, která je vykreslena černou barvou, lze pomocí myši ovládat polohu a velikost renderovaných dat. Je-li stisknuto a taženo levým tlačítkem myši, vykreslený objekt rotuje kolem svého středu. Rotace je mapována na kouli, probíhá tedy tím stylem, jako by byla uchopena koule v daném bodě a bylo s tímto bodem rotováno.

Pomocí pravého tlačítka myši je provedeno přibližování a oddalování objektu. Je-li taženo se stisknutým pravým tlačítkem nahoru, objekt je oddalován, dolů naopak přibližován.

Prostředním tlačítkem lze určit polohu objektu. Je-li stisknuto prostřední tlačítko a s myší pohybováno, objekt mění svou polohu v rámci vykreslovací oblasti.

C.2 Specifikace přenosové funkce

C.2

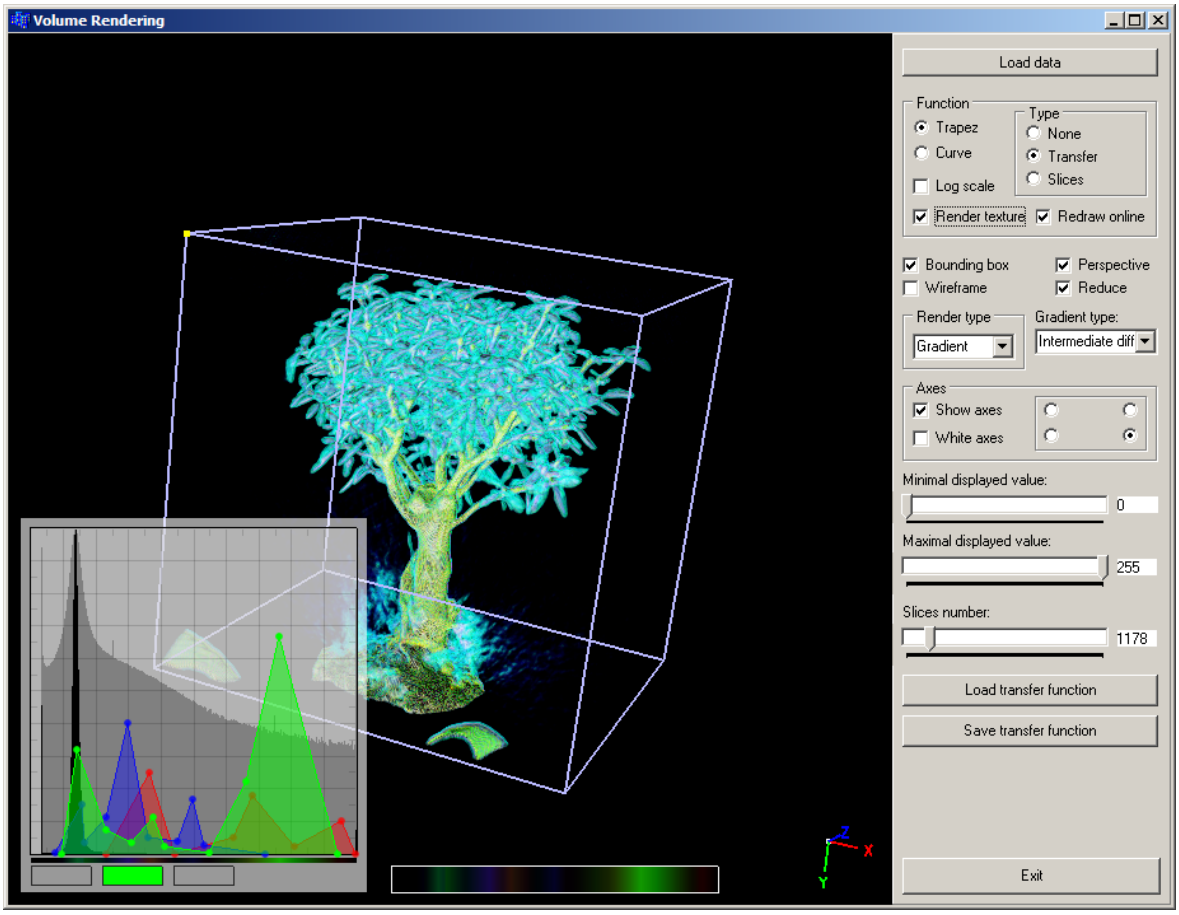
Pokud je aktivní přenosová funkce a je určený požadovaný rozsah zobrazených hodnot, je možné začít s procesem specifikace přenosové funkce. Pro orientačním nalezení všech podstatných částí dat lze nejlépe vyzkoušet pár náhodných nastavení.

Ovládání editoru přenosové (a také příspěvkové) funkce je po prvotním seznámení jednoduché. Ve spodní části se nachází tlačítka pro jednotlivé modfikovatelné kanály. Po stisknutí některého z tlačítek příslušný kanál přepne stav z aktivního na neaktivní a naopak. Aktivní kanály jsou vykresleny intenzivnější barvou a pouze s nimi lze operovat. Poté lze v prostoru editoru stiskem a tažením levého tlačítka myši specifikovat vlastní hodnoty. U polygonální funkce je možno uchopit a přemístit některý z uzlových bodů, hranu, popř. celý polygon. U křivek je automaticky nastavena hodnota na pozici kurzoru. U polygonální funkce lze též stiskem pravého tlačítka myši přidat nový uzlový bod na požadovanou pozici. Pokud je zároveň stisknuta i klávesa `Ctrl`, je z této pozice uzlový bod odebrán, nachází-li se zde nějaký.

Pokud je nalezeno použitelné řešení, lze ho dále přesněji specifikovat pomocí přesnějšího určení rozsahu nastavovaných dat nebo v rámci logaritmického měřítka. Výslednou texturu lze za běhu kontrolovat, pokud je aktivní přepínač `Render texture`.

Protože nyní bylo cílem vytvořit pouze holý renderer, aby poskytoval potřebné výsledky, nelze zatím uložit výsledný obrázek jiným způsobem než s využitím systémové schránky. V další verzi rendereru bude naimplementována možnost uložení výstupu do některého z grafických formátů. Jako nejvhodnější se jeví formát `bmp`, díky jeho jednoduché implementaci a dobré dokumentaci.

Aplikace se specifikovanou přenosovou funkcí může vypadat např. tak, jak je vidět na obrázku C.2.



Obr. C.2: Vzhled aplikace během procesu specifikace přenosové funkce.