VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

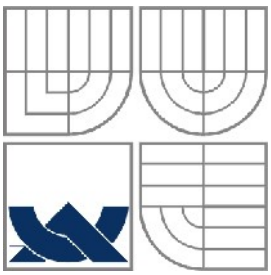
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VIZUALIZACE OBJEMOVÝCH DAT POMOCÍ VOLUME RENDERINGU

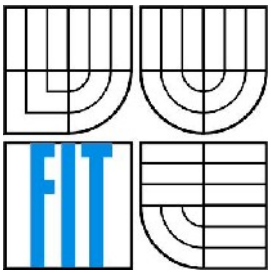
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JIŘÍ KAZÍK
AUTHOR

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 OBJEMOVÝCH DAT POMOCÍ VOLUME RENDERINGU

3D VOLUME RENDERING DATA VISUALIZATION

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ KAZÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing., Ph.D. PŘEMYSL KRŠEK

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Kazík Jiří**
Obor: Informační technologie
Téma: **Vizualizace objemových dat pomocí volume renderingu**
Kategorie: Počítačová grafika

Pokyny:

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

Literatura:

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

Při obhajobě semestrální části projektu je požadováno:

- Splňte první tři body zadání.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% 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: **Kršek Přemysl, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

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

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Jiří Kazík**
Id studenta: 86789
Bytem: Nová Lhota 282, 696 74 Nová Lhota
Narozen: 15. 11. 1982, Kyjov
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Vizualizace objemových dat pomocí volume renderingu
Vedoucí/školitel VŠKP: Kršek Přemysl, Ing., Ph.D.
Ústav: Ústav počítačové grafiky a multimédií
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

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 zveřejněním 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.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	.....
Nabyvatel	Autor

Abstrakt

V teoretické části se práce obecně zaměřuje na problematiku zobrazování objemových dat, vzájemně srovnává a hodnotí jednotlivé přístupy a čtenáři tak poskytuje dobrou základní orientaci v tematice. Podrobně je pak analyzována metoda vizualizace objemových dat pomocí mapování textur, která je rovněž využita pro implementaci grafického systému navrženého v této práci. Systém je tvořen s ohledem na maximální přenositelnost a k jeho realizaci jsou využity jazyk C++ a grafický toolkit Open Scene Graph, jehož popis je v odpovídajícím rozsahu také zahrnut.

Klíčová slova

Zobrazování objemových dat, mapování textur, vrhání paprsku, Voxel Splatting, Shear Warp, Open Scene Graph, OpenGL, GLSL, Pixel Shader, Vertex Shader, Fragment Shader, C++

Abstract

Theoretical part of this project is focused on rendering of volumetric data. It compares and appraise individual methods and thus readers get a good basic knowledge of commonnest causes of problems. Texture Mapped Volume Rendering is described in detail, because it is used in implementation of graphic system designed in this thesis. Whole system is created with maximal attention to platform independence. C++ language and Open Scene Graph toolkit are used for realization, so a brief description of OSG toolkit is also present in this work.

Keywords

Volume Rendering, Texture Mapping, Ray Casting, Voxel Splatting, Shear Warp, Open Scene Graph, OpenGL, GLSL, Pixel Shader, Vertex Shader, Fragment Shader, C++

Citace

Jiří Kazík: Vizualizace objemových dat pomocí volume renderingu, bakalářská práce, Brno, FIT VUT v Brně, 2007

Vizualizace objemových dat pomocí volume renderingu

Prohlášení

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

.....
Jméno Příjmení
Datum

Obsah

1 Úvod

Vizualizace objemových dat se v dnešní době stále častěji dostává do popředí zájmu v mnoha různých oblastech lidské činnosti. Denně pomáhá např. v mikrobiologii, medicíně nebo fyzice, kde umožňuje názorné zkoumání jevů, které běžně zůstávají lidskému oku skryty.

Technicky vzato, volume rendering je nefotorealistickou vizualizační technikou používanou pro zobrazení dvourozměrné projekce trojrozměrných, diskrétních dat, zpravidla získaných pomocí CT scanu (Computed Tomography), případně MRI (Magnetic Resonance Imaging). Většinou se jedná o sadu několika 2D řezů skládaných na sebe s konstantními rozestupy a ve výsledku tedy formujících tzv. objemovou kostku.

Masivní nárůst výpočetního výkonu v dnešní době umožňuje zpracování prostorových dat v reálném čase a zejména použití v oblasti medicíny tak nabývá vysoké užitné hodnoty. Interaktivní práce nabízí u jiných metod nevídanou efektivitu analýzy získaných dat a nezanedbatelným faktorem rázně mluvícím ve prospěch volume renderingu je také ekonomické hledisko – není již nutné pořizovat drahé snímky, které jsou často potřebné jen po omezený čas a v neposlední řadě je umožněna také digitální archivaci získaných dat.

Volume rendering pomocí různých způsobů zobrazení umožní optimální pohled na zkoumaná data. Je možné dynamicky skrýt oblasti, které nejsou předmětem zkoumání a naopak zvýraznit oblasti při pozorování podstatné. Samožřejmou součástí pochopitelně jsou neomezené možnosti rotace objemových dat a případně tvorba vhodných řezů.

Dnes existuje více různých přístupů k technické realizaci volume renderingu. Každý z nich má svá specifika a nelze jednoznačně určit jediný, univerzálně nejvýhodnější přístup. Nejčastěji používané metody jsou zkoumány v úvodu druhé kapitoly, v teoretickém rozboru. Dále v této práci je do detailu zkoumána pouze metoda vizualizace pomocí mapování textur (texture mapped volume rendering), která v posledních letech zažívá velký rozmach způsobený zejména dobrou dostupností výkonných grafických akcelerátorů parametricky vhodných pro hardwarovou implementaci volume renderingu.

Návrh vizualizačního systému je zachycen ve stejnojmenné kapitole a detaily samotné implementace jsou zkoumány v kapitole čtvrté. Za realizační prostředky byly zvoleny programovací jazyk C++ a toolkit pro 3D grafiku Open Scene Graph (OSG) zastřešující rozšířené rozhraní OpenGL. Celý systém je tvořen s ohledem na multiplaformní použití (OSG běží na všech Windows platformách, dále pod OSX, GNU/Linux, IRIX, Solaris, HP-Ux, AIX a FreeBSD).

Kapitola pátá - výsledky - popisuje aktuální stav navrženého systému, rozebírá implementované funkce, různá omezení a v konečné fázi rozvíjí také budoucí možnosti projektu.

Závěr pak již nabízí pouze zhodnocení přínosu této práce, zkoumá získané zkušenosti a shrnuje zkoumanou problematiku na úrovni tohoto projektu.

2 Teoretický rozbor

V teoretickém rozboru bude čtenář seznámen s problematikou objemového zobrazování dat na takové úrovni, aby byl v dalších částech schopen porozumět různým aspektům v návrhu systému a jeho implementaci. Pro zachování celkového přehledu v rozsáhlé problematice objemového zobrazování dat je nastíněno více možných metod vedoucích k výsledku požadovenému od našeho systému. Pouze metoda zvolená pro implementaci (Texture Mapping) je ovšem vysvětlena v rozsahu zahrnujícím co nejvíce detailů.

2.1 Zobrazování povrchu a objemu

V počítačové grafice je možné metody vizualizace rozdělit do dvou základních kategorií:

- surface rendering (zobrazování povrchu)
- volume rendering (zobrazování objemu)

Surface rendering, jak je patrné již z názvu, se zaměřuje výhradně na zobrazování povrchů různých geometrických primitiv, z nichž je případně tvořena celá scéna. Metoda je mimořádně výhodná všude tam, kde pro prezentaci není podstatná vnitřní struktura objektů – metoda nabízí vysokou efektivitu a je velmi dobře hardwarově akcelerovatelná. Právě proto je dnes často využívána např. v počítačových hrách i drtivé většině CAD systémů.

Volume rendering pak vytváří 2D obraz přímo z 3D objemových dat. Použití je předurčeno pro oblasti, kde je povrchová reprezentace nedostatečná pro vypovídající charakter vizualizace.

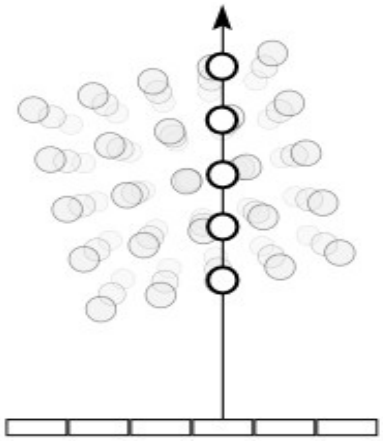
Ačkoliv se jedná o dvě zcela odlišné metody, je vhodné dodat, že nic nebrání implementaci hybridního systému umožňujícího společnou práci s oběma metodami.

2.2 Přímý volume rendering

Jak již bylo řečeno, pro realizaci volume renderingu bylo během času vytvořeno více základních technik. Hlavním hnacím motorem pro vývoj v této oblasti byla především snaha o zvýšení výkonu – a to i za cenu různých ústupků na straně kvality výstupu. Každá metoda tedy má svá pro a proti, nelze jednoznačně určit ideální metodu a v každém případě se jedná o určitý kompromis. Přes tradiční Volume Ray Casting se přes Voxel Splatting a Shear Warp probereme až k Texture Mappingu, tedy ke způsobu vizualizace využitém v implementaci našeho systému a seznámíme se s jejich nejdůležitějšími vlastnostmi.

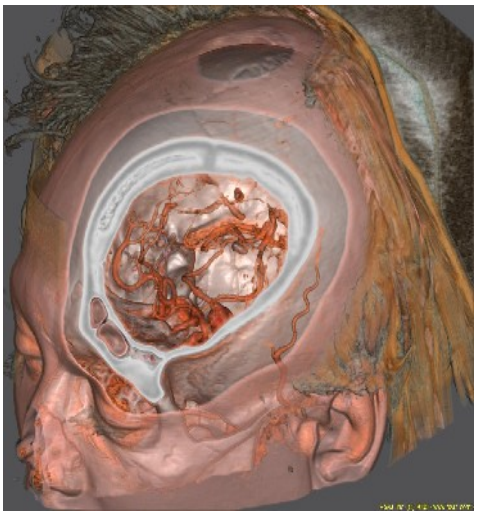
2.2.1 Volume Ray Casting

Nejjednodušším způsobem pro projekci obrazu je vrhání paprsků skrze objem pomocí tzv. Ray Castingu. Pro každý pixel požadovaného výsledného obrazu je vržen paprsek – ze středního bodu projekce skrze obrazovou rovinu (pixel výsledného obrazu) do samotného objemu, který zobrazujeme (viz. obrázek vpravo). Paprsek je pro vyšší efektivitu ořezán ohraničením vizualizovaného objemu a v pravidelném intervalu jsou brány vzorky z protínaného objemu. Na každý vzorek je aplikována přenosová funkce, díky níž získáme z dat hustoty vzorek RGBA (obsahující informaci o barvě). RGBA vzorek je pak přičítán k RGBA součtu daného paprsku a proces je opakován tak dlouho, dokud paprsek neopustí zobrazovaný objem. Pak můžeme RGBA součet převést na výstupní RGB barvu a pokračovat s dalším obrazovým bodem (pixelem), dokud nezpracujeme celý obraz.



Ilustrace 1: Ray Casting

Ilustrace 2: Plíce v přímém pohledu [6]



Ilustrace 3: Řez hlavou [6]

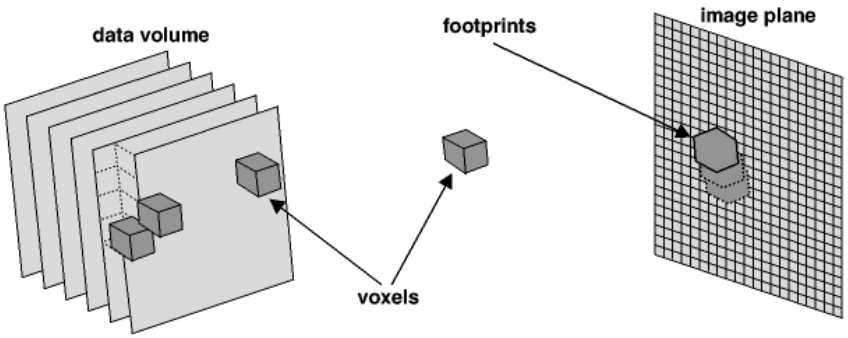
Voxely jsou typicky skládány podle vztahu:

$$I = \sum_{k=1}^{n} Col_k \alpha_k \prod_{i=0}^{k-1} (1 - \alpha_i)$$

Kde I je výsledná barva aktuálního pixelu na obrazovce, Col_k je barva zpracovávaného voxelu získaná z přenosové funkce na základě hustoty voxelu a α_k je průhlednost stejného voxelu. Násobení průhledností pak není nutné provádět pro každý voxel samostatně, stačí zachovávat výsledek posledního součinu a násobit jej průhledností aktuálního voxelu.

2.2.2 Voxel Splatting

Voxel Splatting je typickým zástupcem metod kladoucích důraz na vyšší rychlost za cenu nižší kvality zobrazení. Jednotky objemových dat (voxely) jsou na obrazovou rovinu „naplácávány“ (splatting je možné přeložit právě jako „plácání, naplácávání“) od zadní části objemu směrem dopředu a jsou reprezentovány pomocí stop (footprints) s různými parametry (velikost, tvar, barva...).

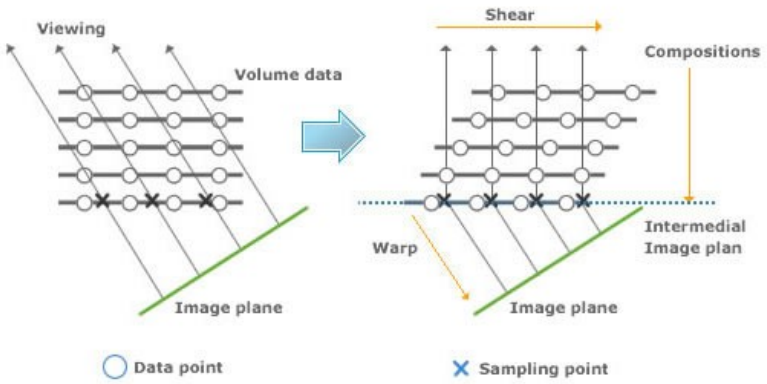


Ilustrace 4: Proces voxel splattingu

2.2.3 Shear Warp

Nový přístup k Volume Renderingu přinesli Philippe Lacroute a Marc Levoy. Techniku publikovali v dokumentu „Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation“. Pohledová transformace je upravena tak, aby nejbližší plocha objemu byla osově zarovnaná s mimo-obrazovým bufferem (Intermedial Image Plan) s konstantním měřítkem voxelů ku pixelům. Objem je záhy renderován do tohoto bufferu a využívá výhodnějších paměťových přístupů (vhodný memory-alignment) a konstantní scaling a blending faktory. Jakmile jsou všechny pláty objemu vyrenderovány, buffer je upraven do požadované orientace při současné změně měřítka.

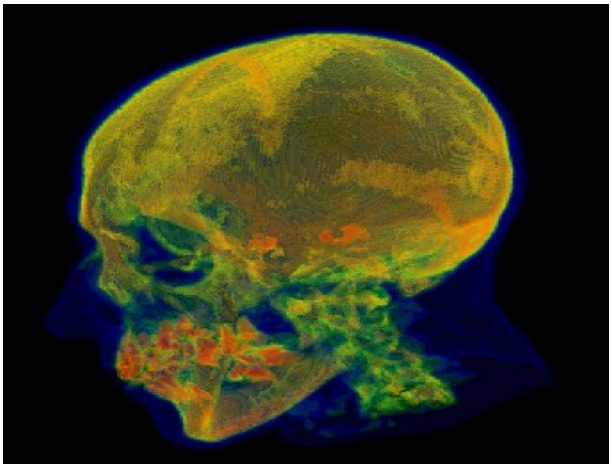
Technika je poměrně rychlá, tentokrát na úkor menší přenosti samplingu a ve výsledku také horší kvality obrazu (ve srovnání s ray castingem).



Ilustrace 5: Proces shear-warpingu

2.2.4 Texture Mapping

Abychom pro objemové zobrazení mohli využít grafický hardware, je potřeba reprezentovat objemová data (Volume Data Set) jako štos (stack) sousedících plátů (slices). Jsou-li hardwarově podporovány 3D textury (OpenGL 1.2), je možné zobrazovat pláty souběžně k obrazové rovině (image plane) s ohledem na současný směr pohledu. Při změně pohledové matice je ovšem potřeba tyto pohledově-orientované (viewport-aligned) pláty přepočítat. Vzhledem k tomu, že je již možná hardwarová trilineární interpolace textur, tato operace je realizovatelná při zachování interaktivity. V závěrečném kombinačním kroku jsou texturované pláty smíchávány odzadu-dopředu na obrazovou rovinu, což vyústí v částečně průhledný pohled na objemová data. U tohoto přístupu snadno zvýšíme obrazovou kvalitu zvýšením počtu plátů. Nicméně, abychom získali ekvivalentní reprezentaci objemových dat při změně počtu plátů, je potřeba hodnoty hustoty přizpůsobit měnící se vzdálenosti plátů. I když správný scaling factor je funkcí hustoty, ve většině případů je vizuálně adekvátní aproximací lineární rozložení hodnot s kontantním faktorem odpovídajícím vzdálenostem plátů.

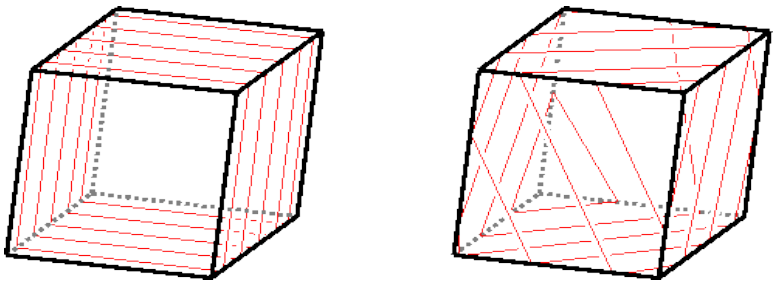


Ilustrace 6: CT scan lidské hlavy, MIP zobrazení (výstup navrženého systému)

Ilustrace 7: Čajová konvice s humrem uvnitř (výstup navrženého systému)

Pokud hardware podporuje pouze 2D textury, pláty jsou objektově zarovnané (object-aligned slices). To umožňuje nahrazení trilineární interpolace za bilineární. Pokud se směr pohledu změní o více než 90 stupňů, orientace normálového vektoru plátů musí být změněna. Je tedy nutné udržovat v paměti tři kopie objemových dat- jednu kopii pro směr každé ze souřadných os (x, y, z). Pláty jsou renderovány jako planární polygony texturované obrazovou informací získanou z 2D textury a přimíchané na obrazovou rovinu. Navzdory vysokým paměťovým nárokům je zásadní nevýhodou této implementace chybějící prostorová interpolace. Výsledkem je, že získané obrazy obsahují silné vizuální artefakty. Pro získání lepších vizuálních výsledků musí být hodnoty hustoty změněny podle vzdálenosti mezi dvěmi sousedícími pláty ve směru pohledového paprsku. Stejně jako

při 3D texturování, i zde vede linární změna hodnot (s konstatním faktorem jako aproximací) k dobrým vizuálním výsledkům.



Ilustrace 8: Objektově zarovnané řezy vlevo, pohledově zarovnané vpravo

2.2.5 Způsoby optimalizace

Vysoké nároky na technické vybavení pochopitelně vedou ke snaze omezit co největší množství nadbytečných operací. Mezi nejčastěji používané způsoby patří:

Empty Space Skipping – objemová data často obsahují oblasti bez viditelných dat, které není nutné zpracovávat. Někdy je tedy vhodné implementovat systém, který tato prázdná místa přeskočí. Metodu je možné s výhodou využít u Voxel Splattingu, naproti tomu u Texture Mappingu ji z principu nelze realizovat.

Early Ray Termination – metoda se používá v případech, kdy jsou data zobrazována ve směru odpředu-dozadu. Jakmile paprsek narazí na dostatečně hustý materiál, změny od následujících voxelů by již nebyly příliš významné a můžeme se rozhodnout výpočet ukončit.

Octree / BSP space subdivision – použití stromových struktur (octree a BSP tree) může být výhodné na jedné straně pro kompresi objemových dat, ale také pro významné zrychlení ray castingu.

Volume Segmentation – segmentace dat spočívá v odstranění části objemových dat, které pro nás během analýzy nejsou významné. Tak lze významně omezit počet prováděných operací během vizualizace a v důsledku podstatně navýšit výkon systému.

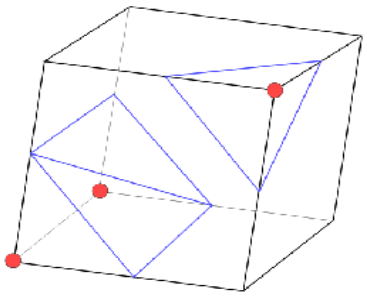
Multiple and Adaptive Resolution Representation – částečně souvisí se segmentací objemu. U těch oblastí dat, které pro nás nejsou významné, můžeme zvolit nižší rozlišení a teprve v případě potřeby data ve vyšším rozlišení načítat, případně získávat interpolací.

Dynamickou změnu obrazové kvality lze uplatnit také při pohybu s objemem (čímž zajistíme vyšší interaktivitu) a teprve při nastavení požadovaného pohledu data zobrazit v maximální kvalitě.

2.3 Nepřímý volume rendering

U nepřímého zobrazování objemových dat jsou do celého procesu zahrnuté určité mezikroky nutné k vizualizaci.Existuje více různých metod pro extrahování povrchu z objemových dat – na tomto místě se pouze pro zachování rozhledu seznámíme s algoritmem Marching Cubes.

Marching Cubes pro nalezení povrchů zkoumá objemová data zkoumáním osmi sousedních voxelů tvořících buňky. Voxely jsou označovány buď jako přítomné (1), případně nepřítomné (0), na základě jejich hustoty porovnávané s určitou hranicí. V závislosti na konfiguraci rozložení přítomných a nepřítomných voxelů v buňce jsou generovány různé polygony. Celkem připadá v úvahu 256 kombinací, které mohou být dále redukovány na pouhých 15 případů, pokud jsou odstraněny kombinace symetrické v určitém otočení. Pro každý z těchto příkazů máme předpočítané vlastnosti generovaného povrchu, tedy vrcholy generovaných polygonů a jejich normálové vektory. Tímto způsobem tedy algoritmus Marching Cubes prochází celým objemem a typicky vytváří trojúhelníky.



Ilustrace 9: polygony generované pomocí MC v jedné buňce

MC ovšem generuje obrovské množšství polygonů zpravidla neumožňující interaktivní práci. Tento problém se řeší optimalizací množiny polygonů (například je možné některé spojovat do větších). Ve výsledku jsou tedy pro potřeby vizualizace data reprezentována jako množina 2D polygonů a ne jako objemová data, což umožňuje hardwarovou akceleraci a maximalizuje tak dostupný výkon počítače.

3 Návrh

3.1 Požadavky na programové řešení

S ohledem na přenositelnost kódu naší aplikace zvolíme také realizační nástroje. Vizualizační toolkit pro 3D grafiku, Open Scene Graph (dále jen OSG), tvoří přístupové rozhraní pro OpenGL API, které přímo komunikuje s grafickým hardwarem. Toolkit je přeložitelný na na všech Windows platformách, dále pod OSX, GNU/Linux, IRIX, Solaris, HP-Ux, AIX a FreeBSD. Jako jazyk implementace využijeme C++, ve kterém je vytvořen také OSG. Vyhneme se použití nestandardních konstrukcí a rozhraní, které by omezovaly přenositelnost (zejména využití Win API).

V tomto projektu se nesoustředíme na návrh dokonalého uživatelského rozhraní (GUI), ale na funkční možnosti a maximálně názorné řešení. GUI bude představovat jednoduchý prohlížeč objemové kostky umožňující vhodné rotace, přibližování a ořezávání objemu v každé souřadné ose (min/max). Volby různých metod zobrazení, nastavování veškerých potřebných parametrů a souboru vstupních dat budou zadány při spouštění aplikace jako parametry příkazové řádky a v průběhu programu s nimi bude možné jednoduchým způsobem manipulovat, aby byla zachována hlavní výhoda volume renderingu – interaktivita práce s daty a okamžitá, názorná prezentace výsledků.

3.2 Cílová platforma, technické vybavení

Jak již bylo zmíněno v úvodu této kapitoly, projekt je nezávislý na platformě a bez větších problémů by měl být přeložitelný na každém PC disponujícím adekvátním výkonem a především moderním grafickým akcelerátorem plně podporujícím standard Pixel Shader 2.0.

Referenční řešení bude testováno na dvou podstatně rozdílných sestavách, což opět umožní nezávislé výsledky reprodukovatelné na co nejširším spektru vybavení. Jedna ze sestav poslouží pouze pro demonstraci kompatibility a definuje minimální nároky projektu.

Minimální konfigurace (první sestava)

- Intel Pentium M 1,6 Ghz
- 1024 MB RAM
- GK AGP ATI Mobility Radeon 9700 - 128 MB RAM

Optimální konfigurace (druhá sestava)

- AMD Athlon 64 X2 3800+
- 2048 MB RAM
- GK PCI-e Leadtek PX8800 GTS – 640 MB RAM

3.3 Uživatelské rozhraní

Uživatelské rozhraní bude tvořeno interaktivním prohlížečem objemové kostky maximální jednoduchosti (cílem práce není tvorba dokonalého uživatelského prostředí, ale návrh a implementace efektivního nástroje pro demonstraci zobrazování objemových dat pomocí mapování textur).

3.3.1 Načítání objemových dat

Načíst objemová data je možné pouze při spuštění programu. Soubor obsahující požadovanou sadu řezů se zadává jako parametr příkazové řádky. Vzhledem ke zpracovávání souborů bez jakýchkoliv hlaviček nesoucích informace o obsažených datech je potřeba v dalších parametrech zadat tyto informace:

- jméno souboru s načítanými daty
- rozměr objemových dat v ose X
- rozměr objemových dat v ose Y
- počet obsažených platů (rozměr Z)
- velikost voxelu v bytech (1 nebo 2 => 8 bit nebo 16 bit zobrazení)
- v případě 2 bytového uložení pořadí (big-endian, little endian)

Rozměry objemových dat musí být z důvodů kompatibility na širším spektru hardware mocniny dvou, v opačném případě budou zvětšeny na nejbližší vyšší mocninu (maximální neefektivity tedy dosáhneme například při práci s objemem o rozměrech 257 x 257 x 257, který ve své interní reprezentaci v paměti obsadí 512 ^ 3 voxelů).

Objemová data jsou načtena do objemové kostky jednotkové velikosti, což nám dále usnadňuje práci s naším rozhraním (např. při práci s ořezávacími rovinami).

3.3.2 Manipulace objemovou kostkou

Manipulace s objemovou kostkou je řízena výhradně pohybem myši:

- **Rotace objemu** – stisknuté ***levé tlačítko + pohyb v ose X/Y.***
- **Přiblížení/oddálení objemu** – stisknuté ***pravé tlačítko + pohyb v ose X/Y.***
- **Posun objemu** – stisknuté ***kolečko myši + pohyb v ose X/Y***.

3.3.3 Ořezávací roviny (clipping planes)

Podstatnou funkcí je možnost výběru určité části dat pro vizalizaci (umožní nám zaměřit se pouze na zkoumaná data a jako vedlejší jev přinese zvýšení výkonu).

Bude použito všech 6 standardních ořezávacích rovin (minimum a maximum v souřadných osách X, Y, Z), přičemž v rozhraní budeme mezi jednotlivými rovinami přepínat a ovládat je samostatně.

- **Přepínání rovin** – stisk klávesy ***Shift* + *C*** mění ořezávací roviny v pořadí min x, max x, min y, max y, min z, max z. Standardně je vybrána ořezávací rovina min x.
- **Změna polohy aktivní roviny** – stisk klávesy ***C* + *pohyb myši v ose Y*** (poloha u horního okraje obrazu = 0,5; střed obrazu = 0,0; dolní okraj obrazu = -0,5).

Ve výchozím nastavení jsou ořezávací roviny umístěny tak, aby byla viditelná celá objemová kostka. Polohy jednotlivých ořezávacích rovin je možné zadat již při spuštění programu jako parametry příkazové řádky.

3.4 Vizualizace, Open Scene Graph

Grafický toolkit Open Scene Graph (OSG) je v této práci využit pro abstrakci práce s grafickým akcelerátorem. OSG sám využívá aplikačního rozhraní OpenGL, které již s hardwarem komunikuje přímo. Široké možnosti toolkitu využijeme pouze okrajově, protože OSG je navržen pro povrchovou vizualizaci a v našem případě tedy budeme zobrazovat pouze vhodně transformovaný štos polygonů.

Naší snahou zde není dokumentovat OSG toolkit a principy jeho funkce, ale především navrhnout postup implementace našeho grafického systému. Soustředit se tedy budeme pouze na části pro náš projekt podstatné.

3.4.1 Graf scény

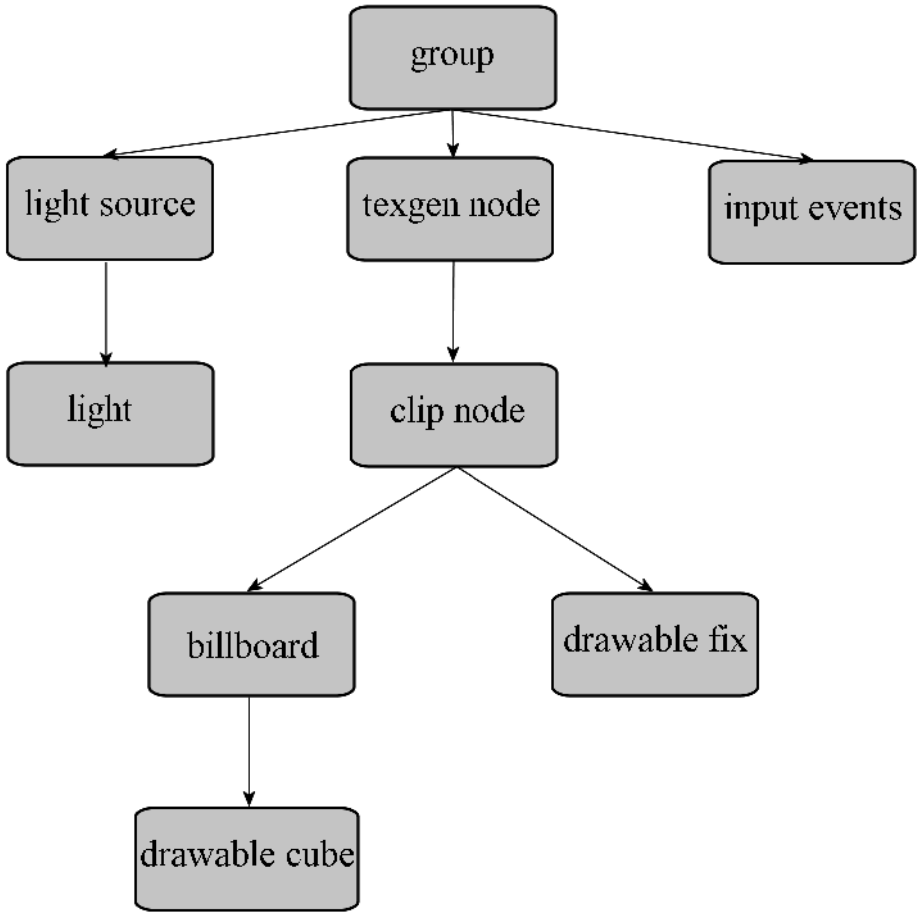
OSG pro reprezentaci scény využívá výhodnou stromovou reprezentaci (acyklický orientovaný graf), ve kterém uzly nesou například informace o geometrii, transformaci, osvětlení, případně o složitějším systému a hrany představují závislosti mezi nimi.

V OSG je uzel reprezentován objektem **osg::Node**, který na jedné straně umožňuje vzájemné spojování uzlů a na straně druhé pak slouží pro vytváření následníků. Mezi základní patří listový prvek **osg::Geode** a naopak **osg::Group** sloužící jako kontejner k napojení dalších uzlů.

V našem projektu tato reprezentace nepřináší velké výhody, ale díky způsobu návrhu bude možné připojit celý náš graf do jiného grafu a vytvořit tak například hybridní systém kombinující zobrazení objemových dat a dat reprezentovaných pomocí ploch.

OSG ovšem obsahuje objekty, které nám zajistí snadnější implementaci. Jedná se zejména o **osg::TexGenNode**, **osg::ClipNode** a **osg::Billboard**.

Na následujícím obrázku je schematicky znázorněn graf scény, který použijeme pro naši implementaci. V dalších odstavcích se tedy postupně seznámíme s jednotlivými objekty OSG, které jsou v něm obsaženy.



Ilustrace 10: Schematické znázornění grafu scény navrženého pro potřeby projektu

3.4.2 Třída osg::Geode

Geode je zkratkou pro Geometric Node – tedy list nesoucí geometrii. Jde sice o list, ale v OSG slouží k napojení dalších objektů třídy **osg::Drawable**, která popisuje zobrazitelné objekty (například sadu polygonů, nebo v kontextu volume renderingu by bylo možné generovat objektově zarovnané štosy polygonů).

3.4.3 osg::Billboard – Transformace objemové kostky

Tato třída je odvozena od **osg::Geode**. Slouží k automatické úpravě transformace tak, aby napojenou geometrii vždy natáčela směrem k pozorovateli. Většinou se používá k zobrazení různých efektů (kouř, mraky, částicové systémy), u kterých k dostatečně věrné vizuální reprezentaci stačí pohled z jedné strany.

Pokud ovšem **osg::Billboard** zasadíme do kontextu volume renderingu, prokáže nám velkou službu při zobrazování pohledově zarovnaných řezů. Jak je patrné již v uvedeném grafu scény, na **osg::Billboard** napojíme objekt třídy **osg::Drawable** obsahující vygenerovaný štos polygonů.

Ten je vytvořen ve zpětném pořadí, takže se jako první bude vždy renderovat pozorovateli nejvzdálenější polygon a navíc budou díky transformaci v této třídě všechny polygony namířeny přímo k pozorovateli.

3.4.4 osg::ClipNode – Ořezání objemové kostky

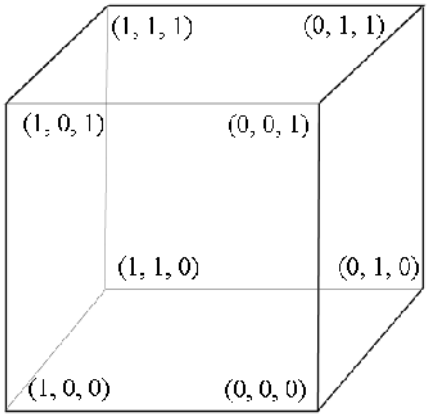
Nyní již tedy máme správně seřazené polygony, které už ovšem po rotaci netvoří objemovou kostku a je nutné je vhodně ořezat, abychom zabránili zbytečnému zobrazování prázdných oblastí. Přestože toto ořezání patří ke složitějším částem volume renderingu s pohledově zarovnanými pláty (tedy pokud bychom jej řešili softwarově), s využitím osg::ClipNode je vše velmi snadné. Třída nám umožní definovat 6 základních ořezávacích rovin, které dokonale poslouží k vyhranění objemové kostky a navíc budeme mít možnost pomocí vhodných posunů zobrazovat také vnitřní řezy objemové kostky. Stručně shrnuto – objem obklopený šesti ořezávacími rovinami definuje objemovou kostku (resp. kvádr při vnitřních řezech), kterou budeme interaktivně otáčet. Za instanci objektu třídy **osg::ClipNode** navážeme instanci objektu **osg::Billboard** popsanou dříve – požadované ořezání je tedy na světě a navíc je realizováno hardwarově (clipping planes jsou součástí pipeline grafického akcelerátoru – nejde tedy přímo o záslužnou činnost třídy **osg::ClipNode** – ta vystupuje v roli prostředníka mezi standardní funkcionalitou OpenGL, resp. samotných grafických akcelerátorů).

V tomto bodě návrhu jsme tedy připraveni implementovat zobrazení objemové kostky s pohledově zarovnanými pláty. Zbývá nám zajistit správné souřadnice pro mapování 3D textury nesoucí data hustoty na korektně otočený a ořezaný štos polygonů.

3.4.5 osg::TexGenNode – Souřadnice pro mapování textur

K zajištění snadného vygenerování správných souřadnic pro mapování 3D textury na polygony objemové kostky využijeme třídu **osg::TexGenNode**. V ní určíme tři roviny zarovnané se souřadnými osami, v nichž bude mapování probíhat (roviny S, T, R), a nastavíme také požadovanou tranformaci souřadnic na **osg::TexGen::EYE_LINEAR**. Díky této možnosti nebudeme později muset provést transformaci podle aktuálního pohledu.

Původní mapování texturovacích souřadnic je u naší jednotkové objemové kostky triviální, jak uvádí obrázek.
K úpravě souřadnic dochází vždy při změně pohledu (ještě před samotným zobrazováním) a díky volbě režimu EYE_LINEAR jsou transformovány podle aktuální pohledové matice (resp. podle transformačních matic předcházejících výskyt instance TexGenNode v grafu scény a navíc pohledové matice).



Ilustrace 11: Texturovací souřadnice

3.5 Metody zobrazení

V systému budeme realizovat dva různé způsoby zobrazení objemových dat – X-rays a Maximum Intensity Projection.

3.5.1 Metoda „X-rays“

X-rays (X-paprsky) nebo také Röntgen-rays (rentgenové paprsky) jsou používány pro diagnostiku například v radiografii a krystalografii. Jedná se o formu elektromagnetické radiace s vlnovou délkou v rozsahu 10 – 0.01 nanometrů (to odpovídá frekvencím v rozsahu 30 – 30000 Phz).

Rentgenové paprsky snadněji prochází měkkými tkáněmi a naopak jsou blokovány tkáněmi tvrdými, případně kostmi. Na výstupní rentgenový snímek tedy dopadá stín zkoumaného objektu (viz. obrázek).

Pro naši práci využijeme tento typický výstup metody používaný v medicíně a budeme jej napodobovat. Prakticky vzato, zjistíme průměrnou hodnotu hustoty podél paprsku vrženého zkoumanými objemovými daty a na základě tohoto průměru provedeme vizualizaci – podle povahy zkoumaného vzorku můžeme zobrazit vyšší hustoty s vyšší intenzitou, případně naopak.

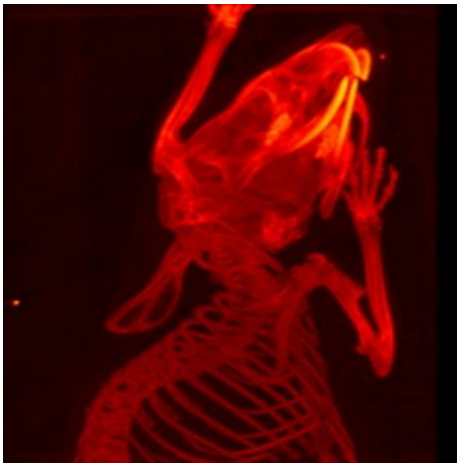
Ilustrace 12: Ruka ženy (X-ray snímek pořídil roku 1896 sám Wilhelm Röntgen)

3.5.2 Maximum Intensity Projection (MIP)

U předchozí metody byly hustoty voxelů podél paprsku akumulovány a na obrazovou rovinu se dostala průměrná hodnota ze všech zpracovaných vzorků.

Jiný přístup je zvolen u Maximum Intensity Projection (MIP). Zde nejsou hustoty voxelů průměrovány, ale je zjištěna pouze maximální hustota (intenzita). Je tedy zřejmé, že výsledkem vizualizace dvou snímků pomocí metody MIP z opačných pohledů získáme dva symetrické obrazy.

Metoda se používá například při analýze CT scanů plic, ve kterých umožňuje snadné vyhledání uzlin.



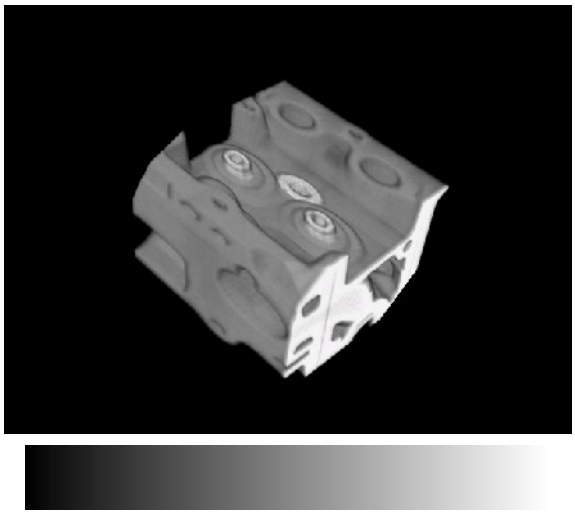
Ilustrace 13: MIP rendering

3.6 Přenosové funkce

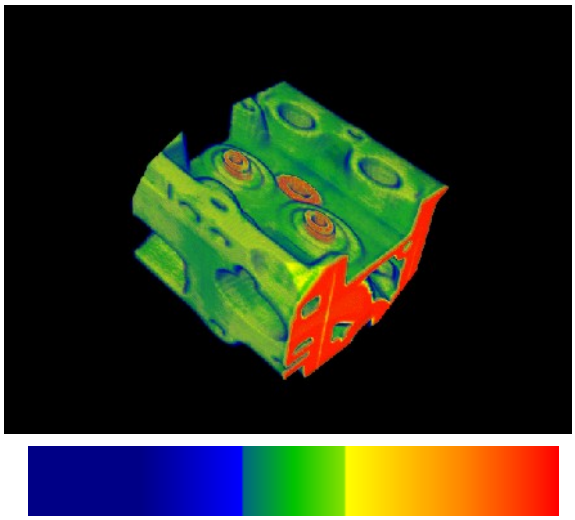
Přenosové funkce (Transfer Functions) jsou používány ke klasifikaci vzorků na základě hodnoty jejich hustoty. V našem případě funkci implementujeme pomocí 1D RGBA textury, ve které bude normalizovaná hustota voxelu udávat texturovací souřadnici. Hodnotu hustoty voxelu tudíž převedeme na RGBA informaci o barvě.

Pro navržení vhodných přenosových funkcí je nutná důkladná analýza zobrazovaného vzorku a přenosové funkce tedy zpravidla nejsou univerzálně použitelné.

V našem systému proto umožníme dynamické přepínání z více různých přenosových funkcí.

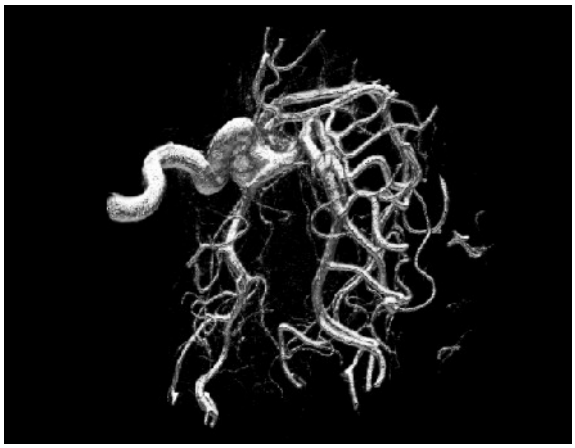


Ilustrace 14: hustota přímo určuje intenzitu výsledné barvy (lineární přenosová funkce)



Ilustrace 15: části s vyšší hustotou jsou vyobrazeny v odstínech žluté a červené

Vhodné přenosové funkce umožňují jednoduchým způsobem významné zlepšení obrazového výstupu. Na okolních obrázcích je zobrazen blok motoru – vlevo s linární přenosovou funkcí, vpravo s přenosovou funkcí složenou z více barev. Na dalších záběrech je vyobrazeno aneurisma – vlevo opět s lineární přenosovou funkcí a vpravo s přenosovou funkcí vhodně zvýrazňující část vzorků.



Ilustrace 16: Aneurisma – Lineární přenosová funkce neposkytuje dobrý vjem

Ilustrace 17: Vhodná přenosová funkce podstatně zpřehlední vizualizaci dat

3.7 Osvětlení objemových dat

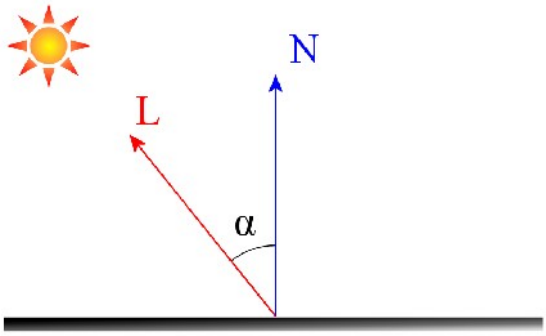
Osvětlení objemových dat významným způsobem zlepšuje výsledný vizuální vjem. Při jeho realizaci však vyvstává problém, protože objemová data zpravidla neobsahují žádné informace o plochách, jejichž normálové vektory jsou při výpočtech osvětlení využívány. Adekvátní náhradu však poskytují gradienty, které budou popsány později. Implementaci osvětlovacího modelu tedy nic nebrání a proto shrneme základní poznatky, kterých využijeme při tvorbě vizualizačního systému.

Výpočet barvy osvětlení můžeme rozložit do dvou oddělených složek:

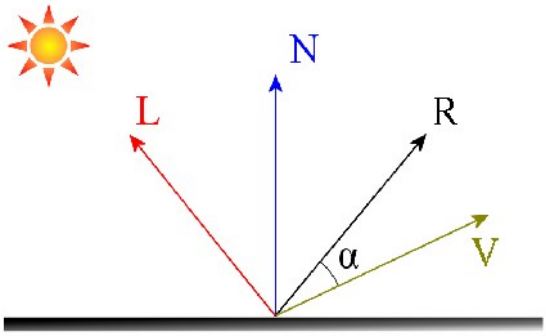
- difůzní složky
- spekulární složky (odlesky)

Při výpočtu **difůzní složky** zohledňujeme pouze normálový vektor osvětlované plochy, směr světla a barvu světla (*LightColor*).

Ve výpočtu **spekulární složky** navíc hraje roli poloha pozorovatele ve scéně a parametr určující odrazivost materiálu (*shininess*), vizuální změny při manipulaci s objemovou kostkou budou u této složky mnohem lépe pozorovatelné na úrovni jednotlivých voxelů. Spekulární složka se projevuje tím více, čím se úhel mezi vektorem odrazu světla a vektorem směřujícím k pozorovateli zmenšuje.



Ilustrace 18: Difůzní složka světla (L - směr světla, N - normálový vektor plochy)



Ilustrace 19: Spekulární složka světla (R - směr odrazu světla, V - směr pozorovatele)

Výpočet difůzní složky světla:

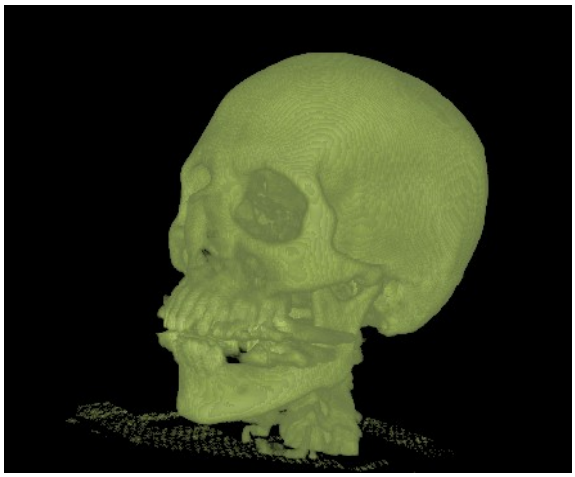
$$diffuse = (\vec{N} \cdot \vec{L}) * LightColor$$

(N a L musí být normalizované vektory)

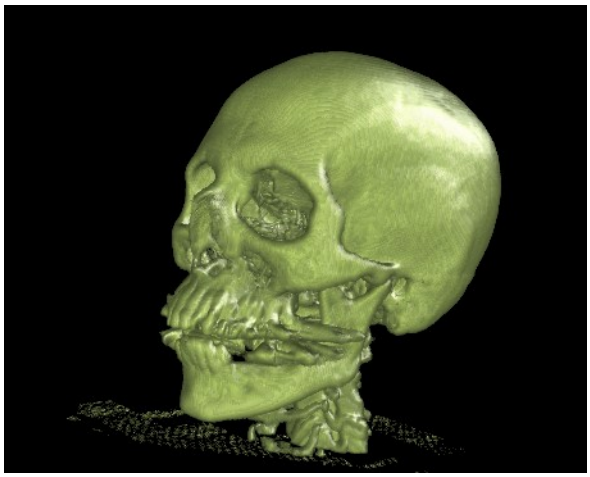
Výpočet spekulární složky světla:

$$specular = (\vec{R} \cdot \vec{V})^{shininess} * LightColor$$

(R a V musí být normalizované vektory)



Ilustrace 20: CT scan - lineární zobrazení bez aplikace osvětlení



Ilustrace 21: Stejná data po aplikaci osvětlovacího modelu

Jak je z výše uvedených ilustrací patrné, aplikace osvětlovacího modelu přináší výrazné zlepšení výsledného vizuálního vjemu.

3.7.1 Gradienty

V popsaném modelu osvětlení namísto normálových vektorů používaných při povrchové reprezentaci využijeme tzv. gradientů. Ty je možné vypočítat pouze na základě známých hodnot hustoty v našich objemových datech.

Výpočet je založen na předpokladu, že plocha se vyskytuje v tom místě, kde dochází k výrazné změně hustoty (typicky přechod mezi prostředím – tuhý materiál / řídký vzduch). Pro každý voxel můžeme na základě prozkoumání jeho hustoty a hustot sousedních voxelů určit gradient – tedy normálový vektor iso-plošky založený na změně hustoty v jednotlivých směrech.

Gradienty můžeme počítat během samotného renderingu, ale jde o velmi neefektivní řešení vzhledem k nepříznivému paměťovému zarovnání (pro každý voxel bude provedeno dalších 6 závislých čtení). Je tedy vhodné gradienty předpočítat a uložit v jedné textuře současně s hodnotou hustoty.

4 Implementace

V této kapitole jsou zachyceny komentáře k programové realizaci postupů navržených v předchozích kapitolách. Často také problematiku dále rozvedeme, bude-li to nutné.

V rámci této kapitoly jsou často uváděny fragmenty zdrojového kódu, popřípadě hlavičky nejdůležitějších funkcí s vysvětlením jejich funkce a předpokládaného chování.

4.1 Načtení objemových dat

Na základě původního návrhu je implementována funkce `readRaw`.

```
    osg::Image* readRaw(int sizeX, int sizeY, int sizeZ, int
numberBytesPerComponent, int numberOfComponents, int normalise,
const std::string& endian, const std::string& raw_filename)
```

Parametry `sizeX, sizeY a sizeZ` udávají rozměr objemových dat uložených v souboru. Ve výsledné textuře jsou zvětšeny na nejnižší vyšší mocninu dvou.

Parametr `numberBytesPerComponent` umožňuje rozlišit 8 bitové a 16 bitové vzorky – přípustné hodnoty jsou tedy 1 a 2.

Většinou v našich testech požadujeme normalizovaná data hustoty v rozsahu 0.0 – 1.0. Automatickou normalizaci vstupních dat provedeme uvedením parametru `normalise = 1`.

Vzhledem k mnoha různým testovacím datům byl přidán parametr `endian`. Má smysl pouze pro objemová data s 16 bitovými vzorky a určuje, který byte je významnější (přípustné hodnoty jsou `"big"` a `"low"`).

Parametr `raw_filename` udává vstupní soubor s objemovými daty.

Návratovou hodnotou funkce je ukazatel na nový objekt třídy `osg::Image`, který je později využit pro přiřazení k požadované texturovací jednotce grafického adaptéru (viz. následující kód).

4.1.1 Uložení objemových dat do 3D textury

Pro abstrakci práce s 3D texturami poskytuje toolkit OSG třídu `osg::Texture3D`. Pomocí metody `SetImage` asociujeme data načtená v `osg::Image`.

Vzhledem k potřebě další práce s uloženými daty v 3D textuře na úrovni Pixel Shaderu je nutné explicitně specifikovat formát, ve kterém chceme texturu uložit. Nabízí se nám dva možné formáty:

- `GL_INTENSITY`
- `GL_RGBA`

GL_INTENSITY využíváme pro načtení objemových dat, u kterých nebudeme ukládat gradienty (tj. normálové vektory jednotlivých voxelů používané pro implementaci osvětlení). Interní reprezentace dat může být jak 8 bitová, tak 16 bitová. Bez komplikací tedy můžeme využívat oba druhy dat.

GL_RGBA naopak využijeme pro uložení objemových dat i s předpočítanými gradienty, které nám umožní implementaci modelu osvětlení pro každý voxel. Zde využijeme pouze 8 bitové reprezentace a 16 bitové vzorky hustoty převedeme pouze na 8 bitové. RGB složky nesou předpočítané souřadnice gradientů a alfa složka obsahuje hodnotu hustoty. Více bude vysvětleno v kapitole věnované implementaci osvětlení.

V obou případech bude dostupná hodnota hustoty v kanálu alfa každého voxelu – pro obě metody tedy dále můžeme výhodně využívat společný kód.

Fragment kódu obsahující asociaci 3D textury a její aktivaci:

```
osg::ref_ptr<osg::Texture3D> texture3D (new osg::Texture3D);

texture3D->SetInternalFormatMode (
     osg::Texture3D::USE_USER_DEFINED_FORMAT);

if (lighting == 1)    texture3D->setInternalFormat (GL_RGBA);
                 else texture3D->setInternalFormat (GL_INTENSITY);

texture3D->setFilter (
     osg::Texture3D::MIN_FILTER,      osg::Texture3D::LINEAR);
texture3D->setFilter (
     osg::Texture3D::MAG_FILTER,      osg::Texture3D::LINEAR);

texture3D->setWrap (osg::Texture3D::WRAP_S, osg::Texture3D::CLAMP);
texture3D->setWrap (osg::Texture3D::WRAP_T, osg::Texture3D::CLAMP);
texture3D->setWrap (osg::Texture3D::WRAP_R, osg::Texture3D::CLAMP);

if (lighting == 1)    texture3D->setImage (image_3d_gradient.get());
                 else texture3D->setImage (image_3d.get());

stateset->setTextureAttributeAndModes (
     TEXUNIT_DENSITY,
     texture3D.get(),
     osg::StateAttribute::ON);
```

V kódu kromě samotné asociace dochází také k nastavení parametrů texturování – je aktivována trilineární interpolace a způsob zacházení s texturovacími souřadnicemi překračujícími hranice načtených dat (ořezávání, CLAMP). V posledním kroku je textura aktivována ve stavové množině uzlu, kde bude používána a je jí přiřazeno číslo texturovací jednotky.

4.1.2 Výpočet a uložení gradientů

Chceme-li při zobrazování objemových dat realizovat i osvětlovací model, je potřeba předpočítat normálové vektory (gradienty) pro každý voxel (isoplochy).

Jak již bylo řečeno v návrhu, je možná také implementace dynamického výpočtu gradientů v pixel shaderech, ale se současným hardwarem je velmi neefektivní (ani aktuální high-end v podobě GeForce 8800 GTX se svými 128 stream procesory nepřináší uspokojivé výsledky – omezení rychlosti totiž vyplývá z neefektivní práce pamětí při tomto výpočtu).

V naší implementaci tedy volíme možnost předpočítání gradientů do jedné textury i s hodnotami hustoty – rozdíl ve výkonu tedy není žádný až do okamžiku, kdy gradienty použijeme k dalším výpočtům.

Bohužel je zřejmá paměťová náročnost – potřebujeme uložit 3 složky gradientu, pro každou vyhradíme jeden byte – celkem tedy 24 bitů. Při zobrazení 8 bitových objemových dat je tedy paměťová náročnost 4x vyšší. Navíc se jedná o data, která musí být bezpodmínečně umístěna přímo v paměti grafické karty, aby nedošlo k drastickému poklesu výkonu souvisejícího s opakovaným načítáním částí 3D textury ze systémové paměti do paměti grafické karty.

Vzhledem k faktu, že gradienty jsou normalizované vektory, nabízí se ještě možnost uložení pouze dvou složek a dynamického výpočtu poslední složky při běhu pixel shaderu. Tento způsob se zdá na první pohled zajímavý, ale přináší také několik komplikací. Proto se jím budeme zabývat až později.

V naší implementaci tedy budeme ukládat objemová data společně se všemi složkami gradientů do jedné textury formátu RGBA:

R – x složka gradientu

G – y složka gradientu

B – z složka gradientu

A – hodnota hustoty daného voxelu

Gradient pro každý voxel vypočteme následovně (vysvětlení je obsaženo v návrhu):

```
osg::Vec3 grad( (float)(*left)  - (float)(*right),
                (float)(*below) - (float)(*above),
                (float)(*out)   - (float)(*in) );
grad.normalize();
```

Před uložením do naší textury je potřeba převést hodnotu z normalizovaného rozsahu *[0 .. 1]* do rozsahu neznaménkového bytu *[0 .. 255]*:

```
grad.x() = osg::clampBetween(
                (grad.x() + 1.0f) * 128.0f, 0.0f, 255.0f);
grad.y() = osg::clampBetween(
                (grad.y() + 1.0f) * 128.0f, 0.0f, 255.0f);
grad.z() = osg::clampBetween(
                (grad.z() + 1.0f) * 128.0f, 0.0f, 255.0f);
```

A v posledním kroku již uložíme data ve správném rozsahu do cílové textury, do složek RGB:

```
*(destination++) = (unsigned char)(grad.x());
*(destination++) = (unsigned char)(grad.y());
*(destination++) = (unsigned char)(grad.z());
```

Poslední složku cílové textury využijeme pro uložení hodnoty hustoty:

```
*(destination++) = *densityMap;
```

Máme tedy vytvořenu RGBA texturu s daty vhodnými pro zobrazení objemových dat včetně osvětlovacího modelu. Jde ovšem o řešení s mnoha kompromisy. Především hodnota hustoty je uložena pouze na osmi bitech, místo běžných deseti a uložení složek gradientu na osmi bitech také není dostatečně přesné. I přes tato omezení osvětlená objemová data působí dobrým vizuálním dojmem a u povrchových částí dat zpravidla působí přesvědčivěji.

4.2 Vytvoření objemové kostky

```
osg::Node* createCube(float xsize, float ysize, float zsize, float
alpha, unsigned int numSlices)
```

Výstupem této funkce je uzel obsahující geometrii objemové kostky o rozměru (`xsize * ysize * zsize`) s daným počtem plátů (`numSlices`), jejichž vertexy mají nastavenu průhlednost `alpha`.

Objemová kostka je napojena na objekt třídy `osg::Billboard`, takže normálové vektory plátů jsou vždy namířeny směrem k pozorovateli.

Ořezávání je zajištěno napojením vytvořené kostky na objekt třídy `osg::ClipNode`, generování texturovacích souřadnic pak napojením na `osg::TexGenNode`.

V prvním kroku tedy vytváříme objekt třídy `osg::TexGenNode`, přiřazujeme jej k texturovací jednotce (`TEXUNIT_DENSITY = 1`), nastavujeme požadovaný režim generování

souřadnic (`osg::TexGen::EYE_LINEAR`) a konečně také nastavujeme roviny pro generování souřadnic v jednotlivých osách.

```
     osg::TexGenNode* texgenNode_0 = new osg::TexGenNode;
    texgenNode_0->setTextureUnit(TEXUNIT_DENSITY);
    texgenNode_0->getTexGen()->setMode(osg::TexGen::EYE_LINEAR);
    texgenNode_0->getTexGen()->setPlane (
                osg::TexGen::S,
osg::Vec4(xMultiplier,0.0f,0.0f,0.5f));
    texgenNode_0->getTexGen()->setPlane (
                osg::TexGen::T,
osg::Vec4(0.0f,yMultiplier,0.0f,0.5f));
    texgenNode_0->getTexGen()->setPlane (
                osg::TexGen::R,
     osg::Vec4(0.0f,0.0f,zMultiplier,0.5f));
```

Dále vytváříme uzel pro ořezávání, nastavujeme ořezávací roviny (boundingbox), navazujeme geometrii objemové kostky a společně s ořezávacím uzlem vše připojujeme k texturovacímu uzlu.

Parametry boundingboxu mohou být libovolně nastavovány uživatelem. Ve výchozím stavu pak obsahují celou objemovou kostku.

```
    clipnode = new osg::ClipNode;
    clipnode->createClipBox (*boundingbox);
    clipnode->addChild(createCube(xSize, ySize, zSize, 1.0f, numSlices));
    texgenNode_0->addChild(clipnode);
```

4.3 Implementace přenosových funkcí

Implementace přenosových funkcí se drží původního návrhu – pro každou přenosovou funkci tedy vytváříme tabulku hodnot (1D texturu – `osg::Texture1D`) a zpřístupňujeme ji dalšímu zpracování v pixel shaderech (jako `TEXUNIT_COLOR=0`), jak je patrné z dále uvedeného fragmentu kódu:

```
    texture_palette = new osg::Texture1D;
    texture_palette->setFilter(
          osg::Texture1D::MIN_FILTER, osg::Texture1D::NEAREST);
    texture_palette->setFilter(
          osg::Texture1D::MAG_FILTER, osg::Texture1D::NEAREST);
    texture_palette->setWrap(
          osg::Texture1D::WRAP_S, osg::Texture1D::CLAMP);
```

```
stateset->setTextureAttributeAndModes(
      TEXUNIT_COLOR,
      texture_palette,
      osg::StateAttribute::ON);
```

V posledním kroku je texturovací jedotka `TEXUNIT_COLOR` aktivována v požadované stavové množině grafu scény.

Během běhu grafického systému je možné libovolně měnit aktuální přenosovou funkci naplněním obsahu textury `texture_palette` novými hodnotami.

```
osg::ref_ptr<osg::Image> image1 (
                              osgDB::readImageFile(fileName));
texture_palette->setImage (image1.get());
```

4.4 Pixel Shadery a GLSL

Při implementaci programů pro jednotky Pixel Shaderů jsme využili jazyka GLSL (GL Shading Language), jehož prostřednictvím je možné k těmto jednotkám přistupovat na úrovni jazyka nápadně podobnému standardnímu C. GLSL je podporován i použitým toolkitem OSG, proto jsme mohli efektivně a bez zbytečných komplikací využít všech jeho výhod (zejména tedy vynikající názornosti, která vyplývá z přehledné syntaxe).

Ačkoliv jsou dnes Pixel Shadery používány jako obecný pojem, je potřeba rozlišit je na dvě oddělené (ale spolupracující) jednotky – Vertex Shadery a Pixel Shadery (v GLSL také Fragment Shadery).

Na podrobné vysvětlení funkce grafické pipe-line a funkce Pixel Shaderů v této práci není dostatek prostoru, omezíme se tedy na vysvětlení jejich využití v našem projektu.

4.4.1 Práce s Vertex Shadery

Vertex Shadery poskytují přístup k zobrazované geometrii – přesněji tedy k jednotlivým polygonům a jejich vrcholům, které grafický akcelerátor zobrazuje. Před samotným zobrazováním máme možnost poskytnout vertex shaderům všechna data, která budeme potřebovat pro výpočet. Často ovšem postačují údaje, které jsou poskytovány implicitně. Výsledky výpočtů provedených ve vertex shaderech mohou být dále využity v pixel shaderech (fragment shaderech) – není tedy nutné pro každý zobrazovaný obrazový bod opakovaně vypočítávat data měnící se pouze na úrovni vertexů.

Úkolem našeho vertex shaderu je pouze výpočet souřadnic pro mapování 3D textury v pixel shaderu. Objemová data jsou předávána ve druhé texturovací jednotce (index 1), v první texturovací jednotce je k dispozici 1D textura obsahující hodnoty použité přenosové funkce – zde žádné

generování texturovacích souřadnic nemá smysl, protože souřadnice se mění pro každý zobrazený bod na základě hodnoty hustoty přečtené z 3D textury až v pixel shaderu.

V našem systému tedy používáme pouze triviální program vertex shaderu, který nahrazuje standardní funkcionalitu Open GL a navíc fragment shaderům předává texturovací souřadnice pro zobrazení objemových dat (zde si všimněme využití rovin EyePlaneS/T/R/Q, které jsou v programu nastavovány v uzlu TexGenNode – viz. Kapitola 4.2 – Vytvoření objemové kostky).

```
void main()
{
    gl_Position=ftransform();
    gl_ClipVertex = gl_ModelViewMatrix * gl_Vertex;
    vec4 eyePos = gl_ClipVertex;

    gl_TexCoord[1].s=dot(eyePos,gl_EyePlaneS[1]);
    gl_TexCoord[1].t=dot(eyePos,gl_EyePlaneT[1]);
    gl_TexCoord[1].p=dot(eyePos,gl_EyePlaneR[1]);
    gl_TexCoord[1].q=dot(eyePos,gl_EyePlaneQ[1]);
    gl_TexCoord[1] *= gl_TextureMatrix[1];
}
```

4.4.2 Práce s Pixel shadery (Fragment Shadery)

Pixel Shadery (případně Fragment Shadery v podání GLSL) poskytují přístup ke každému pixelu, který je grafickým adaptérem zobrazován. Umožňuje nám prakticky libovolnou manipulaci s veškerými svými atributy – barvou, hodnotou v paměti hloubky, je možná dokonce i změna polohy ve scéně, případně i úplné vypuštění pixelu z dalšího procesu zobrazování (čehož můžeme s výhodou použít – včas se vyhneme náročným výpočtům u jakýmkoliv způsobem nezajímavých pixelů).

Implementací několika různých programů pixel shaderů (dále můžeme dle kontextu pod pojmem „pixel shader“ rozumět také program pro pixel shader jednotku) a umožněním jejich interaktivního přepínání v konečné fázi docílíme různých způsobů zobrazení (viz. kapitola 3.5 – Metody zobrazení) bez zásahů do hlavního programu. Než ale rozebereme různá úskalí tvorby pixel shaderů, shrneme veškeré atributy, které budou pro naše programy pevně dané a vytvoříme tak určitou formu minimalistického aplikačního rozhraní.

4.4.2.1 Atributy „uniform“

Uniform proměnné v rámci pixel shaderu představují libovolné neměnné hodnoty, které mohou být programu předány buď z hlavního programu, nebo z programu vertexu shaderu. Neměnnou hodnotou v tomto případě může být (a také téměř vždy je) i textura.

Pro naše programy budou společné tyto hodnoty nastavené z hlavního programu:

```
uniform sampler1D colorMap;
```

Hodnoty přenosové funkce jsou předány v první texturovací jednotce grafického adaptéru a budeme k nim přistupovat jako k 1D textuře (resp. tabulce) obsahující předem definované hodnoty.

```
uniform sampler3D densityMap;
```

V další texturovací jednotce budou vždy předána data obsahující hodnoty hustoty voxelů (ty budou vždy dostupné v alfa-složce RGBA modelu). V případě spuštění hlavního programu s podporou osvětlovacího modelu budou RGB složky obsahovat předpočítaný gradient vektor daného voxelu.

```
uniform float transparency;
```

Uniform „transparency“ pixel shaderu udává koeficient, kterým by měla být v každém programu násobena výstupní hodnota průhlednosti jednotlivých voxelů. Hodnota v původním nastavení přímo souvisí s počtem zobrazovaných plátů objemové kostky – abychom dosáhli podobných vizuálních výsledků s rozlišným nastavením, je totiž nutné výstupní hodnoty odpovídajícím způsobem upravovat. Ve výchozím stavu je hodnota nastavena na *32.0f / počet plátů* – je ovšem uživatelsky nastavitelná i při běhu hlavního programu a může tedy nabývat libovolné hodnoty s rozsahem *0.0f – 1.0f*.

```
uniform float alphaCutOff;
uniform float alphaCutOffMax;
```

Uniform *alphaCutOff* a *alphaCutOffMax* slouží k odstranění požadované části voxelů na základě hodnoty jejich normalizované hustoty. Je-li hodnota hustoty nižší než hodnota *alphaCutOff*, voxel by měl být odstraněn z dalšího procesu zobrazování. Stejně tak by měl být vypuštěn i v případě překročení horní hranice, kterou definuje uniform *alphaCutOffMax*. Obě hodnoty jsou uživatelsky nastavitelné v rozsahu *0.0f – 1.0f*.

Způsob, jakým konkrétní program těchto vstupních dat využije, se již může zásadně odlišovat. Přes programy velmi jednoduché je možné postoupit až k těm nejsložitějším operacím – je ovšem potřeba v každé situaci mít na paměti, že veškeré operace budou vypočteny při zobrazení každého voxelu a i na první pohled drobné změny mohou vážně snížit výkon celého grafického systému. V krajním případě vede výrazně neefektivní program pixel shaderu k nefunkčnosti celého systému.

4.4.2.2 Uniform sampler3D – densityMap

Uniform sampler3D poskytuje přístup k datům v texturovací jednotce interpretovaným ve formě 3D textury. Pro získání hodnoty hustoty (případně gradientů) právě zpracovávaného voxelu slouží konstrukce:

```
vec4 density = texture3D (densityMap, gl_TexCoord[1].stp);
```

Texturovací souřadnice obsažené ve vektoru *gl_TexCoord[1].stp* jsou vypočteny na základě nastavení provedeného v programu vertex shaderu. Hodnota je díky nastavení hlavního programu trilineárně interpolována (hardwarová interpolace je zde velmi rychlá – realizovaná grafickým akcelerátorem).

Po provedení příkazu máme v každém případě k dispozici hodnotu hustoty aktuálního voxelu (*density.a*), případně také hodnoty jednotlivých složek gradient vektorů (*density.r, density.g, density.b*).

4.4.2.3 Uniform sampler1D – colorMap

Pro přístup k hodnotám přenosové funkce využijeme obdobně také první texturovací jednotku. Tentokrát data interpretujeme jako 1D texturu a indexem tedy není vektor, ale pouze jediná hodnota typu *float*, kterou může být také dříve získaná hodnota hustoty upravená dle potřeby. Výstupem je opět vektor se čtyřmi složkami, reprezentujícími jednotlivé složky barevného modelu RGBA (*color.r, color.g, color.b, color.a*).

```
vec4 color = texture1D (colorMap, density.a);
```

4.4.2.4 Výstupní atributy voxelu

V programu pixel shaderu můžeme standardní funkcionalitu zcela změnit modifikací určitých atributů výstupního voxelu. Na tomto místě se seznámíme pouze se dvěmi: *gl_FragColor* a *gl_FragDepth*.

Atribut *gl_FragColor* je hodnotou typu *vec4* a je umožňuje nezávislý přístup ke každé barevné složce i alfa kanálu výstupního voxelu. Je tedy nejdůležitějším atributem, který je využit ve všech programech generujících vizuální výstup.

Atribut *gl_FragDepth* je hodnotou typu *float* umožňující změnu hodnoty hloubky aktuálního voxelu před uložením do paměti hloubky (depth buffer). Jeho funkce je v hlavním programu nastavena tak, aby voxely s vyšší hodnotou hloubky (voxely hlouběji ve scéně) nepřepisovaly voxely s nižší hloubkou (popředí scény) a byly vypuštěny. Pokud hodnotu nezměníme, voxel bude vždy vykreslen, protože hlavní program vždy vykresluje pláty objemové kostky směrem od zadního k přednímu.

4.4.3 Implementace modelu osvětlení

Již během návrhu jsme zdůraznili nejpodstatnější věc – chceme-li implementovat osvětlení, potřebujeme normálové vektory. Vysvětlili jsme si náhradu normálových vektorů gradient vektory, které jsme si předpočítali a uložili do 3D textury společně s hodnotami hustoty. V pixel shaderu je znovu získáme následovně:

```
vec3 normal = normalize ((density.rgb - 0.5) * 2.0);
```

GLSL veškeré hodnoty v textuře interpretuje typem float v rozsahu *[0.0f .. 1.0f]*, proto současně s načtením hodnot jednotlivých složek provedeme jejich převod do rozsahu *[-1.0f .. 1.0f]* a tím znovu získáváme uložený gradient vektor. Přestože jsme jej uložili v normalizovaném tvaru, převodem na celé číslo s pouhými osmi bity a zpětnou interpretací na typ *float* dochází ke ztrátě přesnosti – proto získaný gradient znovu normalizujeme.

Dalším krokem bude získání informací o světelném zdroji. Využijeme proměnné gl_LightSource zpřístupňující běžná světla v OpenGL scéně:

```
vec3 lightDir = normalize (gl_LightSource[0].position);
vec3 halfVector = normalize (gl_LightSource[0].halfVector);
```

Následuje již pouze výpočet difůzní a spekulární složky světla:

```
float diffuse = abs (dot (lightDir, normal));
float specular = max (0.0, pow (dot (halfVector, normal),
                                shininessParam)
                      );
```

V konečné fázi získané složky smícháme podle požadovaného výsledku, případně připočteme trvalé osvětlení určité intenzity, abychom zcela neosvětlená data nezobrazili pouze jednolitou černou barvou.

```
gl_FragColor = ambient + diffuse * color + specular;
```

V tomto kroku se již otevírá celá řada dalších možností jak výsledné vzorky upravit.

Zde uvedené ukázky jsou zcela záměrně minimalistické a částečně zjednodušené, aby byla zachována maximální názornost bez zbytečného zamlžení problematiky implementačními detaily (kompletní zdrojové texty programů fragment shaderů jsou uvedeny ve druhé příloze této práce).

4.4.4 Vizualizace metodou X-rays

Z hlediska implementace jde o nejjednodušší variantu – v hlavním programu jsme již nastavili způsob míchání nových voxelů s dřívější barvou (Blending Mode), nyní tedy pouze zobrazujeme všechny voxely.

Zjištení RGBA hodnoty přenosové funkce:

```
vec4 col = texture1D (colorMap, density.a);
```

Jakmile známe hodnotu hustoty, můžeme ihned rozhodnout, jestli je ve zkoumaném rozsahu, případně jestli můžeme voxel vyřadit z dalšího zobrazení pomocí funkce *discard*:

```
if ((density.a < alphaCutOff) || (densita.a > alphaCutOffMax))
 discard;
```

Pokud voxel vyřazen nebyl, následuje nastavení jeho výstupní barvy – v tomto případě zanedbáváme alfa kanál přenosové funkce (z níž získáme barevnou informaci) a nahrazujeme jej hodnotou hustoty *density.a* násobenou koeficientem *transparency* závislým na počtu zobrazovaných řezů (pro zachování stejného vizuálního vjemu s různými počty řezů, jak bylo vysvětleno dříve).

```
gl_FragColor = vec4 (col.r, col.g, col.b,
                        density.a * transparency);
```

4.4.5 Vizualizace metodou MIP

Postup zobrazení u této metody je totožný jako v předchozím případě s jediným drobným, ale zásadním doplněním. Protože nás u MIP zajímá pouze maximální hodnota hustoty, nebudeme smíchávat výstupní barvy všech voxelů – na výstup pouze nastavíme barevnou hodnotu voxelu s maximální hustotou. Toho docílíme využitím paměti hloubky (depth buffer). Pro každý zpracovaný voxel změníme jeho výstupní hodnotu *gl_FragDepth* na základě hodnoty hustoty. Na výstup se tedy voxel dostane pouze v tom případě, kdy bude k pozorovateli blíže (bude mít vyšší hodnotu hustoty).

Určení hloubky aktuálního voxelu:

```
gl_FragDepth = 1.0 - density.a;
```

Funkce tohoto programu je pochopitelně podmíněna odpovídajícím návrhem hlavního programu – pokud bychom dříve vhodně nenastavili srovnávací funkce paměti hloubky a smíchávací režim, vizuální výstup programu by nebyl korektním zobrazením metody MIP.

5 Výsledky

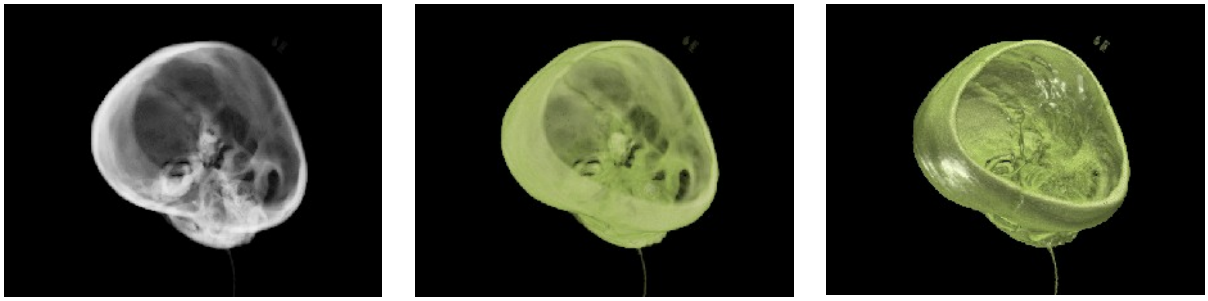
Vypracovali jsme systém pro vizualizaci objemových dat se všemi základními funkcemi podle návrhu. Systém pracuje s grafickými akcelerátory podporující standard Pixel Shader 2.0. Velikost vizualizovaných objemových dat je omezena pouze velikostí paměti akcelerátoru a výsledný výkon je dán zejména rychlostí jednotek Pixel Shaderů, v nichž probíhá převážná část veškerých výpočetních operací.

Efektivita systému na první testovací sestavě, zastupující minulou generaci grafických akcelerátorů a hardware, je podle předpokladu minimální a projevuje se silná závislost výkonu na počtu zobrazovaných řezů – to je dáno nízkým výkonem jednotek Pixel Shaderů. Systém je tedy na této sestavě možné reálně použít s daty do velikosti 256 x 256 x 256 voxelů. Paměťové nároky na grafický akcelerátor jsou v tomto případě 16 MB u 8 bitových vzorků, 32 MB u 16 bitových vzorků a 64 MB u 8 bitových vzorků s předpočítanými gradienty pro implementaci osvětlovacího modelu.

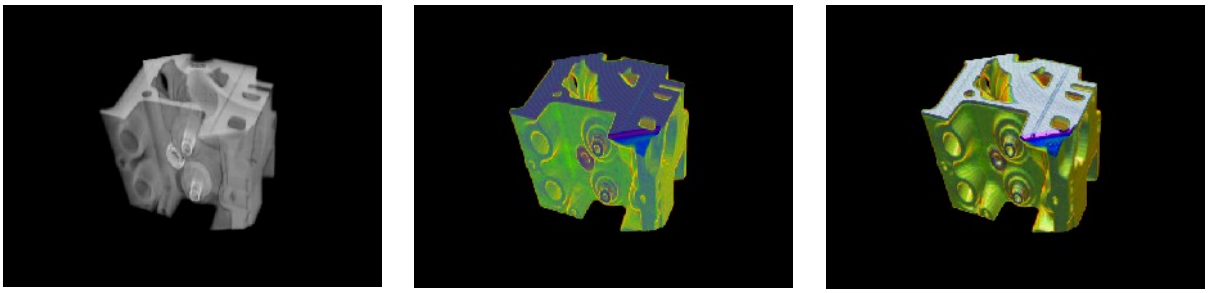
Efektivita systému na druhé testovací sestavě umožňuje plně interaktivní práci s objemovými daty velikosti 512 x 512 x 512 voxelů i se zobrazením pomocí vyššího počtu řezů pro lepší obrazovou kvalitu (512-1024 řezů). Paměťové nároky na zobrazení objemových dat 512^3 činí 128 MB s 8 bitovými vzorky, 256 MB s 16 bitovými vzorky a 512 MB s 8 bitovými vzorky a předpočítanými gradienty pro implementaci osvětlovacího modelu. S těmito daty jsou tedy ideálním způsobem využity schopnosti testovaného akcelerátoru nVidia s vysokým výkonem stream-procesorů použitých pro výpočet Pixel Shaderů a výborná propustnost pamětí. To je možné snadno ověřit přidáním dalších řezů a srovnáním výkonu – proti první testovací sestavě již nedochází k dramatickému poklesu efektivity a běžně je možná interaktivní práce kolem hranice 10 – 20 FPS v rozlišení 1280x1024.

Vytvořený systém byl průběžně testován s téměř 30 sadami objemových dat vzájemně se lišícími v počtu řezů, rozlišení vzorkování, ale i pořizovací metodou (CT scan a MRI). To umožnilo poměrně dobrou kompatibilitu a zachycení dílčích problémů vyskytujících se pouze u některých dat.

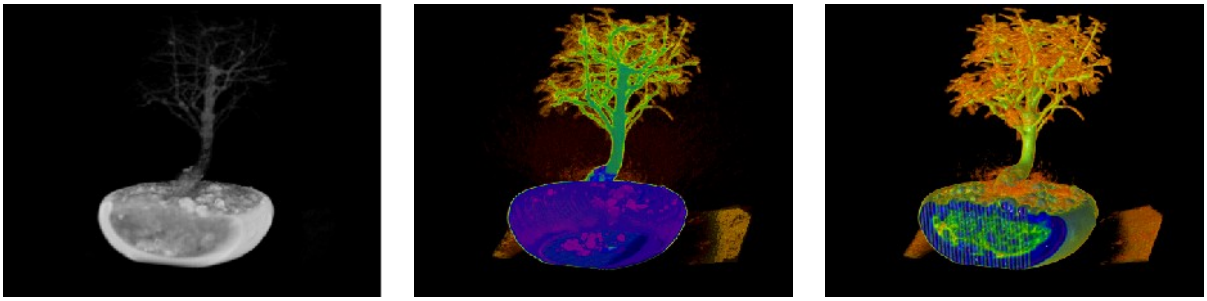
Dále jsou uvedeny vybrané ukázkové výstupy systému poskytující názorné shrnutí možností (veškerá testovací data jsou k dispozici na přiloženém DVD):



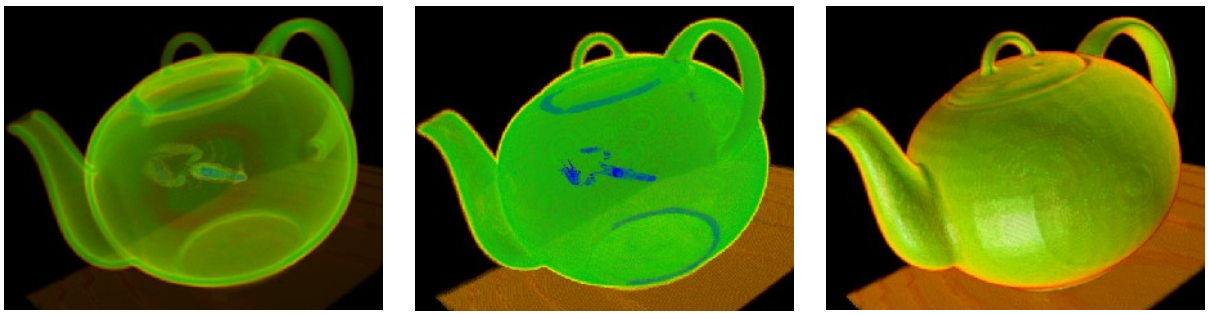
Testovací data: hlava dítěte (X-rays, X-rays + TF, X-rays + light) [BabyHead.raw]



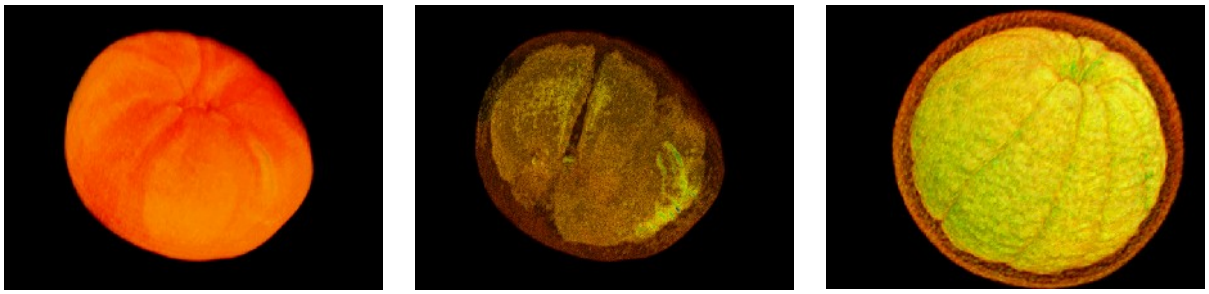
Testovací data: blok motoru (X-rays, X-rays + TF, X-rays + TF + light) [engine.raw]



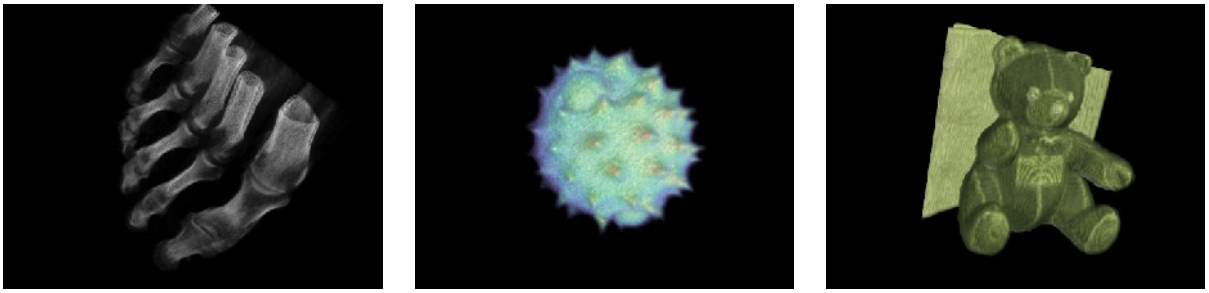
Testovací data: bonsai (X-rays, MIP, X-rays + light) [bonsai.raw]



Testovací data: čajová konvice s humrem (X-rays + TF, MIP, X-rays+TF+light) [BostonTeapot.raw]



Testovací data: rajče (X-rays), rajče (MIP), pomeranč (X-rays + light) [tomato.raw, orange.raw]



Testovací data: [foot.raw] [DaisyPollenGrain.raw] [teddybear_low.raw]

6 Závěr

Cílem tohoto projektu byl návrh a implementace testovacího systému pro vizualizaci objemových dat, který by nám umožnil snadnou manipulaci se zkoumanými vzorky, srovnávání různých vizualizačních metod (MIP, X-rays) a přenosových funkcí. Po této stránce realizovaný systém plně odpovídá všem požadavkům, což je zhodnoceno již v minulé kapitole shrnující dosažené výsledky.

Struktura práce navíc umožňuje velmi pohodlný náhled do světa perspektivní problematiky Volume Renderingu. Jakmile se čtenář seznámí s obecnou teorií utvářející neopominutelný základní přehled, jsou mu v části návrhu v logických krocích předkládány postupy vhodné pro řešení dílčích problémů. Nejpodstatnější části jsou dále detailně vysvětleny v implementační části práce.

Při tvorbě systému byl kladen důraz na maximální jednoduchost řešení, které každému dovoluje plně se soustředit na podstatné části kódu bez zbytečného odvádění pozornosti drobnými detaily, čímž se práce stává velmi dobře využitelnou také ve výuce.

Je ovšem třeba mít na paměti, že rozsah této práce nenabízí dostatek prostoru pro vysvětlení práce se všemi použitými prostředky (ať už jde o Open Scene Graph, rozhraní OpenGL nebo GL Shading Language). Jejich funkcionalita je proto zachycena pouze v nezbytně nutné míře a v případě hlubšího zájmu o pochopení všech principů je nutné využít další, výrazně obšírnější materiály.

6.1 Budoucí práce

V navrženém systému je pochopitelně ponechán značný prostor pro další rovoj. Vzhledem k tomu, že nejpalčivějším problémem jsou dnes u Volume Renderingu paměťové nároky, přímo se nabízí implementace dalších formátů pro uložení objemových dat v paměti. V tomto ohledu již další vývoj probíhá a v budoucnu bude práce jistě rozšířena o interní reprezentaci objemu v RGBA formátu textury, kde hodnota hustoty bude uložena na 16 bitech a gradient vektor na dalších 16 bitech (chybějící složka bude vyčíslována dynamicky v Pixel Shader programu).

Logicky následujícím krokem by mělo být zapouzdření implementované funkcionality do objektového modelu držícího se zvyklostí Open Scene Graph toolkitu, testování systému na podstatně vyšším počtu konfigurací dostupného hardware a odhlédneme-li od funkční stránky projektu, bude jistě vhodné vytvořit uživatelské rozhraní umožňující pohodlnější práci.

Literatura

[1] J. Žára, B. Beneš, J. Sochor, P. Felkel, *Moderní počítačová grafika,* Brno, Computer Press, 2004.

[2] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, T. Ertl, *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization,* Vienna University of Technology, Austria, 2000.

[3] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman, *High-Quality Volume Rendering Using Texture Mapping Hardware,* SIGGRAPH Eurographics Graphics Hardware Workshop, 1998.

[4] K. Engel, *Visualization of Volumetric Data on Consumer PC Hardware,* http://www.vis.uni-stuttgart.de/vis03_tutorial/ (3.5.2007).

[5] Cezary Bloch, *Rendering of multivariate 3D volume data*, Master Thesis, Sweden, Uppsala University, 2006.

[6] Fovia, Inc, http://www.fovia.com/ (3.5.2007).

Seznam příloh

Příloha 1.: Uživatelský manuál

Příloha 2.: DVD se zdrojovými texty, ukázkovými daty a elektronickou verzí práce

Příloha 1.: Uživatelský manuál

Parametry a přepínače příkazové řádky

`--raw XSlices YSlices ZSlices Bytes Components Endian FileName`

po spuštění načte soubor FileName obsahující objemová data o rozměrech `Xslices * Yslices * Zslices`. Parametr `Bytes` udává na kolika bytech je uložen jeden voxel (1 nebo 2), parametr `Components` (1 a vyšší) umožňuje načítání dat, v nichž se pro každý voxel ukládá více informací (pouze hodnota hustoty = 1). Parametr Endian určuje, jsou-li data uložena ve formátu low-endian (`low`) nebo big-endian (`big`).

`-s slices`

slices udává počet řezů, které budou vytvořeny. Výchozí hodnota je 256.

`--normalise`

hodnoty hustoty voxelů jsou před zobrazováním normalizovány. Ve výchozím stavu hodnoty nejsou normalizovány.

`--lighting`

předpočítat gradient vektory používané v osvětlovacím modelu (výsledný formát uložení voxelu v paměti bude RGBA8, RGB = gradient, A = 8 bit hodnota hustoty)

`--xClipMin cmin --xClipMax cmax`

nastavení ořezávání objemu na ose X (objemová kostka je vytvářena s rozměrem 1.0, výchozí `xClipMin = -0.5, xClipMax = 0.5`).

Ekvivalentní parametry jsou dostupné také pro ořezávání na ose Ya Z.

`--xMultiplier xmul, --yMultiplier xmul, --zMultiplier xmul`

parametry umožňují změnu koeficientu texturování v případě, kdy rozměry uložených voxelů nejsou v poměru 1 : 1 : 1. Výchozí hodnota 1.0 může být libovolně zvyšována, čímž dosáhneme zvýšení násobiče vzorkování na vybrané ose.

Př.: `--xMultiplier 1.28 --yMultiplier 2.56 --zMultiplier 1.0`

pro data s poměrem délek hran voxelu 0.78125 : 0.390625 : 1.

Ovládání programu za běhu

- **Rotace objemu** – stisknuté ***levé tlačítko + pohyb v ose X/Y.***
- **Přiblížení/oddálení objemu** – stisknuté ***pravé tlačítko + pohyb v ose X/Y.***
- **Posun objemu –** stisknuté ***kolečko myši + pohyb v ose X/Y***

- **Přepínání ořezávacích rovin** – stisk klávesy ***Shift* + *C*** mění ořezávací roviny v pořadí min x, max x, min y, max y, min z, max z. Jako výchozí je vybrána ořezávací rovina min x.
- **Změna polohy aktivní roviny –** *stisk klávesy* ***C* + *pohyb myši v ose Y*** *(poloha u horního okraje obrazu = 0,5; střed obrazu = 0,0; dolní okraj obrazu = -0,5)*

- **Přepínání programu pro Pixel Shader –** klávesa **M** (method)
  (programy jsou uloženy v adresáři *./shaders*, přepíná se mezi vrt0.frag – vrt9.frag)

- **Přepínání aktuální přenosové funkce –** klávesa **P** (palette)
  (palety jsou uloženy v adresáři *./palettes*, přepíná se mezi palette0.png – palette9.png)

- **Alpha Cut-off minimum –** klávesa ***A* + *pohyb myši v ose Y***
- **Alpha Cut-off maximum** – klávesa ***Shift* + *A* + *pohyb myši v ose Y***

- **Transparency Level** – klávesa ***T* + *pohyb myši v ose Y***