VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

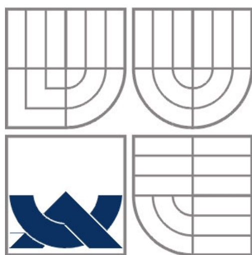
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZOVÁNÍ OBJEMOVÝCH DAT POMOCÍ PROGRAMOVATELNÉHO HW

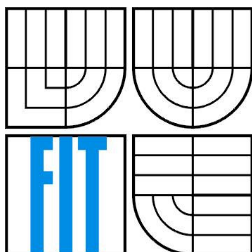
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Radovan Jošth
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

ZOBRAZOVÁNÍ OBJEMOVÝCH DAT POMOCÍ PROGRAMOVATELNÉHO HW

VOLUME RENDERING USING PROGRAMMABLE HW

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Radovan Jošth
AUTHOR

VEDOUCÍ PRÁCE — Ing. Adam Herout, Ph.D.
SUPERVISOR

BRNO 2007

Zadání diplomové práce/5902/2006/xjosth00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií   Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Jošth Radovan**

Obor: Výpočetní technika a informatika

Téma: **Zobrazování objemových dat pomocí programovatelného HW**

Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte a popište problematiku zobrazování objemových dat.
2. Prostudujte a popište možnosti soudobého grafického HW.
3. Prostudujte a srovnejte existující řešení zobrazující volumetrická data na programovatelném grafickém HW.
4. Navrhněte algoritmus zobrazování objemových dat umožňující zvýraznění "důležitých" jevů v datech.
5. Implementujte navržený algoritmus.
6. Vyhodnoťte vlastnosti implementovaného algoritmu na demonstračních datech.
7. 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:

- Bez požadavků.

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.

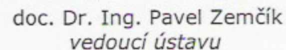
Vedoucí: **Herout Adam, Ing., Ph.D.**, UPGM FIT VUT

Datum zadání: 1. listopadu 2006

Datum odevzdání: 22. 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: **Radovan Jošth**
Id studenta: 22707
Bytem: Kynceľová 37, 974 01 Banská Bystrica
Narozen: 25. 09. 1982, Banská Bystrica
(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):
   diplomová práce

Název VŠKP: Zobrazování objemových dat pomocí programovatelného HW
Vedoucí/školitel VŠKP: Herout Adam, 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

Táto práca popisuje a implementuje metódu zobrazovania volumetrických dát. Hlavným účelom je vizualizovať nasnímané 3D dáta jednou zo súčasných metód pre získavanie 3D volumetrických dát. Takéto metódy sa veľmi často používajú v medicíne a chémii. Systém využíva na zobrazovanie dát programovateľný pipeline súčasných grafických kariet, ktorý umožňuje rýchle paralelné spracovanie veľkých objemov dát. V projekte je popísaný teoretický úvod do problematiky volumetrických dát, tiež obsahuje návrh systému a nakoniec popis implementácie. Výsledkom projektu je aplikácia ktorá s využitím OpenGL vykresľuje volumetrické dáta.

Kľúčové slová

Objemové dáta, Renderovanie do framebufferu, 3D textúrovanie, programovateľný hardware, CG programovací jazyk, Frame buffer object, Projekcia maximalnej intenzity, MRI, CT

Abstract

This work describes and implementing method for volume data rendering. Main purpose of this work is visualization of scanned 3D data with some current method used for the 3D volumetric scanning. The 3D volumetric scanning is mainly used in medicine and chemistry. System is using programmable pipeline of current graphic cards, which provides us fast parallel work with large volumetric data. This paper introduces some basics about the volumetric rendering and scanning, describes design and at the end, the implementation steps. Result of this project is application which renders volumetric data with OpenGL.

Keywords

Volume data, Render to framebuffer, 3D texturing, programmable hardware, CG programming language, Frame buffer object, Maximal intensity projection, MRI, CT

Zobrazovanie objemových dát pomocí programovateľného HW

Prehlásenie

Prehlasujem, že som tuto diplomovú prácu vypracoval samostatne pod vedením Ing. Adama Herouta, Ph.D..
Ďalšie informácie mi poskytol Ing. Přemysl Kršek Ph.D. a Ing. Michal Španěl.
Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……………………
Radovan Jošth
23.4.2007

Poďakovanie

Na tomto mieste by som chcel vyjadriť poďakovanie Ing. Adamovi Heroutovi, Ph.D., ktorý mi poskytol nevyhnutné informácie a rady pri riešení tohto projektu a bez ktorého pomoci by dokončenie práce trvalo oveľa dlhšie. Tak isto by som chcel poďakovať aj Ing. Přemyslovi Krškovi, Ph.D. a Ing. Michalovi Španělovi za poskytnuté konzultácie.

© Radovan Jošth, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod ... 5
2 Získavanie objemových dát ... 7
2.1 CT – Computed Tomogrpahy ... 7
2.2 MRI – Magnetic Resonance Imaging ... 8
2.3 Laser scanner ... 10
2.3.1 Time-of-flight ... 10
2.3.2 Triangulation ... 10
2.4 Structured light ... 11
2.5 Stereoskopická metóda ... 11
2.6 Siluetová metóda ... 11
3 Zobrazovanie 3D dát ... 13
3.1 Volume ray casting ... 13
3.2 Splatting ... 14
3.3 Shear warp ... 15
3.4 Texture mapping ... 15
3.5 Hardwarovo urýchľovaný rendering ... 16
3.6 Shadery ... 17
3.6.1 Vertex Shader ... 19
3.6.2 Pixel Shader ... 20
3.7 FBO – Framebuffer object ... 22
4 Návrh ... 23
5 Implementácia ... 26
5.1 Vstupné dáta ... 26
5.2 Predpríprava dát ... 28
5.3 Renderovanie dát ... 30
5.3.1 Príprava textúr ... 30
5.3.2 Kompilácia CG shaderov ... 32
5.3.3 Zostavenie vrstiev objemových dát ... 34
5.3.4 Renderovanie jednotlivých vrstiev ... 36
5.3.5 Postprocessing efekty ... 37
6 Dosiahnuté výsledky ... 39
6.1 Rýchlosť vykresľovania ... 40
6.2 Kvalita vykresľovania ... 42
7 Záver ... 44

Príloha 1.: Manuál 47
Inštalácia..47
Príkazový riadok 47
Ovládanie 48
Príloha 2.: Ukážky (screenshots) 52

Obrázky

Obrázok 1: CT skener 7
Obrázok 2: MRI skener 8
Obrázok 3: Triangulačný skener 10
Obrázok 4: Structure light metóda 11
Obrázok 5: Stereo fotografia 11
Obrázok 6: Stereo fotoaparát 11
Obrázok 7: Siluetové modelovanie 12
Obrázok 8: Ray Casting 14
Obrázok 9: Splatting 14
Obrázok 10: Shear warp 15
Obrázok 11: Texture Mapping 15
Obrázok 12: NV30 blokový diagram 16
Obrázok 13: Grafický pipelne 18
Obrázok 14: Rastrové operácie 18
Obrázok 15: Vertex Shader (SM 2.0) 20
Obrázok 16: Pixel Shader 21
Obrázok 17: Framebuffer object 22
Obrázok 18: Blend 23
Obrázok 19: MIP 23
Obrázok 20: Blend+MIP 23
Obrázok 21: Vrstvenie - Texture Mapping 24
Obrázok 22: Pixel shader algoritmus 25
Obrázok 23: Dátová matica 27
Obrázok 24: c3DTexture objekt 27
Obrázok 25: Algoritmus predprípravy dát 28
Obrázok 26: Výpočet normál 29
Obrázok 27: Objekt cMask 29
Obrázok 28: Objekt cVolumeObject 29
Obrázok 29: cTextureManager::AddToControl 32
Obrázok 30: cTextureManager::DeleteFromControl 32
Obrázok 31: cTextureManager::InvalidateAll 32
Obrázok 32: Triedy použitých shaderov 33

Obrázok 33: Proces prechodu shadermi 34
Obrázok 34: Výpočet vrstvy objemu dát 36
Obrázok 35: Postprocessing 37
Obrázok 36: Závisloť FPS na Quality class 40
Obrázok 37: Závislosť zaťaženia GPU na počte texelov 41
Obrázok 38: Kvalita obrazu (a: *QC=0.5*; b: *QC=1.0*) 43
Obrázok 39: Vplyv QualityClass parametru 43
Obrázok 40: Po spustení... 48
Obrázok 41: Rozloženie ovládania 49
Obrázok 42: Color Adjuster 49
Obrázok 43: Maskovanie objektu 50

Tabuľky

Tabulka 1: Príklady radio-hustoty niektorých látok 8
Tabulka 2: Testovací hardware 39
Tabulka 3: Framerate závislosť na Quality Class 40
Tabulka 4: Blend Metoda - benchmark 41
Tabulka 5: Minimálna a doporučená konfigurácia HW 47

1 Úvod

Moderná medicína v súčasnosti využíva rôzne metódy na zisťovanie rôznych porúch a chorôb v materiáloch. Za posledné obdobie sa vyvinulo niekoľko metód, ktoré dokážu vytvoriť digitálnu kópiu povrchu, alebo objemu skenovaného objektu. Tento projekt je zameraný hlavne na zobrazovanie dát z CT (Computed Tomography) metódy a MRI (Magnetic Resonance Imaging) metódy, avšak je ho možné použiť aj pri inej metóde 3D skenovania objektu, ako je napríklad LaserScanner, alebo špeciálne mikroskopy. Tieto metódy dokážu vytvoriť 3D snímky, ktoré je možné reprezentovať 3D maticou dát, ktorá obsahuje dáta namerané danou metódou. Tato 3D matica dát predstavuje 3D textúru, ktorá je postupne vykresľovaná po plochách s určitou priesvitnosťou a tým je docielená transparentná rekonštrukcia originálnych dát.

Pri vykresľovaní 3D textúry je nutné reprezentovať namerané dáta farebnými hodnotami. Na toto je veľmi vhodné použiť programovateľné procesory, ktoré sú bežne dostupné na grafických kartách. Tieto procesory je možné naprogramovať tak, aby reprezentovali 3D dáta textúry podľa potreby v reálnom čase. Značnou výhodou programovateľných procesorov grafickej karty, oproti bežným procesorom, je schopnosť pracovať paralelne s vektorovými dátami – MIMD architektúra.

V prvej kapitole si povieme o tom, ako môžeme získavať volumetrické dáta. Popíšeme si niektoré bežne používané spôsoby ich získavania, ich výhody a nevýhody. Niektoré z metód slúžia na získavanie len povrchových dát objemu a niektoré získavajú dáta aj z vnútra objemu, tie budú rozobrané hlbšie.

Tretia kapitola popisuje renderovacie metódy volumetrických dát. Týchto metód je viac, niektoré sú vhodné pre renderovanie klasickým CPU, niektoré sa zase dajú veľmi dobre paralelizovať, prípadne vieme v nich využiť súčasný hardware a jeho možnosti.

V štvrtej kapitole si popíšeme čo je to programovateľný pipeline. V súčasnosti sa veľa prostriedkov sústredilo do jeho rozvoja na grafickej karte. Jeho vysoký paralelizmus a iné výhody nám prinášajú nové možnosti pri spracovaní grafických úloh. Pri jeho použití už nepoužívame fixný pipeline, to nám umožňuje obchádzanie rôznych obmedzení, ktoré vyplývali z fixného grafického pipelinu.

V piatej kapitole si popíšeme tzn. FBO – Framebuffer Object. Toto rozšírenie umožnilo grafickým kartám renderovať do textúry a urýchlilo tak postprocessing operácie na výstupnom obraze. Toto rozšírenie je tiež vhodné pre obchádzanie klasického Alpha Blendingu a naprogramovanie si vlastnej blendovacej funkcie.

Šiesta kapitola bude hovoriť o návrhu systému. Tu si povieme aké princípy využijeme pri renderovaní, ako budú pospájané jednotlivé bloky na spracovanie dát a tiež tam bude popísaný algoritmus Pixel shaderu na vytváranie obrazu rôznymi metódami.

Siedma kapitola bude o implementácii a budú tu popísané základné formulácie algoritmov a ich vnútorný princíp. Bude tu popísané rozdelenie renderovacieho cyklu na bloky, v ktorých sa pripravujú textúry a shadery. Ďalej sa delia na bloky, ktoré budú počítať a renderovať objemové dáta a nakoniec ich upravovať v postprocessingu.

Posledné dve kapitoly búdu hovoriť o dosiahnutých výsledkoch a prípadnej náväznosti na túto problematiku a na tento projekt. Vo výsledkoch bude popísaná kvalita výsledného obrazu a tiež aj rýchlosť renderovania rôznymi metódami.

V tejto práci je použitých veľa anglických výrazov v texte a v obrázkoch, ktoré nemajú vhodný slovenský, alebo česky ekvivalent. Preto pre zachovanie prehľadnosti a jasnosti textu neboli preložené. Pre bližšie informácie o daných slovách je vhodné si preštudovať danú problematiku. Ďakujem za pochopenie.

2 Získavanie objemových dát

Aby sme mohli zobrazovať nejaké dáta musíme ich najprv vytvoriť. Vytváranie (modelovanie) dát (objektov) sa dá robiť rôznymi metódami, ktoré už načítavajú reálny objekt, alebo modelujeme objekt priamo v počítači na základe nejakého modelovacieho softwaru.

V tejto kapitole si ukážeme niektoré metódy, ktoré modelujú objekt už z reálneho objektu. Niektoré z týchto metód dokážu nasnímať objem celého objektu, teda aj vnútornú štruktúru, nielen povrch objektu. Metódy, ktoré toto dokážu si popíšeme hneď na začiatku, lebo sú najvhodnejšie pre vytváranie 3D objemových dát. Práva z týchto metód je CT – Computed Tomography, ktorá sa využíva hlavne v medicíne a je určená hlavne na snímanie tvrdších materiálov. Ďalšou metódou bude MRI – Magnetic Resonance Imgaing, táto metóda je tiež veľmi používaná v medicíne. MRI je určené hlavne na snímanie mäkkých tkanín a má nedeštruktívne účinky na rozdiel od CT.

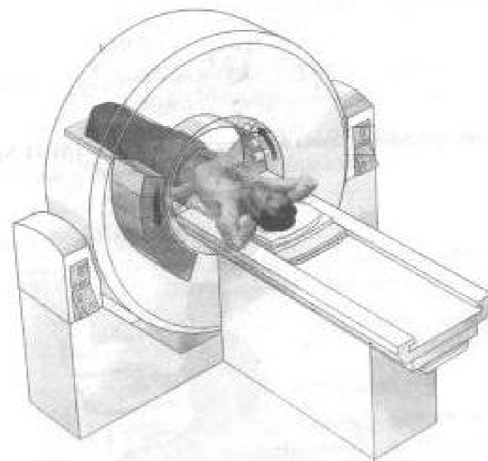
Ďalšie metódy snímania 3D objektov, ktoré si popíšeme dokážu snímať iba povrch objektov. Sú preto skôr vhodnejšie na objekty, ktorých vnútorná štruktúra nie je podstatná a môžeme ju preto zanedbať. Objekty takto nasnímané je možné triangulovať a vytvoriť z nich trojuholníkový model, ktorý môžeme zobrazovať aj klasickými metódami pre zobrazovanie trojuholníkov, ktoré sú podstatne rýchlejšie.

Tento projekt zobrazuje objem dát ako body a nie ako trojuholníky a preto je hlavne určený na zobrazovanie dát z CT a MRI. Nie je však vylúčene ho použiť aj na zobrazovanie dát získaných inými metódami.

2.1 CT – Computed Tomogrpahy

Computed tomography je metóda, ktorou dostaneme 3D obraz zlúčením veľkého množstva 2D röntgenových obrázkov vytvorených rotáciou okolo jednej osi.

Pôvodne CT dokázali získavať dáta v rovine kolmej na os snímania alebo v rovine pozdĺžnej s osou snímania. Moderné CT tomografy dokážu vytvárať snímky v rôznych rovinách, alebo tiež vytvoriť jeden 3D volumetrický snímok, ktorý môžeme následne spracovávať cez požadovaný software.



Obrázok 1: CT skener

CT vytvára jednotlivé snímky použitím ionizujúceho röntgenového žiarenia, ktoré je generované žiaričom, ktorý rotuje okolo skenovaného

objektu. Röntgenový detektor je na opačnej strane rotora. Skenované hodnoty sú automaticky prepočítavané na 2D snímok metódou nazývanou tomographic reconstruction.

Dáta reprezentujú rôznu rádio-hustotu vypočítanú z hladinu útlmu röntgenového žiarenia v skenovanom objekte. Rádio-hustota sa meria v Hounsfieldovej stupnici a má jednotku HU. V tabuľke Tabulka 1 môžeme vidieť príklady rádio-hustoty niektorých látok.

Látka	HU
Vzduch	-1000
Tuk	-120
Voda	0
Svaly	40
Kosti	1000

Tabulka 1: Príklady radio-hustoty niektorých látok

CT je metóda, ktorá vyžaruje pomerne silnú dávku röntgenového žiarenia. Aj keď sa CT metóda vyvíjala pomerne efektívne a zvyšovala sa kvalita senzorov, tým pádom nebolo nutné také silné žiarenie, tak napriek tomu sú požadované vyššie rozlíšenia od tejto metódy a sú požadované aj oveľa komplexnejšie snímky a tieto požiadavky implikujú použitie väčšej intenzity röntgenového žiarenia. Tieto požiadavky teda vylúčili možnosť zníženia rizika pri skenovaní živých tkanín na vznik nádorov. Samozrejme sila žiarenia a s tým spojené riziko závisí na veľa faktoroch, ako sú objem požadovaného snímku, počet a typ skenovaných sekvencii a tiež aj stavba pacientovho tela. CT metóda je pre deti nebezpečná, lebo môže spôsobiť v neskoršej dobe rakovinu, preto ak je nutné použiť CT snímkovanie je nutné nastaviť ho tak, aby dieťa bolo vystavené oveľa menšej dávke žiarenia.

2.2 MRI – Magnetic Resonance Imaging

Magneticka rezonancia je metóda podobná CT metóde avšak nepoužíva nebezpečné röntgenové žiarenie.

Obrázok 2: MRI skener

MRI je známe aj pod názvom MRT ako Magnetic Resonance Tomography, alebo v chémii ako NMR – Nuclear Magnetic Resonance.

MRI je hlavne používané v medicíne na zisťovanie patologických alebo fyziologických zmien živých tkanín. Okrem medicínskych účelov sa MRI používa na zisťovanie permeability skál na základe hydrouhlíkov a ako nedestruktívna metóda skenovania sa používa na zisťovanie kvality potravín, alebo iných surovín ako napríklad drevo.

Medicínske MRI obvykle dosahuje 0,3T-3,0T, výskumne MRI skenery dosahujú až do 10T a špeciálne malé MRI skenery až 20T. Magnetické pole Zeme ma silu 50µT.

Princíp MRI skenera spočíva na relaxačných vlastnostiach vybudených jadier atómov vodíka vo vode a lipidoch. Keď je objekt skenovaný je vložený do silného uniformného magnetického pola a jadrá atómov so spinovým číslom rôznym od nuly sú usmernené vzhľadom na magnetické pole podľa kvantovej mechaniky. Jadrá atómu vodíka majú spinové číslo ½ a preto sa usmernia paralelne alebo anti-paralelne so smerom magnetického pola.

Spinová polarizácia určuje silu MRI signálu. Pre protóny určuje polarizácia rozdiel populácie dvoch energetických stavov, ktoré sú viazané na paralelnú, alebo anti-paralelnú orientáciu protónových spinov v magnetickom poli. Pri magnetickom poli o sile 1,5T je len veľmi malý rozdiel medzi počtom paralelne a anti-paralelne orientovanými jadrami vodíka, avšak vo väčšom objeme už rozdiel týchto paralelne a anti-paralene orientovanými jadrami spôsobí merateľnú odchýlku v magnetickom poli na základe vysielaného RF signálu na Larmorovej frekvencii.

MRI skener sa skladá z troch hlavných komponentov:

1. Magnet vytvárajúci statické magnetické pole
2. RF vysielač a prijímač
3. Nastaviteľné ortogonálne magnetické gradienty

Magnet vytvárajúci statické magnetické pole je najväčší a najdrahší komponent na MRI skenery. Okrem sily magnetického pola, je aj veľmi podstatná presnosť prevedenia (priamosť siločiar v strede magnetu, jeho presný stred, ...). V praxi sa používajú tri typy magnetov.

Permanentný magnet je používaný len v slabých MRI skeneroch do 0,4T a vyrába sa z feromagnetických materiálov. Jeho hmotnosť dosahuje radovo 100 ton. Presnosť takéhoto magnetu nie je veľmi dobrá, ale napriek tomu po inštalácii nie je nutné robiť drahý servis. Takýto magnet je nebezpečný pre kovové predmety v jeho okolí, ktoré môže na seba pripútať a je veľmi obťažne ich oddeliť.

Ďalším typom magnetu je elektromagnet, ktorý pozostáva zo selenoidu, na ktorom je navinutý medený drôt. Výhodou takéhoto magnetu je jeho nízka cena na výrobu. Nevýhodou je jeho zlá stabilita magnetického pola a na jeho operáciu je nutná elektrická energia, ktorá môže predražiť jeho operatívnosť. Tento typ magnetov je zastaraný.

Tretím a zrejme najpoužívanejším je supravodivý elektromagnet. Keď je zliatina nobium-titanium chladená tekutým héliom na 4K (-269ºC), tak sa stáva supravodivým vodičom a nekladie odpor elektrickému prúdu. Takto je možné použiť supravodič na vytvorenie silného a stabilného magnetického pola. Konštrukcia takýchto magnetov je veľmi drahá a tiež aj udržovanie hélia v kryo stave je veľmi zložité a drahé. Na chladenie hélia sa používa tzv. cryocooler.

RF vysielač sa skladá z RF syntetizátora, zosilňovača a vysielacej cievky. V high-end systémoch dokáže vysielať trvale výkon, až 1kW. RF prijímač pozostáva z prijímacej cievky, predzosilňovača a signálového procesoru.

Magnetické gradienty sú tri cievky ortogonálne umiestnené v x,y,z smeroch MRI skenera. Tieto cievky vytvárajú plynulú zmenu magnetického pola v MRI skeneri a tým dovoľujú nastaviť plochu vytvárania snímku.

2.3 Laser scanner

2.3.1 Time-of-flight

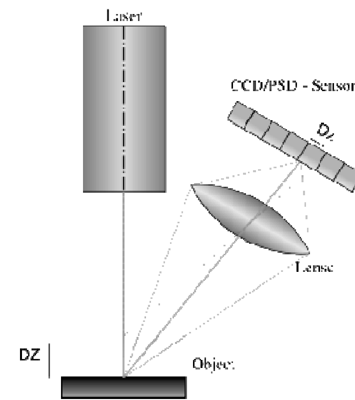
Tento skener je aktívny skener, ktorý používa na meranie laserové svetlo. Srdcom tohto skeneru je tzn. Laser Range Finder, ktorý zisťuje vzdialenosť povrchu na základe času letu svetelného pulzu. Keď poznáme čas letu laserového pulzu môžeme na vypočítať vzdialenosť letu, ktorá je dvojnasobná, skutočnú vzdialenosť dostaneme po vydelení dvomi.

$$d = \frac{c \times t}{2}$$

Presnosť tejto metódy je závislá na tom, ako presne dokážeme merať čas (vzdialenosť 1mm preletí svetlo za 3,3ps). Laserový skener dokáže zmerať maximálne jeden bod naraz, avšak je možné skenovať 10000-100000 bodov za sekundu. Aby sme zo skenovali celý objekt môžeme rotovať laserový skener, objekt, alebo zrkadlá umiestnené okolo objektu.

2.3.2 Triangulation

Triangulačný skener je tiež typ aktívneho skeneru, využíva laserové svetlo na detekciu povrchu objektu. Spôsob skenovania spočíva vo vyžarovaní laserového svetla na povrch a snímanie nasvieteného bodu kamerou. Na snímači kamery sa bod objavuje na rôznych miestach podľa toho ako je bod umiestnený vzhľadom na optický systém skeneru. Triangulačným systémom to nazývame preto lebo laserový žiarič, nasvietený bod, kamera tvoria trojuholník. Vzdialenosť žiariča laseru od kamery je známa, tiež uhol kamery vzhľadom smeru vyžarovania je známy, na základe pozície bodu na snímači kamery. Na základe týchto informácii môžeme jednoducho vypočítať pozíciu objektu vzhľadom na skener. Ak by sme použili viac laserových žiaričov s rôznymi parametrami mohli by sme skenovať viac bodov naraz.

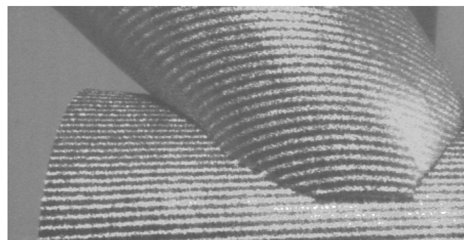


Obrázok 3: Triangulačný skener

Existuje podobná metóda, ktorá využíva conoskopický kryštál na nasmerovanie odrazeného laserového paprsku do senzora, kde potom na základe difrakčného obrazca určíme vzdialenosť snímaného povrchu. Výhodou tohto systému je že využíva len jednu líniu laseru a teda je možné merať hlboké diery.

2.4 Structured light

Spôsob Structure light využíva premietanie známeho obrazcu na skenovanú plochu. Potom spätným nasnímaním obrazcu zistí zakrívenie a vypočíta tvar povrchu metódou podobnou triangulácii. Obrazec môže byť buď 1D, alebo 2D. Jednoduchým príkladom 1D obrazcu je obyčajná čiara, 2D obrazec môže byť mriežka s určitou hustotou. Na vytvorenie obrazcu sa obvykle využíva LCD projektor. Na 1D čiare sa pozoruje zakrivenie z ktorého sa určí poloha bodov. Pri 2D obrazcoch sa používa zložitý algoritmus na detekciu zakrivenia.



Obrázok 4: Structure light metóda

Pri prudkých zmenách tvaru, alebo dier v skenovaných povrchoch môže dochádzať k vynechaniu týchto častí, alebo ovplyvnenia celého výsledku.

Tento typ skenerov je veľmi rýchli kvôli schopnosti skenovania celého povrchu naraz, avšak je ešte stále vo vývoji.

2.5 Stereoskopická metóda

Stereoskopická metóda je založená na systéme ľudského vnímania priestoru. Využíva dve kamery, ktoré sú posunuté od seba a na základe rozdielu v snímaných obrazoch sa dá vypočítať vzdialenosť jednotlivých bodov. Touto metódou nemôžeme snímať vnútro objektov, ale len ich obálku. Ak by sme chceli vymodelovať celý objekt, museli by sme ho nasnímať z viacerých strán a výsledný objekt potom spojiť. Táto metóda nie je veľmi vhodná na modelovanie 3D objektov, avšak je dobrá pre fotografie.

Obrázok 6: Stereo fotoaparát

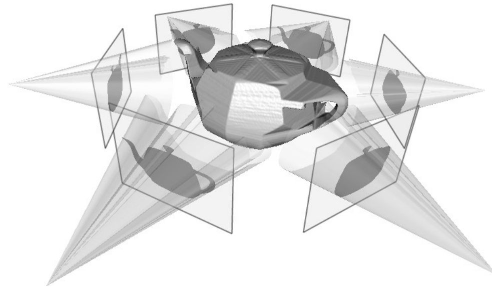
Obrázok 5: Stereo fotografia

2.6 Siluetová metóda

Táto metóda funguje na základe rozpoznávania siluety objektov v jednotlivých obrázkoch získaných jednou kamerou, alebo jedným fotoaparátom. Jednotlivé obrázky sa porovnávajú a vyhľadávajú sa

vzájomné zmeny (posuny) jednotlivých objektov, ktoré určujú ich vzájomnú vzdialenosť. Často túto metódu nepoužívame, lebo nie vždy je jednoduché rozpoznať siluety objektov.

S touto metódou dokážeme veľmi rýchlo vymodelovať veľké objemy dát. Nedá sa však použiť na modelovanie objemu dát, ale len na povrch objektov.



Obrázok 7: Siluetové modelovanie

3 Zobrazovanie 3D dát

V predošlej kapitole sme sa dozvedeli, akým spôsobom môžeme získať 3D dáta. V tejto kapitole si povieme niektoré základné možnosti ich zobrazenia. Každý z nasledujúcich spôsobov môžeme doplniť o množstvo vylepšení, tu si však popíšeme len základné princípy.

Dávnejšie v histórii počítačov bolo veľmi zložité vykresľovať väčšie objemy dát, kvôli nedostatku pamäti a nedostatku výpočtovej sily, ktorá by tento objem dokázala v reálnom čase spracovať. Postupom času sa začali vyvíjať rôzne špecializované platformy, ktoré boli určené na spracovanie takýchto dát, avšak ich cena bola veľmi vysoká. Okrem týchto špecializovaných počítačov sa začali vyvíjať nové procesory na spracovanie obrazu – grafické karty. Hlavným rysom týchto procesorov bol ich vysoký paralelizmus. Boli navrhnuté tak aby dokázali vykresľovať veľký počet trojuholníkov a zároveň ich textúrovať. Nároky na tieto grafické karty postupne vzrastali a ich špeciálne procesory sa stali programovateľnými. Grafické karty dospeli do štádia, kedy sa dajú aj zložité, alebo pôvodne časovo náročné výpočty vykonávať pomerne rýchlo vďaka vysokému paralelizmu a toto umožnilo renderovanie veľkých objemov dát.

Samozrejme aj v súčasnosti existujú špeciálne počítače na plnenie podobných úloh, ale ich cena je pre bežného človeka pomerne vysoká. Použitie bežného procesoru je síce tiež možné, ale jeho rýchlosť je vo veľa prípadoch nedostačujúca.

Pri renderovaní 3D textúry je nutne si stanoviť niekoľko parametrov, ktoré súvisia z projekciou 3D dát do 2D priestoru. Prvým parametrom je pozícia kamery v 3D priestore vzhľadom na renderovaný objem dát. Ďalším parametrom je tzv. transfer funkcia, ktorá určuje pre každý typ voxelu (dát) určitú farbu a priesvitnosť. Táto transfer funkcia je obvykle implementovaná tak, že mapuje RGBA hodnoty na reálne čísla, ktoré reprezentujú dáta. Okrem týchto dvoch parametrov je možné si zvoliť pozíciu svetiel, ktoré budú osvetľovať dáta.

3.1 Volume ray casting

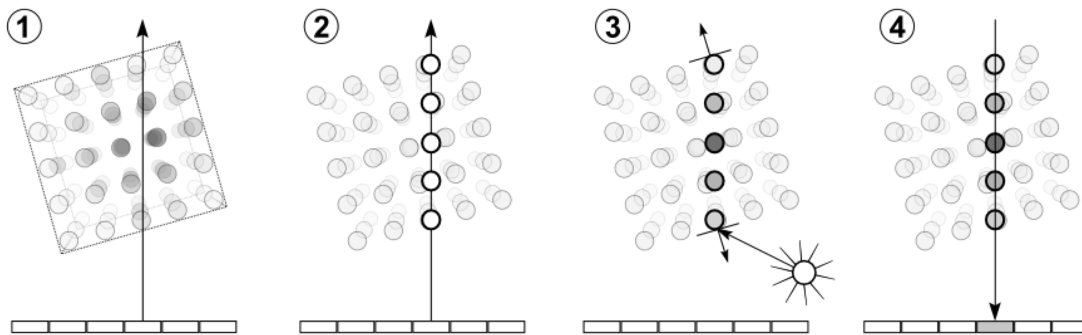
Najzákladnejším spôsobom, ako vykresľovať objem dát je takzvaný Ray Casting. Tento spôsob funguje na princípe premietania lúča objemom v určitom smere a projektovaním farby na tientitko kamery.

Lúč obvykle začína v strede kamery, pokračuje cez rovinu tienitka kamery do objemu dát a postupne ako prechádza dátami tak zhromažďuje hodnoty objemu, aplikuje na ne transfer funkciu a hodnoty RGBA spája dokopy definovanou funkciou. Vysielaný lúč je orezaný na objem dát, aby sa ušetril čas výpočtu. Lúč je samplovaný lineárne, body v okolí sa lineárne interpolujú a pre každý fragment sa pomocou transfer funkcie vypočíta RGBA hodnota, ktorá sa potom akumuluje do RGBA

hodnoty nesenej lúčom. Na konci algoritmu sa RGBA hodnota naakumulovaná lúčom premietne ako jeden pixel na tienitku kamery. Tento algoritmus sa vykonáva pre každý bod na tienitku.

Algoritmus (Obrázok 8):

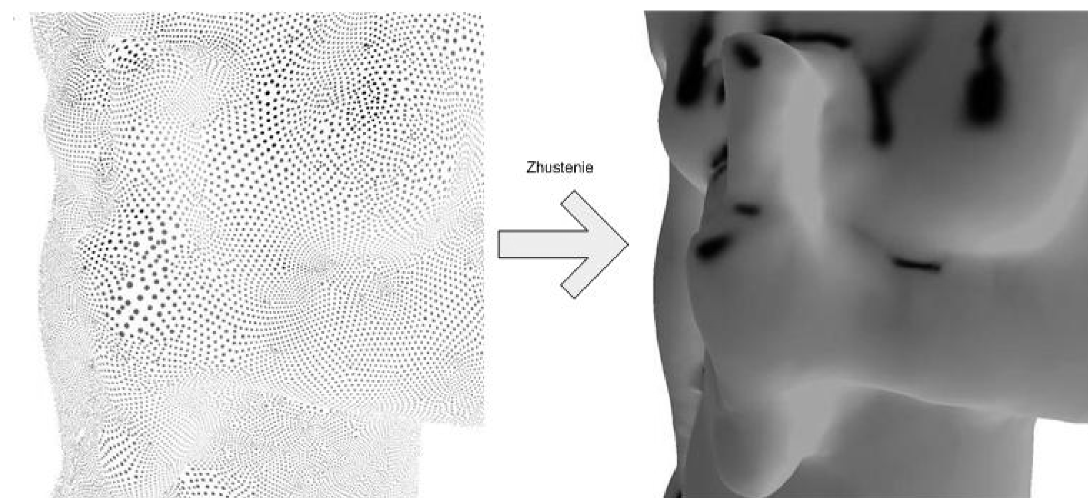
1. Lúč orezávaný okrajmi volumetrického telesa
2. Samplovanie lúča
3. Interpolácia okolia farby a aplikovanie osvetľovacieho modelu
4. Vrátenie naakumulovanej RGBA hodnoty



Obrázok 8: Ray Casting

3.2 Splatting

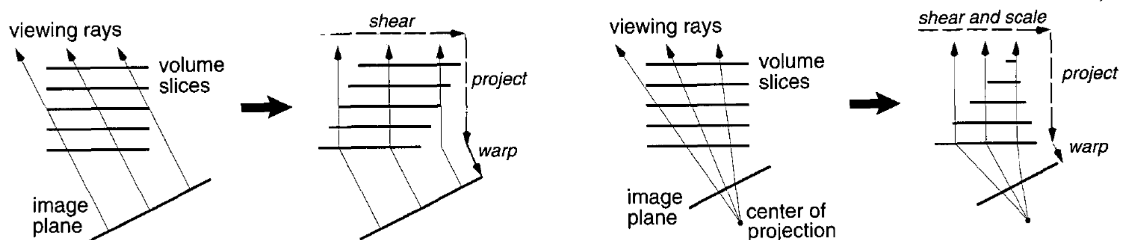
Táto technika vykresľovania volumetrických objemových dát spočíva vo vykresľovaní malých plôch, ktoré sú natočené kolmo k pozorovateľovi, teda ku kamere. Parametre týchto plôch, farba a priesvitnosť, sa menia diametrálne v normálnom gaussovom rozložení.



Obrázok 9: Splatting

3.3 Shear warp

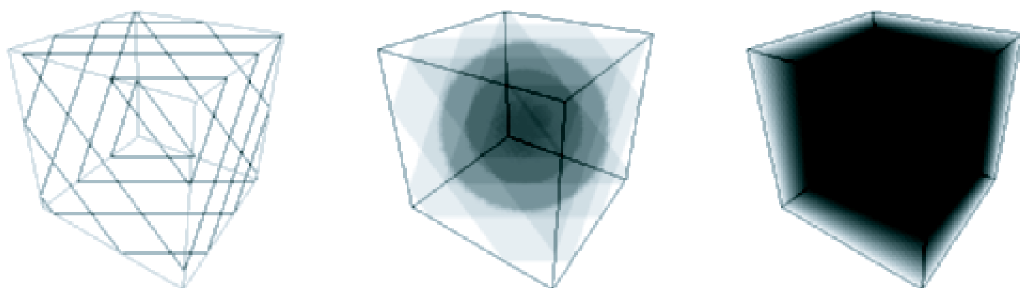
Metóda Shear warp spočíva v transformácii pohľadu tak, aby sa jednotlivé vrstvy zarovnali z osou. Každá vrstva má jednotný pomer veľkosti voxelu k veľkosti premietaného bodu na celej ploche vrstvy. Keď máme takto zrovnané vrstvy, môžeme na ne aplikovať posun, okrem posunu je možné aplikovať aj zväčšenie resp. zmenšenie v prípade, že sa jedna o perspektívnu projekciu. Keď máme takto nastavené vrstvy, tak ich postupne vyrenderujeme do off-screen bufferu, ktorý vo výsledku zdeformujeme do roviny pohľadu.



Obrázok 10: Shear warp

3.4 Texture mapping

Veľa 3D grafických systémov používa mapovanie textúr na geometrické objekty. Bežne dostupné grafické karty sú veľmi rýchle v mapovaní textúr a vedia efektívne renderovať vrstvy 3D objektu. Tieto vrstvy sa zarovnávajú, tak aby boli kolmé k pozorovateľovi. Na tieto vrstvy sa potom mapujú textúry s určitou priesvitnosťou. Avšak, aby sme mohli efektívne mapovať 3D textúru na jednotlivé vrstvy, potrebujeme hardware, ktorý podporuje 3D textúry a dokáže robiť rezy týmito textúrami. Ak by hardware nepodporoval 3D textúry, museli by sme vytvárať rezy 3D textúrou priamo na CPU čo by podstatne spomalilo renderovanie.



Obrázok 11: Texture Mapping

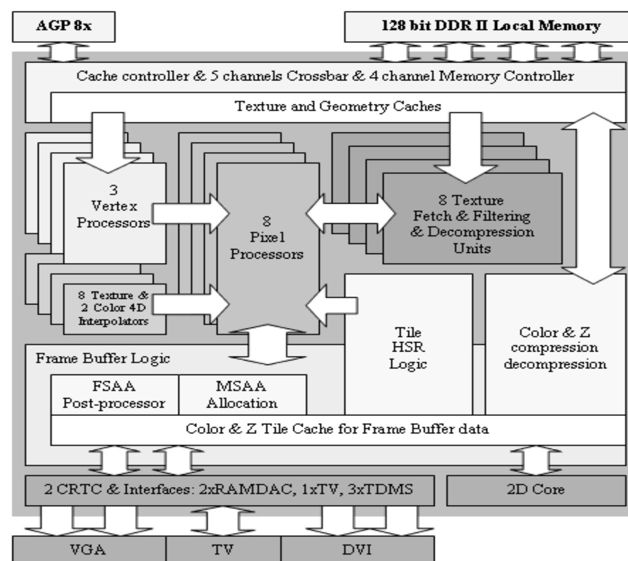
3.5 Hardwarovo urýchľovaný rendering

Súčasné metódy vykresľovania volumetrických dát sú vo väčšine prípadov založené na predošlých štyroch technikách vykresľovania, avšak aby mohli fungovať dostatočne rýchlo tak musia využívať súčasný hardware. Súčasný hardware ponúka veľa hardwarových urýchlení pri výpočtoch, ale hlavné vysoký paralelizmus.

Za posledné roky sa zvyšovaním frekvencie značne zrýchlil hardware, ale frekvencia nie je vždy dostačujúca. Hlavnými zmenami bolo pridávanie výpočtových jednotiek, ktoré dokážu spracovávať viacej streamov naraz. Paralelizmus je síce veľmi efektívny prostriedok na zlepšenie rýchlosti, ale často krát je potrebné riešiť zložité výpočty a tie sa nedali na aktuálnom hardwari vykonávať. Prípady keď sa nedalo niečo riešiť hardwarovo sa vyriešili pridaním programovateľných procesorov, tzv. Pixel Shader a Vertex Shader. Tieto procesory otvorili dvere celej škále nových algoritmov. Pixel Shader je veľmi užitočný hlavne pri renderovaní volumetrických dát, lebo sa dá využiť ako transfer funkcia a tiež ako blender.

Okrem programovateľných procesorov pribudlo na grafických kartách viacej textúrovacích jednotiek, ktoré veľmi rýchlo a efektívne dokážu vybrať z textúry požadované dáta. Toto umožnilo texúrovať viacej bodov naraz, alebo využívať viacej textúr naraz.

V súčasnej dobe procesor grafickej karty využíva technológiu MIMD, čo umožňuje spracovávať viacej inštrukcií a viacej dát naraz, okrem toho procesor grafickej karty je plne programovateľný, čo nám dáva veľkú silu v programovaní algoritmov priamo na grafickej karte s vysokou paralelizáciou. Algoritmy ako Ray Casting sa teda dajú bez problémov riešiť grafickou kartou a hlavný procesor počítača (CPU) už funguje len na manažovanie celého systému. Tiež Texture Mapping metóda je veľmi rýchla vďaka textúrovacím jednotkám.



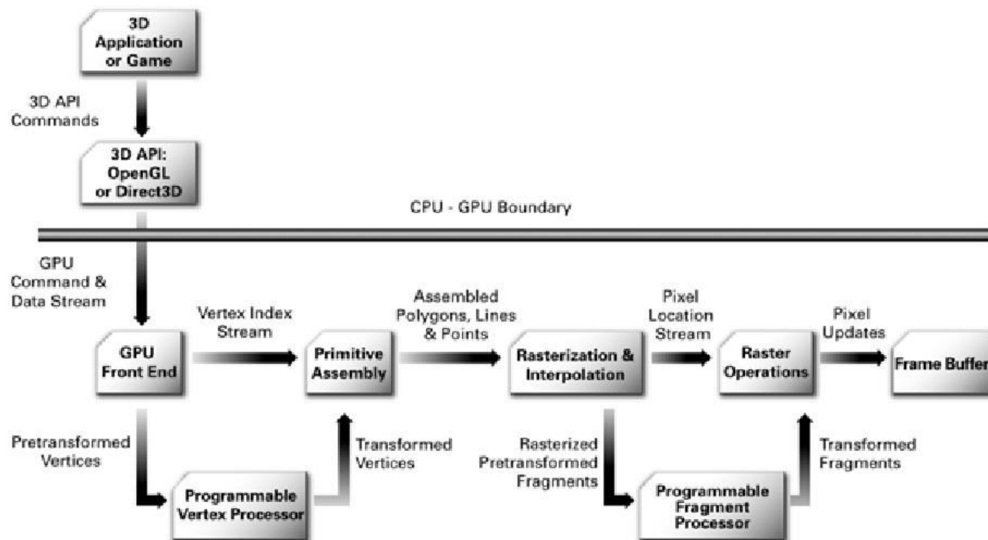
Obrázok 12: NV30 blokový diagram

3.6 Shadery

Keď renderujeme geometrické objekty, tak sú posielané do grafickej karty, ktorá ich spracováva a vykresľuje. V minulosti mali grafické karty určitý počet natvrdo zabudovaných funkcií, ktoré riešili výpočet, osvetlenie, uloženie objektu v priestore, atď.. Tieto natvrdo zabudované algoritmy nazývame Fixed Function Pipeline (FFP). Programátor si mohol zvoliť, ktoré z týchto funkcii chce aplikovať pri výpočte, ale nemohol si navrhovať vlastné funkcie. Taktiež po začatí nebolo možné aby programátor mohol riadiť výpočet. Aplikácie, ktoré využívali grafickú kartu mali teda zhruba rovnaký vzhľad, nebolo možné dodávať nové efekty. Aby mali programátori väčšiu voľnosť, tak sa v roku 2001 objavili na trhu prvé grafické karty, ktoré sa dali čiastočne programovať – objavila sa prvá verzia shadera.

Shadery vznikli v dvoch typoch. Prvý typ bol takzvaný Vertex Shader, ktorý dovoľuje pracovať s vertex dátami a druhý typ bol takzvaný Pixel Shader, ktorý dovoľuje pracovať s pixel dátami. Program shaderu je nahraný do pamäti grafickej karty a nastavený ako program pre grafický pipeline. Kód programu je síce assembler, ale v súčasnosti existuje viacej high-level jazykov, ktoré umožňujú programátorom jednoduchšie programovanie a tiež automatickú optimalizáciu. Najznámejšími jazykmi pre programovanie shaderov sú HLSL(Microsoft Direct3D), GLSL(OpenGL), Cg(Nvidia), ASHLI(ATI). Jazyk HLSL a jazyk Cg sú rovnaké, je to len iné pomenovanie výrobcov toho istého jazyku.

Aby sme mohli programovať shadery musíme vedieť kde je umiestnený Vertex Shader a kde je umiestnený Pixel Shader v grafickom pipeline.

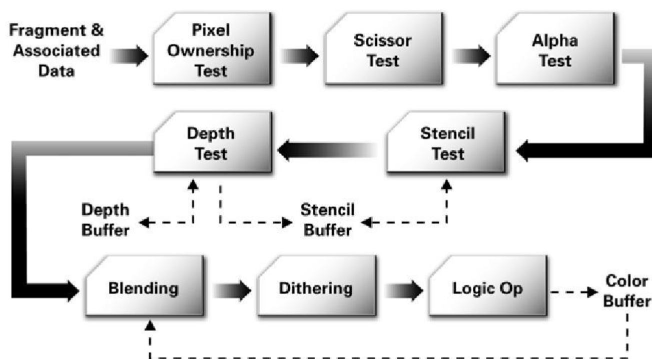


Obrázok 13: Grafický pipelne

Ako môžeme vidieť Vertex Shader – Vertex Processor je umiestnený hneď na vstupe grafického pipelinu. Umožňuje nám teda pracovať z vertexami trojuholníkov. Pixel Shader – Fragment Processor je umiestnený až za rasterizačnou jednotkou, takže v ňom môžeme robiť operácie, ktoré sa týkajú konkrétnych pixelov vo framebuffri.

Počet Shaderov v grafických kartách sa postupom času zvyšuje, Pixel Shaderov býva obvykle viacej ako Vertex Shaderov. Momentálne najnovšie grafické karty GT8800 od Nvidie obsahujú 128 stream procesorov – shaderov, ktoré sú unifikované a dokážu zastávať funkciu Vertex shaderu a tiež aj Pixel shaderu. Prideľovanie stream procesorov sa vykonáva automaticky.

Shadery sú vektorové procesory a dokážu pracovať so štyrmi komponentmi naraz bez akéhokoľvek zdržania. Je možné ich využiť aj na iné účely, než len na grafiku, napr. fyziku, simulácie, atď..



Obrázok 14: Rastrové operácie

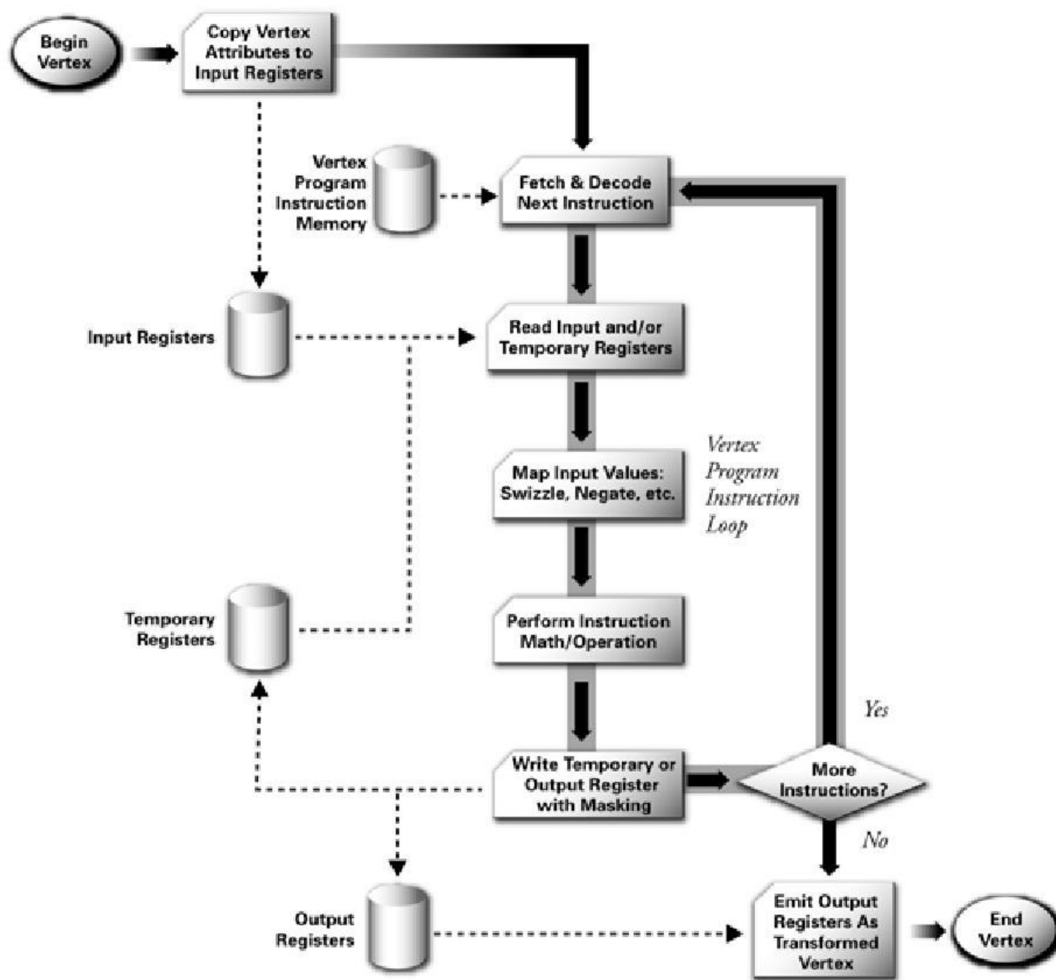
Na obrázku 14. môžeme vidieť ďalší priebeh spracovania jednotlivý fragmentov. Každý fragment je podrobený niekoľkým testom a okrem toho je možné ukladať jeho parametre do dvoch bufferov (Stencil a Depth buffer).

3.6.1 Vertex Shader

Ako sme už hovorili, vertex shader slúži na spracovanie vstupujúcich vertexov trojuholníkov do grafického pipelinu. Každý vertex môže byť definovaný mnohými parametrami, hlavným parametrom je vždy jeho pozícia v 3D priestore, ostané parametre ako napr. farba, normála, textúrovacia pozícia nie sú nutné. Vertex shader nemení typ dát, ale ich hodnoty, čiže v prípade pozície je možné vertex posunúť, v prípade farby je možné ju meniť na základe istých funkcií, podobne to funguje i s ostanými parametrami, ktoré sú poslané na vstup Vertex shader.

Pred uvedením GeForce3 bola práca a výpočty s vertexami veľmi výpočtovo náročné a bolo ich možné vykonávať iba na niekoľko procesorových systémoch. V súčasnosti vertex shadery ponúkajú veľký výpočtový výkon, dokážu spracovávať niekoľko vertexov naraz.

Dávnejšie Vertex shader nedokázal pristupovať k textúram, ale od Shader Model 3.0 (Nvidia GeForce6) je možné čítať dáta aj z textúr. Okrem texúr je možné posielať na vstup konštanty, ktoré slúžia pre nastavenie parametrov scény. Výstup z Vertex shaderu býva v podobnej forme, ako vstupné vertex dáta, čiže pozícia, farba, textúrovacie súradnice, atď.. Pozícia vertexu už musí byť transformovaná do 2D priestoru model-view maticou.

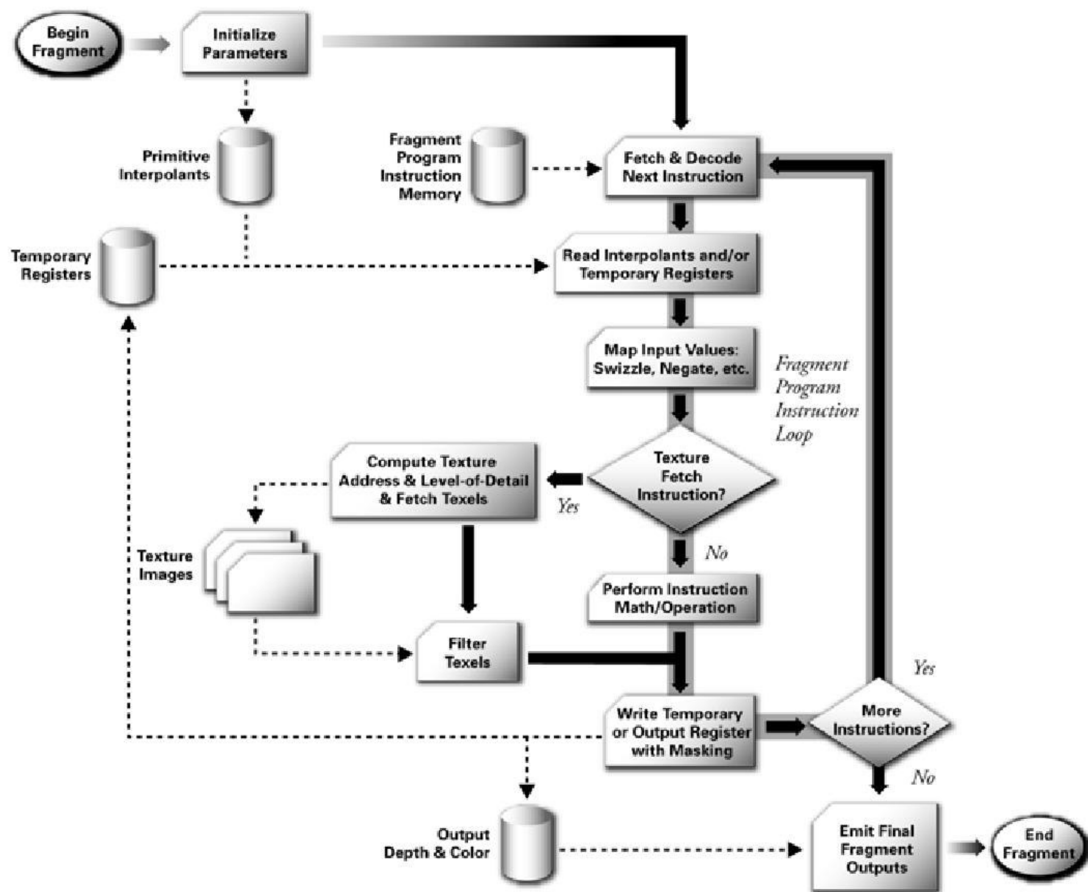


Obrázok 15: Vertex Shader (SM 2.0)

3.6.2 Pixel Shader

Pixel shader je pipeline, ktorý pracuje z jednotlivými pixelmi. Jeho hlavnou úlohou je dodať realistický vzhľad materiálom. Pixel shader je sada veľmi výkonných procesorov, ktoré dokážu spracovávať niekoľko desiatok krát za sekundu stovky tisíc pixelov, tento výkon je oproti bežnému procesoru oveľa väčší.

Pixel shader dostáva na vstup rasterizované dáta z Vertex shaderu, následne ich upraví. Na úpravu môže využívať rôzne textúry a prednastavené dáta – konštanty. Výstupom z pixel shaderu je farba pixelu, prípadne hĺbka (Z-hodnota) pixelu. Je tiež možné využiť tzv. MRT – Multiple Render Target, čo znamená, že je možné mať ako výstup až štyri farby pre jeden pixel. Tento spôsob renderovania šetrí počet renderovaní na jeden frame, pri multipass renderingu.

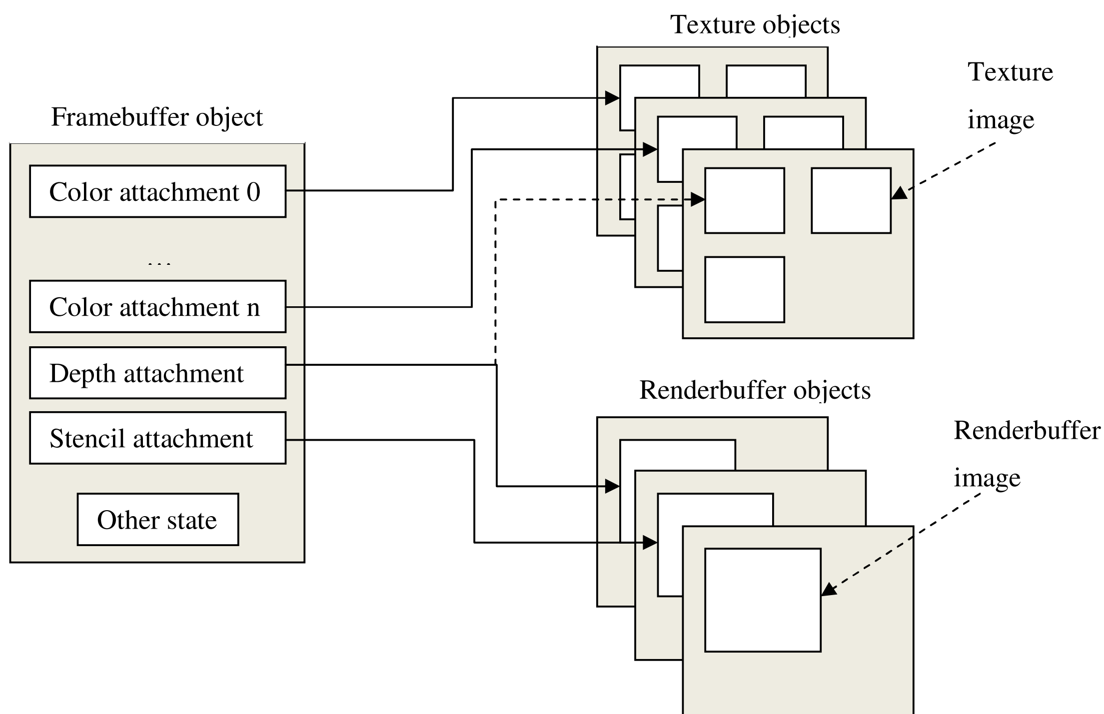


Obrázok 16: Pixel Shader

3.7 FBO – Framebuffer object

Framebuffer object nám dovoľuje renderovať priamo do textúry, čo znamená, že z nej môžeme čítať dáta priamo bez použitia hlavného procesoru – CPU. Výhodou framebuffer objectu je to, že šetrí čas. Nie je nutné renderovať priamo do frambufferu (na obrazovku) a potom previesť obsah framebufferu do textúry. Tiež je to výhodnejšie pre pamäť, lebo nemusíme držať obsah textúry a tiež framebufferu. Do framebuffer objectu môžeme renderovať okrem farebného výstupu, aj depth buffer a tiež stencil buffer.

Texture object je objekt, ktorý uchováva dáta ako textúry. Je možne ich teda bindovať a využívať ako bežné textúry. Render object obsahuje jednoduché 2D obrazky, žiaden mipmaping, alebo iné špecifické vlastnosti textúry.

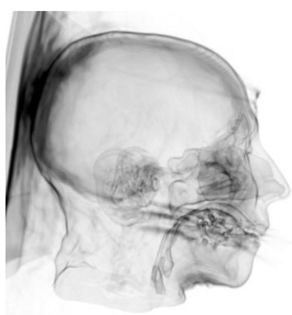


Obrázok 17: Framebuffer object

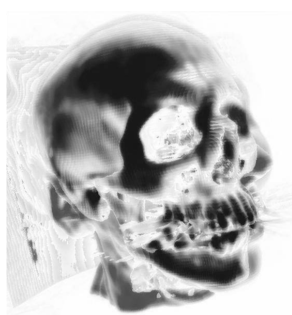
4 Návrh

Zobrazovať 3D dáta je možné viacerými spôsobmi. Zobrazovacia metóda veľmi závisí od našich požiadaviek čo chceme vidieť. V tomto projekte sme sa rozhodli používať nasledujúce tri projekčné metódy:

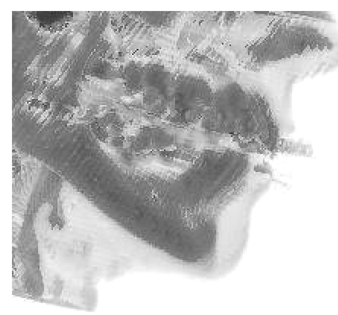
1. Blend projection
2. MIP – Maximal Intensity Projection
3. Blend+MIP (Kombinované)



Obrázok 18: Blend



Obrázok 19: MIP



Obrázok 20: Blend+MIP

Blend projekcia zobrazuje celý objem dát s určitou nastavenou priesvitnosťou. Je vhodná na zobrazovanie mäkkých, najmenej výrazných tkanín, ktorých je väčšia vrstva. Pri tejto projekcii veľmi dobre vynikajú krivky rôznych typov materiálov, prípadne dutiny v objeme.

MIP (Maximal Intensity Projekcia) vychádza z požiadavky vidieť tie najvýraznejšie tkanivá v celej hĺbke objemu. Táto metóda projekcie je teda veľmi dobrá na zvýrazňovanie najtvrdších, najvýraznejších tkanín. Metódu je ju vhodné kombinovať s osvetľovacím modelom scény, aby vynikli krivky nepriesvitných častí objemu.

Kombinovaná projekcia spája niektoré výhody Blend projekcie a MIP projekcie. Výhodou tejto metódy je zobrazenie najtvrdších (najvýraznejších) častí objemu a následné prekrytie týchto pevných časti mäkkými (menej výraznými) vrstvami objemu.

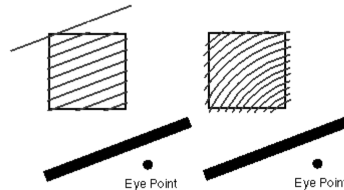
V tomto projekte budeme využívať len Blend projekciu a kombinovanu (Blend+MIP) projekciu. Čistú MIP projekciu vynecháme z toho dôvodu, že kombinovaná projekcia ju dokáže nahradiť.

Teraz, keď už vieme čo očakávame vyrenderované na obrazovke, tak sa musíme rozhodnúť, ako objem dát vyrenderujeme. V predošlých kapitolách sme sa dozvedeli, aké sú možnosti súčasného hardwaru a vieme, aké metódy renderovania 3D objemu môžeme použiť. Súčasný hardware má veľmi rýchle textúrovacie jednotky a tiež možnosť per-pixel spracovania obrazu. Z týchto dvoch skutočností môžeme vyselektovať metódu RayCasting a Texture Mapping. Ray Casting je síce veľmi

kvalitná metóda ale po pár testoch a úvahách o rýchlosti sa ukázala, ako nie úplne vhodná, lebo presúvala celý výpočet na streamovacie procesory grafickej karty. Metóda Texture Mapping ponecháva určité zaťaženie na procesore počítača a časť na procesoroch grafickej karty. Toto rozdelenie výpočtu vzniká kvôli tomu, že okraje objemu sa riešia už na procesore, pri RayCastingu sa okraje a presah za okraj objemu rieši v grafickej karte.

Aby sme mohli Texture mapping použiť, musíme si celý objem rozdeliť na vrstvy. Táto časť výpočtu je veľmi jednoduchá analytická geometria. Vieme, že naše dáta majú tvar kvádra, táto skutočnosť nám určuje 12 hrán kvádru. Pre výpočet jednotlivých vrstiev použijeme rovinu ktorá je kolmo naklonená k pozorovateľovi. Na začiatku si rovinu umiestnime do najvzdialenejšej pozície, akú môže objem nadobudnúť a potom tou rovinou posúvame a počítame priesečníky roviny a 12-tich úsečiek. Rovinou posúvame až dovtedy dosiahneme najbližší možný bod pri kamere, ktorý môže objem dát nadobúdať. Priesečníky ktoré vypočítame nám určujú prostredníctvom n-uholníkov jednotlivé vrstvy objemu. Tieto n-uholníky potom renderujeme a mapujeme na ne 3D textúru nášho objemu.

Okrem roviny kolmej k pozorovateľovi, môžeme použiť aj zakrivené plochy, ktoré majú v každom bode rovnakú vzdialenosť od pozorovateľa. Túto variantu nebudeme používať, lebo jej použitie má význam len v prípade, žeby sme zobrazovali dáta veľmi blízko pozorovateľa a s veľkým uhlom pohľadu, čo by v tomto projekte nemalo zmysel, pri uhle pohľadu 30º.



Obrázok 21: Vrstvenie - Texture Mapping

Vrstvy renderujeme ako polygóny a aby sme na ne mohli mapovať 3D textúru podľa nášho uváženia musíme použiť shadery. Tieto shadery nastavíme ešte pred renderovaním. Vertex Shader použijeme na výpočet textúrovacích súradníc a na posunutie iných parametrov pre Pixel Shader. Textúrovacie súradnice môžeme poslať aj priamo z CPU do GPU ale to by zbytočne zaťažovalo systém. Pixel Shader naprogramujeme tak, aby na základe textúrovacích súradníc textúroval a upravoval priesvitnosť vykresľovaných bodov.

Na zvýrazňovanie materiálov použijeme transfer funkciu vo forme 1D textúry. V tejto textúre budeme vyhľadávať intenzitu voxelu z dátovej textúry a tým dostaneme RGBA hodnoty pre ten voxel. Textúru transfer funkcie budeme nazývať kolorovacia textúra.

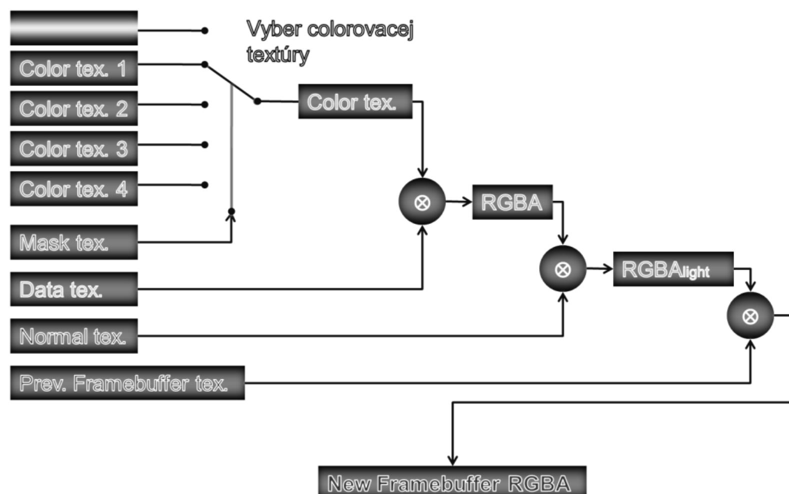
Pri MIP metóde musíme osvetľovať RGBA hodnoty voxelu. Kvôli tomuto potrebujeme normálovú 3D textúru, ktorá bude obsahovať predpočítané hodnoty normál. Okrem tejto textúry potrebujeme ešte do Pixel shadera nastaviť pred spustením renderovania pozíciu svetla. Normálová

mapa môže obsahovať dva údaje. Ako sme si už povedali, prvý údaj je normála ukazujúca smer gradientu materiálu. Druhý parameter môže byť dĺžka normály, ktorá bude určovať silu gradientu zmeny materiálu. Tento druhý parameter sa nám bude hodiť pri tvorení vrstiev materiálov naskenovaných dát.

Veľmi často si je nutné zvýrazňovať určitú oblasť dát. Zvýrazňovanie je možné vytvoriť na základe rôznych transfer funkcií v rôznych oblastiach dát. Oblasti dát je možné vytvoriť maskovacou textúrou, ktorá bude určovať index kolorovacej textúry. Kolorovacích textúr si preto vytvoríme viac a nastavíme ich všetky na vstup Pixel shaderu. Aby sme mohli zvolené časti odstraňovať, orezávať, tak zavedieme pravidlo, že ak maskovacia textúra obsahuje index s číslom 0, tak daný materiál nastavíme na maximálnu priesvitnosť.

V predošlých odstavcoch sme si povedali ako vytvoríme farbu a priesvitnosť renderovaného voxelu. Teraz ešte potrebujeme spájať jednotlivé vrstvy dokopy. Na toto využijeme renderovanie do textúry, teda FBO – Framebuffer object. FBO nám umožní renderovať do textúry, akoby to bol klasický framebuffer. Id tejto textúry nastavíme Pixel Shaderu ako vstupnú textúru, v pixel shaderi môžeme takto čítať hodnoty ktoré sme zapísali o jednu vrstvu objemu skôr. Renderovanie a čítanie predošlých výsledkov z FBO nám nahradí klasický spôsob blendovania pixelov, teraz si blendovanie môžeme robiť plne podľa svojej vôle. V prípade MIP bude blending podmienený tvrdosťou materiálu a v prípade Blend projekcie budú hodnoty sčítané.

Na nasledujúcom obrázku môžeme vidieť navrhnutý algoritmus Pixel shaderu. Ako prvá sa určí kolorovacia textúra (v projekte sú 4 kolorovacie textúry) na základe indexu maskovacej textúry. Po vyselektovaní kolorovacej textúry sa lokalizujú dáta z dátovej textúry a vyberie sa z kolorovacej textúry, na základe týchto dát, farba RGBA pre spracovávaný voxel. Ak používame MIP metódu, tak sa na RGBA hodnotu voxelu aplikuje osvetľovací model spolu z normálovou textúrou. Na záver sa osvetlená RGBA hodnota voxelu spojí s pôvodnou hodnotou voxelu z textúry vytvorenej pomocou FBO a pošle ako výstup z pixel shaderu. Tento algoritmus sa vykoná pre všetky polygóny a ich všetky body, ktoré pokrývajú (rádovo milióny bodov)



Obrázok 22: Pixel shader algoritmus

5 Implementácia

Celý projekt bude implementovaný v jazyku C++ a ako vývojové prostredie bude použité Microsoft Visual Studio 2005. Na zobrazovanie 3D scény použijeme OpenGL. Výhodou OpneGL je jeho prenositeľnosť na rôzne platformy. Shadery budeme programovať v jazyku CG, ktorý je veľmi dobre okomentovaný a má veľkú podporu Nvidie. Na pri vývoji sa projekt bude testovať na grafickej karte Nvidia GeForce 6600GT. ATI karty nebudú zatiaľ oficiálne podporované kvôli nedostupnosti hardwaru.

V nasledujúcich kapitolách si postupne vysvetlíme celý priebeh programu, od načítavania dát až po ich zobrazenie.

5.1 Vstupné dáta

Na začiatku tohto dokumentu sme si hovorili o tom, ako získavame 3D dáta a aké metódy nato môžeme použiť. Tam sme sa dozvedeli, že dáta sú reprezentované 3D maticou, ktorá obsahuje hodnoty, ktoré vyjadrujú určitú vlastnosť materiálu, napríklad tvrdosť.

Ako základ pre načítavanie 3D dát si zoberieme VLB formát, ktorý je veľmi jednoduchý. Obsahuje textovú hlavičku, ktorá popisuje základné parametre dát a za hlavičkou sa nachádzajú dáta. Príklad takejto hlavičky môžeme vidieť na nasledujúcich riadkoch:

```
VLIB.1
256 256 113
int16
little
1. 1. 2.
-1117 2248
0 1
```

Hlavička sa skladá z niekoľkých riadkov, ktoré obsahujú jednotlivé dáta.

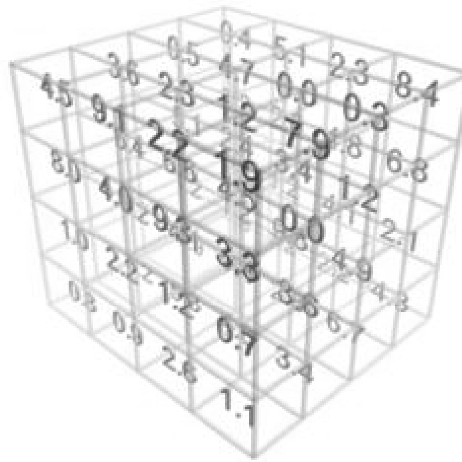
1. Verzia VLB dátového formátu (string)
2. Rozmer dátovej matice (unsigned long)
3. Typ dát a počet bitov ([int,uint][8-32],float, double)
4. Endian (little, big)
5. Mierka dát (float)

6. Min a Max hodnota dát (float)
7. Doporučená Min a Max hodnota zarovnania dát (float)

Za textovou hlavičkou nasledujú binárne dáta, ktoré majú presne toľko položiek, koľko bolo špecifikované rozmerom dátovej matice.

Okrem VLB vstupných dát je podporovaný aj zložitejší ICS formát, na ktorý je využitá ICS knižnica LibICS pre načítanie súborov. ICS podpora je čiastočná a nevyužíva všetky výhody a funkcie ICS formátu.

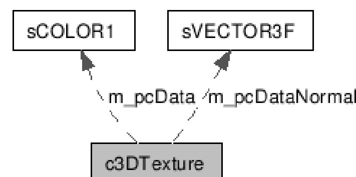
Po načítaní dát zo súboru, sa dáta preformátujú do typu FLOAT, aby bola jednoduchšia práca pri ich následnom spracovaní. Dáta sú v momentálnej implementácii skalárne čísla, čiže obsahujú len jednu informačnú hodnotu na voxel. Príklad veľmi malej dátovej matice môžeme vidieť na nasledujúcom obrázku.



Obrázok 23: Dátová matica

O proces nahrávania dát zo súborov sa stará objekt c3DTexture. Jeho dva konštruktory dokážu generovať testovací objem dát, alebo nahrávať zo spomínaných dvoch typov súborov. Generovanie 3D objemu dát je vhodné na rýchle testovanie programu a hardwaru. Okrem vytvárania dát sa objekt stará o predprípravu vstupných dát.

Objekt c3DTexture obsahuje dva pointre na zmienené dve štruktúry. Dátová matica je typu sCOLOR1, čo je typ, obsahujúci jednu skalárnu hodnotu typu float. Normálová matica je uložená vo formáte sVECTOR3F, čo je typ obsahujúci vektor s tromi float hodnotami.



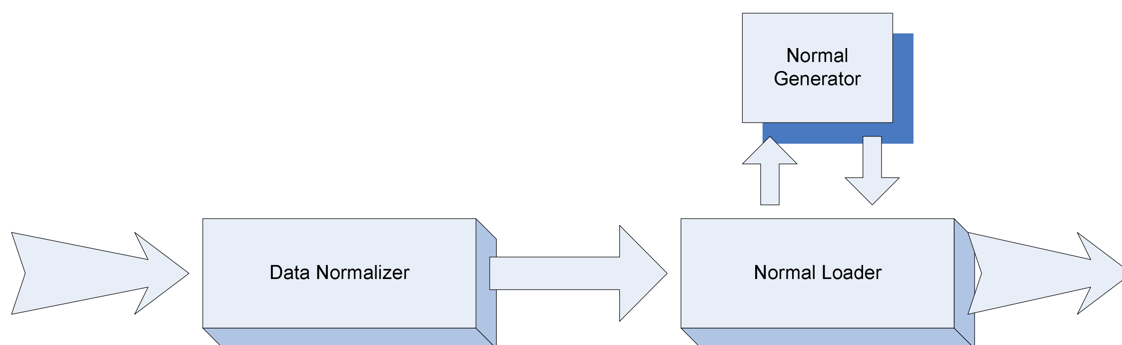
Obrázok 24: c3DTexture objekt

5.2 Predpríprava dát

Vstupné dáta, ktoré vzniknú vytvorením, alebo nahratím dát zo súboru, sú len obyčajná matica čísel. Na základe tejto matice čísel budeme vyhľadávať v kolorovacej matici príslušnú farbu pre daný materiál. Ak by dáta boli v ľubovoľnom rozsahu, tak by nastal problém pri vyhľadávaní v kolorovacej textúre kvôli nekompatibilite rozsahov. Kvôli tejto nekompatibilite je vhodné tieto dáta normalizovať do rozsahu [0.0-1.0], to nám umožní využiť klasické mapovanie textúr. Na nasledujúcom obrázku môžeme vidieť časť Data Normalizer, ktorá sa stará o túto funkciu.

Pri Blend metóde zobrazovania by postačovala dátová matica, ale pri MIP sú tieto dáta nedostatočné, lebo je nutné vedieť smer zmeny gradientu materiálu a na ňom závislú normálu. Veľkosť normály tiež určuje, ako prudko sa mení gradient v danom smere. Keďže budeme normály používať hlavne na osvetľovanie objemu dát a tiež na vyhľadávanie okrajov materiálu, tak je ich vhodné mať normalizované na veľkosť [0.0-1.0]. Potom pri vyhľadávaní okrajov vrstiev materiálov budeme môcť označiť najväčšiu zmenu číslom 1.0 a najmenšiu zmenu číslom 0.0.

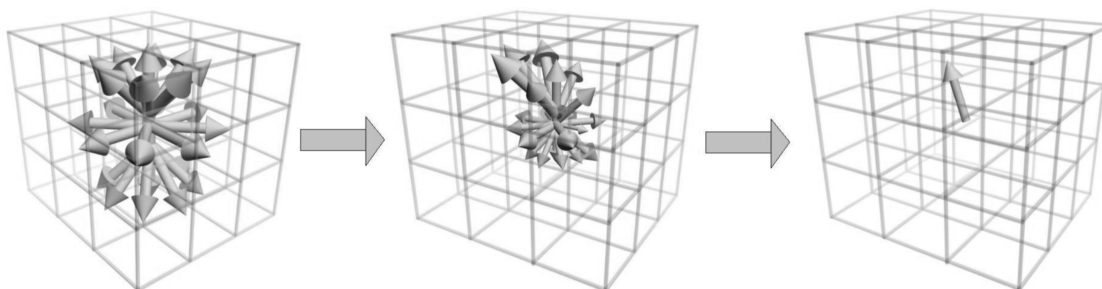
Na nasledujúcom obrázku vidíme Data Normalizer, ktorý si po načítaní dát nájde maximum a minimum dátových hodnôt a potom všetky dáta normalizuje na základe minima a maxima na rozsah [0.0-1.0]. Algoritmus potom ďalej postupuje do časti Normal Loader, ktorý sa pokúsi načítať z binárneho súboru normály. Ak sa mu to nepodarí, vytvorí ich z dátovej matice a následné uloží do súboru.



Obrázok 25: Algoritmus predprípravy dát

Ak sa normály vytvárajú, nie nahrávajú so súboru, postupuje sa tak, že sa prechádza celá dátová matica a v najbližšom okolí sa vytvoria vektory na všetky smery. Každý z týchto vektorov sa potom vynásobí rozdielom hodnoty voxelu v okolí a voxelu pre ktorý sa vytvára normála (stredovým voxelom). Takto upravené vektory sa potom sčítajú a výsledkom tejto celej operácie je jeden vektor, ktorý má smer najväčšieho gradientu a tiež veľkosť, ktorá určuje rýchlosť zmeny.

Na nasledujúcom obrázku vidíme všetky vektory v okolí vzdialenom o 1 voxel. Na ľavej strane sú tieto vektory ešte v pôvodnej jednotkovej dĺžke a v strede sú tieto vektory vynásobené zmenou hodnoty oproti stredovému voxelu a na prvom obrázku z ľava sú sčítane do jedného vektoru.

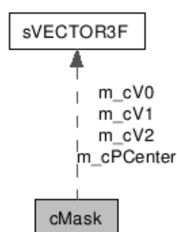


Obrázok 26: Výpočet normál

Pri vytváraní vektorov sa hľadá tiež minimum a maximum, ako je to v prípade dátovej matice a následne sa normálová matica normalizuje na rozsah [0.0-1.0]. Normálová matica má vždy rovnaký rozmer, ako dátová matica.

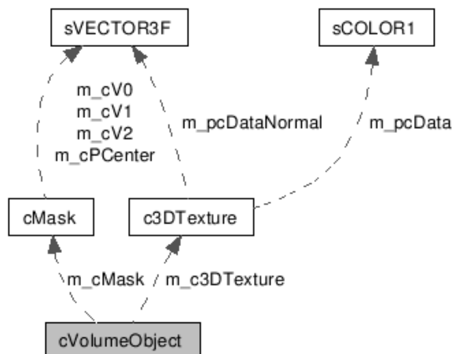
O predošlé dve operácie (normalizácia dát, vytvorenie normál) sa stará ten istý objekt c3DTexture, ktorý je tiež zodpovedný aj za načítanie dát. Objekt c3DTexture je vytváraný objektom cVolumeObject, ktorý je základným kameňom renderovania dát.

Objekt cVolumeObject tiež obsahuje maskovaciu maticu, potrebnú na indexovanie kolorovacej textúry podľa miesta výskytu voxelu. Maskovacia matica môže mať rozmer iný, ako dátová a normálová matica v objekte c3DTexture. Maskovaciu maticu vytvárame z objektu cMask. Na začiatku je táto matica nastavená na hodnotu 1, čo znamená, že sa používa kolorovacia matica č.1..



Obrázok 27: Objekt cMask

Objekt cVolumeObject obsahuje, okrem objektu c3DTexture a cMask, aj dáta o tvare a okrajoch volumetrických dát a hlavne funkcie na vykresľovanie celého objemu dát.



Obrázok 28: Objekt cVolumeObject

5.3 Renderovanie dát

Pri návrhu zobrazovacieho systému sme si povedali, jednoduchý princíp zobrazovania. Zobrazovanie 3D dát je robené objektom cVolumeObject, ktorý má za úlohu pripravovať textúry, starať sa o shadery, vytvárať a zoraďovať vrstvy zobrazovaného objemu dát a na koniec tieto vrstvy zobraziť. Okrem zobrazovania objemu dát, dokáže zobraziť normálovú maticu vo forme malých normál vychádzajúcich z každého voxelu objemu.

V nasledujúcich kapitolách si popíšeme postupný proces celého zobrazovacieho podsystému. Najprv sa budeme zaoberať ako dostaneme dáta z hlavnej pamäti počítača do pamäti grafickej karty a ako ich budeme manažovať. V druhej časti si povieme ako vytvoríme a nahráme programy pre programovateľné procesory do ich vnútornej pamäti. Následne potom si popíšeme ako vytvoriť vrstvy objemových dát a ako ich zobrazíme. A v poslednej časti budeme hovoriť o farebných úpravách výsledného obrazu.

5.3.1 Príprava textúr

Príprava textúr je veľmi podstatná časť v tomto projekte. Je dôležitá hlavne kvôli tomu, že textúry načítané v grafickej karte rozhodujú o rýchlosti vykresľovania. Vykresľovanie je závislé na textúrach preto, lebo pri textúrovaní sa dáta čítajú z pamäti grafickej karty, v horšom prípade z hlavnej pamäti počítača. Pri lokalizácii potrebného texelu je nutná veľká rýchlosť čítania z tejto pamäte. Ako každý iný systém, má aj grafická karta hierarchiu pamätí, ktoré sa líšia okrem svojej veľkosti aj svojou rýchlosťou. Ak je dát veľmi veľa, nemusia sa zmestiť do najrýchlejšej cache pamäte a musia sa čítať z pamäte vyššej úrovne. Aby sa dát zmestilo čo najviac do cache pamäte je nutné ich minimalizovať na čo najmenší obsah. Minimalizácia obsahu textúry sa robí už pri jej vytváraní, tak, že sa udá formát v akom má byť uložená. Formát špecifikovaný pri vytváraní textúry je len odporučenie pre OpenGL aký formát by sme chceli využiť na uloženie danej textúry, nie je teda isté či budú takto uložené. Ak OpenGL nenájde požadovaný formát, tak sa pokúsi zvoliť najbližší možný formát, ktorý použitá grafická karta podporuje.

Formáty, ktoré sú ponúkané na uloženie dát v grafickej karty môžeme rozdeliť do dvoch skupín. Prvá skupina sú formáty typu *integer*, ktoré môžu mať rozsah 4bity, 8bitov, 16bitov. Druhá skupina sú *float* formáty, ktoré sú podporované len v novších grafických kartách. *Float* formáty môžu mať veľkosť 16bitov alebo 32bitov. *Float* formáty sú samozrejme náročnejšie na výpočty a preto ich nie je vhodné používať tam, kde sa dajú klasické *integer* formáty použiť. Každý integer a float formát môže byť vektorový, čiže počet komponentov vo vektore je 1-4 [r,g,b,a].

Ako sme si už spomínali, pracujeme s tromi hlavnými textúrami, dátovou textúrou, normálovou textúrou a maskovacou textúrou. Každá z týchto textúr uchováva iný typ dát a toto musíme zohľadniť aj pri volení interného formátu v grafickej karte.

Najprv si rozoberieme dátovú maticu. Tento typ dát je špecifický tým, že obsahuje len jednu informačnú hodnotu, ktorá však musí byť veľmi presná. Presná musí byť kvôli tomu, že často krát sa pôvodné dáta nachádzajú vo formáte ktorý je 16bitový. Aby sme túto požiadavku splnili bude pre dátovú maticu vhodný interný formát GL_ALPHA16. Je to 16bitový formát z jednou zložkou vektoru. Je teda dosť presný a zároveň neobsahuje zbytočné dáta. Na grafických kartách, kde nie je tento formát podporovaný, tak driver automatický zvolí formát GL_ALPHA8, čo je ešte prípustná strata kvality pre starší hardware.

Ďalšia matica je normálová matica. Táto matica už obsahuje vektory s tromi položkami x, y, z. To znamená, že v pamäti môže zaberať až 3× väčší obsah, ako dátová matica a preto nie je vhodné voliť príliš vysoké bitové rozlíšenie. Na túto maticu sa nám teda môže veľmi dobre hodiť formát GL_RGB8. Tento formát ma 8bitové rozlíšenie pre každú zložku vektoru. To znamená, že sme schopný smerovať vektor z najhoršou presnosťou 1.4º, čo je celkom dostačujúce. Normálová matica sa nevyužíva len na normály pre osvetľovací model scény, ale tiež aj na určenie gradientu zmeny materiálu. Tento gradient je určený dĺžkou a smerom normály. S 8bitovou presnosťou dosiahneme rozlíšenie gradientu na 256 hodnôt.

Poslednou maticou je matica maskovacia, táto matica obsahuje len skalárne hodnoty v ktorých sú uložené indexy kolorovacích matíc, ktoré nadobúdajú hodnoty od 0 do N. Na takéto je vhodný formát GL_ALPHA4, obsahuje iba jednu zložku vektoru v rozlíšení 16 hodnôt. 16hodnôt nám určite dostačuje, lebo budeme používať len 4 kolorovacie matice + 1 pseudo čirno-priesvitnú kolorovaciu maticu, to nám dáva len 5 indexov.

Teraz skúsme vypočítať pamäťové nároky na tieto tri textúry. Dátova a normálová matica budú mať rozmer 250×250×150 voxelov. Maskovacia matica bude mať rozmer 100×100×100 voxelov. Budeme predpokladať, že grafická karta plne podporuje požadované interné formáty textúr.

250×250×150=9375000 voxelov
100×100×100=1000000 voxelov
dátová matica zaberá v pamäti 1×2B na voxel, to nám dáva 18750000B
normálová matica zaberá v pamäti 3×1B na voxel, to nám dáva 28125000B
maskovacia matica zaberá v pamäti 1×0.5B na voxel, to nám dáva 500000B

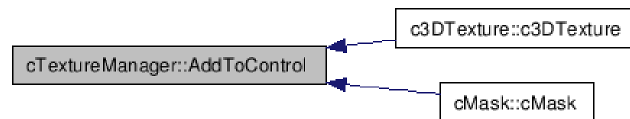
---

dokopy teda dostávame 47375000B = 45.2MB pamäti

Okrem týchto troch základných textúr, sú tam aj textúry, ktoré slúžia ako pomocné pri zobrazovaní určitých informácii a tiež sú tam textúry ktoré nahrádzajú Framebuffer. Textúry pre FBO sú vytvorené pri prvom použití. To je v prvom renderovacom cykle. Musíme uvažovať s textúrami typu *float,* kvôli prípadným úpravám obrazu v postprocessingu a okrem toho musia tiež spĺňať čo najmenší bitový rozsah. Z týchto požiadaviek je teda jasne, že použijeme interný formát GL_RGBA16F_ARB.

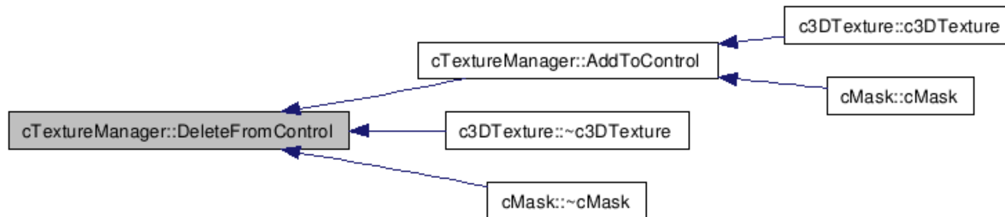
Ak meníme parametre zobrazovacieho okna, tak je nutné ho reinicializovať. Pri reinicializácii, vytvorené textúry budú neplatné, tak ako odkazy na ne, ktoré sú uložené v objektoch ktoré ich využívajú, budú tiež neplatné. Z tohto dôvodu je nutné spraviť mechanizmus, ktorý bude manažovať tieto odkazy. Objekt bude udržiavať odkazy na tieto textúry. V prípade reinicializácie vymaže všetky textúry. Objekty, ktoré budú podliehať tomuto manažovaniu, automaticky detekujú, že ich textúry sú zmazané a vytvoria si nové.

Pri vytváraní objektu spravovaného cTextureManagerom, bude pomocou funkcie AddToControl pridaný odkaz na miesto, kde sa v tomto objekte bude udržovať ID textúry. Takto bude cTextureManager obsahovať všetky ID na textúry.



Obrázok 29: cTextureManager::AddToControl

Ak sa bude objekt rušiť, tak v deštruktore každého objektu bude odkaz na funkciu DeleteFromControl a tým sa zruší správa textúry ktorá bola spravovaná.



Obrázok 30: cTextureManager::DeleteFromControl

Pri reinicializácii renderovacieho okna musí byť vždy zavolaná funkcia InvalidateAll. Táto funkcia zabezpečí bezpečné odstránenie ID textúr zo všetkých objektov ktoré tieto textúry využívajú.

Obrázok 31: cTextureManager::InvalidateAll

Pri zmazaní všetkých ID textúr zistí každý objekt, že jeho textúra je neplatná a vytvorí si nové ID a zároveň ju nainicializuje na požadované hodnoty.

5.3.2 Kompilácia CG shaderov

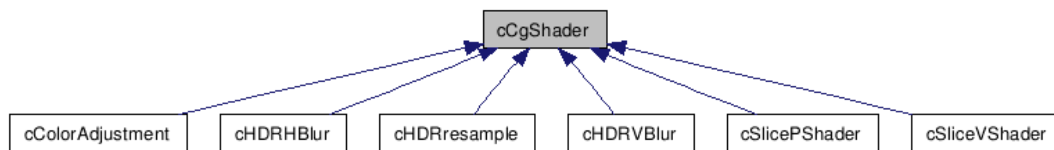
Shadery v tomto projekte hrajú hlavnú úlohu, vďaka schopnosti ich programovať. Nahrádzajú nám blendovací blok grafickej karty a umožňujú nám robiť rôzne špeciálne projekcie objemu dát a iné efekty.

Aby sme mohli shadery začať využívať, tak ich musíme pripraviť a nahrať do grafickej karty. Všetky shadery, ktoré využívame v tomto projekte využívajú rovnaký programovací jazyk – CG. Tento jazyk má výhodu v tom, že je veľmi dobre prenositeľný na iné grafické karty. Prenositeľnosť je

dosiahnutá jeho runtime kompiláciou. To znamená, že programy v jazyku CG môžu byť dodávané nekompilované, v zdrojovom kóde, alebo už skompilované. Vtedy je ich prenositeľnosť obtiažnejšia.

Shadery majú rovnaký interface pre ich načítanie, skompilovanie, aktivovanie a tiež deaktivovanie. Aby sme toto všetko nemuseli programovať pre každý shader zvlášť, tak si vytvoríme triedu cCgShader. Táto trieda bude obsahovať všetky potrebné funkcie na prácu so shadermi. Tiež ju budeme dediť do shaderov, ktoré budeme vytárať.

Na nasledujúcom obrázku môžeme vidieť všetky triedy používaných shaderov v tomto projekte. Všetky sú odvodené od hlavného objektu cCgShader. Trieda cCgShader obsahuje systém automatickej kontroly používaného shaderu. Pri aktivácii shaderu sa automaticky kontroluje, či je shader skompilovaný a nastavený. V prípade dákeho nedostatku sa automaticky pokúsi odstrániť tento problém. Tento systém je veľmi dobrý kvôli tomu, že nie je nutné inicializovať shadery, všetko je automaticky spravené pri prvom použití.



Obrázok 32: Triedy použitých shaderov

Prvým z týchto shaderov je cColorAdjustment. Tento shader slúži na úpravu kontrastu, svetlosti a gama korekcie. Na vstup ide textúra obrazu, ktorý chceme upraviť a vektor parametrov nastavení funkcii.

Trieda cHDRHBlur spracováva obraz tak, že vytvára okolo bodov, ktoré presahujú maximálnu zobrazovanú intenzitu, jemnú žiaru v horizontálnej polohe. Shader dostáva na vstup textúru obrazu na spracovanie a výstupom je gaussovovsky ožiarený bod z bodov v jeho horizontálnom okolí.

Trieda cHDRresample slúži na redukciu textúry na menší rozmer. Na vstup je vložená textúra na spracovanie a na výstupe je zredukovaná textúra na polovičnú veľkosť.

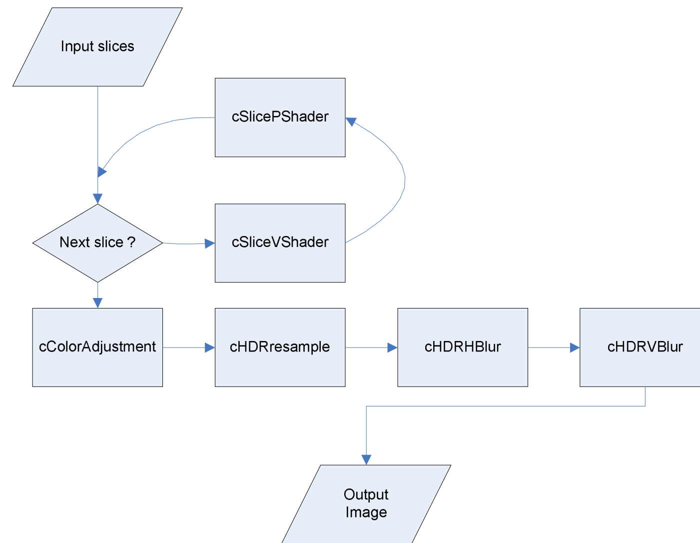
Trieda cHDRVBlur pracuje rovnako, ako trieda cHDRHBlur, len s tým z rozdielom, že vytvára ožiarenie vo vertikálnom smere.

Trieda cSliceVShader je iný typ, ako shadery, ktoré sme si doteraz popísali. Tento shader je určený pre Vertex pipeline, teda je to vertex shader. Tento shader slúži na predprípravu niektorých dát, ktoré potom využívame v Pixel shadery. Konkrétne sa jedná o transformáciu 3D priestoru do 2D priestoru, tiež o predanie netransformovaných súradníc a nakoniec o predanie vypočítaných textúrovacích súradníc.

Trieda cSlicePShader je shader pre Pixel shader , ktorý spracovaáva dáta z vertex shaderu a dáva im grafickú podobu. Jeho hlavnou úlohou je zostaviť z maskovacej, kolorovacej a dátovej matice farebnú reprezentáciu dát. Okrem toho sa stará o osvetľovanie a o spájanie (blendovanie) predošlej vrstvy, ktorá je daná na vstup ako textúra. Tento shader najprv podľa maskovacej textúry zvolí príslušnú kolorovaciu textúru a potom z nej vyberie na základe dátovej matice príslušnú farbu.

Na túto farbu aplikuje podľa normálovej matice osvetľovací model. Podľa alfa kanálu vypočítanej farby sa rozhoduje, či je pixel úplne viditeľný, alebo sa pričíta k pôvodnej farbe pixelu na pôvodnom mieste.

Na nasledujúcom obrázku môžeme vidieť blokový diagram ako postupne pracujú jednotlivé shadery. Shader cSlicePShader je podrobnejšie popísaný na Obrázok 22.



Obrázok 33: Proces prechodu shadermi

5.3.3 Zostavenie vrstiev objemových dát

Na zobrazovanie budeme používať metódu Texture Mapping, to znamená, že budeme postupne vrstviť celý objem dát a mapovať na tieto vrstvy textúru. Vrstvy ktoré budeme tvoriť budú plochy ohraničené prienikom plochy a strán kvádru. Touto úvahou by sme však dostali, pri prieniku plochy kolmej k pozorovateľovi a plochy strany kvádra ohraničujúceho objem, len rovnicu priamky. Zobrazovanie v grafike funguje na princípe vertexov, čo sú vlastne rohy trojuholníka alebo prípadne n-uholníka. Z tohto dôvodu musíme definovať vrstvu tak aby sme dostali jej priesečníky so stranami kvádru ohraničujúceho dáta.

Takže keď chceme dostať jednu vrstvu dát naklonenú kolmo k pozorovateľovi, musíme začať s rovnicou jej plochy.

$$ax + by + cz + d = 0$$

V tejto rovnici môžeme vidieť vektor, ktorý je kolmý na nami požadovanú vrstvu, to je vektor *[a,b,c]*. Premenná *d* vyjadruje posun plochy v smere vektoru kolmého na plochu. Túto premennú budeme využívať v cykle na postupný posun plochy.

Ďalšiu rovnicu, ktorú potrebujeme je rovnica priamky. Budeme pomocou nej špecifikovať jednotlivé strany kvádru ohraničujúceho dátovú maticu. Použijeme parametrické rovnice.

$$v_x t + p_x = x$$

$$v_y t + p_y = y$$

$$v_z t + p_z = z$$

Vektor $[v_x, v_y, v_z]$ je smerový vektor priamky, $[p_x, p_y, p_z]$ je počiatočný bod priamky a t je parameter určujúci relatívnu vzdialenosť bodu na priamke od počiatočného bodu. Keď hľadáme priesečník priamky a roviny tak dostaneme nasledujúcu rovnicu.

$$t = -\frac{ap_x + bp_y + cp_z + d}{av_x + bv_y + cv_z}$$

Výsledkom tejto rovnice je parameter t, ktorý keď dosadíme späť do rovníc priamky, dostaneme priesečník strany kvádru a požadovanej vrstvy. Tento algoritmus opakujeme pre všetky strany kvádru a tak dostaneme sadu priesečníkov.

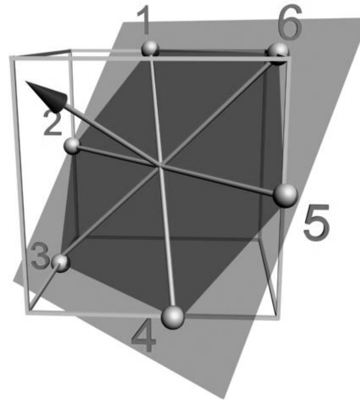
Tieto priesečníky usporiadame v smere pravotočivom, resp. ľavotočivom. Využijeme nato rovnicu na výpočet uhlu dvoch vektorov.

$$\propto = \operatorname{acos}\left(\frac{\overline{v_1} \cdot \overline{v_2}}{|\overline{v_1}||\overline{v_2}|}\right)$$

Táto rovnica však dáva výsledok v rozsahu [-π, π], to znamená, že nevieme určiť smer natočenia kvôli tomu, že nevieme ako boli merané vektory navzájom usporiadané. Aby sme určili smer uhlu, tak vypočítame nasledujúcu rovnicu, ktorá nám určí vzájomnú polohu dvoch vektorov vzhľadom na os danú vektorom v_{norm}. Vektor v_{norm} je v našom prípade kolmica na počítanú vrstvu.

$$sign = sgn\left((\overline{v_1} \times \overline{v_2}) \cdot \overline{v_{norm}}\right)$$

Podľa znamienka *sign* určíme, či sú vektory pravotočivo, alebo ľavotočivo usporiadané. Podľa uhlu, ktorý sme vypočítali predtým, môžeme usporiadať všetky priesečníky tak, že po vymenovaní dostaneme konvexný polygón.



Obrázok 34: Výpočet vrstvy objemu dát

Po usporiadaní priesečníkov, postupne meníme parameter *d* v rovnici plochy a posúvame túto plochu smerom ku kamere.

5.3.4 Renderovanie jednotlivých vrstiev

Renderovanie plôch spočíva v stanovaní najvzdialenejšej plochy, ktorou sa začne renderovať a ukončení renderovania vtedy, keď už je objem dát celý vyrenderovaný. V predošlej kapitole bol ukázaný spôsob ako sa vypočítavajú polygóny plôch. Teraz si povieme ako ich vyrenderujeme a presne od kade začneme renderovať plochy.

Na začiatku renderovania si najprv inicializujeme shadery a nastavíme ich parametre, ako je napríklad pozícia svetla, rozmery dátovej matice, atď.. Po inicializácii shaderov sa pustíme do cyklu, ktorý bude počítať a vykresľovať jednotlivé vrstvy.

Rovnica plochy obsahuje normálový vektor, ktorý je rozdiel stredového bodu objemu a polohy kamery v 3D priestore, okrem normálového vektoru, je tam aj parameter *d,* ktorý značí posunutie plochy v smere normálového vektoru. Parameter *d* stanovíme v rozmedzí:

$$d = \langle -(|n_x| l_x + |n_y| l_y + |n_z| l_z), |n_x| l_x + |n_y| l_y + |n_z| l_z \rangle$$

Vektor popisuje dĺžky strán dátového kvádru, vektor je normála na vrstvu. Parameter *d* je určený ako maximálna možná hodnota, ktorá spĺňa podmienku rovnice plochy, keď vieme, že kváder je uložený jedným rohom do počiatku súradnicového systému. Minimálna hodnota parametru je opačná hodnota maxima, lebo kameru, resp. kváder je možné rotovať o 180º.

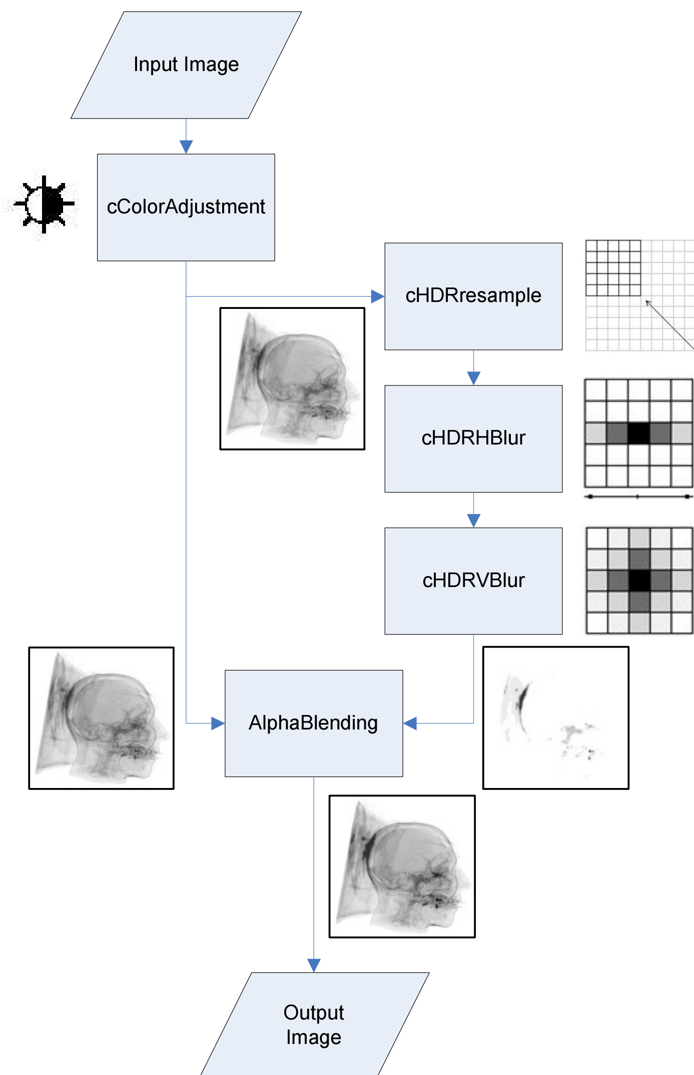
Takto, keď už poznáme hranice cyklu, v ktorom sa bude postupne meniť parameter d, môžeme pre každú vrstvu vypočítať jej priesečníky s dátovým kvádrom a následne tieto priesečníky usporiadať, tak aby popisovali konvexný polygón. Keď máme celú túto prípravu spravenú, tak vykreslíme cez OpenGL polygón. Tento algoritmus sa opakuje v cykle a jednotlivé polygóny sa skladajú na seba.

Pri skladaní polygónov by bolo možné využiť Alpha Blending, avšak v tomto projekte je pre lepšie možnosti využité renderovanie do textúry a Alpha Blending je tak naprogramovaný v Pixel

Shadery. Vždy keď je vyrenderovaný jeden polygón, tak sa pri renderovaní ďalšieho dá predošlý výsledok renderovania na vstup pixel shaderu ako textúra. Využitie FBO je v tomto prípade veľmi účinné, lebo nespomaľuje kopírovanie medzi framebufferom a textúrou.

5.3.5 Postprocessing efekty

Po vyrenderovaní dátovej matice, máme výsledok v textúre. Táto textúra je typu FLOAT, kvôli väčšiemu rozsahu bitovej hĺbky. Celý postprocessing algoritmus sa delí na niekoľko častí, ktoré spracovávajú vstupnú textúru. Na nasledujúcom obrázku vidíme, aký priebeh má spracovanie výsledného obrazu.



Obrázok 35: Postprocessing

Vstupný obraz, teda FBO textúra vyrenderovaná z objemu dát ide na vstup shaderu ColorAdjustment. Ten vyrenderuje do novej FBO textúry pôvodnú textúru s upraveným kontrastom, svetlosťou a gama korekciou.

Ďalej je obraz predaný do shadera HDRresample, ktorý priemeruje 2x2 maticu bodov a vytvára z nich jeden bod. Pre použitím tohto shadera je nutné nastaviť viewport na polovicu, aj výšku aj šírku, aby shader z pôvodnej veľkej textúry zapisoval do polovičnej. Okrem zmenšenia obrazu má tento shader za úlohu odfiltrovať body, ktoré majú menšiu úroveň osvetlenia, ako je požadovaná (70%) – sú nastavené na 0. Body, ktoré majú väčšiu hodnotu osvetlenia, tak sa dostanú do ďalšieho spracovania.

Ďalšie dva shadery HBlur a VBlur slúžia na vytvorenie horizontálneho ožiarenia a vertikálneho ožiarenia.

Po prejdení všetkých týchto stupňov sa vyrenderuje pôvodná textúra obrazu a cez ňu sa vyrenderuje redukovaná textúra s ožiarením. Vo výslednom obraze sa môžeme stretnúť s efektom vyžarovania silného svetla z veľmi svetlých bodov.

6 Dosiahnuté výsledky

Zadanie sa mi podarilo úspešne naprogramovať. Navrhnutým spôsobom sa podaril dosiahnuť predpokladaný výstup. Pri programovaní vznikali problémy, hlavne pri využívaní shaderov, ktoré pracujú s dátami iným spôsobom. Napriek tomu kompilátor CG jazyku vždy úspešne odhalil chyby v CG kóde.

Shadery pri predávaní dát z Vertex shaderu do Pixel Shaderu upravujú hodnoty v premenných. Toto spôsobovalo problémy hlavne pri prenášaní dát ktoré neboli normalizované na rozsah [0.0-1.0]. Tento problém vznikal kvôli špecializovanej komunikácii a využívaniu k tomu viazaných premenných, ako je TEXCOORD, POSITION, COLOR, WPOS. Niektoré z týchto premenných musia byť normalizované na požadovaný rozsah, niektoré nesmú obsahovať záporné hodnoty. Kvôli týmto problémom boli niektoré časti CG kódu, ktoré mohli byť vo Vertex shadery, presunuté do Pixel shaderu. Toto ovplyvnilo rýchlosť výpočtu, lebo Pixel shader musel počítať tieto časti kódu sám a nemohli byť využité interpolačné jednotky umiestnené medzi Vertex shaderom a Pixel shaderom.

Testovania boli robené na grafickej karte NVidia 6600GT, ktorá bola ako jediná dostupná počas celého vývoja tohto projektu. ATI grafické karty nie sú bohužiaľ zatiaľ podporované kvôli ich zlej kompatibilite OpenGL a najnovších rozšírení ARB. Kompletná zostava na ktorej boli dosiahnuté nasledujúce výsledky je popísaná v nasledujúcej tabuľke.

CPU	AMD Athlon 3000+
RAM	2GB 400MHz
MainBoard	Asus A7N8X Deluxe
Graphic card	NV 6600GT, 128MB, AGP

Tabulka 2: Testovací hardware

Minimálne požiadavky na procesor nie sú stanovené, na pamäť RAM sú stanované na 1GB a grafická karta musí spĺňať špecifikáciu OpenGL 2.0, podporovať Shader Model 3.0 a musí to byť NVidia. Na iných zostavách ako boli tu popísané nie je zaručené, že aplikácia pojde.

Projekt bol testovaný aj na grafickej karte ATI X1400, avšak po testoch bolo zistené, že grafická karta, aj keď podporuje OpenGL 2.0, nepracovala správne s FBO, okrem toho bola zistená aj nekompatibilita so Shader Modelom 3.0, ktorý je ako hlavná požiadavka na skompilovanie použitých shaderov.

Objemové dáta sa nahrávajú do pamäti grafickej karty. NVidia GT6600 dokáže spracovávať maximálnu textúru o veľkosti 512×512×512. Tento rozmer síce nie je veľký, ale bohato dostačuje pri rýchlosti renderovania grafickej karty, to znamená že už pri rozmeroch 200×200×200 je framerate vykresľovania dosť nízky a nebolo by reálne využiť celú pamäť pre jednu textúru. Táto pamäť je síce

okupovaná niekoľkými textúrami, ale z testov bolo viditeľné, že za normálnych okolností pamäť o veľkosti 128MB bude dostačovať.

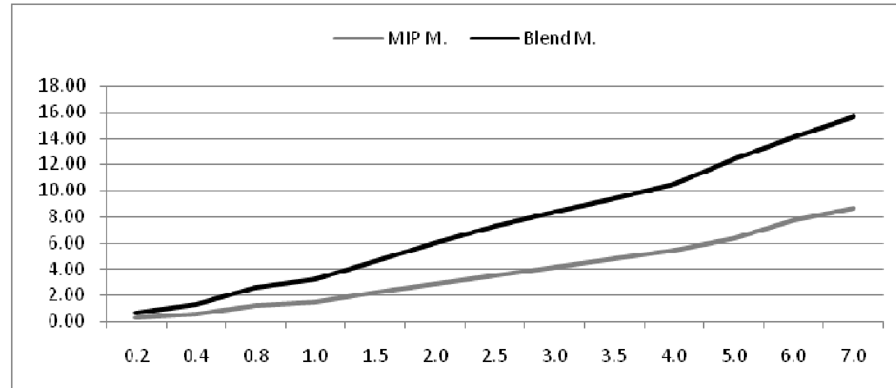
6.1 Rýchlosť vykresľovania

Rýchlosť vykresľovania je veľmi závislá na zobrazovanom objeme, samozrejme čím je objem väčší tým sú nároky na hardware väčšie. Súčasná implementácia obsahuje možnosť nastavenia, ako kvalitne sa má objekt renderovať.

Na nasledujúcej tabuľke a grafe vidíme, ako sa mení rýchlosť vykresľovania. Je tam naznačená závislosť frameratu od nastaveného Quality class. Quality class umožňuje znížiť, resp. zvýšiť počet renderovaných vrstiev objemu. Nasledujúce výsledky boli robené pri rozlíšení 1000×730px a objekt pokrýval zhruba 70% okna.

Quality C.	**0.2**	**0.4**	**0.8**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**5.0**	**6.0**	**7.0**
MIP M.	0.31	0.61	1.21	1.48	2.20	2.90	3.50	4.20	4.83	5.40	6.40	7.70	8.65
Blend M.	0.67	1.33	2.60	3.24	4.60	6.00	7.30	8.40	9.40	10.50	12.40	14.00	15.60
difference	*2.16x*	*2.18x*	*2.15x*	*2.19x*	*2.09x*	*2.07x*	*2.09x*	*2.00x*	*1.95x*	*1.94x*	*1.94x*	*1.82x*	*1.80x*

Tabulka 3: Framerate závislosť na Quality Class



Obrázok 36: Závisloť FPS na Quality class

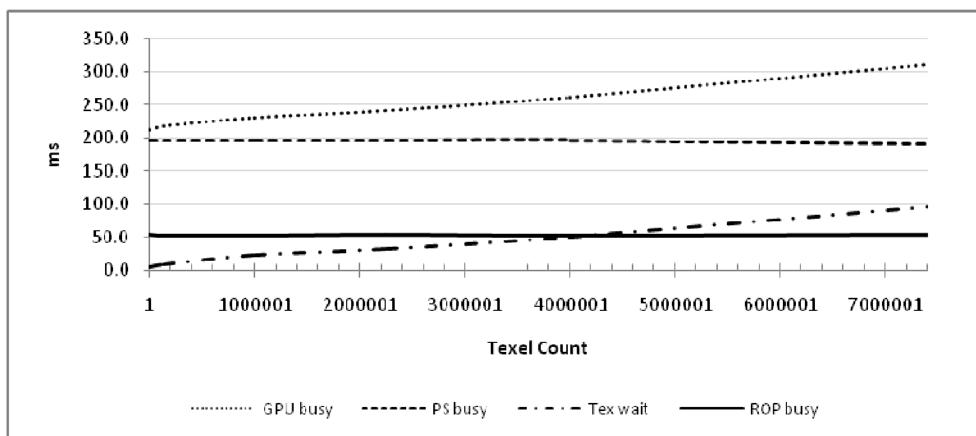
Ako vidíme na grafe (Obrázok 36) a v tabuľke (Tabulka 3), tak pri renderovaní Blend metódou je rýchlosť renderovania zhruba 2× väčšia ako pri renderovaní metódou MIP. Čím je väčšie Quality class tým sa zväčšuje rozdiel násobku medzi Blend a MIP metódou.

Blend Method	Texels	*GPU busy*	VS busy	PS busy	Tex wait	ROP busy	Fill [px]	FPS
DataTexture 1x1x1	1	*99.81%*	0.00%	92.33%	2.48%	24.92%	52351250	4.70
DataTexture 50x50x50	125000	*99.81%*	0.00%	90.11%	4.20%	24.29%	52351254	4.60
DataTexture 100x100x100	1000000	*99.84%*	0.00%	84.86%	9.76%	22.89%	52351222	4.33
DataTexture 150x150x100	2250000	*99.85%*	0.00%	81.09%	13.04%	22.07%	52351192	4.15
DataTexture 200x200x100	4000000	*99.86%*	0.00%	75.04%	19.33%	20.07%	52351242	3.82
DataTexture 256x256x113	7405568	*99.87%*	0.00%	61.45%	30.80%	17.20%	52350834	3.22

Tabulka 4: Blend Metoda - benchmark

V tabuľke (Tabulka 4) vidíme výsledky testov pri ktorých bolo merané zaťaženie jednotlivých častí programovateľného pipelinu v závislosti na veľkosti dátovej textúry pri konštantnom počte vrstiev, teda rovnakom počte renderovaných pixelov. Percentuálne vyjadrenie jednotlivých hodnôt vyjadruje koľko percent času tá daná jednotka pracovala za posledný jeden renderovací cyklus. Vidíme, že Vertex Shader (VS) vôbec nezaťažuje grafický pipeline. Pixel shader zaťažuje pipeline najviac. Jeho pomer vzhľadom na Vertex shader nie je ideálny, lebo nevyužívame plnú kapacitu grafického hardwaru. Nesmieme tiež zabudnúť na textúrovacie jednotky, tie veľmi ovplyvňujú rýchlosť vykresľovania, pri práci s veľkými textúrami. Hodnota Tex wait vyjadruje koľko percent času čakal Pixel shader na dáta z textúrovacích jednotiek. Pri väčších textúrach vidíme, že čas strávený čakaním na textúrovacie jednotky nie je zanedbateľný.

V tabuľke sú vyjadrené hodnoty percentami. Tie však hovoria iba o relatívnom zaťažení. Aby sme vedeli či zmeny vo veľkosti textúry ovplyvňujú záťaž jednotlivých jednotiek, musíme si ich prepočítať na čas. Tento prepočet je možné spraviť pomocou FPS, teda počtu obrázkov za sekundu. Na nasledujúcom grafe vidíme zaťaženie niektorých jednotiek grafického pipelinu.



Obrázok 37: Závislosť zaťaženia GPU na počte texelov

Na grafe vidíme, že veľkosť textúry neovplyvňuje zaťaženie ROP (Raster Operations Unit) a ani Pixel shaderu. Jediné čo je závisle na veľkosti textúry sú textúrovacie jednotky. Čas spotrebovaný pri výbere texelov z textúry sa značne prejavuje aj na výslednom zaťažení celého GPU.

Tieto merania síce ukazujú, veľkú závislosť rýchlosti na veľkosti textúry, ale tiež si musíme uvedomiť, že čím je väčšia textúra tým máme väčší počet vrstiev renderovaného objemu a teda väčší

počet renderovaných pixelov. Posledne meranie v Tabuľke 4, je nameraná hodnota pri nezmenenej textúre a pri potrebnom počte vrstiev na túto textúru. Z tohto riadku môžeme usúdiť, že pri Blend metóde vykresľovania, je pixel shader zaťažený na 60% a k tomu sa zhruba 30% času čaká na textúry.

Meranie Kombinovanej metódy zobrazovania volumetrických dát ukázalo, že Pixel shader pracuje 66% času a k tomu sa 18% čaká na textúry. Pri tejto metóde je počítané osvetlenie objektu a teda je aj väčšie zaťaženie na Pixel shader.

Pri vývoji sa občas stávalo, že niektoré textúry boli odstránené z pamäti grafickej karty pri zmene nastavení okna a OpenGL, to spôsobovalo znehodnotenie obrazu. Kvôli tomuto boli implementované určité časti kódu, ktoré sa starajú o textúry a pri prípadných chybách ich reloadujú. Tieto opatrenia priniesli do behu programu určité spomalenie, ktoré sa prejavuje len v prípade chybných textúr, teda pri ich znovu načítaní.

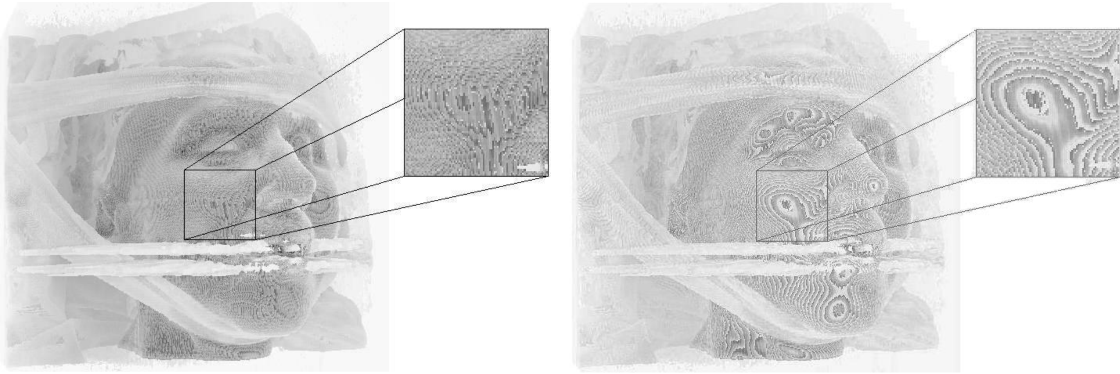
Podobne ako s textúrami, boli aj shadery ovplyvňované oknom a nastavením OpenGL. Preto aj shadery dostali vylepšenie kódu, ktorý sa stará o ich kontrolu a prípadné nové načítanie. Okrem programu shaderov sa tieto časti kódu starajú aj o kontext a nastavenie programovateľného pipelinu. Spomalenie, ktoré je s týmto spojené, sa tiež prejavuje len pri znovu načítaní a iných úpravách na programovateľnom pipeline.

6.2 Kvalita vykresľovania

Kvalita výsledného obrazu je veľmi závislá na Quality class, lebo on určuje ako husto sa ma samplovať objem dát na vrstvy. Okrem Quality class sú pri metóde MIP veľmi podstatné aj normály z normálovej matice. Ich smer a ich presnosť určuje presnosť osvetlenia. Osvetlenie na objeme dát je veľmi viditeľná a dôležitá vec a preto je nutné dávať pozor aj na normály.

Ako sme si už hovorili, normálová matica sa generuje z okolia bodu do vzdialenosti jedného voxelu. Toto nastavenie nie je však vždy ideálne. Z testov sa ukázalo, že je výhodné vytvárať normály z okolia bodu do vzdialenosti dva voxely. Väčší rádius vytvárania normál by mohol mať význam, ale tiež už môže deformovať odtiene povrchu. Nastavenie rádiusu vytvárania normál má podobný vplyv na renderovanie ako nastavenie Quality class.

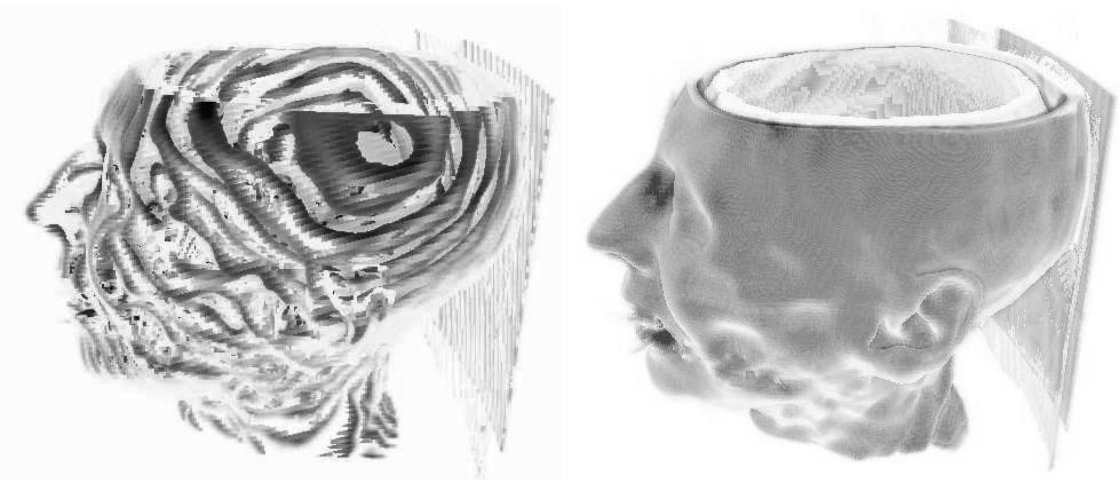
Normálová matica je do grafickej karty nahrávaná ako klasická integer textúra s tromi komponentmi. Pri šetrení miestom v pamäti grafickej karty je možné zvoliť GL_RGB4 interný formát. Ale ako sa ukázalo takéto normály majú nedostatočnú presnosť a spôsobovali ostré prechody svetla medzi zakrivenými plochami. Pri použití GL_RGB8 sa presnosť smerovania normál podstatne zlepšila a kvalita obrazu a osvetlenia viditeľne stúpla.



Obrázok 38: Kvalita obrazu (a: *QC=0.5*; b: *QC=1.0*)

Na predchádzajúcom obrázku vidíme aký vplyv má na výsledný obraz Quality class. Čím je Quality class nižší tým je samplovanie vrstiev väčšie a z toho dôvodu sú odtiene zakrivených povrchov presnejšie a jemnejšie.

Nasledujúci obrázok ukazuje použitie veľmi vysokého Quality class (veľmi zlá kvalita) a použitie veľmi nízkeho Quality class (veľmi dobrá kvalita). Podobný efekt využívame na zrýchlenie renderovania pri pohybe renderovaným objektom.



Obrázok 39: Vplyv QualityClass parametru

7 Záver

Cieľom tohto projektu bolo preštudovať, navrhnúť a implementovať aplikáciu na zobrazovanie volumetrických dát. Súčasný grafický hardware umožňuje implementovať pomerne zložité algoritmy na spracovanie veľkých objemových dát. Tento hardware sa nám podarilo využiť v plnej miere a umožnil nám úspešne tento projekt implementovať a tiež otestovať. Súčasná implementácia bola značne orientovaná na medicínske dáta, kvôli ich jednoduchšej dostupnosti. Pri implementácii nastali problémy s implementáciou shaderov, ale tieto problémy sa podarili vždy vyriešiť. Zadanie bolo kompletne splnené a výsledný projekt dokáže spoľahlivo a pomerne efektívne zobrazovať volumetrické objekty v real-time rýchlosti.

Uplatnenie tohto projektu by mohlo byť v medicíne, alebo v iných odvetviach, kde je nutné zobrazovať volumetriké dáta. Keby bol nasadený do reálneho života, musel by prejsť množstvom ďalších úprav, ktoré by sa hlavne týkali ovládacieho interfacu. Interface pre tento projekt bol navrhovaný len ako vedľajšia úloha. Preto nemôžeme počítať s jeho ideálnou ovládateľnosťou. Popis aktuálneho interfacu a jeho ovládania nájdeme v prílohe.

Okrem interfacu by boli veľmi nutné úpravy na kompatibilite s hardwarom. Tento problém je však možné riešiť len vtedy, ak by bol dostupný požadovaný hardware. Veľké problémy, ktoré nastali boli hlavne s ATI grafickými kartami a s ich implementáciou driverov a OpenGL špecifikácie. V súčasnej realease verzii nie sú tieto problémy vyriešené a preto nie je vhodné testovať tento projekt na ATI kartách.

Tento projekt tiež značne pomohol pri rozšírení obzoru v programovaní grafickej karty a jej programovateľného pipelinu, volumetrických renderovacích techník a získavaniu volumetrických dát.

Literatúra

[1] Randima Fernando: GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics. Addison-Wesley Professional; Bk&CD-Rom edition, 2004

[2] WWW stránky. CG homepage
http://developer.nvidia.com/page/cg_main.html

[3] WWW stránky. VLIB Volume Graphic API
http://vg.swan.ac.uk/vlib/

[4] WWW stránky. Medical Datasets
http://www.volvis.org/

[5] WWW stránky. Volume Rendering
http://en.wikipedia.org/wiki/Volume_rendering

[6] WWW stránky. Magnetic resonance imaging
http://en.wikipedia.org/wiki/MRI

[7] WWW stránky. 3D scanners
http://en.wikipedia.org/wiki/3D_scanner

[8] WWW stránky. Fast Volume Rendering Using a Shear-Warp Factorization
http://graphics.stanford.edu/papers/shear/shearwarp.pdf

Zoznam príloh

Príloha 1.: Manuál

- inštalácia
- príkazový riadok
- ovládanie

Príloha 2.: Ukážky (screenshots)

Príloha 3.: DVD

- zdrojové texty
- doxygen manuál
- ukážky (screenshots)
- release verzia projektu
- rôzne dokumenty ku knižniciam a knižnice potrebné na kompiláciu projektu

Príloha 1.: Manuál

Inštalácia

Inštalácia aplikácie je veľmi jednoduchá a je ju nutné robiť len v prípade, žeby sme chceli ukladať predvolené nastavenia aplikácie (testovacie dáta a ich konfiguračné súbory).

Inštalácia spočíva v skopírovaní programu na vami zvolené miesto. Pri kopírovaní nie je nutné kopírovať súbory z adresáru ./data/, súbory v tomto adresári sú len testovacie a nie sú nutné k behu programu.

Ak budeme spúšťať testovacie dáta priamo z DVD média nesmieme zabudnúť, že nebude možné ukladať konfiguračné súbory. Aplikácia ukladá konfiguračné dáta k dátovému súboru, ak nie je inak špecifikované cez príkazový riadok.

	Konfigurácia PC	
	Minimálna	Doporučená
CPU	1.5GHz	3GHz
RAM	1GB	2GB
GPU	NV 6600 (128MB)	NV 7900GTO (256MB)

Tabulka 5: Minimálna a doporučená konfigurácia HW

Príkazový riadok

Príkíkazový riadok sĺúži na špecifikáciu niektorých počiatočných nastavení programu. Tiež je možné pomocou neho vygenerovať testovacie dáta.

VolumeRendering.exe [-ar] [-c confname] [-d datafile] [-f fpslimit] [-h] [-q qualityclass] [-t w h d]

Options:

-ar	*Always redraw volume object (disable redraw optimization)*
-c confname	*Load configs from specified file base name*
-d datafile	*Load object from data file*
-f fpslimit	*Set max FPS*
-h	*Show this help*
-q qualityclass	*Quality class [0.1-inf]*
-t w h d	*Generate test object WxHxD size*

Parameter "-ar" slúži na vypnutie optimalizácie vykresľovania objektu, je vhodne ho zapnúť pri testovaní FPS renderovania objektu.

Parametre "-c" a "-d" dovoľujú špecifikovať konfiguračné súbory a dátoví súbor. Ak sa nešpecifikujú konfiguračné súbory, tak pri ukladaní nastavení sa konfiguračné dáta ukladajú do súborov k dátovému súboru.

Parameter "-f" nastavuje maximálne dovolené FPS. Tento parameter je vhodný na zníženie záťaži CPU ak je to možné.

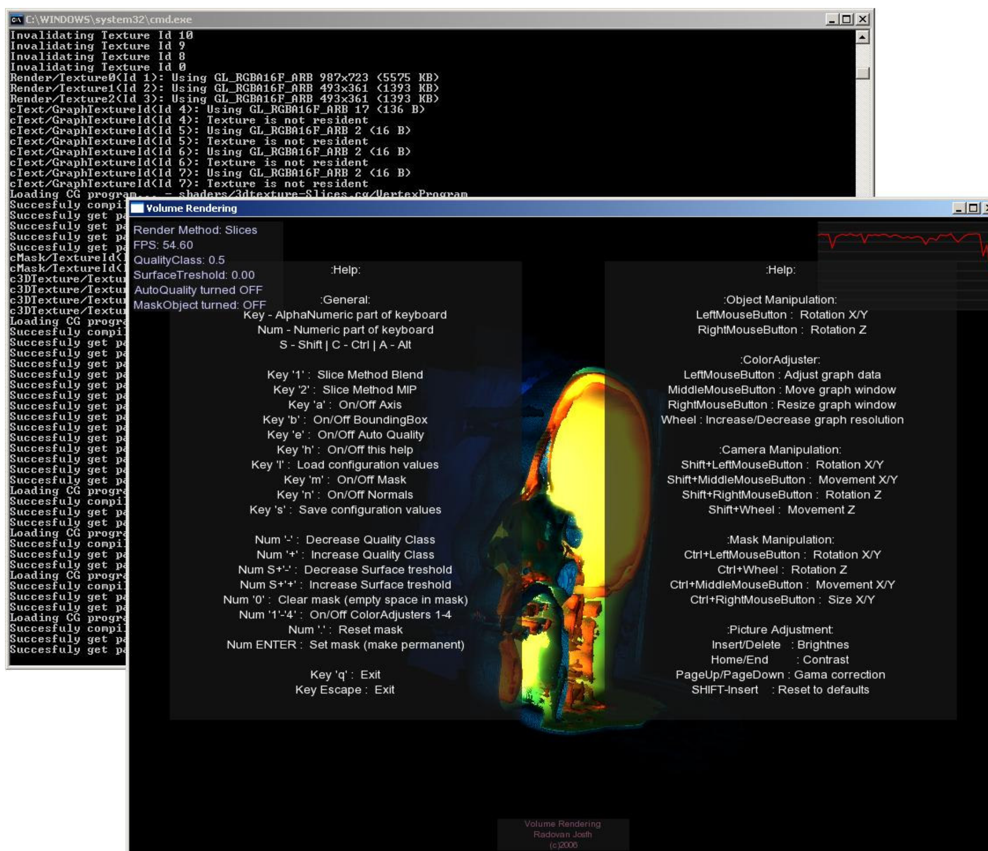
Parameter "-q" nastavuje počiatočný Quality class pre renderovanie vrstiev.

Parameter "-t" umožňuje vygenerovať testovacie dáta so špecifikovanou veľkosťou.

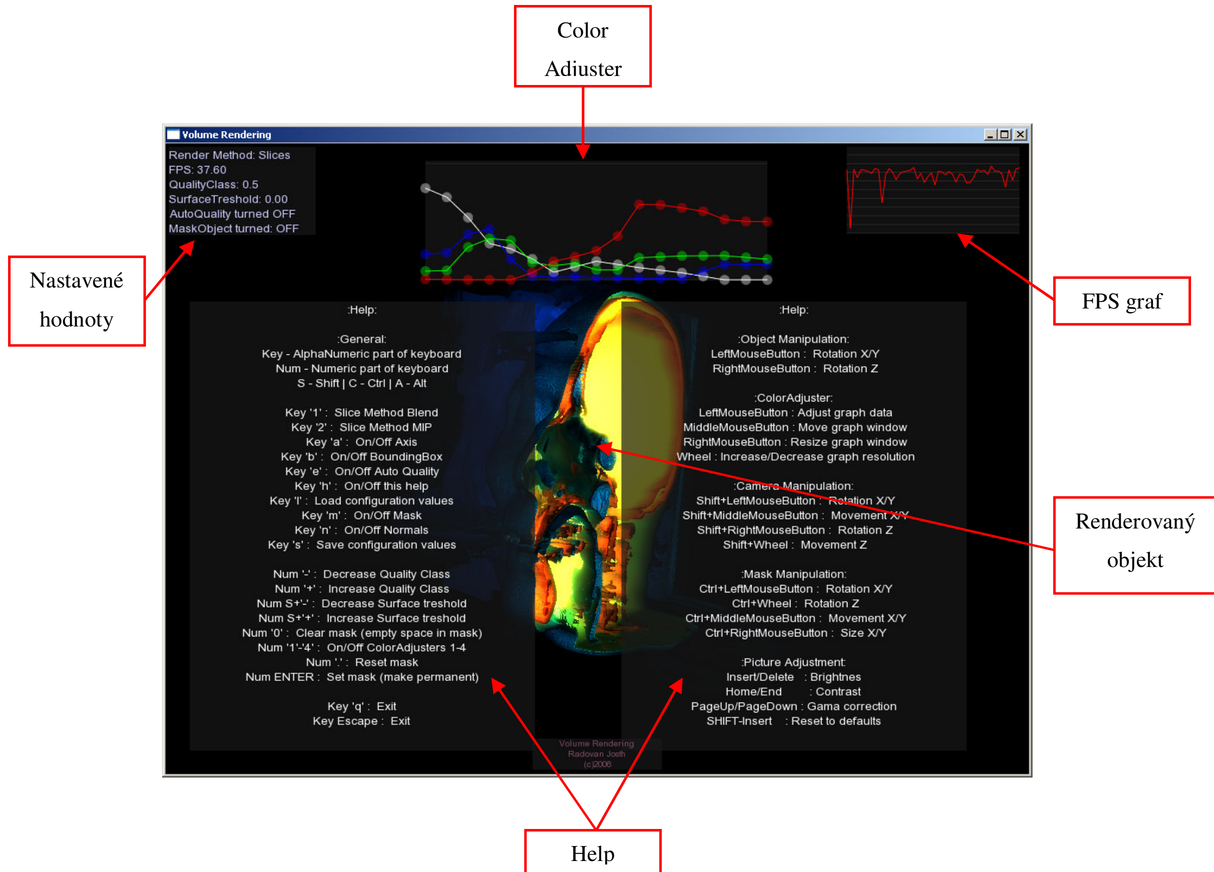
Ovládanie

Okrem príkazového riadku, ktorý je popísaný v predošlej časti, je ovládanie aplikácie robené pomocou klávesnice a myši. Ovládacích prvkov nie je síce veľa ale niektoré nastavenia sú pomerne zložité a vyžadujú určité skúsenosti! Preto je vhodné si najprv chvíľku testovať ovládanie na pripravených dátach a získať základnú predstavivosť o význame ovládania.

Po spustení aplikácie nás privíta úvodná obrazovka na ktorej je zobrazený jednoduchý návod na použitie. V pozadí uvidíme textovú konzolu, ktorá slúži na podrobnejšie informácie o aktuálnom stave aplikácie.



Obrázok 40: Po spustení...

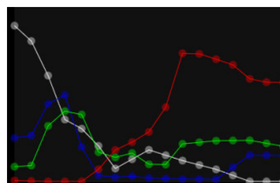


Obrázok 41: Rozloženie ovládania

Aby sme mohli ovládať tento projekt, museli byť k nemu navrhnuté ovládacie prvky. Tieto prvky vidíme na predchádzajúcom obrázku. Plocha pre Nastavené hodnoty, FPS graf a Help sú čisto informačné. Plocha Color Adjustera je jedna z najdôležitejších súčastí ovládania. Nastavenie color Adjustera ovplyvňuje transfer funkciu, ktorá výsledne ovplyvňuje reprezentáciu dátových hodnôt.

Color Adjuster:

Color adjusterom nastavujeme reprezentáciu dát na farebné hodnoty, tzv. transfer funkciu. V aplikácii sú štyri Color Adjustery, ktoré zapíname klávesmi “1“-“4” na numerickej klávesnici. Os X reprezentuje dátové hodnoty. Os Y reprezentuje intenzitu farby RGBA.



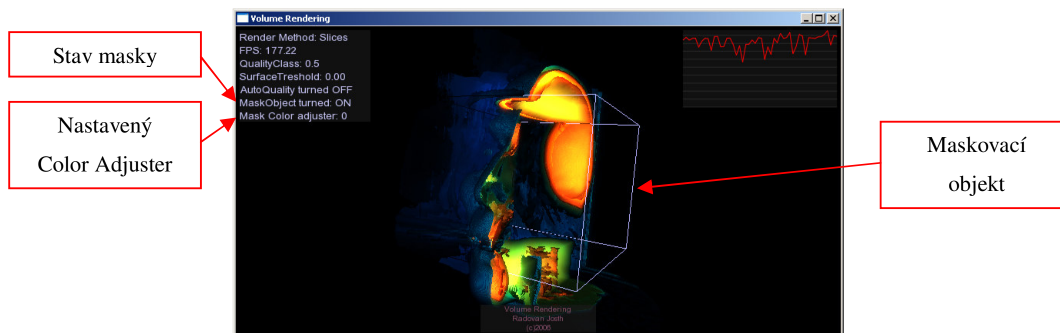
Obrázok 42: Color Adjuster

Pri Blend metóde, vyjadruje A-alpha skutočnú priesvitnosť. Pri MIP metóde, vyjadruje A-alpha intenzitu, podľa ktorej sa rozhoduje, ktorý materiál má najväčšiu intenzitu.

Nastavenie bodov sa vykonáva **ľavým tlačidlom** na myši. **Prvým tlačidlom** myši nastavujeme veľkosť okna Color Adjustera. **Stredným tlačidlom** posúvame okno Color Adjustera. **Kolieskom** myši nastavujeme hustotu referenčných bodov.

Maskovanie dát:

Maskovanie objektu slúži na rôzne zvýrazňovanie určitých častí dát. Inými slovami to znamená, že pre rôzne miesta môžeme nastaviť rôzne Color Adjustery, prípadne spriesvitniť danú oblasť.



Obrázok 43: Maskovanie objektu

Masku aktivujeme klávesov “m”, po aktivácii uvidíme v objeme dát maskovací objekt. Dáta v maskovacom objekte budú podliehať nastavenému Color Adjusteru. Ak je Id Color Adjusteru “0“, tak to znamená, že dáta budú priesvitne (odstránené), inač budú dáta ofarbené na farbu aktívneho Color Adjusteru.

Aktívny Color Adjuster nastavíme tak, že si otvoríme niektorý zo štyroch Color Adjusterov a klikneme ľavým tlačidlom na jeho okno. Ak chceme aby dáta boli priesvitné, stlačíme na numerickej klávesnici “0“.

Rotáciu, pohyb a veľkosť nastavujeme tlačidlami a kolieskom na myši. Tak, že držíme tlačidlo a pohybujem myšou.

Po nastavení maskovacieho objektu a príslušného Color Adjusteru, stlačíme ENTER na numerickej klávesnici a maska sa permanentne uloží. Po tomto uložení masky môžeme aktivovať masku znovu a vymaskovať inú časť dát na iný Color Adjuster.

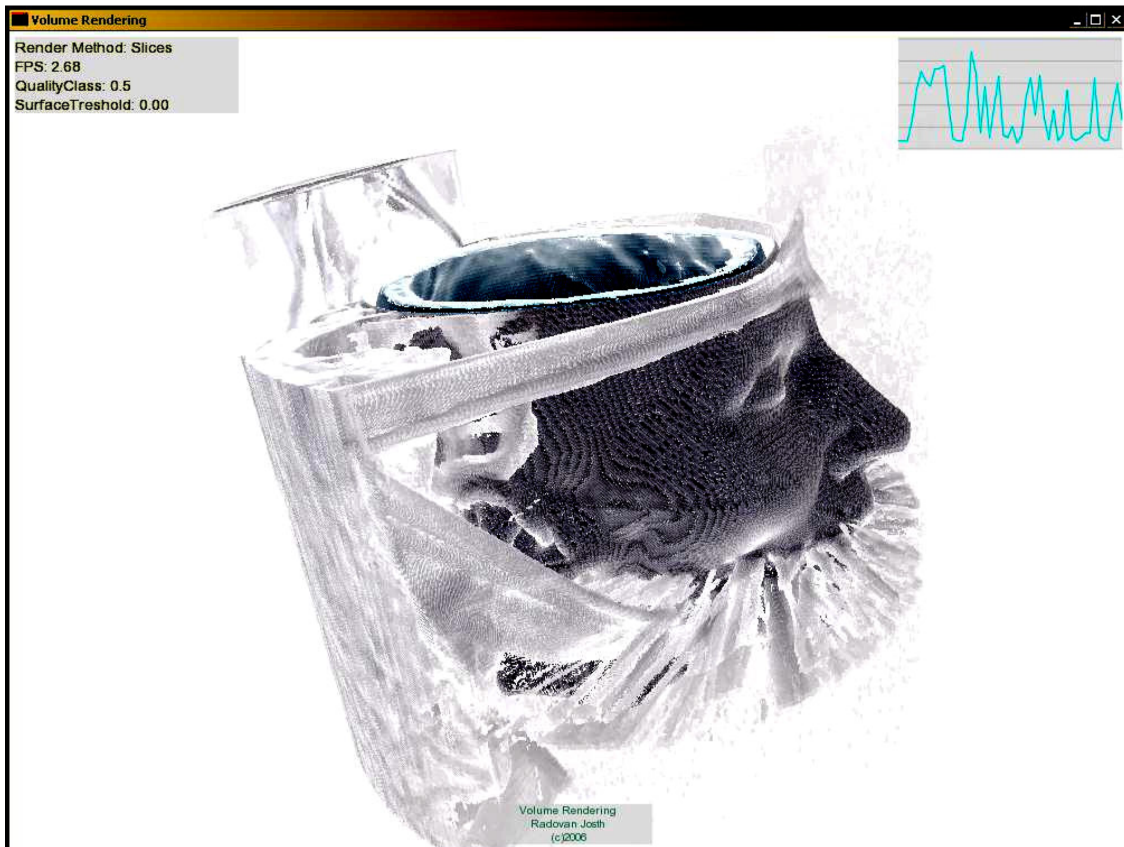
Tipy na ovládanie:

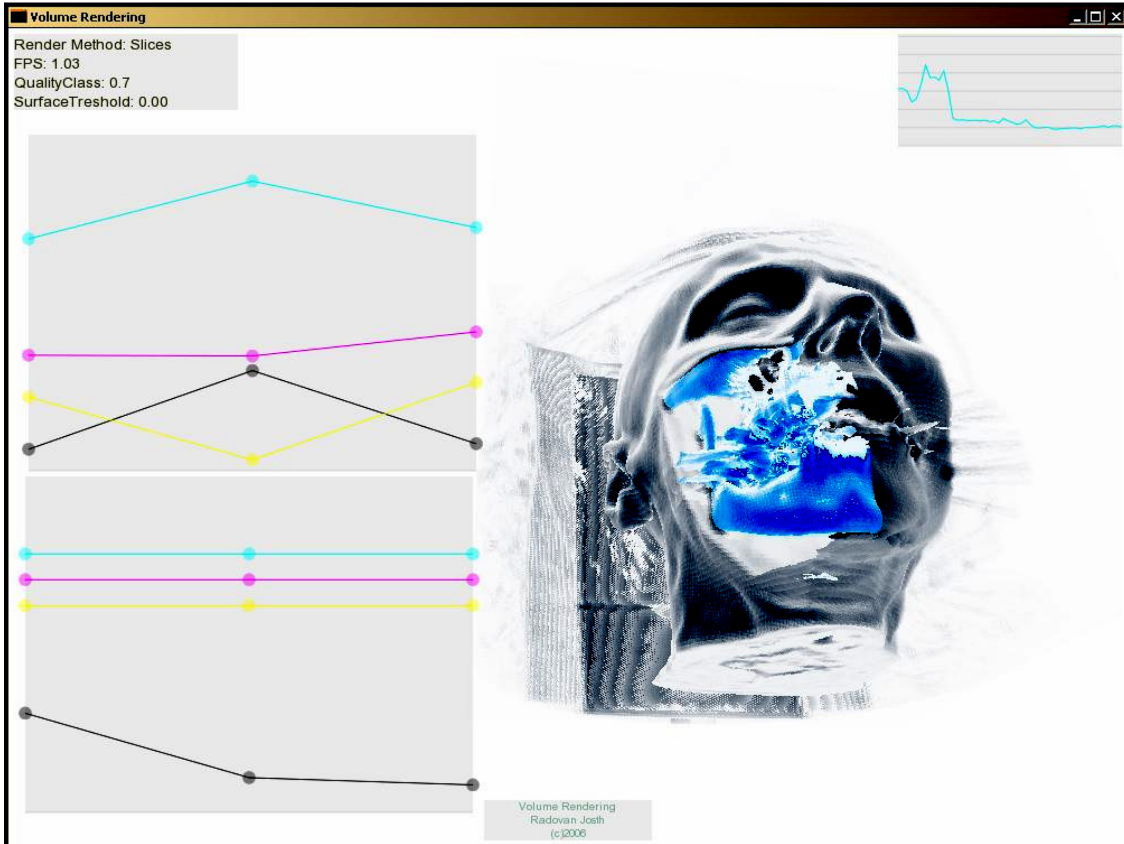
- **Quality class** nastavujeme klávesmi “+” a “-“ na numerickej klávesnici. Nevoľte hneď najväčšiu kvalitu (najmenší quality class), aby ste predišli problémom s FPS a stým spojeného ovládania
- Pri manipulácii z objektom si zapnite **Auto Quality** – “e” – umožňuje automatickú zmenu Quality class počas pohybu – lepšie ovládanie
- Ohraničenie objektu je možné vidieť po zapnutí **Bounding Box** – “b”

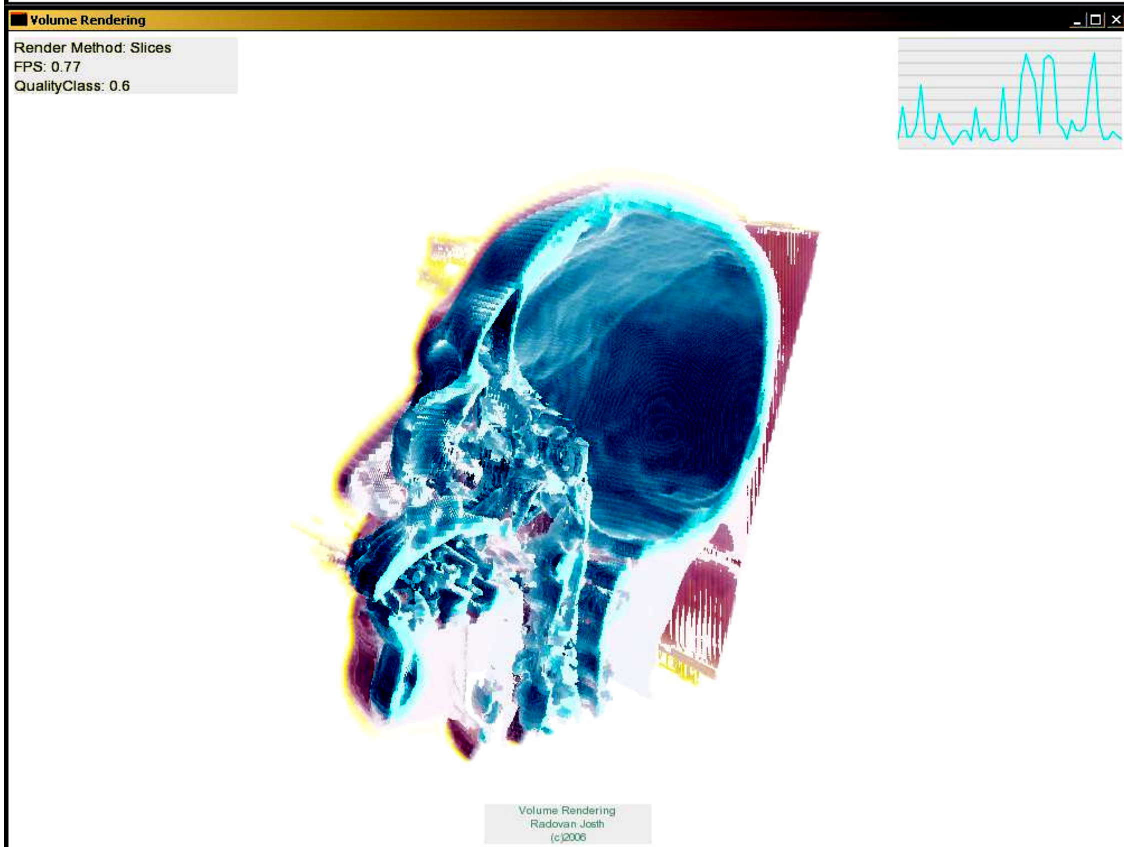
- **Vykresľovanie normál** značne spomaľuje vykresľovanie – “n“
- **Uloženie do konfiguračného súboru** spravíte klávesov “s” – predtým je vhodne špecifikovať cez príkazový riadok názov konfiguračného súboru, inač bude konfigurácia uložená pod názvom dátového súboru
- **Veľkosť zobrazovacieho okna** má vplyv na zaťaženie grafickej pamäte – nenastavujte veľké okno ak máte staršiu grafickú kartu (128MB)
- **Color adjuster** je možné rozšíriť za okraje okna – pomôže to pri zobrazovaní hustého grafu
- **Maskovanie** využíva štyri Color Adjustery + jedenu transparent masku, to znamená, že len 4+1 oblastí môžeme maskovať

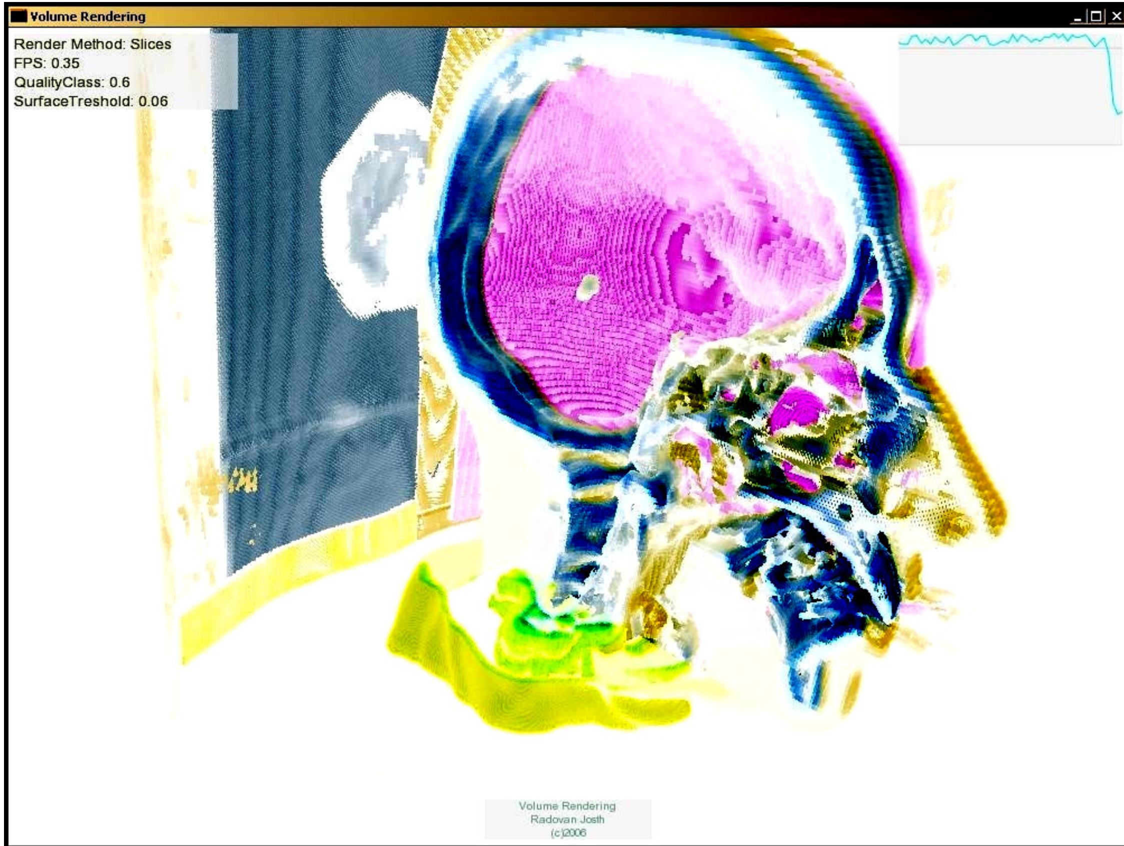
Príloha 2.: Ukážky (screenshots)

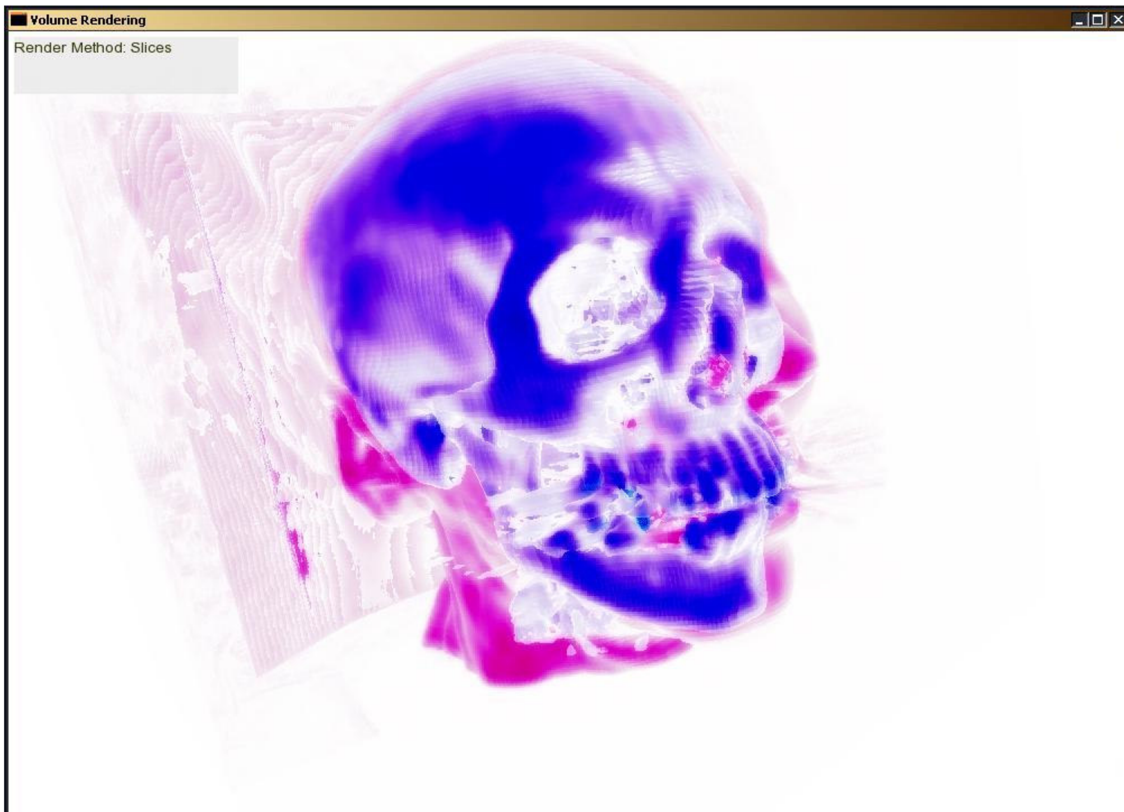
Nasledujúce ukážky sú farebne invertované kvôli tlači. Všetky tieto ukážky a plno ďalších si môžete pozrieť na priloženom médiu.

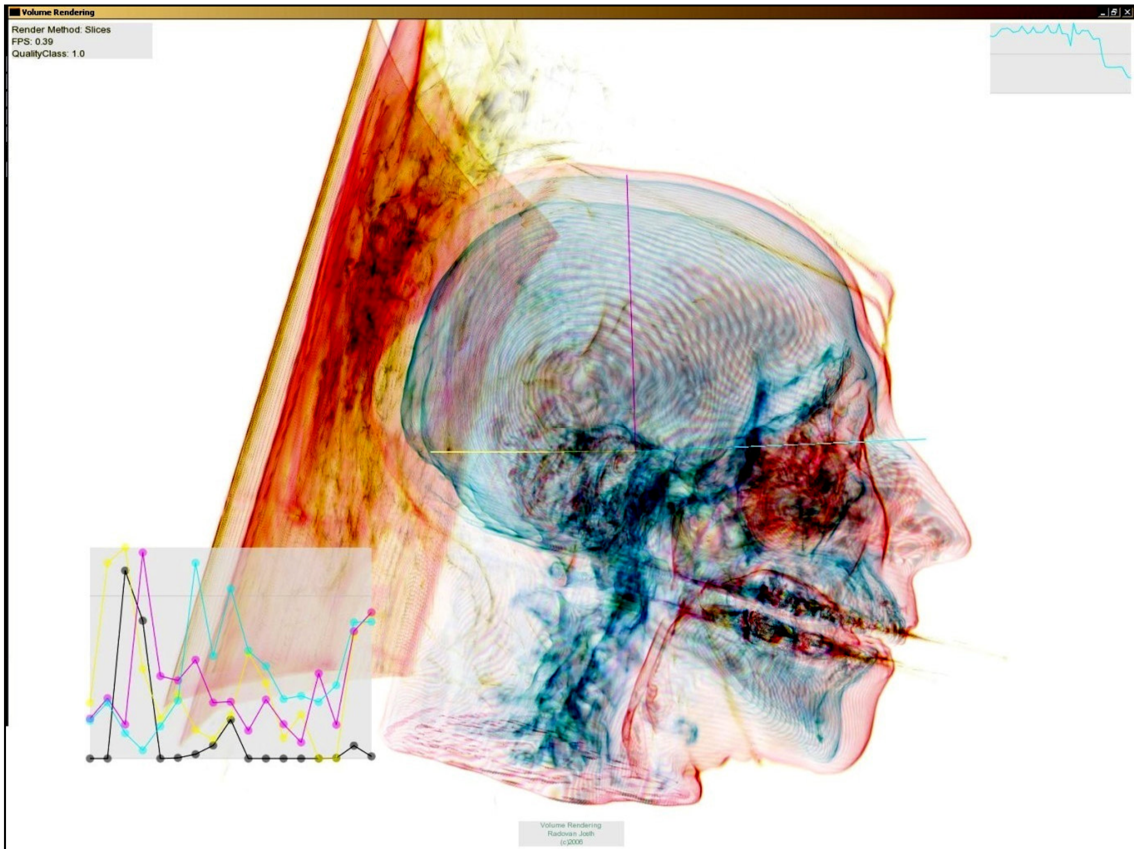


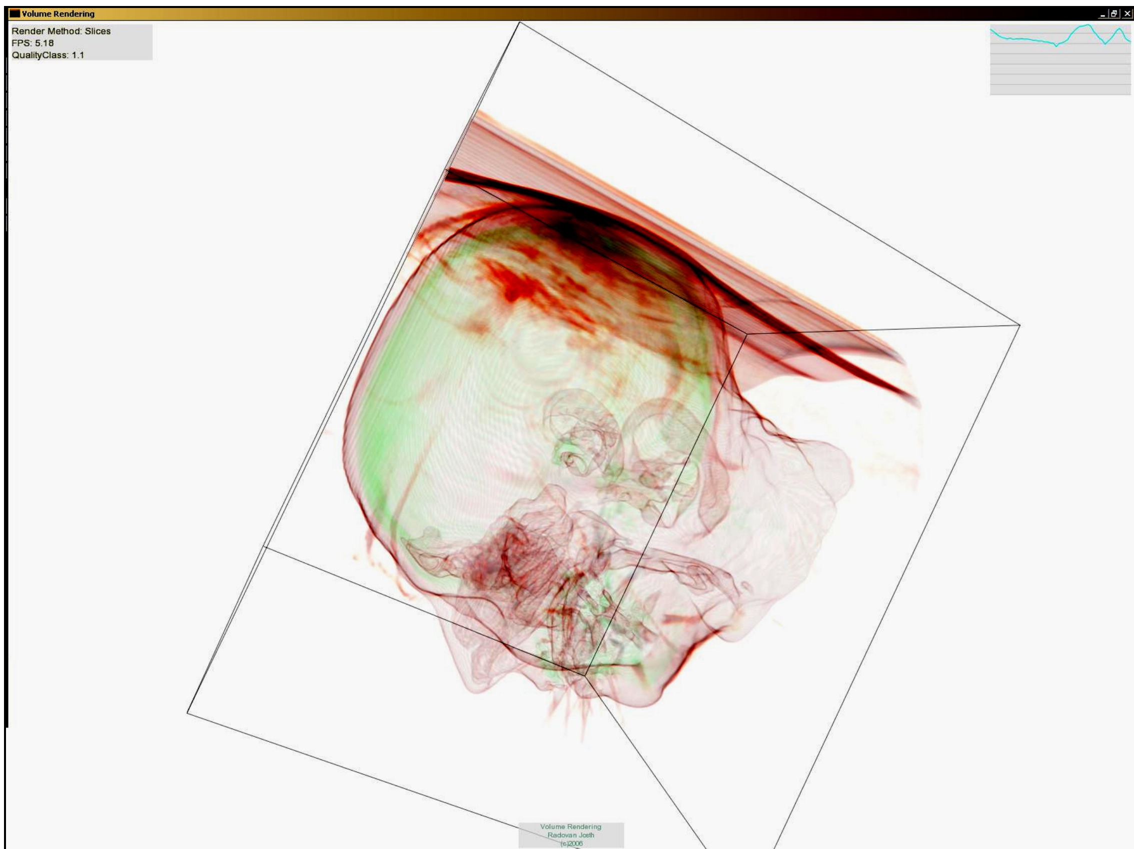

Volume Rendering
Render Method: Slices
FPS: 1.03
QualityClass: 0.7
SurfaceTreshold: 0.00
Volume Rendering
Radovan Josth
(c)2006

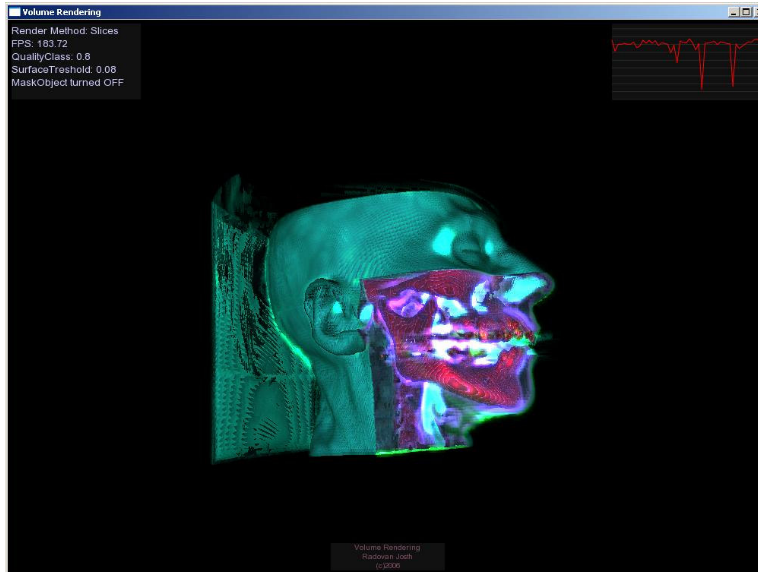

Volume Rendering
Render Method: Slices
FPS: 0.77
QualityClass: 0.6
Volume Rendering
Radovan Josth
(c)2006

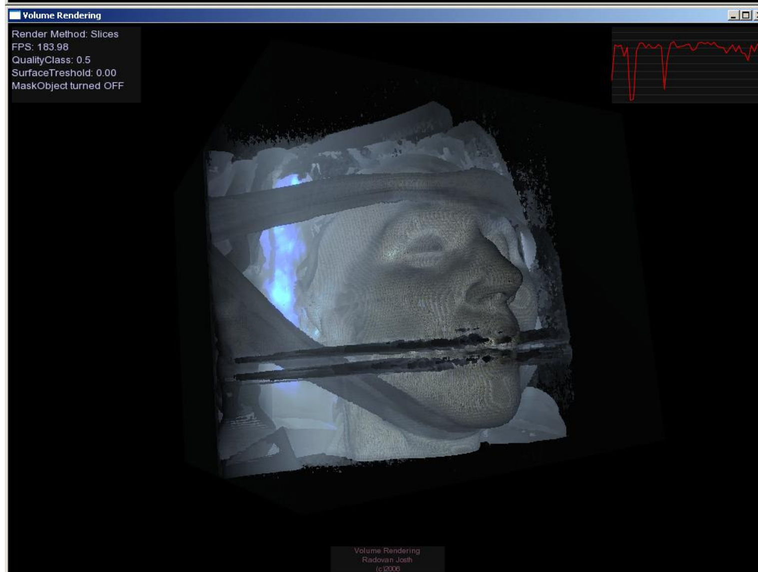

Volume Rendering
Render Method: Slices
FPS: 0.35
QualityClass: 0.6
SurfaceTreshold: 0.06
Volume Rendering
Radovan Josth
(c)2006

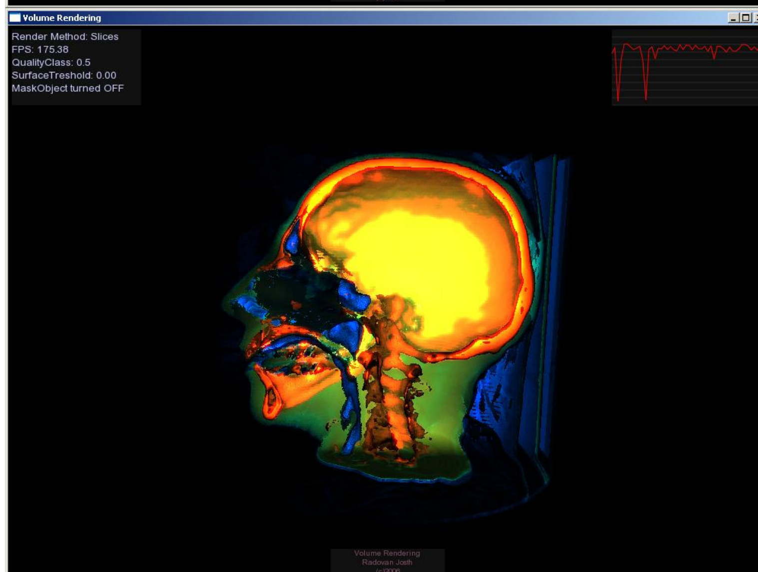

Volume Rendering
Render Method: Slices


Volume Rendering
Render Method: Slices
FPS: 0.39
QualityClass: 1.0
Volume Rendering
Radovan Josth
(c)2006


Volume Rendering
Render Method: Slices
FPS: 5.18
QualityClass: 1.1
Volume Rendering
Radovan Josth
(c)2006


Volume Rendering
Render Method: Slices
FPS: 183.72
QualityClass: 0.8
SurfaceTreshold: 0.08
MaskObject turned OFF


Volume Rendering
Render Method: Slices
FPS: 183.98
QualityClass: 0.5
SurfaceTreshold: 0.00
MaskObject turned OFF


Volume Rendering
Render Method: Slices
FPS: 175.38
QualityClass: 0.5
SurfaceTreshold: 0.00
MaskObject turned OFF