Fakulta matematiky, fyziky a informatiky
Univerzita Komenského, Bratislava

Diplomová práca

Michal Červeňanský 2004

Fakulta matematiky, fyziky a informatiky
Univerzita Komenského, Bratislava
Katedra počítačovej grafiky a spracovania obrazu

Diplomová práca

Využitie komerčných grafických akcelerátorov pre vizualizáciu a spracovanie objemových dát

Diplomant: Michal Červeňanský
Školiteľ: Ing. Miloš Šrámek

Bratislava 2004

Čestné prehlásenie:
Čestne prehlasujem, že prácu som vypracoval samostatne pod odborným vedením školiteľa a len s použitím uvedenej literatúry.

.....
Michal Červeňanský

Poďakovanie:

Chcel by som sa poďakovať môjmu školiteľovi Ing. Milošovi Šrámkovi za odbornú pomoc pri tejto diplomovej práci, jeho cenné rady a pripomienky. Poďakovanie mu patrí aj za sprostredkovanie študijného pobytu v Rakúsku. Ďalej ďakujem svojmu kamarátovi Jurajovi Starinskému za pomoc pri problémoch s OpenGL. Taktiež ďakujem aj rodine a priateľke za morálnu podporu.

Obsah

1 Úvod

Počítačová grafika má veľké uplatnenie v mnohých vedných disciplínach, rovnako ako aj v zábavnom priemysle. Rozvoj počítačovej grafiky je hnaný používateľmi, ktorý chcú stále reálnejšie stvárnenie okolitého sveta pomocou počítača. Z tohoto dôvodu vznikajú nové algoritmy, ktoré sa snažia dosiahnuť vysokú dôveryhodnosť zobrazovania. Tento rozvoj by nemohol napredovať bez neustáleho vývoja nových grafických akcelerátorov. Výkon nových akcelerátorov enormne narastá a v niektorých parametroch prekonáva aj parametre hlavných procesorov osobných počítačov. Avšak požiadavky používateľov väčšinou o krok napredujú. Preto sa pri návrhu nových algoritmov berie na zreteľ aj optimalizácia. Jednou z významných častí počítačovej grafiky je aj objemové zobrazovanie. V tejto oblasti je mnoho algoritmov, ktoré sa snažia využívať možnosti grafických kariet na urýchlenie aplikácií. Témou predkladanej diplomovej práce je využitie nových možností grafických kariet pre zobrazovanie objemových dát. Jej teoretické výsledky sú implementované v aplikácii, na ktorej je možné si tieto nové možnosti vyskúšať.

2 Vizualizácia objemových dát

Objemové zobrazovanie reprezentuje metódy zamerané na spracovanie trojrozmerných skalárnych dát za účelom ich zobrazenia v dvojrozmernej rovine [6,18]. Prvé požiadavky na zobrazovanie objemových dát prišli z medicínskych potrieb, avšak postupne sa oblasť používateľov objemového zobrazovania rozrástla aj do iných odvetví ako sú napríklad meteorológia, analýza molekulárnych štruktúr, fyzikálne simulácie, seizmické merania a ďalšie.

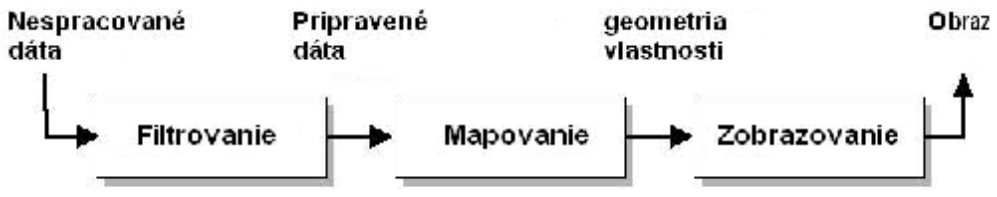
2.1 Objemové dáta

Objemové dáta, v porovnaní s povrchovými dátami, ktoré reprezentujú len daný povrch telesa, sú používané na popísanie celej vnútornej štruktúry objektu. Objemové dáta dovoľujú modelovať tekuté a plynné objekty rovnako, ako sa vyskytujú v prírode (napr. oblaky, hmla, oheň a voda).

Objemové dáta sa dajú získať z meraní, alebo matematickým výpočtom. Pod meraním si predstavujeme získanie dát na určitom snímacom zariadení, napríklad počítačová tomografia (computer tomography - CT), magnetická rezonancia (magnetic resonance - MRI) a ďalšie. Pod matematickým výpočtom rozumieme popis nejakej simulácie pomocou rovníc, kde hodnoty v jednotlivých bodoch získavame výpočtom z popisujúcich rovníc. Ak tento model opisuje trojrozmerný objekt hovoríme o *voxelizácii.*

2.3 Objemové zobrazovanie

Proces zobrazovania objemových dát je často popisovaný ako *visualization pipeline*, kde na obrázku 2.1 je zobrazená jeho základná verzia. Vstupom do procesu zobrazovania sú nespracované dáta, z ktorých je zvyčajne len istá časť zaujímavá. Filtrovaním sú dáta analyzované a prevzorkované (interpolované) podľa používateľom definovaných kritérií. Mapovanie tvorí podstatnú časť, kde sú detekované štruktúry. Tento krok môže byť jednoduchý a to definovaním, ale aj komplexný a to prehľadávaním celého objemu. Cieľom je previesť abstraktnú informáciu do zobrazovateľných tvarov, ktoré sú potom rozložené na základné primitívy (napr. trojuholníky) a určenie vizuálnych veličín (farba, priehľadnosť). V zobrazovacom kroku sa použijú základné primitívy na zobrazenie výsledného objektu (objemu). Časti týchto krokov môžu byť urýchlené pomocou grafických akcelerátorov.



obrázok 2.1: Proces zobrazovania objemových dát (visualization pipeline).

3 Algoritmy zobrazovania objemových dát

Medzi 1D, 2D dátami a 3D dátami je zreteľný rozdiel, ktorý spočíva v priamej vizuálnej reprezentácii. Jednorozmerné dáta môžeme reprezentovať ako čiaru alebo nejaký graf a dvojrozmerné dáta ako obrázok, či fotografiu. Pre trojrozmerné dáta nemáme podobnú vnímateľnú reprezentáciu [6,18]. Pokiaľ máme napríklad 3D maticu dát z CT tomografu reprezentujúcu povedzme ľudskú hlavu, nemôžeme ju vziať a pozrieť sa čo je vo vnútri. Z vývojového hľadiska sme zvyknutí vnímať objekty a ich povrchy nachádzajúce sa vôkol nás. Náš systém vnímania je teda povrchovo orientovaný, avšak v tomografických dátach nie sú objekty a povrchy. Sú v nich iba body a hodnoty reprezentujúce materiálové vlastnosti nasnímaného tkaniva. Preto potrebujeme špeciálne algoritmy na reprezentáciu týchto dát.

3.1 Nepriame metódy – povrchové zobrazovanie

Nepriame metódy vyberajú homogénne oblasti s rovnakými alebo podobnými vlastnosťami a zobrazujú povrch danej oblasti. Do tejto kategórie spadajú všetky prístupy, ktoré transformujú *voxel (volume element)* na povrchovú reprezentáciu v rámci predspracovania. Tento výsledný povrch je v závere zobrazený tradičnými zobrazovacími technikami. Vo vzťahu k zobrazovaciemu procesu zaberá najväčšiu časť výpočtu mapovanie.

Hlavné nevýhody týchto techník:

- model je oddelený od pôvodných dát a tým sa stráca podstatná časť informácií
- je potrebný veľký počet mnohouholníkov na určenie komplexného povrchu
- je ťažké simulovať neostrý, beztvarý povrch (oblaky, oheň)

Na druhej strane geometrické mnohouholníky sú ľahko popísateľné s možnosťou vhodného uloženia, kompresie a prenositelnosti. Interaktivita je dobre dosiahnuteľná použitím štandardných grafických akcelerátorov (pokiaľ trojuholníkov nie je príliš veľa).

3.1.1 Vyhľadávanie kontúr

Väčšina tomografických snímačov produkuje dáta v rezoch. Základný princíp týchto algoritmov je načrtnutý v nasledujúcich dvoch krokoch:

1. v každom reze sa detekuje hranica, kontúra objektu daného segmentáciou pre určitú hraničnú hodnotu.
2. určí sa približný povrch objektu medzi susednými rezmi pomocou záplat, najčastejšie pomocou trojuholníkov.

Sú používané rôzne metódy na určenie povrchu ako maximalizovanie vnútorného objemu, minimalizovanie celkového povrchu objektu alebo iné.

Avšak automatické určovanie kontúry nemusí viesť ku korektnému povrchovému modelu. Počet kontúr pre susedné rezy nemusí byť totožný a môžu nastať problémy aj medzi konvexnými a konkávnymi kontúrami.

3.1.2 Cuberille model

Tento algoritmus je založený na prirodzenom ponímaní povrchu daného objektu [18], definovaného pre binárnu scénu zjednotením voxelov s hodnotou 1. Ďalej predpokladajme, že objekt je pospájaný tak, že dva ľubovolné voxely sú spojené iným voxelom, alebo minimálne majú spoločnú jednu hranu. Tento algoritmus má dve výhody:

1. dokáže izolovať objekt záujmu z množiny objektov, ktoré sú nespojité a
2. môže rátať objem objektu.

Tak dostávame povrch objektu definovaný ako sadu stien voxelov s hodnotou rovnej 1, oddelené od okolitých voxelov tvoriacich pozadie. Algoritmus nekladie obmedzenia na komplexnosť objektu. Povrch je definovaný jednoznačne a je tvorený z útvarov rovnakého tvaru, čo zjednodušuje jeho spracovanie. Takto definovaný povrch môžeme reprezentovať pomocou orientovaného grafu (*boundary diagraph*), alebo pomocou stromovej štruktúry.

3.1.3 Implicitné povrchové pokrývanie

Implicitné povrchové algoritmy aproximujú vnútorný povrch, napríklad: povrch definovaný funkciou $f(x,y,z) = T$ sadou mnohouholníkov. Delia priestor na neprekrývajúce sa, najčastejšie kockové bunky. Pre každý vrchol bunky je definovaná jeho hodnota. Algoritmy nachádzajú povrchové bunky, ktorých hodnoty vo vrcholoch sú v nejakom okolí hraničnej hodnoty T. V nich nájdu priesečníky s hranami bunky a spájajú ich do mnohouholníkov, trojuholníkov. Priesečníky môžu byť nájdené buď interpoláciou príslušných vrcholov, alebo priamo z funkcie f.

Najznámejším algoritmom spadajúci do tejto kategórie je algoritmus *Marching Cubes* [12], navrhnutý pre vytváranie trojuholníkového modelu z 3D medicínskych dát. Algoritmus spracováva scénu bunka po bunke s nasledujúcimi krokmi:

1. detekuje bunky s danou hraničnou hodnotou a počíta index do tabuľky definovanej 256-mi možnosťami usporiadania trojuholníkov,
2. definuje pozície vrcholov trojuholníkov pomocou interpolácie medzi vrcholmi bunky,
3. ráta jednotkové vektory pre každý vrchol bunky na základe gradientu a interpolácie.

Konečný krok, zobrazenie trojuholníkovej siete môže byť prenechané grafickému zariadeniu s použitím Gouraudovho tieňovania. Pri počítaní sa môžu vyskytnúť na povrchu diery.

3.2 Priame metódy – objemové zobrazovanie

Objemové zobrazovanie (*volume rendering*) zobrazuje dáta priamo bez vytvárania pomocnej povrchovej reprezentácie. Tento prístup má výhodu v tom, že môžeme vizualizovať polopriehľadné materiály a štruktúry vnorené do iných štruktúr. Tieto techniky umožňujú zobrazovať rozhranie medzi materiálmi a aj ich vnútro.

Pre objemové zobrazovanie sa používajú dva hlavné prístupy. Objektovo a obrazovo orientované alebo hybridné techniky, ktoré kombinujú prvé dve. Objektovo

orientované zobrazovanie využíva mapovanie dát na zobrazovaciu rovinu. V obrazovo orientovaných algoritmoch je pre každý pixel zo zobrazovacej roviny vrhaný lúč cez celé dáta na určenie výslednej hodnoty daného pixela. Niektoré algoritmy na objemové zobrazovanie pozostávajú z niekoľkých krokov, kde sa ako prvé používajú objektovo orientované techniky nasledované obrazovo orientovanými technikami, ktoré určujú hodnotu výsledného pixelu, alebo naopak. Tieto algoritmy spadajú do kategórie hybridného objemového zobrazovania.

3.2.1 Objektovo orientované techniky

Objektovo orientované techniky [19] začínajú s jedným voxelom a počítajú jeho príspevky, ktoré sú premietnuté do výsledného obrazu. Tento výpočet sa iteratívne vykonáva pre všetky dátové voxely. Algoritmy pracujúce v poradí zozadu dopredu, prehľadávajú dáta v danom smere a premietajú dané hodnoty do obrazu. Ak je potrebné, tak ich prepíšu, alebo využijú na výpočet výslednej hodnoty. V opačnom poradí pracujú algoritmy, ktoré využívajú výhodu elementov premietnutých do už vykreslených regiónov, pretože ich nie je potreba uvažovať. Medzi objektovo orientované techniky spadajú aj algoritmy popísané nižšie, ktoré využívajú na zobrazovanie objemových dát textúry.

3.2.2 Obrazovo orientované techniky

Tieto techniky posudzujú každý pixel vo výslednom obraze samostatne. Pre každý pixel je rátaná výsledná hodnota z jednotlivých príspevkov celého objemu, ktoré zodpovedajú prechodu pohľadového lúča cez dáta. Typickým predstaviteľom tejto metódy ja algoritmus sledovania lúča (*ray casting*) [7,10].

3.3 Interakcia svetla

Pri objemovom zobrazovaní potrebujeme definovať model, ktorý bude simulovať interakciu svetla s objemom. Od požadovaných výsledkov závisí aj voľba modelu a to pohlcovanie, vyžarovanie a rozptyl svetla [14,18,20].

Interakciu svetla s prostredím popisuje *Beer-ov zákon* daný predpisom:

$$dI(t)/(dt) = \rho(t)I(t) - k(t)\rho(t), \qquad (3.1)$$

kde $I(t)$ je naakumulovaná svetelná intenzita, $\rho(t)I(t)$ koeficient popisujúci útlm a $k(t)\rho(t)$ vyžarovanie svetla. V tomto predpise sa t zväčšuje so zväčšujúcou vzdialenosťou od pozorovateľa. Tento predpis popisuje metódu prechodu lúčom spredu dozadu (ray tracing). Obrátenie smeru t vedie k zmene znamienok na pravej strane rovnice (3.1). Po úpravách môžeme dostať rovnicu pre diskrétny priestor [20], ktorá popisuje interakciu svetla s prostredím:

$$I(s_k) = I(s_{k-1})v_k + b_k \qquad (3.2)$$

Táto rovnica je už prevedená na popisovanie metód pracujúcich zozadu dopredu, kde v_k je priehľadnosť a b_k je vyžarovanie.

4 Grafické akcelerátory

V súčasnosti sú grafické akcelerátory podporujúce efektívne 2D a 3D operácie dostupné na takmer každom bežnom počítači. Ich výkon enormne vzrástol a cena v pomere k výkonu výrazne poklesla. Tento hardvér je vo väčšine prípadov využívaný na multimediálne aplikácie, počítačové hry a zábavné programy.

4.1 Minulosť

Pri dnešnom pohľade na údaje starých grafických kariet, sa len pousmejeme. Postupne výrobcovia vyrábali novšie a lepšie karty. Na začiatku gradoval počet farieb, ktorý sa postupne prepracoval z 2, 4, 16 farebnej škály až k 256 farbám. Aj grafické rozlíšenia, ktoré boli karty schopné zobraziť postupne rástli. Tieto karty nedisponovali buď žiadnym 3D výkonom a ak áno, tak veľmi slabým, ale aj to sa postupne menilo. Rapídny rozvoj 3D kariet prišiel s príchodom čipu Voodoo1. Postupne sa pridávali aj iní výrobcovia až sa postupne môžeme dostať po súčasnosť.

4.2 Súčasnosť

V súčasnej dobe obsahujú grafické karty vysoko výkonné grafické procesory (*GPU – graphics processor unit*), ktoré v sebe integrujú všetky potrebné grafické výpočty. Grafický procesor preberá výpočtovo náročnú časť zobrazovania (rasterizácia, osvetlenie, tieňovanie) z hlavného procesora, čím mu umožňuje vykonávať iné potrebné výpočty. Grafický procesor je veľmi vhodne navrhnutý pre prácu s vektormi, maticami a geometrickými primitívami hlavne trojuholníkmi, preto dosahujú vysoký výkon. S toho dôvodu sa začal využívať pojem grafický akcelerátor, ktorý výrazným podielom urýchľuje grafické aplikácie.

4.3 Grafické zobrazovanie – graphics pipeline

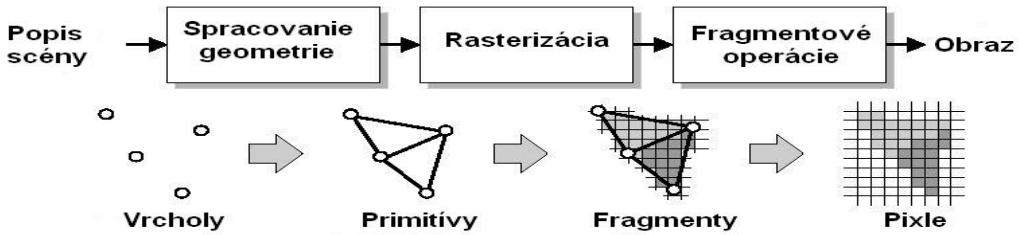
Pre grafické zariadenia musí zobrazovaná scéna pozostávať z rovinných útvarov. Proces, ktorý prevedie množinu mnohouholníkov reprezentujúcich scénu do rastrového obrazu, sa nazýva *display traversal*. Tento proces sa skladá z určitej postupnosti krokov, ktoré sú pre väčšinu grafických kariet obdobné. Poradie týchto krokov sa opisuje ako *graphics pipeline* [5] a je zobrazené na obrázku 4.1. Vstupom tohoto procesu je zoznam vrcholov primitív (trojuholníkov) s dodatočnými údajmi o farbe, normále atď.. Proces dekompozície trojrozmernej scény na rovinné útvary sa nazýva *teselácia (tesellation)*. Výstupom display traversal procesu je rastrový obraz s farebnými hodnotami, ktoré sú zobrazené na obrazovke. Graphics pipeline sa delí na tri základné vrstvy.

Spracovanie geometrie transformuje (rotuje, posúva, škáluje atď.) vstupné údaje do dvojrozmernej roviny, kde sú spoločné vrcholy pospájané do geometrických primitív, útvarov (body, čiary, trojuholníky ...)

Rasterizácia rozkladá dané primitívy na fragmenty, ktorým priraďuje textúrové hodnoty – mapovanie textúry. Každý fragment zodpovedá jednému pixelu na monitore.

Fragmentové (*Per-fragment*) operácie sú aplikované potom čo boli fragmentom priradené hodnoty ako farba a priehľadnosť v rasterizačnej jednotke. Posledným krokom pred vykreslením daného fragmentu na monitor sa uskutočňujú fragmentové testy, ktoré rozhodujú či bude daný fragment vykreslený alebo nie.

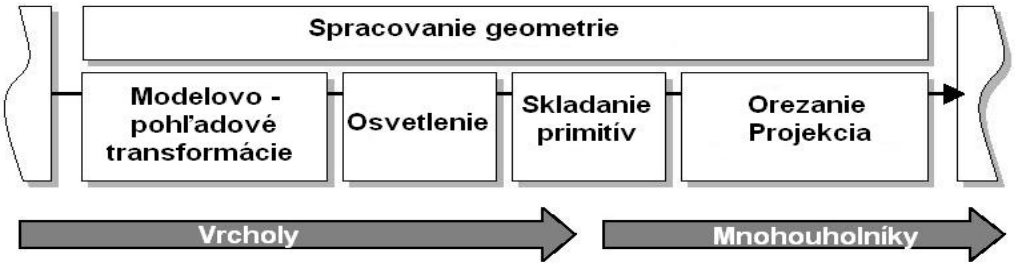
Pre dobré porozumenie niektorých algoritmov je dobré poznať proces grafického prechodu z trojrozmernej scény na rastrový obraz. Preto si ešte podrobnejšie vysvetlíme jednotlivé kroky.



obrázok 4.1: grafické zobrazovanie – graphics pipeline

4.3.1 Spracovanie geometrie

Toto predspracovanie prichádzajúcich vrcholov sa aj nazýva per-vertex operations. Táto jednotka počíta lineárne transformácie ako posunutie, otočenie, škálovanie a projekcie do premietacej roviny na vstupných vrcholoch. Ďalej priraďuje jednotlivým vrcholom farebné hodnoty a zlučuje ich s farebnou hodnotou vypočítanou z daného svetelného modelu a z jeho vlastností. Názornejší pohľad je na obrázku 4.2.



obrázok 4.2: spracovanie geometrie

Modelové transformácie: Transformácie, ktoré majú usporiadať a umiestniť vrcholy v scéne. Tieto transformácie sú reprezentované ako matice 4x4 s použitím homogénnych súradníc.

Pohľadové transformácie: Transformácie, ktoré majú na starosti nastavenie pozície kamery a jej smer. Táto transformácia je reprezentovaná maticou 4x4. Modelovacia a pohľadová matica je predspracovaná a udržiavaná ako modelovo - pohľadová matica.

Osvetlenie: Po tom čo sú vrcholy správne umiestnené v scéne sa im priradí farba, alebo ak je povolené osvetlenie a sú k dispozíci informácie o normálach a svetle(ách), sa vypočíta farba pomocou Phongovho svetelného modelu.

Skladanie primitív: Výsledné vrcholy sú pospájané do čiar a tie do mnohouholníkov. Mnohouholníky sú zvyčajne rozložené na trojuholníky kvôli zabezpečeniu planárnosti.

Orezávanie: Mnohouholníky a čiary sú orezané a časti mimo zobrazovaného priestoru sú vylúčené z ďalších výpočtov.

Projekčné transformácie: Dané vrcholy, mnohouholníky sú stredovým alebo rovnobežným premietaním premietnuté do premietacej roviny.

Výsledkom týchto operácií je premietnutá scéna do premietacej roviny. Ďalšie úpravy sa dejú už len v tejto rovine. Nové grafické karty umožňujú upravenie týchto krokov podľa požiadaviek aplikácie. Tento spôsob bude vysvetlený nižšie.

4.3.2 Rasterizácia

Rasterizácia je proces konverzie geometrických dát v obrazovej rovine na fragmenty. Každý fragment zodpovedá jednému pixelu vo výslednom obraze. Proces rasterizácie sa môže ďalej deliť na tri rozdielne časti, tak ako znázorňuje obrázok 4.3.

obrázok 4.3: rasterizácia

Rasterizácia mnohouholníka: Rasterizáciou sa určia vnútorné oblasti, fragmenty mnohouholníka. Týmto fragmentom sa priradí interpolovaná farba, osvetlenie, priehľadnosť, prípadne jeho vrcholom sa priradia textúrové súradnice.

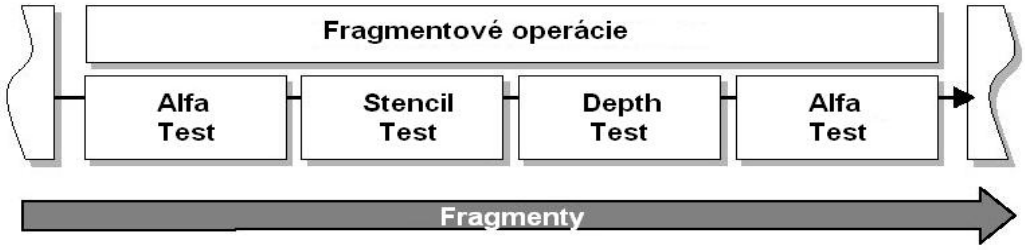
Generovanie textúr: Textúry sú 1, 2, 3 - rozmerné rastrové obrázky, ktoré sú nanesené na mnohouholník podľa textúrových súradníc určených pre vrcholy. Pre každý fragment sa interpoláciou textúrových súradníc vrcholov získajú hodnoty, ktoré odkazujú na príslušné miesto v danej textúre. Táto hodnota sa nazýva texel (texture element).

Aplikovanie textúr: Ak je povolené mapovanie textúr, farba z textúry prislúchajúca danému texelu sa kombinuje z vypočítanej farby používateľom definovanou cestou. Týmto krokom má príslušný fragment priradenú výslednú farebnú hodnotu a priehľadnosť.

Aj v tejto časti grafického zobrazovania umožňujú nové grafické karty aplikovať svoje vlastné algoritmy na dosiahnutie žiadaných efektov.

4.3.3 Fragmentové *(per-fragment)* operácie

Fragmenty sú zapísané do *frame buffer-a*, čo je dvojrozmerné pole, ktoré obsahuje aj časť, ktorá sa zobrazuje na monitore. Keď sa fragment zapisuje, zapisujú sa s ním aj iné hodnoty do iných častí. Pred vykreslením sa prevádzajú operácie, podľa ktorých môže byť vykreslenie fragmentu zakázané. Tieto operácie sa nazývajú fragmentové *(per-fragment)* operácie. Ukážka postupnosti operácií je načrtnutá na obrázku 4.4.



obrázok 4.4: fragmentové operácie

Alfa test: Alfa test dovoľuje zakázať vykreslenie fragmentu podľa jeho priehľadnosti a nastavenej hodnoty.

Stencil test: Test, ktorý porovnáva príslušnú hodnotu fragmentu s hodnotou prislúchajúcou v *stencil buffer-i* (maskou). Jedným z využití stencil buffer-u je generovanie tieňov.

Depth test: Využíva sa na riešenie viditeľnosti. V *depth buffer-i* sú uchovávané vzdialenosti daného fragmentu od pozorovateľa.

***Alfa blending*:** Technika, ktorá umožňuje simulovať priehľadnosť alebo priesvitnosť v scéne. *Alfa blending* kombinuje farbu prichádzajúceho fragmentu s farbou príslušného fragmentu už uloženého v frame buffer-i.

Po tom, čo je scéna kompletne vykreslená vo frame buffer-i, môže byť vykreslená na obrazovku. Ďalšie detaily ohľadom grafického procesu môžete nájsť v [5]. Tento proces sa môže čiastočne meniť s vlastnosťami grafických kariet.

5 API (application programming interface)

Programovacie jazyky (*C, C++, Java* atd.), ktoré chcú využívať vymoženosti grafických kariet, musia vedieť komunikovať s ovládačom karty. Toto umožňuje aplikačné programové rozhranie (*application programming interface - API*), ktoré efektívne využíva grafické karty od rôznych výrobcov. API odstraňuje nízkoúrovňové programovanie zobrazovacieho zariadenia a robí programovanie omnoho jednoduchším a dostupnejším a umožňuje využívanie všetkých možností tohoto zariadenia. Samozrejme dobré API uľahčuje prácu vývojárom pri vyvíjaní nového hardvéru, pretože dobre pracuje v rôznorodom prostredí operačných systémov. Najrozšírenejšie API v súčasnosti sú *Direct3D* a *OpenGL*. Obe rozhrania (štandardy) majú rovnaký koncept grafického procesu ako je znázornené na obrázku 4.1. V skratke budú popísané nižšie.

Každé rozhranie by malo spĺňať určité kritéria, ktoré môžu byť rôzne pre programátora zábavného softvéru a vedeckého pracovníka.

Efektívnosť: API by malo umožniť programovanie vysoko výkonných aplikácií. Malo by byť čo najjednoduchšie a zároveň využívať všetky možnosti zariadenia.

Rozšíritelnosť: Dobré API by malo byť schopné rýchlej adaptácie nových vlastností zariadení, ktoré sa ešte nestali štandardom.

Kompatibilita: Aplikácie, ktoré sú založené na danom API by mali byť kompatibilné nezávisle na jeho verzii. Program napísaný na nižšej verzii by mal fungovať na novších verziách a naopak.

Platformová nezávislosť: Pre vedecký výskum je nežiadúce obmedziť aplikáciu na jeden typ zariadenia alebo operačný systém. Toto je obzvlášť pravdivé pre prototypové aplikácie, ktoré sú použité na vyhodnotenie výkonnosti v rôznych prostrediach.

5.1 Direct3D

Direct3D [8] je grafické API, ktoré je súčasťou DirectX [1], čo je technológia firmy Microsoft pre programovanie počítačových hier a zábavných aplikácií. DirectX obsahuje niekoľko súčastí, ktoré formujú dve softvérové vrstvy. *Foundation layer* zabezpečuje hlavnú funkcionalitu, ktorá riadi podporu hardverových zariadení. *Media layer* sa nachádza na vrchu foundation layer a zabezpečuje obsluhu pre animáciu a multimédia.

Direct3D pôvodne pozostáva z dvoch oddelených foriem a to *retained mode*, ktorá bola súčasťou media leyer a *immediate mode*, ktorá patrí do foundation layer. Retained forma poskytuje knižnicu vysokej úrovne a príkazy pre multimédia a virtuálnu realitu. Vývoj retained formy bol zastavený firmou Microsoft s verziou DirectX 6.0. Immediate forma predstavuje výkonnú nižšiu úroveň API, ktorú si prisvojilo veľa programátorov pre programovanie počítačových hier.

5.2 OpenGL

OpenGL [25] je softvérové prepojenie na grafické zariadenie, ktoré obsahuje viac ako 200 odlišných príkazov (funkcií). Toto predstavuje jadro, ktoré je doplňované mnohými voliteľnými rozšíreniami. Od uvedenia OpenGL v roku 1992 sa stalo široko používaným a podporovaným API pre 2D a 3D aplikácie.

OpenGL je voľný, platformovo nezávislý grafický štandard. Špecifikáciu OpenGL má na starosti nezávislé konzorcium, OpenGL *Architecture Review Board (ARB),* ktoré tiež zabezpečuje spätnú kompatibilitu. Programovacie jazyky C, C++, Fortran, Ada, Phyton, Perl a Java môžu využívať OpenGL. Všetky komerčné aplikácie využívajúce OpenGL musia spĺňať jednoduchú špecifikáciu a musia prejsť sadou testov. OpenGL funguje na rôznych operačných systémoch vrátane Mac OS, OS/2, UNIX, Windows 95/98, Windows NT/2000,XP, Linux a BeOS.

Aj keď sú základy OpenGL založené na grafickom procese popísanom v kapitole 4, nové technické zmeny môžu vytvoriť nový prístup pomocou OpenGL rozšíreniam (*OpenGL extensions*). Vývojári tak majú voľnú ruku pri tvorení OpenGL implementácií na špecifickom grafickom akcelerátore. OpenGL rozšírenia dovoľujú vývojárom prispôsobiť program individuálne podľa daného hardvéru. OpenGL rozšírenia budú vysvetlené v ďalšej kapitole.

OpenGL štandard je široko používaný v priemysle a vedeckej komunite. Direct3D je API, ktoré je hlavne používané na programovanie hier a multimediálnych aplikácií. OpenGL je voľne šíriteľný štandard, pričom Direct3D je vlastníctvom firmy Microsoft. DirectX API je založené na *Microsoft's component object* model (COM) a je tak dostupný všetkým programovacím jazykom, ktoré podporujú COM. Toto však obmedzuje DirectX iba na platformy s operačným systémom Windows. Napokon je nižšie priložená tabuľka 5.1 na porovnanie medzi Direct3D a OpenGL. Rozdiel v základnej funkčnosti oboch API je okrajový. Hlavný rozdiel je v prístupe akým jednotlivé štandardy pristupujú k ich rozšíreniu a novým inováciám hardvéru. OpenGL definuje štandartnú sadu vlastností a špeciálnu sadu pomocou rozšírení. Po čase, keď sa ukáže, že sú dané rozšírenia užitočné a sú osvojené viacerými implementáciami, sú zahrnuté v novej špecifikácii OpenGL. Naopak, DirectX nepodporuje hardvérové rozšírenia. Množina funkcií definovaná štandardom DirectX je väčšia než funkčnosť navrhnutá súčasným hardvérom. Vlastnosti, ktoré nie sú podporované hardvérom sú nahradené softvérovou emuláciou. Toto dovoľuje zahrnúť do špecifikácie DirectX aj budúce funkcie (predpokladané). Avšak neexistuje záruka, že dané nové funkcie bude podporovať hardvér. V súčasnosti je DirectX sila, ktorá zabezpečuje rýchly vývoj nových hardvérových vlastností aj vďaka jeho častému aktualizovaniu.

	OpenGL	Direct3D
Operačné systémy	Unix, Linux, MacOS, OS/2, Windows 95/98/ME, Windows NT/2000/XP	Windows 95/98/ME, Windows 2000/XP
Programovacie jazyky	C, C++, Fortran, Perl, Python, Java, Ada	Všetky, ktoré podporujú COM
Voľne šíritelný štandard	Áno	Nie (vlastníctvo Microsoft-u)
Rozšíritelnosť	Áno (OpenGL rozšírenia)	Nie
Plusy a mínusy		

tabuľka 5.1: Porovnanie medzi OpenGL a Direct3D

6 Objemové zobrazovanie pomocou textúr

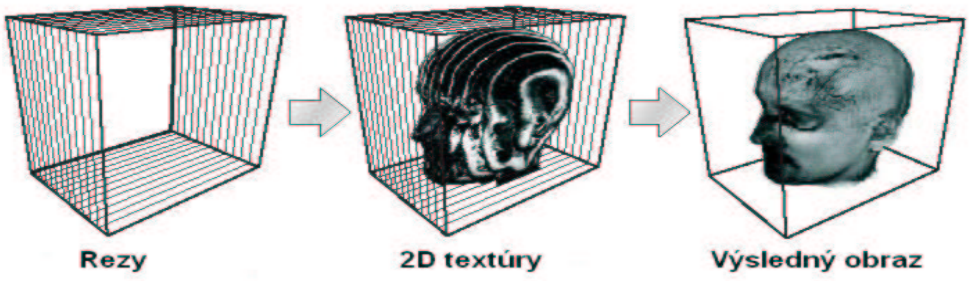
Pre priame objemové zobrazovanie bol navrhnutý špeciálny hardvér, ktorý je žiaľ použiteľný len pre túto oblasť (*VolumePro*,[15]). Je využívaný najmä univerzitami a laboratóriami, pričom jeho cena rádovo prevyšuje cenu súčasných najvýkonnejších grafických kariet pre osobné počítače. V tejto kapitole si ukážeme metódy ako využiť grafické akcelerátory pre osobné počítače pre interaktívne objemové zobrazovanie. Na interaktivitu objemového zobrazovania majú vplyv rôzne faktory. Jedným z významných prispievateľov k výpočtovej zložitosti je veľký počet interpolácií na vykreslenie daného obrazu, ktorý je požiadavkou na prevzorkovanie dát pozdĺž pohľadových lúčov. Ďalším prispievateľom je veľkosť dát. Jednotlivé skalárne voxely sú väčšinou reprezentované ako 8 alebo 16 bitové čísla. Obvyklé rozmery reálnych dát sú 256^3 a viac, menšie dáta sú väčšinou využívané len ako testovacie vzorky. Dáta z počítačovej tomografie mávajú rozlíšenie až 512^3, ale často sa môžeme stretnúť aj s dátami oveľa väčšími. Teda veľkosť dát má tiež na svedomí výpočtovú náročnosť a presuny medzi hlavnou pamäťou a video pamäťou sú tiež obmedzujúcim faktorom.

V algoritmoch popísaných nižšie sa obmedzíme na pravouhlú karteziánsku mriežku, s ktorou pracuje aj OpenGL. Ďalším obmedzením je reprezentácia dát, ktorá reprezentuje skalárne dáta.

6.1 Objemové zobrazovanie pomocou 2D textúr

V súčasnosti každé grafické zariadenie podporuje mapovanie 2D textúr. Časť grafických kariet, ktorá je zodpovedná za mapovanie textúr, povoľuje bilineárnu interpoláciu, ktorá je vo veľkej miere zodpovedná za výpočtovú zložitosť objemového zobrazovania. Preto je dobré využiť túto možnosť grafických kariet na jeho urýchlenie.

Objem je rozložený do sady *objektovo orientovaných rezov*, štvoruholníkov, v závislosti na aktuálnom smere pohľadu. Grafické karty dovoľujú priamo zobrazovať jednotlivé rezy, tak ako je načrtnuté na obrázku 6.1. Orientácia rezov je závislá na minimálnom uhle medzi vektorom pohľadu a normálou rezu, čo vedie k nutnosti mať tri sady dát (textúr), ktoré sú kolmé na súradnicové osi. Vybraná sada rezov je zobrazená transformáciou každého mnohouholníka a aktuálnou transformačnou maticou zobrazovaných v poradí od zadných rezov ku predným. Počas rasterizácie je na každý rez mapovaná prislúchajúca textúra. Bilineárna interpolácia počas mapovania textúry je urýchľovaná grafickým zariadením.



obrázok 6.1: objekt rozložený na sadu objektovo orientovaných rezov

6.2 Skladanie

Jednotlivé zobrazené rezy musia byť nejakou vhodnou formou pospájané do výsledného obrazu tak, aby vyhovovali teoretickým predpokladom uvedením v časti 3.3. Ďalšou úpravou tejto rovnice získame rovnicu [14,18,20]:

$$I(s_k) = I(s_{k-1})v_k + I_{emit}(s_k)(1-v_k) \qquad (6.1)$$

Použité 2D textúry môžeme reprezentovať na grafickej karte ako štvoricu pevne daných zložiek, R - červená, G – zelená, B – modrá a hodnotu pre priehľadnosť A – alfa. Pre každý voxel je koeficient vyžarovania I_{emit} uložený ako farebná hodnota (RGB) v prislúchajúcej textúre. Pomocou hodnoty alfa, ktorá reprezentuje priehľadnosť, môžeme ukladať inverzný koeficient pohlcovania $(1-v_k)$. Použitím metódy alpha blending-u spomenutej v kapitole 4.3.3 sa dá aproximovať skladanie svetla pozdĺž pohľadového lúča. Miešanie (blending) nám umožňuje kombinovať RGBA zložku vstupového (source) fragmentu s cieľovou (destination) hodnotou zapísanou v frame buffer-i. Pokiaľ je blending zakázaný, tak sa cieľová hodnota nahradí vstupnou hodnotou. Použitím OpenGL funkcie, `glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA)`, ktorá nastavuje zmiešavaciu rovnicu

$$C'_{dest} = C_{src}A_{src} + C_{dest}(1-A_{src}) \qquad (6.2)$$

a použitím substitúcie nahrádzajúce koeficienty pohlcovania a vyžarovania,

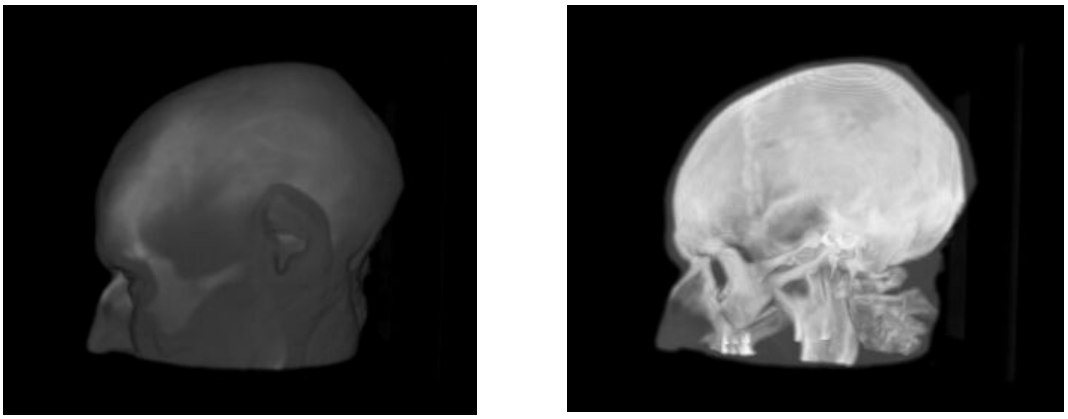
$$C_{src} = I_{emit} \quad \text{a} \quad A_{src} = (1-v_k) \qquad (6.3)$$

dosiahneme rovnicu (6.1).

Pomocou alpha blendingu môžeme nastavovať rôzne zmiešavacie rovnice, ktoré vedú k rôznym výsledkom.

6.2.1 Maximálna intenzitová projekcia

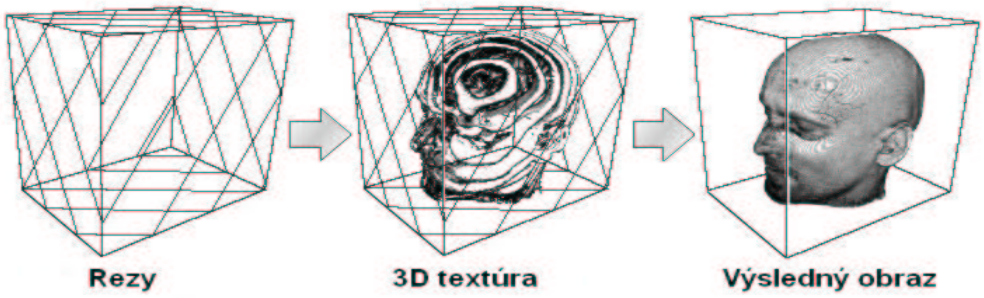
Maximálna intenzitová projekcia (MIP) je hlavne využívaná pre medicínske aplikácie, kde sa zobrazujú tomografické dáta s použitím kontrastnej látky na zvýraznenie (cievy). Pre použitie MIP potrebujeme nastaviť dodatočnú zmiešavaciu rovnicu príkazom `glBlendEquationEXT(GL_MAX_EXT)`. Táto funkcia je zahrnutá v OpenGL rozšíreniach, s názvom `GL_EXT_blend_minmax` , ktoré sú popísané nižšie. Na obrázku 6.2 je vidieť rôzne výsledky pre tie isté dáta.



Obrázok 6.2: použitie rôznych zmiešavacích rovníc pre skladanie rezov

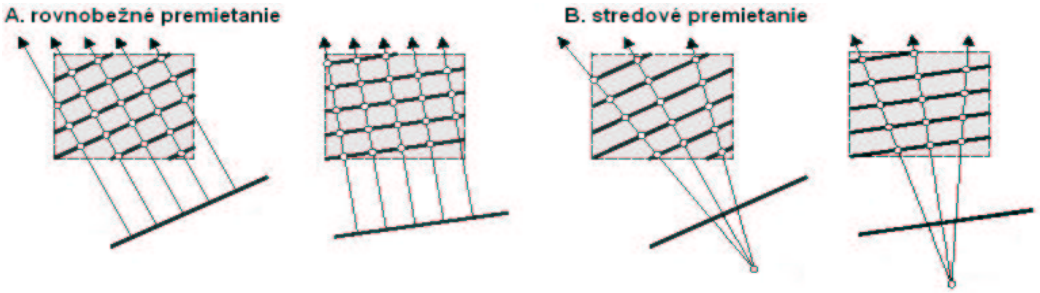
6.3 Objemové zobrazovanie pomocou 3D textúr

Metóda využívajúca 3D textúru [2,21] odstraňuje niektoré nedostatky predchádzajúcej metódy, ktoré boli zapríčinené využívaním len bilineárnej interpolácie. Tieto nedostatky pre obe metódy sú popísané nižšie. Pri 3D textúrach sa využíva trilineárna interpolácia, čo vedie k novému prístupu objemového zobrazovania. Využívanie 3D textúr nevedie k odstráneniu potreby rozložiť objekt na rovinné mnohouholníky. Aj keď je celý objem dostupný na grafickej karte ako 3D textúra, grafické zobrazovanie (graphics pipeline) nepodporuje priame zobrazovanie objemových primitív (kocka a iné), ale len rovinných. Použitie 3D textúry dovoľuje umiestniť rezové mnohouholníky ľubovolne, tak aby bola zachovaná konštantná vzdialenosť rezov pre rôzne uhly pohľadu. Môžeme využiť rezy rovnobežné s premietacou rovinou, *pohľadovo orientované rezy* (obrázku 6.3). Avšak súradnice mnohouholníkov musia byť prepočítavané vždy keď dôjde k zmene smeru pohľadu, pričom tento výpočet je zložitejší ako pri objektovo orientovaných rezoch. V prípade rovnobežného premietania vedie tento spôsob k rovnakému vzorkovaniu pre všetky lúče (obrázok 6.4A). Pre stredové premietanie nie je vzorkovanie rovnaké pre všetky lúče (obrázok 6.4B). Vzdialenosť medzi vzorovými bodmi stúpa s veľkosťou uhla medzi vektorom pohľadu a normálou rezu.



obrázok 6.3: rozloženie objemu na pohľadovo orientované rezy

Pre skladanie jednotlivých rezov v tomto prípade platia rovnaké podmienky ako pre zobrazovanie pomocou 2D textúr popísané vyššie, rovnako aj pre maximálnu intenzitovú projekciu.



obrázok 6.4: vzorokovanie pozdĺž pohľadových ľúčov pre rovnobežné (A) a stredové (B) premietanie

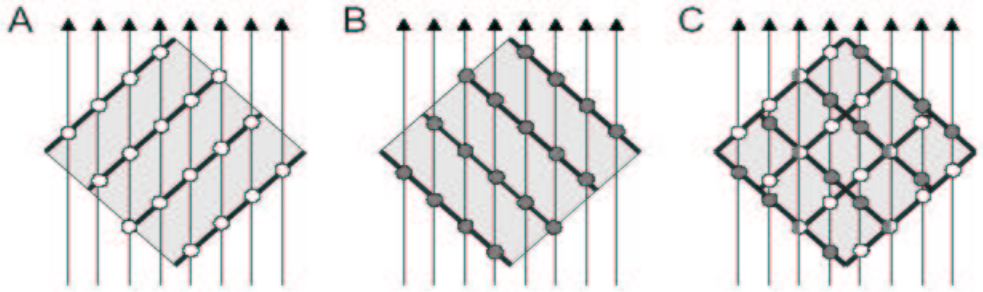
6.4 Zhrnutie

V tejto časti zhrnieme hlavné výhody a nevýhody každého prístupu pomocou textúr.

Tieto algoritmy sú založené na podpore grafických akcelerátorov a preto pre dáta vyplývajú obmedzenia, ktoré požadujú jednotlivé zariadenia. Jedným z obmedzení je veľkosť textúry. V súčasnosti podporujú grafické karty so 128MB pamäťou rozmer 3D textúry 512^3, pričom táto textúra musí byť celá v pamäti, z čoho vyplýva 8bitová presnosť pre jednotlivé voxely. Pri väčších rozmeroch sa dá využiť rozdelenie celého objemu na menšie časti a zobrazovať ich postupne v rámci jedného zobrazovacieho cyklu [16]. Pri 2D textúrach sú dnes maximálne povolené rozmery 4096x4096, ktoré sú zatiaľ dostačujúce, ale nastáva problém s priepustnosťou zbernice medzi hlavnou pamäťou a video pamäťou pri prenose dát. Dnešné AGP8x porty umožňujú prenos rýchlosťou 2.1GB/s. Týmito problémami sa ale v tejto práci nezaoberáme.

6.4.1 2D textúry

Hlavná výhoda tohoto prístupu je presunutie potrebného výpočtu interpolácie na grafický akcelerátor, ktorý je na to vhodne navrhnutý. Prácu s 2D textúrami umožňuje takmer každá grafická karta, preto možno aplikovať tento algoritmus na takmer každom počítači. Zobrazovací výkon je vysoký za predpokladu, že je k dispozícií dostatok video pamäti, od čoho závisí aj rozmer dát. Rozmer dát je obmedzený aj veľkosťou hlavnej pamäti na udržiavanie troch sád potrebných rezov. Pri zväčšovaný obrazu dochádza k typickým aliasingovým artefaktom, ktoré sú hlavne viditeľné na hranách rezov a sú spôsobené nedostatočným vzorkovaním. Počet rezov je stanovený rozmerom dát v danom smere. Pri väčšom počte rezov sa na daný rez mapuje najbližšia textúra z danej sady. Existujú metódy, ktoré využívajú grafické karty na získanie potrebnej textúry zo susedných textúr a potom ju mapujú na daný rez [16]. Ďalší rušivý efekt je možné pozorovať pri zmene medzi rôznymi sadami rezov (obrázok 6.5). Tento problém je odstrániteľný pri použití 3D textúry. Celkovo sú v tabuľke 6.1 zhrnuté výhody a nevýhody tohoto algoritmu.



obrázok 6.5: vzorkovacie artefakty zapríčinené zmenou sady rezov

6.4.2 3D textúry

Prístup využívajúci 3D textúry zvyšuje kvalitu obrazu. Hardvérová podpora pre trilineárnu interpoláciu nám umožňuje zvýšenie vzorkovania na získanie kvalitnejších výsledkov a odpadá nutnosť mať v pamäti tri sady dát. Obrazovo orientované rezy zaručujú rovnaké vzdialenosti medzi susednými vzorkami pre rovnobežné premietanie. Problém meniaceho sa vzorkovania stále nie je odstránený pre stredové premietanie. Táto nekorektnosť je však z vonkajších pohľadov málo viditeľná, pri pohľadoch z vnútra objemu sú viditeľné rušivé artefakty. Tento problém môže byť odstrániteľný použitím guľových zobrazovacích primitív namiesto rovinných [9]. Výhody a nevýhody tohoto prístupu sú zhrnuté v tabuľke 6.2. Aplikovatelnosť tohoto prístupu závisí od hardvérovej podpory 3D textúry. V špecifikácii OpenGL verzie 1.2 sú zahrnuté aj 3D textúry, ale pokiaľ nie sú podporované kartou sú vykonávané softvérovou emuláciou na hlavnom procesore, čo vedie k slabému výkonu.

2D textúry	
Plusy	mínusy
Vysoký výkon veľká dostupnosť	vysoké pamäťové požiadavky iba bilineárna interpolácia vzorkovacie artefakty prepínací efekt medzi sadami textúr nekonzistentné vzorkovanie

tabuľka 6.1: výhody a nevýhody zobrazovania pomocou 2D textúr

3D textúry	
Plusy	mínusy
Vysoký výkon trilineárna interpolácia	rozmerové obmedzenie nekonzistentné vzorkovanie pre stredové premietanie

tabuľka 6.2: výhody a nevýhody zobrazovania pomocou 3D textúr

7 OpenGL rozšírenia

Aj keď základná knižnica OpenGL poskytuje dôležitú časť funkčnosti, pokrok grafického hardvéru rapídne rastie a nové zobrazovacie techniky nie sú pokryté v základnej knižnici. Našťastie rozširovanie OpenGL bolo navrhnuté vhodným spôsobom, a to pomocou OpenGL rozšírení (*OpenGL extensions*) [11,26]. Výrobcovia grafických kariet môžu definovať nové OpenGL rozšírenia, ktoré zavádzajú nové možnosti grafického zobrazovania podporované ich hardvérom a nie sú zahrnuté v základnej knižnici OpenGL. Toto je silná zbraň, ktorá sa vo veľkej miere používa na rozvoj OpenGL. V súčasnosti je definovaných viac ako 300 rozšírení, z ktorých je podstatná časť zahrnutá aj v nových verziách OpenGL.

OpenGL rozšírenia zvyčajne začínajú ako mechanizmus pre prístup k novým vymoženostiam hardvéru ponúknutým výrobcom. Výrobca definuje nové symboly a funkcie, ktoré obsluhujú nové hardvérové možnosti. Rozšírenia ponúkané jedným výrobcom sa prezývajú *vendor-specific extensions* a väčšinou sú dostupné len na hardvéry od daného výrobcu. Ak viacero výrobcov súhlasí zo zavedením novej možnosti hardvéru, tak sa použije jedno rozšírenie pre viacero výrobcov nazývané *multivendor extension*. OpenGL rozšírenia môžu definovať rovnaké využitie grafického hardvéru, ale pod iným názvom, čo vedie pri programovaní k rôznym zdrojovým kódom. Pri využívaní multivendor rozšírení je zdrojový kód rovnaký pre grafické karty od rôznych výrobcov, ktoré podporujú dané rozšírenia. Preto je pre dobrú prenositeľnosť danej aplikácie vhodné využívať hlavne multivendor rozšírenia.

OpenGL rozšírenia sú definované svojim názvom, ktorý sa skladá s predpony, vo väčšine prípadov trojznakovej a názvu daného rozšírenia, ktoré vhodne popisuje jeho funkčnosť, kde sú jednotlivé výrazy oddelené podtržítkom. Multivendor rozšírenia využívajú predponu `ARB` a `EXT`. Výrobcom definované rozšírenia používajú predponu súvisiacu s názvom výrobcu (`ATI`, `NV`, `APPLE`, `SGI` ...).

Firma *Silicon Graphics* má na starosti správu OpenGL rozšírení na internetovej stránke [http://oss.sgi.com/projects/ogl-sample/registry/], ktorá je známa ako *OpenGL Extension Registry*, kde sú oficiálne špecifikácie ku všetkým OpenGL rozšíreniam.

7.1 Využitie OpenGL rozšírení pre objemové zobrazovanie

Počet rozšírení OpenGL rastie s vývojom nových grafických kariet a s potrebou vývojárov. V súčasnosti je ich viac ako 300 od verzie OpenGL 1.1. Tieto sú využívané v rôznych odvetviach počítačovej grafiky. Niektoré sú priamo navrhnuté pre špeciálne aplikácie, čo bol ich hlavný význam. Niektoré sa dajú použiť vo viacerých odvetviach. To už závisí od programátora ako ich dokáže využiť a implementovať v danej aplikácii, čo môže zreteľne urýchliť a skvalitniť proces zobrazovania. V tejto kapitole sú uvedené niektoré rozšírenia a ich využitie pre objemové zobrazovanie. Avšak nie sú tu uvedené presné riešenia problémov a ani špecifikácia daných funkcií, ktorá sa dá nájsť na [26].

7.2 Textúry

V tejto časti sa zaoberáme rozšíreniam, ktoré nejakým spôsobom zavádzajú, nastavujú textúry alebo definujú nové operácie, ktoré pracujú s textúrami.

7.2.1 3D textúry

Do tejto časti spadá rozšírenie **GL_EXT_texture3d**. Toto rozšírenie umožňuje prácu s 3D textúrami, ktoré sú využívané na reprezentáciu daného objemu. Ako je to aj u 2D textúr, tak aj u 3D textúr grafická karta obmedzuje jej veľkosť (rozmer). Pomocou 3D textúr nemusí byť definovaný len daný objem, ale aj iné vlastnosti (gradient a rôzne masky) potrebné na výpočty.

7.2.2 Mapovanie textúr

Tu zhrnieme pár rozšírení, ktoré podporujú nastavovanie rôznych veličín pre textúry ako je orezávanie, filtrovanie, zarovnávanie a ďalšie.
GL_EXT_texture_filter_anisotropic toto rozšírenie povoľuje používanie *anizotropného* filtrovania namiesto štandardne definovaného lineárneho alebo výberu najbližšieho suseda.
GL_ARB_texture_cube_map definuje použitie šiestich 2D textúr korešpondujúcich so šiestimi stranami kocky použitím 3D textúry. Definuje tiež postup generovania textúrových súradníc pre danú textúru. Táto textúra je využiteľná na mapovanie okolia na daný objekt, kde v textúre je zobrazené okolie (*enviroment mapping*) [3].
GL_ARB_texture_non_power_of_two odstraňuje obmedzenie kladené na textúry špecifikáciou OpenGL všetkých dimenzií (1, 2, 3D) a *cube map*, kde mohol byť rozmer tektúry iba mocnina dvoch. Toto rozšírenie by malo podporovať všetky doteraz definované nastavenia ohľadom textúr ako sú: hranice, naväzovanie, palety, *mipmaping*. Textúrové súradnice sú definované na interavle <0,1>. Rozšírenie `GL_EXT_texture_rectangle`, ktoré odstraňovalo obmedzenie ohľadom rozmerov textúr bolo prakticky nepoužiteľné, pretože nepodporovalo dané nastavenia.
GL_EXT_texture_edge_clamp, GL_ARB_texture_border_clamp definujú orezávanie a hranice textúry potrebné pri filtrácii. Pri štandardnom definovaní hraníc a orezávania mohli byť viditeľné artefakty na hraniciach.
GL_ARB_texture_mirrored_repeat umožňuje pri používaní symetrických textúr použiť len istú časť za pomoci zrkadlového napájania.

7.2.3 Kompresia textúr

Sem spadá viacero rozšírení, ktoré využívajú kompresiu na zníženie potrebnej pamäte a zrýchlenie zobrazovacieho procesu.
GL_ARB_texture_compression nedefinuje nové kompresné pomery a algoritmy. Jeho úlohou je hlavne definovanie interných textúrových formátov pri zavádzaní textúr do video pamäti. Je schopné prenášať komprimované textúry z video pamäti a naopak, môžu byť aj takto uchovávané. Poskytuje mechanizmus na obdržanie textúrnych kompresných formátov podporovaných aplikáciou.

GL_EXT_texture_compression_s3tc definuje tri kompresné pomery pre RGBA zložky a jeden pre RGB, ktoré pracujú na každom 4x4 bloku pixelov. Podporuje iba textúry definované bez okrajov a 2D textúry.
GL_NV_texture_compression_vtc rozširuje S3TC kompresiu pre 3D textúry.

Kompresný pomer týchto rozšírení nie je až taký veľký aby významne obmedzil veľkosť dát. Súčasne definované kompresné algoritmy vedú k viditeľným artefaktom.

7.2.4 Operácie na textúrach

Tieto rozšírenia povoľujú prácu s viacerými textúrami v jednom prechode zobrazovacím procesom a definujú operácie ktoré môžu byť aplikované na dané textúry.
GL_ARB_multitexture rozšírenie, ktoré rozširuje možnosť použitia viacerých textúr naraz v textúrovej jednotke. Maximálny počet textúr, ktoré môžu byť použité sú obmedzené grafickou kartou.
GL_ARB_texture_env_add, GL_ARB_texture_env_combine, GL_ARB_texture_env_crosbar, GL_ARB_texture_env_dot3 tieto rozšírenia definujú rôzne operácie pracujúce s viacerými textúrami, pomocou ktorých môžeme ovplyvňovať výslednú hodnotu fragmentu.

Týmito rozšíreniami sa dajú dosiahnuť rôzne efekty ako je napríklad tieňovanie, kde v jednej textúre sú objemové dáta a v druhej prislúchajúce gradienty.

7.2.5 Indexovacie textúrové palety

Sem spadajú rozšírenia, ktoré využívajú palety na priradenie výslednej farby daného fragmentu. Tieto rozšírenia budú vysvetlené v časti o prenosových funkciách.
GL_EXT_paletted_texture zabezpečuje priradenie pomocou textúrovej hodnoty, ako index do palety, výslednú hodnotu fragmentu. Táto paleta sa aplikuje pred interpoláciou textúry.
GL_EXT_shared_texture_palette zabezpečuje použitie jednej palety pre viaceré textúry.
GL_EXT_color_subtable umožňuje zmeniť len určitú časť palety.
GL_SGI_texture_color_table rovnaké priradenie ako vyššie spomenuté rozšírenie, ale aplikuje sa až po interpolovaní textúrnych hodnôt.

7.3 Miešanie - blending

Pod blendingom sa rozumie skladanie výsledných hodnôt fragmentu (pixela) z hodnoty vo frame buffer-i a aktuálne spracovávaného fragmentu. Rozšírenia **GL_EXT_blend_func_separate, GL_EXT_blend_subtract, GL_EXT_blend_color, GL_EXT_blend_minmax** definujú nové rovnice, ktoré môžu byť použité na výpočet výslednej hodnoty fragmentu. Pomocou týchto rovníc sa dajú dosiahnuť rôzne efekty a výsledky, ktoré požaduje aplikácia. Rozšírenie `GL_EXT_blend_minmax` je použité v kapitole 6 na dosiahnutie projekcie maximálnej intenzity.

7.4 Fórmát pixelov

Rozšírenia **GL_EXT_bgra, GL_EXT_abgr** definujú opačné poradie farebných zložiek. V OpenGL sa pracuje s farebnými zložkami v poradí RGBA, avšak niektoré oknové systémy pracujú v opačnom poradí (ABGR) s farebnými zložkami.

GL_EXT_cmyka zavádza metódy na čítanie a zapisovanie obrazov, ktorých farebný formát je typu CMYK(A).

GL_EXT_packed_pixels zabezpečuje nový typ pixelov, ktorý dovoľuje viacerým komponentom (zložkám) byť zbalený do jednotlivých dátových typov. Toto môže byť tiež využité na kompresiu, ale na úkor presnosti.

7.5 Vertex program

Nižšie uvedené rozšírenia poskytujú úplné programovanie na vrcholoch v jednotke, ktorá spracováva geometriu vrátane transformácií súradníc, rátania osvetľovacích modelov a generovania textúrových súradníc. Štandardné výpočty na vrcholoch sú nahradené asemblerovým kódom, ktorý je skompilovaný do *vertex programu*. Pokiaľ je vertex program povolený, programátor preberá takmer všetky operácie na vrcholoch, za ktoré automaticky zodpovedá OpenGL. Veľkosť programu a počet premenných je obmedzený grafickou kartou. Po povolení vykonávania daného programu sa bude aplikovať na každý vrchol. Rozšírenia definujú syntax príkazov, definovanie premenných a sadu inštrukcií, ktoré sú k dispozícií programátorovi. Istú časť operácii vykonáva stále OpenGL.

Časti, za ktoré je zodpovedný programátor sú:

1. transformácia súradníc vrcholov objektového priestoru do homogénneho priestoru,
2. výpočty primárnej a sekundárnej farby prislúchajúcej vrcholu,
3. škálovanie normálových vektorov a normalizácia,
4. normalizovanie ohodnotených normálových vektorov,
5. generovanie textúrových súradníc,
6. aplikácia textúrnych matíc,
7. výpočty potrebné pre nastavenie hmly,
8. útlm intenzity bodov so vzdialenosťou od kamery,
9. váhovanie vrcholov.

GL_ARB_vertex_program, GL_NV_vertex_program umožnujú využívanie a tvorbu vertex programov.

GL_NV_vertex_program1_1 definuje verziu 1.1 vertex programu a pridáva štyri nové inštrukcie.

GL_NV_vertex_program2 definuje verziu 2.0 vertex programu a nové aritmetické inštrukcie.

7.6 Fragment program

Tak ako rozšírenia pre vertex program existujú aj rozšírenia, ktoré nahrádzajú výpočty fragmentov a definujú *fragment program*. Tieto výpočty sú nahradené asemblerovým kódom, ktorý je preložený do fragment programu. Ak je povolené vykonávanie fragment programu, tak je aplikovaný na každý fragment. Fragment

program môže vykonávať komplexné výpočty, vrátane závislých textúr a ďalších. Rozšírenia definujú syntax a sadu inštrukcií, ktoré sú využiteľné na výpočty. Počet inštrukcií a potrebných premenných je obmedzený v závislosti od danej grafickej karty. Úlohou fragment programu je určiť farebnú hodnotu výsledného fragmentu. Fragment program neodstraňuje testy pred zobrazením fragmentov na monitor (alfa, scissor, depth, ...), je zavedený pred tento stupeň zobrazovacieho procesu a za rasterizáciu primitív. Je zodpovedný za mapovanie textúry, mapovanie interpolovanej farby z vrcholov, nastavovanie hmly. Pomocou fragment programu sa dajú použiť rôzne osvetľovacie modely priamo pre každý fragment.
GL_ARB_fragment_program, GL_NV_vertex_program rozšírenia povoľujú a definujú inštrukcie pre fragment program.

7.7 Pixel buffer (pbuffer)

Pixel buffer (pbuffer), definovaný rozšírením popísaným nižšie, je prídavný buffer pre nie viditeľné zobrazovanie (*off-screen rendering*). Využíva sa pre renderovanie potrebných dát (textúr), ktoré sa potom používajú klasickým spôsobom pre zobrazovanie. Pbuffer je možné vytvoriť väčší než je aktuálne rozlíšenie monitora, čo nie je dovolené pri definovaní veľkosti okna a teda aj zobrazovacieho OpenGL buffera. Jeho využitie je preto vhodné pre zobrazovanie veľkých dát v originálnom rozlíšení a požadovanej kvalite. Pbuffer je definovaný rozšírením, ktoré je závislé na oknovom systéme.
WGL_ARB_pbuffer definuje pbuffer a ponúka funkcie na jeho vytvorenie.
WGL_ARB_render_texture dovoľuje použitie color buffer-a na zobrazovanie (renderovanie) doňho a použitie aj ako textúry.
WGL_ARB_make_current_read umožňuje nastaviť rôzne kontexty zariadení (*device context DC*) na čítanie a na zapisovanie do pbuffer-a.
WGL_ARB_pixel_format definuje funkcie na zisťovanie podporovaného pixel formátu, ktorý sa využíva pri vytváraní pbuffer-a.
WGL_ARB_extensions_string dovoľuje zistiť, ktoré WGL rozšírenia sú podporované grafickým zariadením.
WGL_ARB_buffer_region dovoľuje uložiť časť OpenGL okna do nezobrazovateľnej pamäti buď vo video pamäti, alebo v hlavnej pamäti.

OpenGL rozšírenia s predponou `WGL` spadajú pod operačné systémy Windows. Obdobné rozšírenia existujú aj pre operačné systémy typu Unix, Linux, ktoré sú označené predponou `GLX` (`GLX_SGIX_pbuffer`).

7.8 Rôzne

GL_ARB_transpose_matrix toto rozšírenie povoľuje prácu s transponovanými maticami.
GL_EXT_histogram definuje operácie, ktoré rátajú výskyt špecifikovanej farebnej zložky a hľadajú jej maximum a minimum.

Sú tu uvedené rozšírenia OpenGL, ktoré sa môžu priamo použiť na zobrazovanie objemových dát, ale aj také, ktoré skvalitňujú vzhľad aplikácie a uľahčujú prácu programátora. Táto kapitola nie je vôbec uzavretá danými rozšíreniami, určite sú ďalšie, ktoré sa dajú využiť pre skvalitnenie aplikácií alebo návrh nových algoritmov. Výrobcovia budú zavádzať ďalšie, ktoré môžu značne ovplyvniť vývoj nových aplikácií.

8 Cg (C for graphics)

Cg je programovací jazyk vyvinutý firmou *NVidia Corporation* v spolupráci s firmou Microsoft [23,4]. Prekladač jazyka Cg prekladá program do asembleru grafických kariet. Tento prekladač využíva OpenGL rozšírenia (vertex, fragment program), ktoré podporujú programovanie grafických kariet v ich vlastnom asembleri, čím odpadá nutnosť ovládať programovanie v jazyku nízkej úrovne. Volania príkazov Cg nastavujú potrebné parametre pre funkcie OpenGL a využívajú aj vyššie uvedené rozšírenia. Cg využíva syntax z programovacieho jazyka C a je kompatibilné s OpenGL API a Microsoftovským *High-Level Shading Language (HLSL)* pre DirectX 9.0.

8.1 Programovanie v Cg

Cg program pracuje s vrcholmi a fragmentmi, ktoré vchádzajú do programu a vychádzajú po určitých transformáciach [4]. Cg dovoľuje nahradiť určité časti zobrazovacieho procesu daným programom. V závislosti od toho, kde sa aplikuje tento program hovoríme o *vertex programe*, ktorý nahrádza určité časti spracovania geometrie a pracuje na vrcholoch alebo *fragment programe*, ktorý pracuje v rasterizačnej jednotke na fragmentoch. Pre DirectX sú zaužívane názvy *vertex shader* pre vertex program a *pixel shader* pre fragment program. Cg prekladač preloží program do assembleru (sady inštrukcií) definovaného v OpenGL rozšíreniach uvedených vyššie. Pri prekladaní programu treba špecifikovať meno vstupnej funkcie a meno *profilu,* od ktorého závisí funkčnosť prekladača. Meno vstupnej funkcie je názov hlavnej funkcie programu. Profilom sa definuje, či sa jedná o vertex alebo fragment program a špecifikuje sa sada požadovaných funkcií podporovaných grafickou kartou. Profily sú rozdelené podľa určeného programu (vertex, fragment), API (OpenGL, DirectX) a podporovaných funkcií grafickou kartou. Pre príklad uvedieme názvy profilov pre OpenGL od základného po pokročilejšie. Vertex profily: *arbvp1* (`ARB_vertex_program`), *vp20* (`NV_vertex_program`), *vp30* (`NV_vertex_program2`). Fragment profily: *fp20* (`NV_texture_shader`, `NV_register_combiners`), *arbfp1* (`ARB_fragment_program`), *fp30* (`NV_fragment_program`). V zátvorkách sú uvedené OpenGL rozšírenia, ktoré sú potrebné pre jednotlivé profily, lebo prekladač využíva inštrukcie definované v týchto rozšíreniach. Profily arbvp1 a arbfp1 sú založené na multivendor rozšíreniach, čo značí že dané programy fungujú aj na grafických kartách iných výrobcov ako je NVidia. Od podpory profilu závisia aj funkcie, ktoré môžu byť použité v programe a ktoré podporuje grafická karta.

8.2 Premenné

V tomto jazyku sa využívajú dve hlavné skupiny premenných. Do jednej skupiny spadajú premenné, ktoré obdrží program zo zobrazovacieho procesu (graphics pipeline). Pre vertex program sú to hlavne vrcholy, ich normály, farba, prípadne ďalšie. Tieto premenné vstupujú do vertex programu pod svojim kľúčovým menom POSITION, NORMAL, COLOR. S týmito hodnotami vykonávame rôzne operácie a potrebné premenné sú posielané ďalej do fragment programu. Výstupnou premennou z vertex programu musí byť pozícia daného vrcholu v 2D pre ďalšie

spracovanie. Obdobe je to aj s fragment programom, kde obdržíme pre každý fragment jeho farbu, textúrovú súradnicu označené kľúčovými slovami COLOR,TEXCOORD0. Povinným a potrebným výstupným parametrom z fragment programu je jeho farba označená slovom COLOR. Do fragment programu z vertex programu sa môžu poslať aj iné potrebne premenné (normály) pod kľúčovými slovami TEXCOORDX, kde X označuje počet parametrov.

Ďalšou skupinou premenných sú premenné označované slovom uniform, kde spadajú matice, vektory (1,2,3,4 rozmerné) a textúry, prípadne ďalšie. Tieto premenné sú rovnaké pre všetky vrcholy a fragmenty pokiaľ nie sú zmenené programátorom priamo v OpenGL kóde, pred volaním funkcie `glBegin()`.

8.3 Využitie

Vertex a fragment program dáva programátorovi možnosť využiť nové algoritmy a aplikovať ich do zobrazovacieho procesu, ktorý bol doteraz nemenný, alebo len čiastočne pozmenitelný pomocou OpenGL rozšírení. Vo vertex porgrame sa pracuje s vrcholmi, čiže sa naskytuje možnosť meniť pozíciu vrcholu, čím sa môžu dosiahnuť animačné efekty a iné. Tieto možnosti sa dajú v istej miere dosiahnuť aj v rámci predspracovania vrcholov pred posielaním do OpenGL, ale vertex a fragment programy sú vykonávané na GPU, ktorá je špeciálne navrhnutá na prácu s vektormi a maticami, čím dosiahneme lepšie výsledky vzhľadom na rýchlosť vykresľovania. Vo fragment programe sa hlavne naskytujú možnosti využiť rôzne reálnejšie osvetľovacie modely aplikované na každý fragment nie ako to je v štandardnom zobrazovacom procese, kde sa iba interpoluje farba vrcholov. Pri fragment programe môžeme využívať aj rôzny počet textúr a získavať výslednú farbu fragmentu rôznymi operáciami na týchto textúrach.

V ďalších častiach načrtneme možnosti využitia Cg pre objemové zobrazovanie. Využívame hlavne fragment program. Vertex program má potenciálne využitie v povrchových algoritmoch, kde sa využíva dostatok geometrie.

8.3.1 Normály

Pre techniky tieňovania je potrebné poznať normálu povrchu v danom bode. Túto normálu môžeme vyrátať priamo z dát. Princíp získavania normály voxela je založený na rozdiele intenzít susedných voxelov daného voxelu [18]. Uvedieme si teraz kód v Cg, ktorý počíta potrebné normály.

```
#define K=0.01

t0 = IN.texcoord1; t1 = IN.texcoord1;
t0.x-=K;    t1.x+=K;
normal.x = tex3D(volume,t1).r-tex3D(volume,t0).r;

t0 = IN.texcoord1;      t1 = IN.texcoord1;
t0.y-=K;    t1.y+=K;
normal.y = tex3D(volume,t1).r-tex3D(volume,t0).r;

t0 = IN.texcoord1;      t1 = IN.texcoord1;
t0.z-=K;    t1.z+=K;
normal.z = tex3D(volume,t1).r-tex3D(volume,t0).r;
```

Funkcia tex3D() definuje hodnotu z textúry (`volume`), ktorá je určená textúrovými súradnicami (`t0,t1`).

Tento výpočet je ale aj dosť pomalý (30 inštrukcií), pretože uvedený výpočet sa vykonáva opakovane pre každý fragment. Pri použití ďalšej 3D textúry s uloženým gradientom získaným v predspracovaní sa dosahuje lepšia efektivita avšak na úkor potrebnej pamäte.

Tieto normály môžeme použiť ako výslednú farbu daného fragmentu, čím sa dajú dosiahnuť efektné výsledky (Príloha A(5), (6)) .

8.3.2 Tieňovacie modely

Štandardný zobrazovací proces v OpenGL využíva rozšírené Gouradovo tieňovanie. Avšak pre reálnejšie aplikácie je vhodné iné tieňovanie také, ktoré počíta osvetľovací model pre každý fragment. V Cg sa dajú veľmi jednoducho implementovať rôzne osvetľovacie modely. Ukážeme základný osvetľovací model, ktorý je zjednodušením Phongového osvetľovacieho modelu. Predpokladáme, že máme potrebnú normálu fragmentu z výpočtu uvedeného vyššie, prípadne priamo z textúry reprezentujúcej gradient.

```
normal = normalize(normal);

float3 vecToEye   = normalize(posEye   - IN.position);
float3 vecToLight = normalize(posLight - IN.position);

float3 vecHalf    = normalize(vecToLight + vecToEye);

float difuseLight = max(dot(normal,vecToLight),0);
float specLight = pow(max(dot(normal,vecHalf),0), 15 );

if(difuseLight <= 0) specLight = 0;

float3 difuse = float3(1.0,1.0,0.0)*difuseLight;
float3 specular = float3(1.0,1.0,1.0)*specLight;

OUT.color.xyz = difuse + specular + float3(0.1,0.1,0.1);
OUT.color.a = color.a;
```

Potrebné parametre môžu byť poslané z OpenGL ako uniform parametre, alebo vypočítané vo fragment programe, prípadne vertex programe. Pozíciu oka (`posEye`) a pozíciu svetla (`posLight`) sme získali z OpenGL. `IN.position` je vstupný parameter z vertex programu, ktorý definuje 3D pozíciu voxela. `Color.a` je alfa hodnota daného fragmentu získaná z 3D textúry. Tento model sa dá vylepšovať o ďalšie parametre, ako sú farba svetla, koeficienty difúzneho a zrkadlového odrazu, okolité osvetlenie. Dá sa rozšíriť aj pre použitie viacerých svetelných zdrojov, útlmové faktory a mnoho ďalších.

Programovanie v Cg zjednodušuje programovanie grafických kariet v ich asembleri pre vývojárov, ale zároveň umožňuje robiť zmeny v Cg kóde aj programátorom, ktorý nepoužívajú OpenGL, ale vyznajú sa v počítačovej grafike. Keďže Cg programy sa prekladajú počas behu aplikácie, jednoduchú zmenu kódu v rámci možností môže robiť aj používateľ na dosiahnutie pre neho vhodného vizuálneho efektu.

Týmto niesú vyčerpané možnosti Cg pre objemové zobrazovanie a vonkoncom nie pre všeobecné grafické zobrazovanie. V nasledujúcej kapitole uvádzame ešte jedno využitie Cg pre objemové zobrazovanie.

9. Prenosové funkcie

Pri skladaní výslednej hodnoty pixela pozdĺž pohľadového lúča vo výslednom obraze je vhodné špecifikovať koeficienty vyžarovania a pohlcovania energie. Ale dáta získané z merania alebo simulácie vo väčšine prípadov tieto hodnoty neobsahujú a neexistuje prirodzená cesta, ktorá by ich definovala. V praxi sú tieto koeficienty priraďované používateľom pre dané vzorky dát. Tento proces priraďovania je označovaný ako klasifikácia a môže byť popísaný pomocou prenosových funkcií

$$I_{emit} = T_{emit}(v) \qquad \text{a} \qquad v_k = T_{abs}(v),$$

kde I_{emit} reprezentuje koeficient vyžarovania a v_k koeficient pohlcovania.

Postupy ako automaticky generovať prenosové funkcie z dát alebo výsledného obrazu sú vo väčšine prípadov závislé na vstupných dátach [28]. Vo všeobecnosti navrhovanie prenosových funkcií je manuálny a zdĺhavý proces, ktorý vyžaduje dôkladné znalosti a štruktúru, ktorú dáta reprezentujú. Iné výsledky sa obdržia pri použití tých istých prenosových funkciách, ale rôznych dátach rovnakého objektu (CT, MRI). Pre vhodné definovanie daných funkcií v procese klasifikácie je veľmi dôležitá viditeľná odozva používateľových operácií. Preto je žiaduce pri modifikovaní prenosových funkcií aj súčasná interaktívna zmena počas zobrazovania objemových dát.

9.1 Teória klasifikácie

Hoci sú použiteľné spojité funkcie, v praxi sa prenosové funkcie realizujú ako indexové tabuľky pevnej dĺžky. Koeficient vyžarovania I_{emit} je zvyčajne reprezentovaný ako RGB zložka, ktorá dovoľuje vyžarovanie farebného svetla, čím získavame farebné dáta. Koeficient pohlcovania je reprezentovaný ako skalárna veličina v rozmedzí 0 až 1, ktorý definuje priehľadnosť danej vzorky. Tieto koeficienty môžeme skombinovať do RGBA hodnoty.

Objemové dáta sú reprezentované ako trojrozmerné pole bodov (voxelov). Podľa teórie vzorkovania môže byť spojitý signál rekonštruovaný z týchto bodov *konvolúciou* použitím príslušného filtra (interpoláciou daného stupňa). Prenosová funkcia môže byť preto aplikovaná priamo na diskrétne body pred rekonštrukciu alebo až na spojitý signál po rekonštrukcii [16]. Obe metódy vedú k viditeľne rozdielnym výsledkom.

Pred-klasifikácia označuje aplikovanie prenosovej funkcie na diskrétne vzorky bodov pred interpoláciou. Rekonštrukcia spojitého signálu je vykonaná na hodnotách s už priradením vyžarovaním a pohlcovaním.

Po-klasifikácia obracia poradie vykonávaných operácií. Aplikovanie prenosovej funkcie sa dostáva až za proces rekonštrukcie. Transfer funkcia je tak aplikovaná na spojitý signál namiesto aplikovania na diskrétne body.

9.2 Implementácia

V tejto časti ukážeme ako implementovať prenosové funkcie na grafickom hardvéri využívajúc techniky textúrového mapovania vysvetlené vyššie. Sú tu

popísané obe techniky pred-klasifikácie a aj po-klasifikácie. Tieto techniky vedú pri zmenách transfer funkcie k interaktívnym zmenám zobrazovaných dát za predpokladu dostupnosti využívaných funkcií grafickým akcelerátorom.

9.2.1 Pred-klasifikácia

Pred-klasifikácia vyžaduje zavedenie farebnej palety pred interpoláciou texelov, čo môžeme dosiahnuť aj bez využitia grafického zariadenia v predspracovaní dát. Ale OpenGL poskytuje možnosti, ktoré môžeme na to využiť.

9.2.1.1 Pixel Transfer (prenos dát)

Štandardná špecifikácia OpenGL ponúka možnosť ako aplikovať farebnú masku počas prenosu dát z hlavnej pamäti počítača do video pamäti na grafickej karte. Pri zmene prenosovej funkcie je potrebné opakované zavedenie dát na grafickú kartu, čo vyžaduje prenos veľkého objemu dát medzi hlavnou pamäťou a video pamäťou, čím sa výrazne obmedzí interaktivita danej aplikácie. Táto funkcia má názov `glPixelMap()`, ktorá požaduje 3 parametre. Prvým sa nastavuje mapovanie zložiek na iné zložky, druhý popisuje veľkosť tabuľky a tretí je smerník na tabuľku s danými hodnotami. Túto funkciu treba povoliť príkazom `glPixelTransfer(GL_MAP_COLOR, GL_TRUE)` a obdobným príkazom zakázať. Aby bol dosiahnutý žiadaný efekt musí byť povolené a nastavené mapovanie pred špecifikovaním textúr príkazom `glTexImage()`.

9.2.1.2 Textúry s paletami

Pre nevýhody uvedené vyššie, výrobcovia grafických zariadení zaviedli mechanizmus, ktorý dovoľuje mať vo video pamäti spolu s textúrou aj farebnú tabuľku. Tento mechanizmus je nazývaný *paletted textures*. Textúrová paleta môže zreteľne znížiť veľkosť potrebnej pamäte na uloženie textúry vo video pamäti. Namiesto ukladania RGBA hodnôt pre každý texel je uložený iba index do tabuľky pevnej dĺžky. Táto tabuľka je uložená spolu s danou textúrou vo video pamäti. Počas generovania príslušnej textúry v rasterizačnej jednotke grafického zobrazovania sú indexy v textúre nahradené farebnými hodnotami uloženými v tabuľke. Interpolácia textúrových hodnôt je aplikovaná až po nahradení indexov príslušnými hodnotami z palety, čo zodpovedá pred-klasifikácii pre prenosové funkcie.

OpenGL umožňuje využívanie týchto možností pomocou dvoch rôznych rozšírení. Prvé rozšírenie `EXT_paletted_texture` dovoľuje používanie paliet pre textúry. Textúry, ktoré využívajú palety sú vytvárané štandardným spôsobom ako RGBA textúry. Jediná zmena nastáva pri špecifikovaní textúry, kde interný formát RGBA textúry (`GL_RGBA`) je nahradený jedným z možných indexových formátov rôznej presnosti, napríklad `GL_COLOR_INNDEX8_EXT` zodpovedá 8bitovým dátam a veľkosti tabuľky. Pri použití tohoto rozšírenia musí každá textúra obsahovať vlastnú paletu, preto bolo zavedené ďalšie rozšírenie `GL_EXT_shared_texture_palette`, ktoré umožňuje používanie jednej palety viacerými textúrami. Toto taktiež redukuje veľkosť potrebnej video pamäti. Povolenie spoločných paliet pre textúry sa dosiahne príkazom `glEnable( GL_SHARED_TEXTURE_PALETTE_EXT)` a príkazom `glColorTableEXT()` sa definuje konkrétna paleta, kde prvý parameter je `GL_EXT_shared_texture_palette`, druhý zodpovedá farebným zložkám

v palete (GL_RGBA), tretí definuje veľkosť palety, štvrtý dátový typ (GL_BYTE) a posledný je smerník na danú paletu.

Hlavnou výhodou tejto metódy je možnosť zmeniť paletu prislúchajúcu textúram bez potreby nového zavedenia danej textúry. Pokiaľ grafické zariadenie povoľuje tieto rozšírenia, tak dosiahneme interaktívne výsledky pri definovaní prenosovej funkcie.

9.2.2 Po-klasifikácia

Po-klasifikácia vyžaduje mechanizmus zavedenia farebnej palety až po interpolácii texelov. Toto mapovanie farieb musí byť vykonané medzi generovaním textúry a aplikáciou textúry v rasterizačnej jednotke zobrazovacieho procesu. Toto požaduje špeciálne hárdvérové možnosti, ktoré umožňujú meniť texel priamo v textúrovacej jednotke.

9.2.2.1 Farebné palety pre textúry

Priame hardvérove využitie po-klasifikácie umožňuje OpenGL rozšírenie SGI_texture_color_table, ktoré je ale dostupné iba na výkonných pracovných staniciach firmy Silicon Graphics (SGI) [17]. Toto rozšírenie je veľmi podobné textúrovým paletám. Rozšírenie musí byť povolené príkazom glEnable(GL_TEXTURE_COLOR_TABLE_SGI) a farebná palete sa nastavuje príkazom glColorTableSGI() s rovnakými parametrami ako glColorTableEXT() okrem prvého, ktorý musí byť GL_TEXTURE_COLOR_TABLE_SGI. V tomto rozšírení je farebná paleta mapovaná až po textúrovej interpolácii a nie je obmedzená na jednu textúru, ale môže byť použitá na viaceré textúry.

9.2.2.2 Závislé textúry

Grafické akcelerátory pre bežné osobné počítače nedisponujú OpenGL rozšírením, ktoré by priamo aplikovalo farebnú paletu po textúrovej interpolácii ako je uvedené vyššie. Na po-klasifikáciu však možno využiť aj inú možnosť, ktorú poskytujú grafické zariadenia, nazývanú závislé textúry [3,16].

Myšlienka závislých textúr je založená na tom, že hodnoty z textúry sa použijú ako textúrové súradnice do inej textúry (palety). Farebná zložka RGBA získaná z druhej textúry je použitá ako konečná textúrová hodnota pre daný fragment. Využitie závislých textúr pre grafické karty od firmy NVidia definuje OpenGL rozšírenie NV_texture_shader. Obdobné rozšírenie poskytujú aj grafické zariadenia od firmy ATI, čo ale vedie k rozdielnym OpenGL programom. Preto si v nasledujúcej časti ukážeme spôsob ako využiť program v Cg na metódu po-klasifikácie pri použití rovnakého kódu pre grafické karty od rôznych výrobcov.

9.2.2.3 po-klasifikácia pomocou Cg

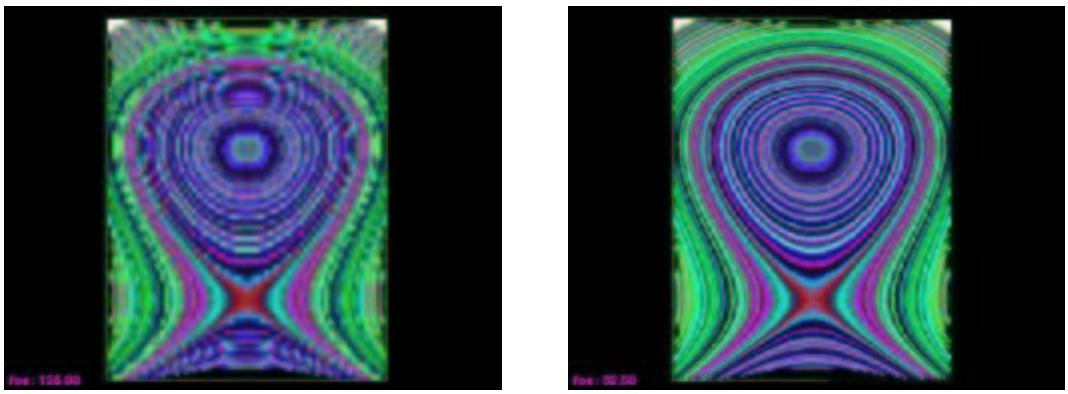
Na dosiahnutie efektu po-klasifikácie využívame fragment program. Táto metóda pracuje na obdobnom princípe ako závislé textúry popísanom vyššie. Princíp algoritmu spočíva v tom, že intenzitu z textúry reprezentujúcu objemové dáta použijeme ako index, textúrové súradnice, do textúry reprezentujúcu paletu. Na výpise 9.4 je uvedený fragment program napísaný v Cg.

```
struct input_data {
      float3 texcoord : TEXCOORD0;
};

struct output_data{
      float4 color    : COLOR;
};

//fragment program for post classification
output_data main( input_data IN, uniform sampler3D volume,
                        uniform sampler2D palette){
      output_data OUT;
      OUT.color = tex2D(palette, tex3D(volume,IN.texcoord).gb);
      return OUT;
}
```

Vstupnými dátami získanými z grafického procesu (graphics pipeline) sú textúrové súradnice daného fragmentu (`texcoord`). Z OpenGL obdržíme textúry reprezentujúce dáta (`volume`) a paletu (`palette`), ktorá reprezentuje prenosovú funkciu. Zložky RGB palety prislúchajú koeficientom vyžarovania a zložka A prislúcha koeficientu pohlcovania.
Výsledkom fragment programu je farba daného fragmentu, ktorú sme získali v hlavnom programe (`main`). Príkazom *tex3D(volume, IN.texcoord)* získame hodnotu intenzity z 3D textúry pre daný fragment. Zelenú a modrú zložku tejto hodnoty použijeme ako textúrové súradnice pre získanie farebnej hodnoty z palety. Môžeme použiť aj funkciu tex1D namiesto tex2D vyžadujúcu vyšší profil. Pri použití profilu fp20 program vykoná 3 inštrukcie, pri použití vyšších profilov sa počet inštrukcií zníži na 2 inštrukcie.



Obrázok 9.1: Rôzne výsledky dosiahnuté metódou pred-klasifikácie (vľavo) a *po-klasifikácie (vpravo)*

9.3 Zhodnotenie

Obe metódy aplikovania prenosovej funkcie sú vykonateľné interaktívne pri definovaní prenosovej funkcie za spolupráce grafického zariadenia. Pre jednotlivé metódy dostávame viditeľne rozdielne výsledky (obrázok 9.1) a to najmä pri veľkých zmenách prenosovej funkcie (vysoké frekvencie).

10 f3dvr program

Vyššie uvedené algoritmy sú implementované v programe f3dvr, ktorý dáva na jednoduchý prehľad jednotlivých techník. Program môže slúžiť aj ako doplňujúca výuková pomôcka pri študovaní objemového zobrazovania. V nasledujúcich častiach popíšeme jeho základné funkcie.

10.1 Knižnice

V programe sú použité rôzne knižnice využívané na skvalitnenie a urýchlenie aplikácie. Na tvorbu grafického používateľského rozhrania (GUI) je použitá knižnica wxWidgets verzie 2.4.0 [27], ktorá je voľná, platformovo nezávislá pre tvorbu GUI. Na využívanie výkonu grafických akcelerátorov je použitá knižnica OpenGL [25]. V aplikácii sú použité Cg knižnice na priame programovanie grafických kariet [23]. Ďalšou knižnicou je knižnica f3dformat [24], ktorá slúži na jednoduchý popis objemových dát a poskytuje funkcie na načítanie a ukladanie týchto dát. Na kompresiu objemových dát slúži knižnica zlib [28]. Aplikácia ešte využíva ďalšie štandardne používané knižnice. Pri výbere knižníc bol zohľadnený fakt, že sa jedná o vedeckú aplikáciu, čiže by mala korektne pracovať aj pod rôznymi operačnými systémami. Aplikácia bola úspešne testovaná pod operačnými systémami Windows XP a Linux Mandrake 9.2.

10.2 Zobrazovacie algoritmy

Algoritmy využité v programe spadajú do kategórie algoritmov pre priame objemové zobrazovanie. Ich konštrukcia pomocou mapovania textúr vychádza z algoritmov popísaných v kapitole 6. Program zobrazuje objemové dáta pomocou 2D textúr (*2D texture*) (Príloha A(1), (2)) a 3D textúr, kde je možnosť voľby objektovo orientovaných rezov (*3D texture OAS*) alebo pohľadovo orientovaných rezov (*3D texture VAS*) (Príloha A(3),(4)). Ďalšími dvoma zobrazovacími módmi sú tie, ktoré priraďujú výslednú farbu fragmentu konverziou normály daného fragmentu na farbu. Pod konverziou rozumieme prevod normálových súradníc z intervalu <-1,1> na interval <0,1>. Tieto módy sa líšia iba vo výpočte normál. Prvý mód pracuje s normálami uloženými v 3D textúre získanými v predspracovaní (Príloha A(5)). Druhý mód tento výpočet vykonáva pre každý fragment tak ako v kapitole 8.3.1 (Príloha A(6)). Posledný zobrazovací mód počíta jednoduchý osvetľovací model uvedený v časti 8.3.2 (Príloha A(7)). Normálové a osvetľovací mód využíva 3D textúru na zobrazovanie dát a to buď ako objektovo orientované rezy, alebo pohľadovo orientované rezy.

Na skladanie farieb jednotlivých textúrových rezov je použité miešanie (blenidng), (viď. časť 6.2). Používateľ má možnosť výberu zo štandardného súčtového modelu alebo maximálnej a minimálnej intenzitovej projekcie.

Špecifikácia OpenGL ponúka možnosť využiť orezávacie roviny, s ktorými je možné rotovať, posúvať a meniť orezávací polpriestor.

Aplikácia ponúka informácie o zobrazovaných dátach a základné vlastnosti a rozšírenia OpenGL.

Zobrazované dáta môžu byť zobrazované pomocou rovnobežného premietania alebo stredového. Daný objem je možno presne natočiť podľa jeho šiestich strán.

Aplikácia poskytuje základné nastavenia ako je počet zobrazovaných rezov. Pri menej výkonných grafických kartách je nastavením menšieho počtu rezov možné dosiahnuť rýchlejšie zobrazovanie. Pre zobrazovanie pomocou 2D textúr sa dá zakázať prepínanie medzi sadami 2D textúr (*switching*). V nastaveniach sa dá nastaviť hraničná hodnota pre alfa test a mód aplikovania. Týmto testom zakážeme zobrazovanie fragmentov s alfa hodnotou väčšou, menšou alebo nerovnou podľa zvoleného módu ako je hraničná hodnota.

V aplikácii sú ďalej implementované prenosové funkcie. Na ich editovanie slúži jednoduchý editor, v ktorom sa nastavujú jednotlivé farebné zložky po jednej alebo ich vzájomnou kombináciou. Pre zobrazovanie kriviek reprezentujúce prenosové funkcie môžeme zvoliť lineárnu lomenú krivku, beziérovu alebo voľne rukou definovanú funkciu. Pri lineárne lomenej a beziérovej krivke sa zadávajú kontrolné body, s ktorými je možné hýbať a mazať ich. Tieto prenosové funkcie podporujú oba varianty aplikovania (*pred a po klasifikácia*) ako sú popísané v kapitole 9.

11 Hardvérové urýchľovanie pre AngioVis projekt

AngioVis projekt je projekt na vyhľadávanie ciev a zobrazovanie objemových dát so zvýraznenými cievami pre potreby *angiografie* dolných končatín [22]. Rozmer dát je 512x512x1298 pri 12 bitovej presnosti, ale uložené ako 16 bitové, čo predstavuje 650MB. Preto nie je cieľom tejto časti projektu dosiahnuť interaktívne zobrazovanie daných dát, ale využiť grafické akcelerátory na urýchlenie generovania snímok potrebných pre lekárov na posúdenie stavu pacienta. Pod pojmom hardvérového urýchľovania rozumieme využitie grafických kariet na zobrazovanie, pričom pod softvérovým zobrazovaním rozumieme vykonávanie všetkých potrebných úkonov pri zobrazovaní (spracovanie geometrie, osvetľovacie modely a iné) na hlavnom procesore počítača.

11.1 Požiadavky

Výstupom danej aplikácie má byť sada obrázkov, najčastejšie 24, v originálnom rozlíšení, na ktorých sú dáta kombinované s maskou reprezentujúcou cievy v dátach s použitím určitého tieňovania a osvetľovacieho modelu na zvýraznenie ciev (Príloha A(8)). Na riešenie týchto požiadaviek využívame metódy popísané vyššie a to 3D textúry a pohľadovo orientované rezy. Pre medicínske aplikácie je vhodnejšie kolmé premietanie. Na kombinovanie masky s dátami využívame fragment program napísaný v Cg a v ňom kombinujeme dáta s prenosovými funkciami.

Pri hlbšej analýze problému sa vyskytli určité problémy s veľkosťou dát a požadované rozmery na výstupné snímky.

Kvalita snímkov: Nezaoberáme sa možnými metódami na zlepšenie kvality obrazu, iba požadujeme rozmer snímkou v originálnom rozlíšení pri kolmom premietaní. Toto vyžaduje minimálne rozlíšenie 725x2517 z čoho vyplýva ďalší problém a to veľkosť zobrazovacieho okna. Oknový systém nepovolí vytvorenie okna s rozmermi väčšími ako aktuálne rozlíšenie monitora. S požiadavkou na off-line generovanie snímkov a nie ich zobrazovaním sa pokúsime navrhnúť aplikáciu, ktorá nepotrebuje žiadne okno a je ovládaná z príkazového riadku.

Veľkosť dát: Dáta potrebné na generovanie majú veľké rozmery presahujúce povolené hodnoty pre 3D textúry. Naviac, pre dané výpočty potrebujeme ešte masku, ktorá definuje pozíciu cievy v dátach, reprezentovanú ako trojrozmerné pole s rovnakým rozlíšením a s 8 bitovou presnosťou, čo predstavuje ďalších asi 345MB dát.

11.2 Riešenie – kvalita snímkov

Pre obmedzenie dané nemožnosťou vytvorenia okna väčšieho ako rozlíšenie monitora sa naskytá možnosť využiť pixel buffer (pbuffer). Aj na rozmery pbuffera sú kladené obmedzenia, ale len grafickou kartou a tie nám postačujú. Keďže nepotrebujeme hneď vidieť dané snímky, tak nie je potrebné vytvoriť konvenčné okno, čo umožňuje využiť výkon grafických kariet iba na renderovanie dát. Pre vytvorenie pbuffer-a však potrebujeme určité nastavenia, ktoré sa získavajú

z OpenGL okna. Preto využívame OpenGL okno len na získanie daných nastavení a potom okno zrušíme. Pri ich získavaní nie je okno zobrazované.

11.3 Riešenie - veľkosť dát

V súčasnej dobe nie je reálne použiť celé dáta ako jednu 3D textúru. Preto sme navrhli algoritmus na ich rozloženie do menších častí - blokov, ktoré je možné reprezentovať pomocou 3D textúr a tie potom postupne renderovať.

Dôležitú úlohu hrá aj konkrétny rozmer bloku, ktorý musí byť tak veľký, aby sa zmestil do video pamäte, ale aby nebol zase príliš malý. Pre našu testovaciu zostavu sme zvolili bloky veľkosti 512x512x128, pričom vo video pamäti potrebujeme k danému bloku aj blok s maskou s rovnakými rozmermi.

11.3.1 Sériové generovanie snímkov (metóda 1)

Generovanie snímkov touto metódou vedie k vytvoreniu celých snímkov postupne jeden po druhom. Pre jeden snímok zoberieme spodný blok (prvý), vyrenderujeme ho a postupujeme ďalej až k hornému (poslednému) bloku. Po vygenerovaní tohoto snímku vyčistíme renderovací buffer a pokračujeme generovaním ďalšieho snímku až po daný počet snímkov. Pri tomto postupe potrebujeme mať pre generovanie jedného snímku postupne všetky dáta v pamäti, čo vedie k veľkému prenosu dát medzi hlavnou pamäťou a video pamäťou. Tento prenos všetkých dát sa vykonáva pre všetky snímky. Preto bola navrhnutá iná metóda, ktorá eliminuje tento prenos.

11.3.2 Paralelné generovanie snímkov (metóda 2)

Pri tejto metóde sa generujú všetky snímky naraz postupne zdola nahor. Generovanie snímkov začína spodným (prvým) blokom. Vygeneruje sa potrebný počet obrazov daného bloku pre potrebné smery pohľadu, ktoré zodpovedajú počtu snímkov. Tieto obrazy sú uložené na ďalšie spracovanie. Po vygenerovaní obrazov spodného bloku sa vygenerujú nasledujúce postupne po blokoch až k vrchnému (posledný). Po tomto procese máme sadu obrazov, ktoré pospájajú do výsledných snímkov. Prislúchajúce obrazy blokov daného pohľadu sa spoja. Pri tejto metóde dochádza pri generovaní k prenosu dát medzi hlavnou pamäťou a video pamäťou iba raz.

Ďalším faktorom, ktorý ovplyvňuje výsledný čas generovania snímkov je aj prenos vyrenderovaného snímku z video pamäti do hlavnej pamäti na ďalšie spracovanie. K prenosu dát z video do hlavnej pamäte používame príkaz `glReadPixels`, ktorý je vo všeobecnosti dosť pomalý. PBuffer ponúka možnosť reprezentovania buffer-a, do ktorého renderuje (color buffer) ako textúry. Preto sa nám naskytla možnosť využiť príkaz `glGetTexImage` na prenos dát z video do hlavnej pamäte s očakávaním lepších výsledkov.

11.4 Výsledky

Výslednú aplikáciu sme testovali na dvojprocesorovom počítači AMD Athlon 1.6Mhz, 2GB RAM a grafickej karte MSI GeForce FX 5700Ultra TD 128. Dané výsledky sú zhrnuté v tabuľke 11.1, kde sú pre jednotlivé metódy udané časové hodnoty v sekundách potrebné na vyrenderovanie a prenos dát do hlavnej pamäti. V tabuľke 11.2 sú nachádzajú testy bez prenosu výsledných snímkov do hlavnej pamäte, jedná sa iba o renderovanie snímkov.

Pri renderovaní je použitá metóda pohľadovo orientovaných rezov, ktorých počet bol 800. Na dáta sa využíva prenosová funkcia kombinovaná vo fragment programe s maskou na dosiahnutie dobrých vizuálnych efektov. Fragment program je preložený do 49 inštrukcií. Normály pre potreby tieňovania sú rátané vo fragment programe.

počet snímkov	paralelné generovanie (s)	sériové generovanie (s)
1	32.015	32.140
4	120.875	132.424
12	362.125	404.204
24	720.514	809.514

tabuľka 11.1: rennderovanie snímkov a prenos do hlavnej pamäte

počet snímkov	paralelné generovanie (s)	sériové generovanie (s)
1	27.576	27.718
4	111.140	122.282
12	338.250	379.234
24	676.732	762.640

tabuľka 11.2: renderovanie snímkov bez prenosu

Z daných meraní sa potvrdil predpoklad, že metóda vytvárajúca snímky paralelne (metóda 2) je rýchlejšia ako metóda generujúca snímky sériovo (metóda 1). Využitie pbuffera ako textúry a následné použitie príkazu `glGetTexImage` neviedlo k výraznému zlepšeniu časov potrebných na prenos dát z video do hlavnej pamäte. Toto urýchlenie predstavuje približne 4%.

12 Záver

V predkladanej diplomovej práci boli ukázané možnosti využitia grafických akcelerátorov pre zobrazovanie objemových dát za pomoci OpenGL rozšírení. Jej cieľom nebolo hodnotiť dané algoritmy vzhľadom na rýchlosť vykresľovania, pretože táto úloha v dnešnej dobe silne závisí od výkonu grafického akcelerátora. Zamerali sme sa skôr na popis možností a funkcií, ktoré sú zahrnuté v OpenGL rozšíreniach pre programátorov aplikácií pracujúcich s objemovými dátami.

Pre mnohých začínajúcich programátorov sú OpenGL rozšírenia veľkou neznámou. Pre nich sa ukázalo, na čo sa jednotlivé rozšírenia môžu využiť. V časti o Cg sme ukázali základné možnosti tohoto programovacieho jazyka, ktorý je dostatočne názorný a v istých mierach aj jednoduchší ako OpenGL rozšírenia. Vďaka jeho syntaxy prebratej z jazyka C sa dá veľmi rýchlo naučiť a určite je názornejší ako programovanie grafických kariet v ich vlastnom asembleri. Preklad tohoto jazyka do asembleru grafických kariet firmy NVidia je veľmi efektívny. Ukázali sme aj algoritmy, ktoré sa dajú implementovať pomocou OpenGL rozšírení, ale tieto sú názornejšie, ak sú uvedené v zrozumiteľnejšom tvare, a to v Cg. Tieto algoritmy boli aj aplikované v konkrétnom programe zobrazujúcom objemové dáta uložené v f3d formáte. Program využíva algoritmy popísané v tejto práci. Naviac bola vyhotovená časť AngioVis projektu spočívajúca v off-line renderovaní rozsiahlych objemových dát, ktorá tiež využíva popísané algoritmy a OpenGL rozšírenia.

Literatúra

[1] B. Bargen, P. Donelly. *Inside DirectX*. Microsoft Press, 1998.

[2] B.Cabral, N. Cam, J. Fortan. *Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware*. ACM symp. On Vol. Vis., 1994

[3] K. Engel, T. Ertl. *Interactive High-Quality Volume Rendering with Flexible Consumer Graphics Hardware*. In Proc. Eurographics, 2002

[4] R. Fernando, M.J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003

[5] J.Foley, A van Dam, S. Feiner, J. Hughes. *Computer Graphics, Principles And Practice*. Addison-Weseley, 1993

[6] R.S. Galagher. *Comupter Visualization: graphics techniques for scientific and engineering analysis.* CRC Pres, 1995

[7] J. Kajiya, B. Von Herzen. *Ray Tracing Volume Densities*. In Proc. Sigraph, 1984

[8] P. Kovach. *Inside Direct 3D*. Microsoft Press, 2000

[9] E. LaMar, B. Hamman, K. Joy. *Multiresolution Technioques for Interactive Texture-based Volume Visulation*. In Proc. IEEE Visualization, 1999

[10] M. Levoy. *Display of surfaces from Volume Data*. IEEE Comp. Graph. Appl., 8(5):29-37, 1988

[11] E. Lengyel. *The OpenGL Extensions Guide*. Charles Rivier Media, Inc., 2003

[12] W. Lorensen, H. Cline. *Marching Cubes: A High Resolution 3D Surface ConstructionAlgorithm.* Comp. Graphics, 21(4):163-169, 1987

[13] J. Marks, W. Ruml, K. Ryall, J. Seims, et al. *Design galleries: a general approach to setting parameters for computer graphics and animation*. Siggraph, 1997

[14] N. Max. *Optical models for direct volume rendering.* IEEE Transactions on Visualization and Computer Graphics, 1(2):99-108, 1995

[15] H. Pfister, J. Hardenberg, J. Knittel, H. Lauer, L. Seiler. *The VolumePro Real-Time Ray-casting System*. Sigraph, 1999

[16] Ch. Rezk-Salama. *Volume Rendering Techniques for General Purpose Graphics Hardware*. Der Technischen Fakultät der Universität Erlangen-Nürnberg, 2001

[17] M. Segal, K.Akley. *The OpenGL Graphics System: A Specification*. http://www.opengl.org.

[18] M.Šrámek. *Visualization of Volumetric Data by Ray Tracing*. Österreichische Computer Ges., 1998

[19] C. Upson, M. Keeler. *V-BUFFER: Visible Volume Rendering.* In Proc. Sigraph, 1988

[20] P.L. Wiliams, N. Max. *A volume density optical model.* 1992 Workshop on Volume Vizualization, pages 61-68, 1992

[21] O.Willson, A. Van Gelder, J. Wilhelms. *Direct Volume Renderng via 3D-textures.* Technical Report UCSC-CRL-94-19, Univ. of Carolina, Santa Cruz, 1994

[22] The Official AngioVis project Website. http://www.cg.tuwien.ac.at/research/vis/angiovis/

[23] The Official Cg Website. http://developer.nvidia.com/Cg/

[24] The Official f3d format Webiste. http://pirin.viskom.oeaw.ac.at/~milos/f3d/

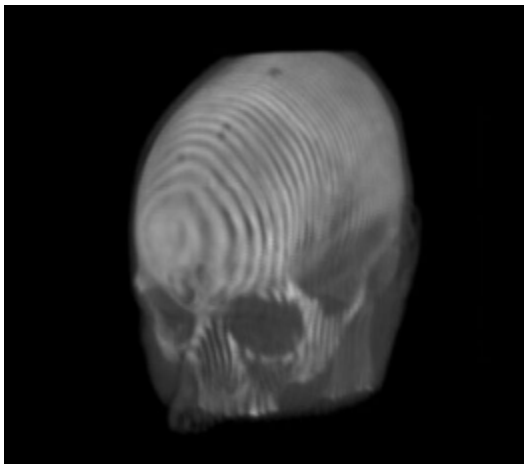
[25] The Official OpenGL Website. http://www.opengl.org

[26] The Official OpenGL extensions Website.

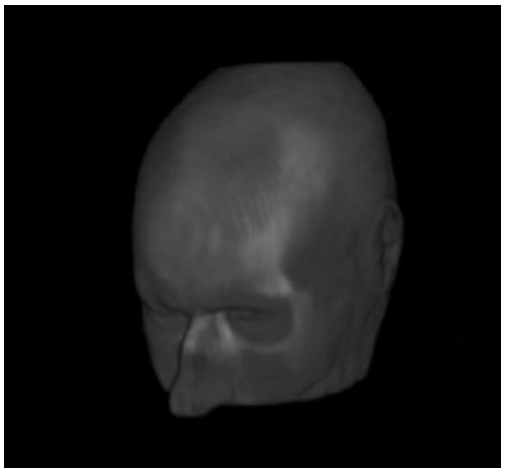
http://oss.sgi.com/projects/ogl-sample/registry/
[27] The Official wxWidgets Website. http://www.wxWidgets.org

[28] The Official zlib Website. http://www.gzip.org/

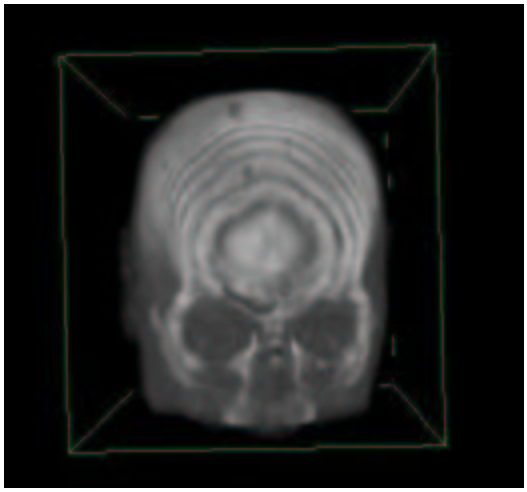
Príloha A



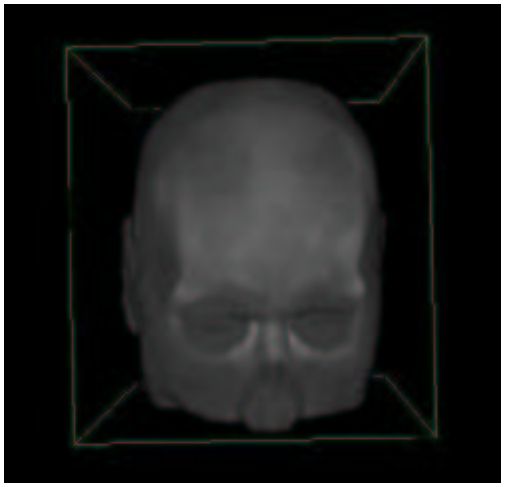
(1)



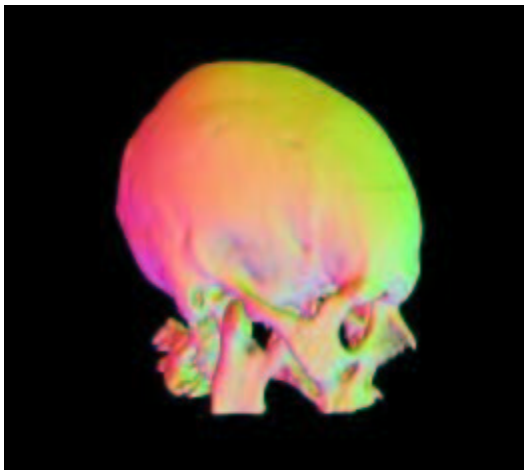
(2)



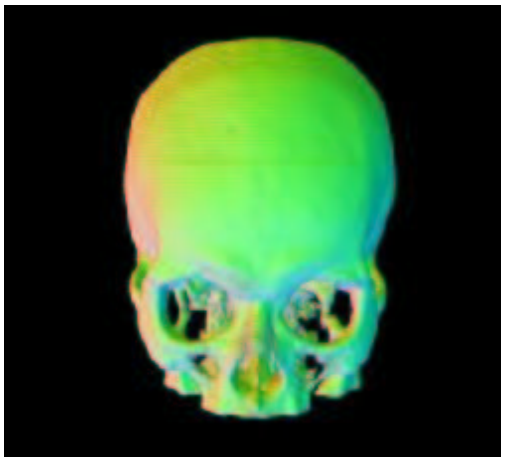
(3)



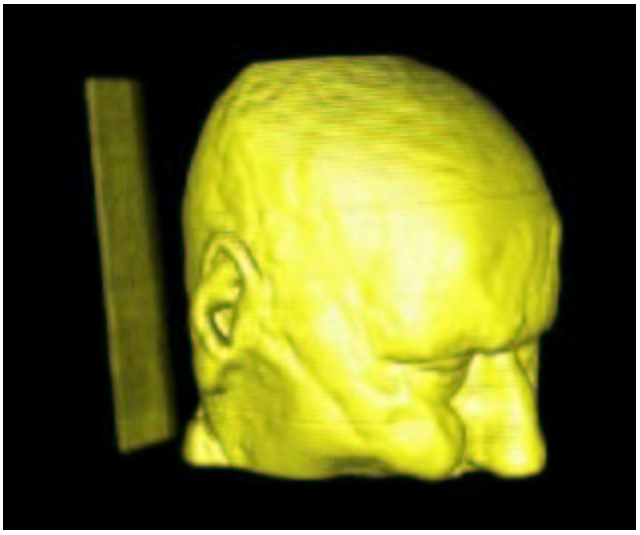
(4)



(5)

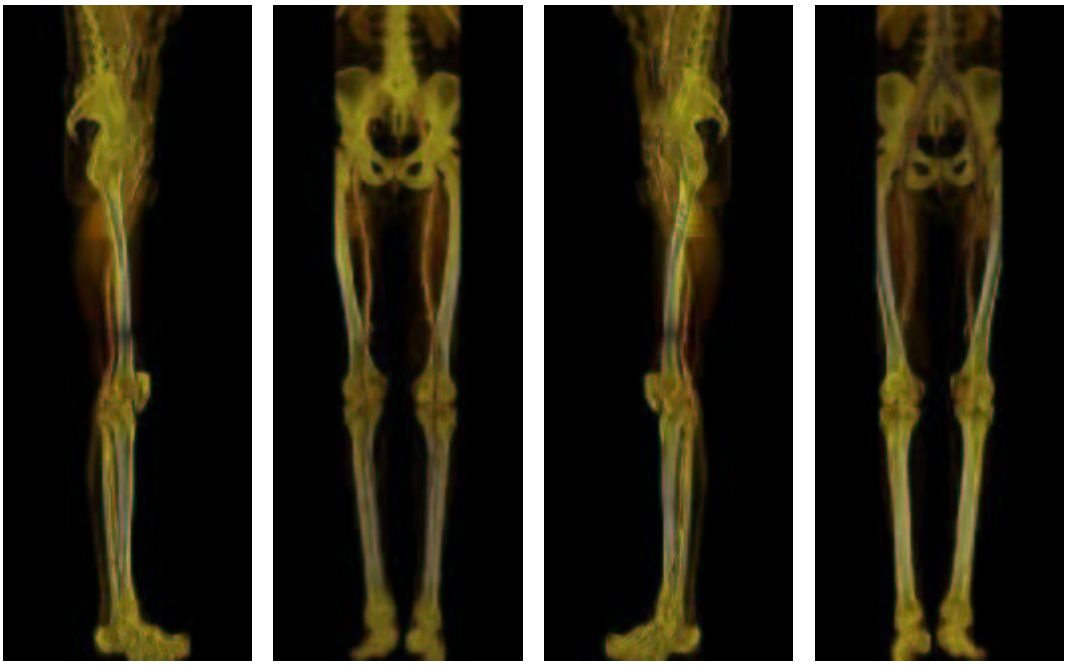


(6)



(7)

obrázky príloha A: *Na všetkých obrázkoch boli použité dáta s rozlíšením 170x190x180. Mapovanie pomocou 2D textúr, 50 rezov (1), 260 rezov (2). Mapovanie pomocou 3D textúry, pohľadovo orientované rezy, 50 rezov (3), 260 rezov (4). Priradenie farby pomocou normály, 512 rezov, alfa test 35%, normály uložené v 3D textúre (5), normály počítané pomocou fragment programu (6). Tieňovaný povrch pomocou fragment program, 512 rezov (7). Na obrázkoch (8) sú snímky urobené v rámci projektu AngioVis pomocou hardvérového urýchlenia.*



(8)

Príloha B

Súčasťou diplomovej práce je aj CD disk, na ktorom je možné nájsť informácie súvisiace s diplomovo prácou.

Zoznam:

Text diplomovej práce.
Program f3dvr.
Vstupné dáta.
Zdrojový kód programu f3dvr.
Testovacie palety pre prenosové funkcie.
Knižnice