Fakulta matematiky, fyziky a informatiky

Univerzita Komenského, Bratislava

Rigorózna práca

Mgr. Michal Červeňanský 2006

Fakulta matematiky, fyziky a informatiky
Univerzita Komenského, Bratislava
Katedra aplikovanej informatiky

Vybrané techniky objemového zobrazovania s použitím komerčných grafických akcelerátorov

Rigorózna práca

Mgr. Michal Červeňanský

Bratislava 2006

Čestné prehlásenie:

Čestne prehlasujem, že som túto rigoróznu prácu vypracoval samostatne s použitím uvedenej literatúry.

.....

Michal Červeňanský

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 žiadajú stále reálnejšie stvárnenie okolitého sveta pomocou počítača. Ako vznikajú nové a väčšinou náročnejšie algoritmy, tak spolu s týmito nárokmi na výpočtový výkon sú vyvíjané aj rýchlejšie procesory a grafické karty, ktoré sú primárne určené na zobrazovanie grafiky. Výkon týchto grafických akcelerátorov narastá a v niektorých parametroch dokonca prekonávajú parametre hlavných procesorov osobných počítačov. Tieto grafické akcelerátory sa v poslednej dobe stávajú nástrojmi na rôzne výpočty, nie len z oblasti počítačovej grafiky.

Objemové zobrazovanie (*volume rendering*) je súčasťou počítačovej grafiky, ktorá v dnešnej dobe zaberá veľké pole pôsobnosti. Je využívané hlavne na medicínsku vizualizáciu, ktorá je hnacím motorom tejto oblasti a v iných oblastiach ako sú geológia, geografia, archeológia, rôzne simulácie prúdenia a podobne.

Keďže objemové zobrazovanie spadá pod počítačovú grafiku, boli navrhnuté algoritmy využívajúce silu grafických akcelerátorov na potrebné výpočty, ktoré potrebujú vo väčšine prípadov vysoký výkon na dosiahnutie požadovaných výsledkov. V tejto práci je uvedený stručný prehľad niektorých techník z podoblastí objemového zobrazovania s využitím tohto hardvéru. V práci sú uvedené techniky priamo súvisiace so zobrazovaním, generovaním obrázkov poskytujúcich potrebu a prehľadnejšiu informáciu pre používateľa. Sú v nej uvedené aj techniky, ktoré sa zameriavajú na určité časti v dátach, ktoré sú podstatnejšie ako ostatné a slúžia na lepší vnem pre používateľa. V práci je navrhnutý nový algoritmus na generovanie týchto obrázkov.

Cieľom práce je dať čitateľovi základný prehľad rôznych techník súvisiacich s objemovým zobrazovaním a využitím grafických akcelerátorov. Uvedené techniky sú implementované v programe, ktorý demonštruje výkon grafických akcelerátorov a prehľad daných techník v praxi.

2 Vizualizácia objemových dát

Objemové zobrazovanie reprezentuje metódy zamerané na spracovanie trojrozmerných skalárnych dát za účelom ich zobrazenia v dvojrozmernej rovine [9,28]. 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

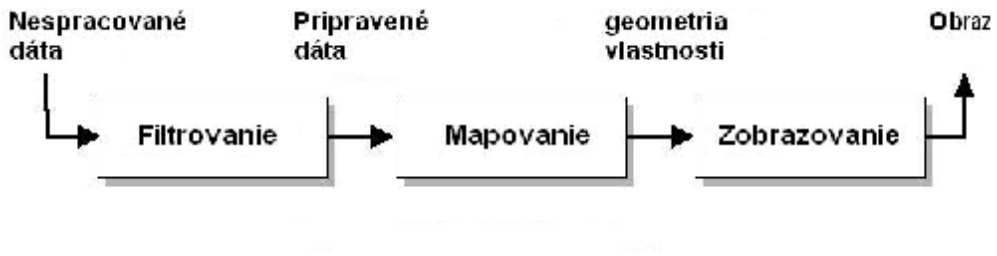
Objemové dáta, v porovnaní s povrchovými dátami, ktoré reprezentujú len daný povrch telesa, sú používané na popísanie celej vnútornej štruktúry objektu. Objemové dáta dovoľujú modelovať tekuté a plynné objekty rovnako, ako sa vyskytujú v prírode (napr. oblaky, hmla, oheň a voda).

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.*

2.2 Objemové zobrazovanie

Proces zobrazovania objemových dát je často popisovaný ako *visualization pipeline*, 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 detegované štruktúry. Tento krok môže byť jednoduchý a to definovaním, ale aj komplexný a to prehľadávaním celého objemu. Cieľom je previesť abstraktnú informáciu do zobraziteľných 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 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 [9,28]. 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 opísateľné s možnosťou vhodného uloženia, kompresie a prenositeľnosti. 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 [28] definovaného pre binárnu scénu zjednotením voxelov s hodnotou 1. Ďalej predpokladajme, že objekt je pospájaný tak, že dva ľubovoľné 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* [19], navrhnutý pre vytváranie trojuholníkového modelu z 3D medicínskych dát. Algoritmus spracováva scénu bunku 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 polo priehľadné 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 pixla. 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 pixlu alebo naopak. Tieto algoritmy spadajú do kategórie hybridného objemového zobrazovania.

3.2.1 Objektovo orientované techniky

Objektovo orientované techniky [29] 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 počítaná výsledná hodnota z jednotlivých príspevkov celého objemu, ktoré zodpovedajú prechodu pohľadového lúča cez dáta. Typickým predstaviteľom tejto metódy ja algoritmus sledovania lúča (*ray casting*) [12,18].

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 [21,28,32].

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 [32], 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 rástol 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 žiadnym 3D výkonom alebo 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. Z toho dôvodu sa začal využívať pojem grafický akcelerátor, ktorý výrazným podielom urýchľuje grafické aplikácie.

Firmy ponúkajúce grafické akcelerátory začali implementovať aj multi GPU systémy (SLI, CorssFire), čo znamená že v jednom počítači sa nachádza viacero grafických kariet (dve až štyri), ktoré sa využívajú na zobrazovanie. Tieto systémy pracujú na princípe rozdelenia vykresľovanej scény buď na fixné časti, kde každá grafická karta renderuje

svoju časť, alebo dynamicky, podľa náročnosti daných scén. V ponuke sú aj multi GPU systémy, ktoré na jednej doske grafickej karty majú 2 grafické procesory.

V dnešnej dobe grafické karty umožňujú mimo fixného grafického zobrazovania aj priame programovanie niektorých častí, čo dáva voľnosť vývojárom na rôzne výpočty, nie nutne súvisiace s grafikou, ktorým sa venuje viacero oblastí [39]. Tieto programovateľné časti sú v súčasnosti dve a to časť spracujúca vrcholy (*vertex shader*) a časť spracujúca fragmenty (*fragment shader*), každá táto časť má fixný počet jednotiek (*shaderov*), ktoré vykonávajú danú funkcionalitu. Viacej o tejto problematike sa venujeme v nasledujúcej časti.

4.3 Budúcnosť

Budúca generácia grafických kariet by mala podporovať *unifikované shadery*, programovateľné jednotky spájajúce vertex a fragment jednotky do jednej spoločnej jednotky. Vývojár určí koľko jednotiek bude pridelených daným (vertex, fagment) častiam. Malo by to podporovať aplikácie, ktoré využívajú vo väčšej miere len jednu časť a druhá je nevyužitá. Programátor (ovládače) pridelí počet jednotiek tak, aby bol výpočtový výkon rozdelený čo najefektívnejšie.

Ďalším pokusom pre rozšírenie GPU ako koprocesora pre CPU budú grafické karty od firmy ATI s názvom *FireStream.* Tieto karty majú byť špeciálne navrhnuté na výpočty rôzneho druhu a majú mať špeciálne ovládače, ktoré budú podporovať výpočty v plávajúcej desatinnej čiarke. Majú byť využívané pre aplikácie zaoberajúce sa napríklad finančnými predpoveďami, databázami ale aj pre rôzne vedecké výpočty.

4.4 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* [8] a je zobrazené na obrázku 4.1. Vstupom tohto procesu je zoznam vrcholov primitív (trojuholníkov) s dodatočnými údajmi o farbe, normále atď.. Proces

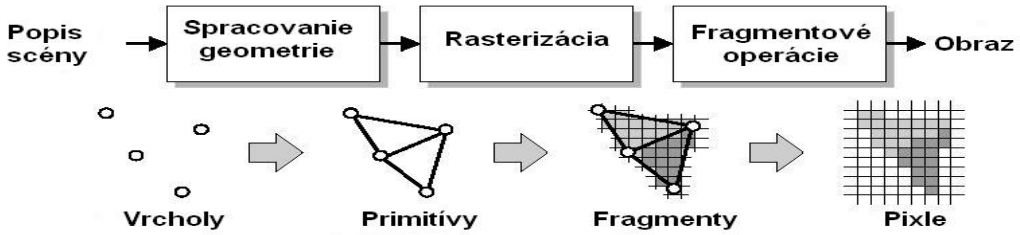
dekompozície trojrozmernej scény na rovinné útvary sa nazýva *teselácia (tesellation).* Výstupom display traversal procesu je rastrový obraz s farebnými hodnotami, ktoré sú zobrazené na obrazovke. Graphics pipeline sa delí na tri základné vrstvy.

Spracovanie geometrie transformuje (rotuje, posúva, škáluje atď.) vstupné údaje z modelových súradníc do svetových súradníc a potom premieta 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 pixlu na monitore.

Fragmentové (*Per-fragment*) operácie sú aplikované následne 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.

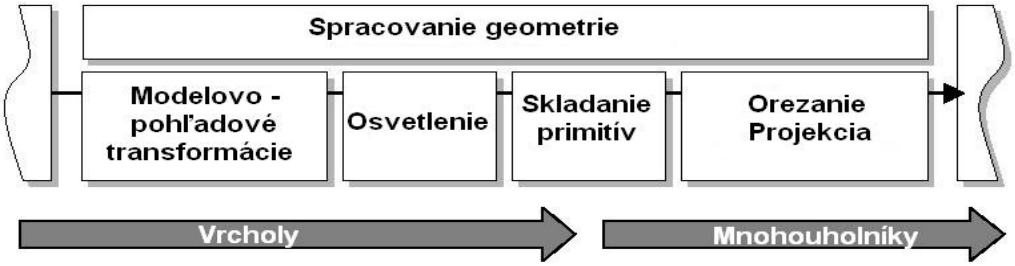
Pre dobré porozumenie niektorých algoritmov je dobré poznať proces grafického prechodu z trojrozmernej scény na rastrový obraz. Preto si ešte podrobnejšie vysvetlíme jednotlivé kroky.



obrázok 4.1: grafické zobrazovanie – graphics pipeline

4.4.1 Spracovanie geometrie

Spracovávenie prichádzajúcich vrcholov sa nazýva aj per-vertex operations. Táto jednotka počíta lineárne transformácie ako posunutie, otočenie, škálovanie a projekciu 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. Modelová 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, ak je povolené osvetlenie prebehne výpočet farby 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 programovateľné grafické karty umožňujú nahradiť pevne danú postupnosť výpočtu programom (vertex program), ktorý je aplikovaný na každý prichádzajúci vrchol. Tento vertex program je aplikovaný vo vertex jednotke (vertex shader) a nahrádza prvé tri časti spracovania geometrie.

4.4.2 Rasterizácia

Rasterizácia je proces konverzie geometrických dát v obrazovej rovine na fragmenty. Každý fragment zodpovedá jednému pixelu vo výslednom obraze (pokiaľ nie je nastavené inak). Proces rasterizácie sa môže ďalej deliť na tri rozdielne časti, 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.

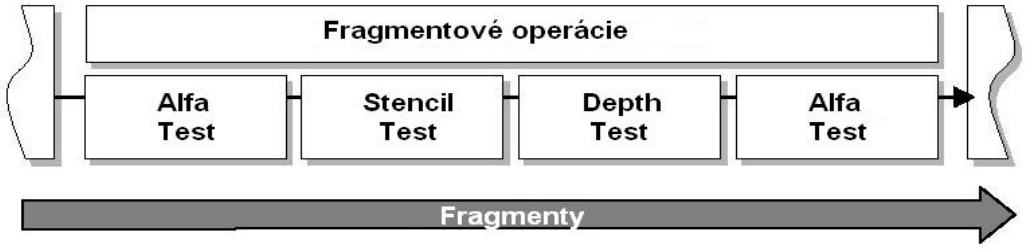
Generovanie textúr: Textúry sú 1, 2, 3 - rozmerné reprezentácie dát, ktoré sú nanesené na mnohouholník podľa textúrových súradníc určených pre vrcholy. Pre každý fragment sa interpoláciou textúrových súradníc vrcholov získajú hodnoty, ktoré odkazujú na príslušné miesto v danej textúre. Táto hodnota sa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

Aplikovanie textúr: Ak je povolené mapovanie textúr, farba z textúry prislúchajúca danému texelu sa aplikuje na daný fragment. Týmto krokom má príslušný fragment priradenú výslednú farebnú hodnotu a priehľadnosť.

Aj v tejto časti grafického zobrazovania umožňujú nové grafické karty aplikovať svoje vlastné programy, fragment programy, na dosiahnutie žiadaných efektov. Fragment programy sú aplikované vo fragment jednotke, ktorá úplne nahrádza posledné dve časti rasterizácie a časť z rasterizácie mnohouholníka.

4.4.3 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 do frame buffera, 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 zakazuje 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 prekrývajúcich sa objektov. 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 je možné nájsť v [8]. 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 tohto zariadenia. Samozrejme vhodne navrhnuté API uľahčuje prácu vývojárom pri vyvíjaní nového hardvéru. 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.

Rozšíriteľnosť: 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 nežiaduce obmedziť aplikáciu na jeden typ zariadenia alebo operačný systém. Toto je vhodné pre prototypové aplikácie, ktoré sú použité na vyhodnotenie výkonnosti v rôznych prostrediach.

Pre programovanie grafických kariet boli vyvinuté samostatné programovacie jazyky, ktoré kooperujú s jednotlivými API. Pre Direct3D to je HLSL (*high level shader langiuage*)[36] a pre OpenGL to je assembler podporovaný priamo v OpenGL rozšíreniach. Pre tento asembler bol vyvinutý jazyk Cg (*C for Graphics*) firmou nVidia [35], ktorý je

kompatibilný so syntaxou HLSL. Pre OpenGL bol navrhnutý aj jazyk glsl (*OpenGL shading languge*)[38], ktorý je priamo zahrnutý do špecifikácie 2.0. Programovací jazyk Cg bude popísaný nižšie, pretože v ňom budú popísané algoritmy v tejto práci.

5.1 Direct3D

Direct3D [15] 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 hardvérových 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 média layer 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 [40] 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. 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 rozšírení (*OpenGL extensions*). Vývojári tak majú voľnú ruku pri tvorení OpenGL programov na špecifickom

grafickom akcelerátore. OpenGL rozšírenia dovoľujú vývojárom prispôsobiť program individuálne podľa daného hardvéru.

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. V tabuľke 5.1 je porovnanie 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 štandardnú 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 funkcionalitu (predpokladanú). Ale 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.

Plusy a mínusy	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 šíriteľný štandard	Áno	Nie (vlastníctvo Microsoftu)
Rozšíriteľnosť	Áno (OpenGL rozšírenia)	Nie

tabuľka 5.1: Porovnanie medzi OpenGL a Direct3D

5.2.1 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*) [17,41]. 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 grafických kartách 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 podtržníkom. 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.

5.2.2 Cg – C for graphics

Cg je programovací jazyk vyvinutý firmou n*Vidia Corporation* v spolupráci s firmou Microsoft [7, 35]. Cg prekladač využíva OpenGL rozšírenia, ktoré podporujú programovanie grafických kariet v ich vlastnom asembleri, čím odpadá nutnosť ovládať programovanie v jazyku nízkej úrovne. Volania príkazov jazyka Cg nastavujú potrebné parametre pre funkcie OpenGL a využívajú rozšírenia, ktoré definujú asembler grafických kariet. Cg využíva syntax z programovacieho jazyka C a je kompatibilné s OpenGL API a Microsoftovým *High-Level Shader Language (HLSL)* pre DirectX 9.0.

Cg program pracuje s vrcholmi a fragmentmi, ktoré vchádzajú do programu a vychádzajú po určitých transformáciách [7]. 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, je vykonávaný vo vertex shader-i, a pracuje na vrcholoch. *Fragment program*, ktorý pracuje v rasterizačnej jednotke na fragmentoch a vykonáva ho fragment shader. Cg prekladač preloží program do assembleru (sady inštrukcií) definovaného v OpenGL rozšíreniach. Prekladač podporuje viacero rozšírení, ktoré špecifikujú rôzne asemblerové inštrukcie. Aj keď je Cg výtvor firmy nVidia, podporuje aj multivendor rozšírenia definujúce asembler grafických kariet, preto by mali programy založené na týchto rozšíreniach fungovať aj na grafických kartách iných firiem.

6 Objemové zobrazovanie s použitím grafického hardvéru

Pre priame objemové zobrazovanie bol navrhnutý špeciálny hardvér, ktorý je žiaľ použiteľný len pre túto oblasť (*VolumePro*,[22]). 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 512x512x512 a viac, menšie dáta sú väčšinou využívané len ako testovacie vzorky. Dáta z počítačovej tomografie mávajú rozlíšenie rezov 512x512 a počet rezov často dosahuje počet 2000 kusov. Veľkosť dát zodpovedá za 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.

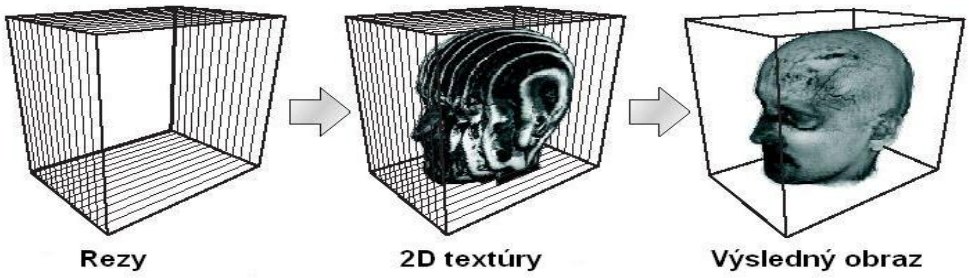
6.1 Objemové zobrazovanie pomocou textúr

Grafické karty sú navrhnuté pre prácu s textúrami, ktoré sa dajú využiť pre objemové zobrazovanie. V tejto časti práce budú popísané algoritmy, ktoré využívajú 2D a 3D textúry pre zobrazovanie.

6.1.1 Objemové zobrazovanie pomocou 2D textúr

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

Objem je rozložený do sady *objektovo orientovaných rezov*, štvoruholníkov, v závislosti na aktuálnom smere pohľadu [3]. 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. Zobrazovanie sady rezov je vykonávané 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

6.1.2 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 rovnice (3.2) získame rovnicu [21,28,32]:

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

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 vstupného (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,

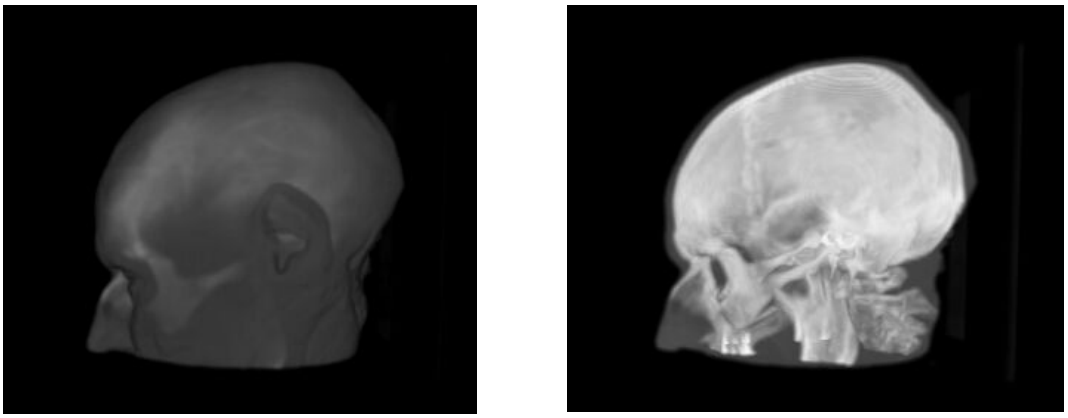
$$C_{src} = I_{emit} \quad a \quad A_{src} = (1-v_k) \qquad (6.3)$$

dosiahneme rovnicu (6.1).

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

Maximálna intenzitová projekcia

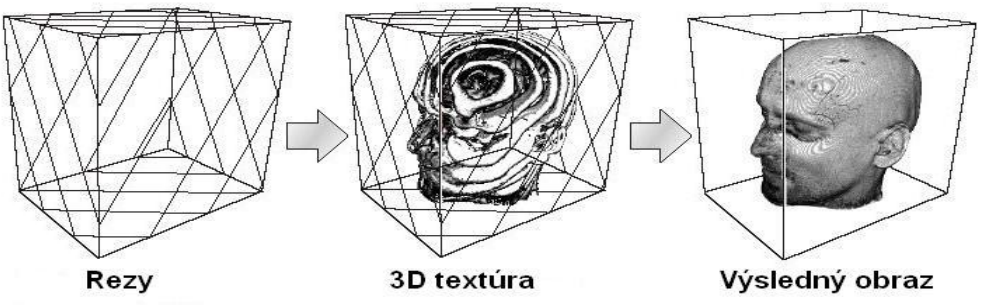
Maximálna intenzitová projekcia (MIP) je hlavne využívaná pre medicínske aplikácie, kde sa zobrazujú tomografické dáta s použitím kontrastnej látky na zvýraznenie (cievy) [34]. Pre použitie MIP potrebujeme nastaviť dodatočnú zmiešavaciu rovnicu príkazom `glBlendEquationEXT(GL_MAX_EXT)`. Táto funkcia je zahrnutá v OpenGL rozšíreniach, s názvom `GL_EXT_blend_minmax`. Na obrázku 6.2 je vidieť rôzne výsledky pre tie isté dáta.



Obrázok 6.2: použitie rôznych zmiešavacích rovníc pre skladanie rezov

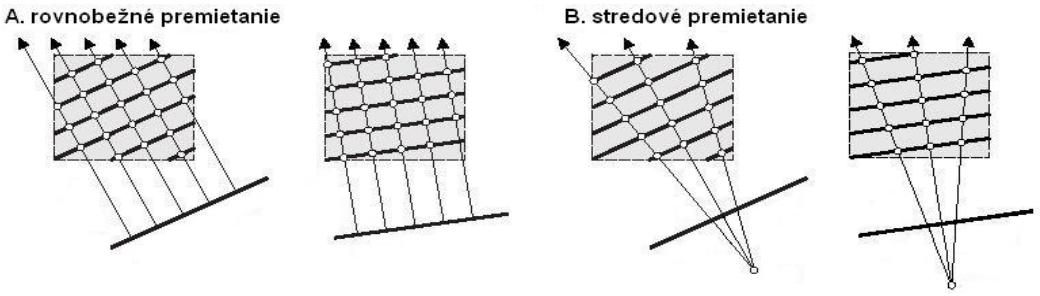
6.1.3 Objemové zobrazovanie pomocou 3D textúr

Metóda využívajúca 3D textúru [3,33] 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 ľubovoľne, tak aby bola zachovaná konštantná vzdialenosť rezov pre rôzne uhly pohľadu. Môžeme využiť rezy rovnobežné s premietacou rovinou, *pohľadovo orientované rezy* (obrázok 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 rastie 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.

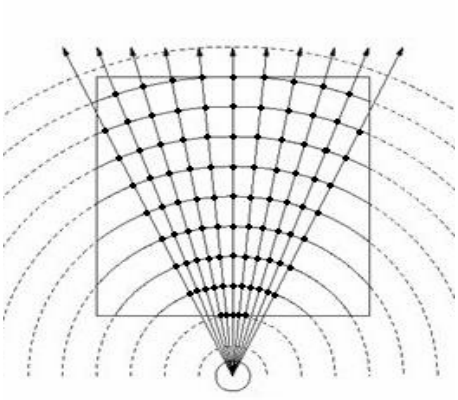


obrázok 6.4: vzorkovanie pozdĺž pohľadových lúčov pre rovnobežné (A) a stredové (B) premietanie

6.2 Metóda vrhania lúča (*ray casting*)

Metóda vrhania lúča spadá do algoritmov typu obrazovo orientovaných, ako bolo uvedené vyššie, kde pre každý pixel výsledného obrazu je vrhaný lúč, ten hľadá prienik s dátami, potom postupne prechádza dátami a akumuluje hodnoty z dát pozdĺž tohto lúča. Výsledná nakumulovaná hodnota je zobrazená na obraze. Algoritmus vrhania lúča odstraňuje problém nekonzistentného vzorkovania pre stredové premietanie (obrázok 6.5) a je to korektný algoritmus pre obe premietania.

Tento algoritmus je možné priamo implementovať ako jednoprechodový algoritmus na grafických kartách, ktoré podporujú dynamické cykly (karty podporujúce *shader model 3.0c*) [25]. Implementovať tento algoritmus bolo možné aj na starších kartách ako kartách, ktoré podporujú SM 3.0c, ale iba ako viacprechodový algoritmus [14]. Pod viacprechodovým algoritmom rozumieme algoritmus, ktorý vykonáva viacero rôznych programov, ktoré môžu, ale aj nemusia závisieť na výsledkoch predchádzajúcich programov.



obrázok 6.5: vzorkovanie pozdĺž pohľadových lúčov pre stredové premietanie

6.2.1 Implementácia

Tento algoritmus je implementovaný priamo na grafickej karte, kde sa vykresľujú iba bočné steny datasetu a ďalšie výpočty sú vykonávane priamo na GPU. Pre každý vrchol datasetu (kocky) sa do vertex programu na grafickú kartu posiela prislúchajúca textúrová súradnica a potrebné matice. Vo vertex programe sa vyráta pohľadový lúč (pozorovateľ je umiestnený v bode 0,0,0, pre stredové premietanie) pre každý vrchol a príslušné textúrové súradnice, ktoré sú ďalej spracovávané vo fragment programe.

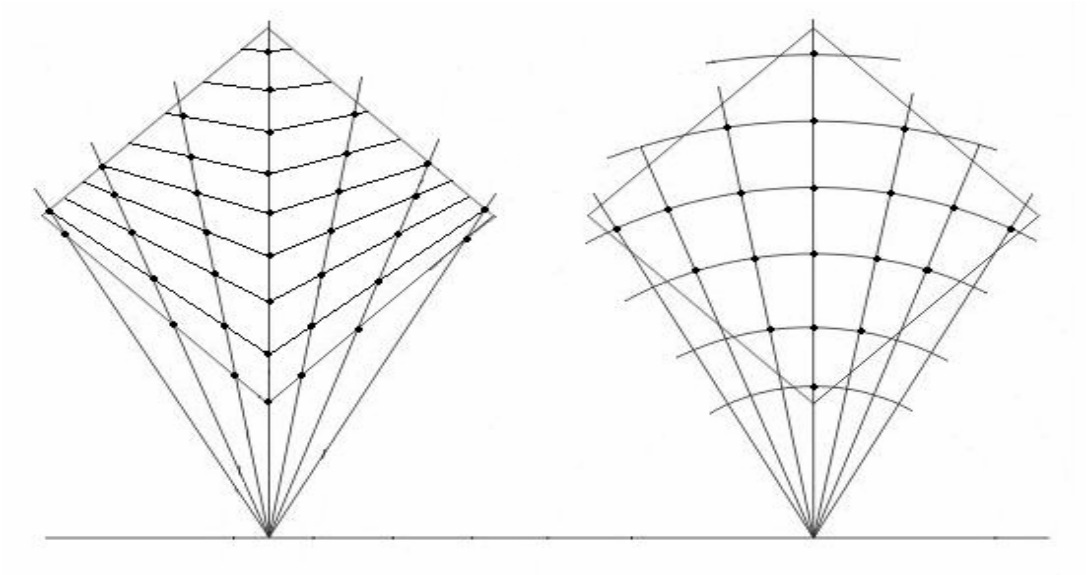
```
struct input_data{
    float4 position     : POSITION;
    float4 texCoord0    : TEXCOORD0;
};

struct output_data{
    float4 position      : POSITION;
    float4 texCoord0  : TEXCOORD0;
    float4 dirVect    : TEXCOORD1;
};

output_data main(input_data IN, uniform float4x4 matrixModelProj,
                 uniform float4x4 matrixInvertModel,
                 uniform float4x4 matrixTexture )
{
  output_data OUT;
  float4 eyeToVert = IN.position-mul(matrixInvertModel,float4(0,0,0,1));
  OUT.dirVect      =  mul( matrixTexture, eyeToVert );
  OUT.position     =  mul( matrixModelProj, IN.position );
  OUT.texCoord0    =  mul( matrixTexture, IN.texCoord );
  return OUT;
}
```

Ďalej vo fragment programe prebieha už výpočet algoritmu vrhania lúča. Tento fragment program sa skladá z dvoch častí. V prvej sa vyráta počiatočná poloha lúča a v druhej sa už v cykle prechádza cez objem a akumulujú výsledné hodnoty. Grafická karta vygeneruje pre každý fragment textúrovú súradnicu, ktorá leží na hranici datasetu. Táto súradnica slúži ako počiatočný bod pre lúč, ktorý prechádza cez dáta. Počiatočnú polohu lúča treba posunúť

v smere pohľadového lúča k najbližšej „pologule“, aby sa dosiahol korektný algoritmus (obrázok 6.6), kde počet pologúľ (počet vzoriek) závisí na vstupnej hodnote slice_dist. Tento počiatok lúča vstupuje do cyklu, kde sa načíta hodnota z 3D textúry. Podľa tejto hodnoty (intenzity) sa načíta farebná hodnota z palety (2D textúra), ktorá sa podľa svetelných rovníc zmiešava s predošlými hodnotami a prispieva tak k výslednej hodnote pixela. Tento cyklus je ukončený, keď lúč opustí daný objemu, alebo ak naakumulovaná hodnota priehľadnosti dosiahne určitú úroveň a ďalšie vzorky by už nezmenili aktuálnu hodnotu, prípadne len nepatrne.

```
struct input_data {
    float3 TexCoord0    : TEXCOORD0;
    float3 dirVect      : TEXCOORD1;
};

struct output_data {
      float4 color      : COLOR;
};

output_data main(input_data IN, uniform sampler3D volume,
                  uniform sampler2D palette, uniform float slice_dist )
{
  output_data OUT;
  float3 ray_step = normalize(IN.dirVect)*slice_dist;
  float lastHSphere = floor(length(IN.dirVect) / slice_dist)*slice_dist;

  float addDist    = slice_dist - (length(IN.dirVect) - lastHSphere);
  float3 shiftVect = normalize(IN.dirVect)*addDist;
  float3 tex_pos   = IN.TexCoord0 + shiftVect;


  float4 res  = float4(0,0,0,0);
  float alpha = 1.0;
  float eps   = 0.0001;

  while (alpha > 0.005 && tex_pos.x > -eps && tex_pos.x < 1.0 + eps &&
      tex_pos.y > -  eps && tex_pos.y < 1 + eps && tex_pos.z > -eps &&
      tex_pos.z < 1.0 + eps)
  {
    float4 col = tex3D(volume, tex_pos);
```

```
    tex_pos+=ray_step;
    res+=col*col.a*alpha;
    alpha*=(1-col.a);
  }

  OUT.color = res;
  OUT.color.a =1- alpha;
  return OUT;
}
```



Obrázok 6.6: rozdiel pri raycastingu bez korektúry (vľavo) začiatku pohľadového lúča a s korektúrou (vpravo).

6.3 Zhrnutie

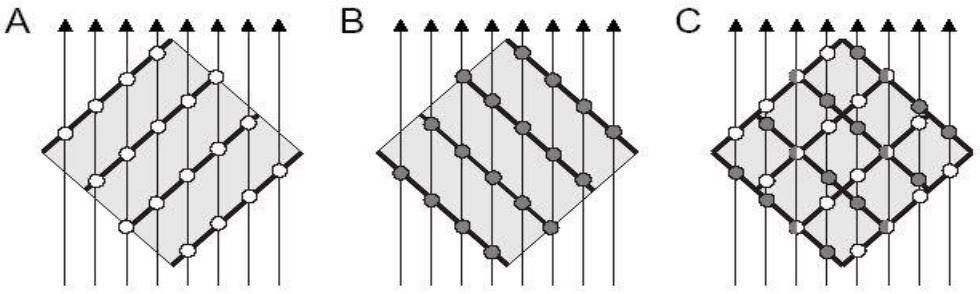
V tejto časti zhrnieme hlavné výhody a nevýhody každého prístupu objemového zobrazovania popísaného vyššie.

Tieto algoritmy sú založené na podpore grafických akcelerátorov, preto pre zobrazované dáta vyplývajú obmedzenia, ktoré požadujú jednotlivé zariadenia. Jedným z obmedzení je veľkosť textúry. V súčasnosti podporujú grafické karty so 128MB pamäťou a väčšou rozmer 3D textúry 512x512x512, 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 (*bricking*) [23]. 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 PCI Express zbernice umožňujú prenos teoreticky rýchlosťou až 8GB/s. Tieto problémy ale v tejto práci nebudú rozoberané.

6.3.1 2D textúry

Hlavná výhoda tohto 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 [23]. 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 tohto algoritmu.



obrázok 6.7: vzorkovacie artefakty zapríčinené zmenou sady rezov

6.3.2 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 [16], alebo pomocou metódy vrhania lúča. Výhody a nevýhody tohto prístupu sú zhrnuté v tabuľke 6.2. Aplikovateľnosť tohto 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.

6.3.3 Metóda vrhania lúča

Táto metóda využíva 3D textúru v ktorej sú uložené zobrazované dáta. Odstraňuje nevýhody vyplývajúce zo stredového premietania. Keďže táto metóda využíva 3D textúry, tak je obmedzená veľkosťou textúry. Tento algoritmus je vhodný na rôzne optimalizačné techniky, ako je preskakovanie prázdnych miest [14], adaptívne vzorkovanie, generovanie isoplôch, prípadne sa dá doplniť o lámanie a odraz lúča [25]. Možnosťou pre toto použitie sú aj rôzne osvetľovacie modely a prenosové funkcie, ktoré budú popísané nižšie. Zhodnotenie tejto metódy je možné vidieť v tabuľke 6.3.

2D textúry	
Plusy	mínusy
vysoký výkon veľká dostupnosť	vysoké pamäťové požiadavky iba bi-lineárna 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

Vrhanie lúča	
plusy	mínusy
trilineárna interpolácia korektný pre obe premietania modulárny	rozmerové obmedzenie menší výkon a ako 2D/3D textúry podpora shader modelu 3.0c

tabuľka 6 .3: výhody a nevýhody metódy vrhania lúča

7. Prenosové funkcie

Pri skladaní výslednej hodnoty pixela pozdĺž pohľadového lúča vo výslednom obraze je vhodné špecifikovať koeficienty vyžarovania a pohlcovania energie. 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

$$I_{emit} = T_{emit}(v) \qquad \text{a} \qquad v_k = T_{abs}(v), \tag{7.1}$$

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.

V tejto práci sú uvedené techniky prenosových funkcií, ktoré sú založené len na jednom parametri, jednorozmerné prenosové funkcie, a to hustote (intenzite) dát. Existujú techniky, ktoré pre základ prenosových funkcií používajú viaceré parametre, napríklad veľkosť gradientu, hlavné krivosti, druhé derivácie a iné, tieto funkcie sú označované ako viacrozmerné prenosové funkcie, alebo dátovo zamerané (*data centric*) [11, 13]. Medzi viacrozmerné prenosové funkcie sa radí aj použitie LH Histogramu [27]. Ďalšími metódami na generovanie prenosových funkcií sú algoritmy označované ako obrazovo zamerané (*image centric*) [20] .

7.1 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 [23]. Obe metódy vedú k viditeľne rozdielnym výsledkom.

Pred klasifikácia označuje aplikovanie prenosovej funkcie na diskrétne vzorky bodov pred interpoláciou. Rekonštrukcia spojitého signálu je vykonávaná na hodnotách s už priradeným vyžarovaním a pohlcovaním, pričom sa nezachovávajú vysoké frekvencie z prenosových funkcií

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. Pre zachovanie vysokých frekvencií v prenosových funkciách, je treba zobrazovať veľa rezov (Nyquistovo kritérium)

Vyššie frekvencie v prenosových funkciách zvyšujú potrebu väčšieho počtu vzorkovania. Pred klasifikácia neprenesie vysoké frekvencie z prenosovej funkcie do výsledného renderovania, po klasifikácia zachováva vysoké frekvencie, ale iba na danom reze. Na reprodukciu vysokých frekvencií, medzi rezmi, treba renderovať viacej rezov, čo vedie k zvýšenému času potrebnému na zobrazovanie.

Pred integrovaná klasifikácia, je metóda klasifikácie, ktorá odstraňuje problém vysokých frekvencií v prenosových funkciách [6]. Táto metóda simuluje renderovanie bloku po bloku a nie rez po reze, kde pod blokom rozumieme objem medzi dvoma rezmi. Pri použití tejto techniky sa docielia kvalitatívne podobné výsledky ako pri po klasifikácii, ale s menším počtom rezov (blokov), čo vedie k rýchlejšiemu zobrazovaniu.

Generovanie palety potrebnej pre pred integrovanú klasifikáciu je časovo náročnejší proces ako generovanie paliet pre predošlé metódy. Predošlé metódy si vystačia s jednorozmernou paletou zloženou s RGBA hodnôt, týmto RGBA hodnotám priamo

prislúchajú jednotlivé hodnoty prenosných funkcií pre daný index. Paleta pre pred integrovanú klasifikáciu je dvojrozmerná a symetrická podľa diagonály. Konkrétne generovanie potrebnej palety bude uvedené neskôr.

7.2 Implementácia

V tejto časti ukážeme ako implementovať prenosové funkcie na grafickom hardvéri využívajúc techniky textúrového mapovania vysvetlené vyššie. Sú tu popísané techniky pred klasifikácie, po klasifikácie a pred integrovanej klasifikácie. Tieto techniky vedú pri zmenách transfer funkcie k interaktívnym zmenám zobrazovaných dát za predpokladu dostupnosti využívaných funkcií grafickým akcelerátorom.

7.2.1 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.

7.2.1.1 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()`.

7.2.1.2 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 tohto 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. Súčasný graficky hardvér už prestáva podporovať tieto rozšírenia a nie sú ani zahrnuté v novej špecifikácii.

7.2.2 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 hardvérové možnosti, ktoré umožňujú meniť texel priamo v textúrovacej jednotke.

7.2.2.1 Farebné palety pre textúry

Priame hardvérové využitie po-klasifikácie umožňuje OpenGL rozšírenie `SGI_texture_color_table`, ktoré je ale dostupné iba na výkonných pracovných staniciach firmy Silicon Graphics (SGI) [24]. 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.

7.2.2.2 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 [5,23].

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

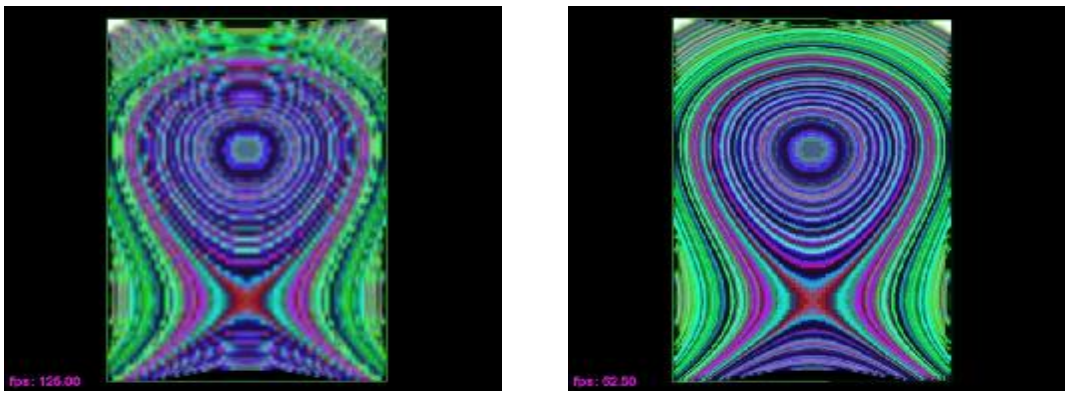
Cg na metódu po-klasifikácie pri použití rovnakého kódu pre grafické karty od rôznych výrobcov.

7.2.2.3 Po klasifikácia pomocou Cg

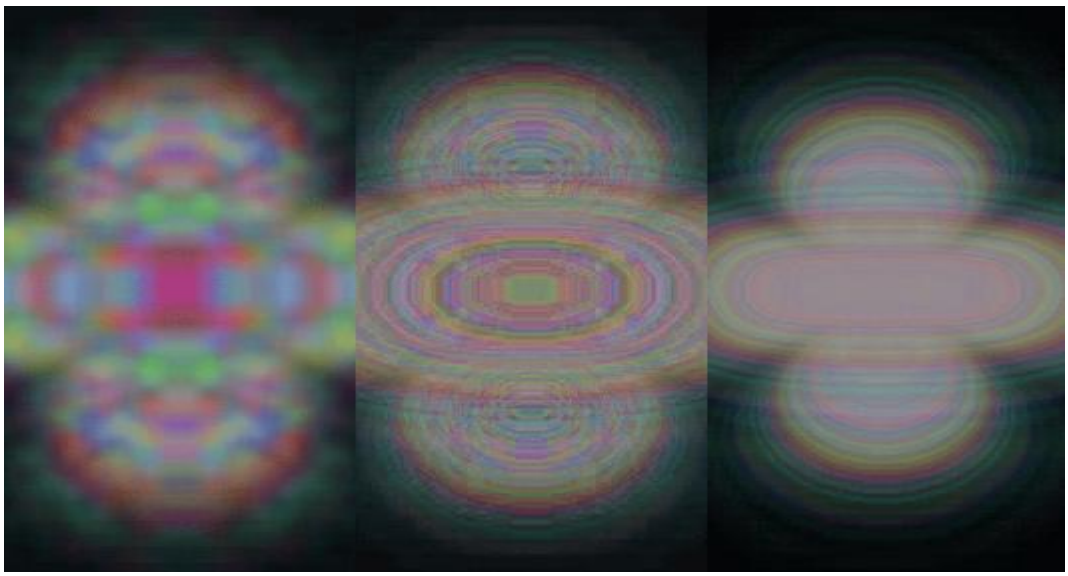
Na dosiahnutie efektu po klasifikácie využívame fragment program. Táto metóda pracuje na obdobnom princípe ako závislé textúry popísanom vyššie. Princíp algoritmu spočíva v tom, že intenzitu z textúry reprezentujúcu objemové dáta použijeme ako index, textúrové súradnice, do textúry reprezentujúcu paletu. Na výpise 7.4 je uvedený fragment program napísaný v Cg.

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

struct output_data{
      float4 color      : COLOR;
};

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)



obrázok 9.2: rôzne výsledky metód klasifikácie (zľava doprava: pred klasifikácia, po klasifikácia, pred integrovaná klasifikácia) pri aplikovaní rovnakej prenosovej funkcie

7.2.3 Pred integrovaná klasifikácia

Na implementovanie metódy pred integrovanej klasifikácie môže byť použitá metóda závislých textúr z kapitoly 9.2.2.2, alebo priamo implementovaná vo fragment programe s použitím jazyka Cg. Nižšie je uvedený fragment program jazyka Cg pre pred integrovanú klasifikáciu. Aplikovanie fragment programu je vhodnejšie aj na dodatočné výpočty, ktoré bývajú potrebné pri objemovom zobrazovaní nie len na klasifikáciu.

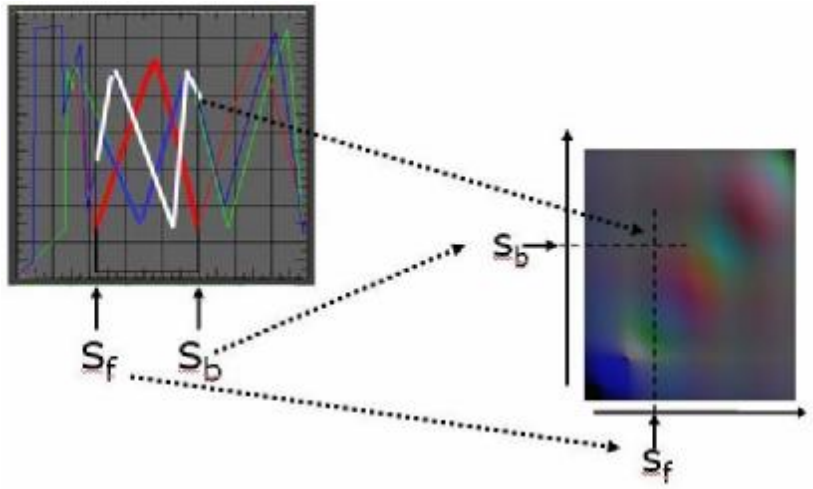
```
struct input_data {
    float3 TexCoord0 : TEXCOORD0;
    float3 TexCoord1 : TEXCOORD1;
};

struct output_data {
      float4 color      : COLOR;
};

output_data main(input_data IN,  uniform sampler3D volume,
                  uniform sampler2D palette)
{
    output_data OUT;
    float2 lookup;
    lookup.x = tex3D(volume, IN.TexCoord1.xyz).x;
    lookup.y = tex3D(volume, IN.TexCoord0.xyz).x;
    OUT.color = tex2D(palette, lookup.xy);
    return OUT;
}
```

Vstupné premenné do framgnet programu sú dve textúrové súradnice z vertex programu, kde prvá zodpovedá prednému rezu a druhá zodpovedá druhému rezu (zadnej strane bloku), tieto dve súradnice slúžia na získanie skalárnych hodnôt z datasetu (*tex3d()*). Pomocou týchto skalárnych hodnôt sa získa z palety výsledná farebná hodnota, ktorá je výsledkom daného programu.

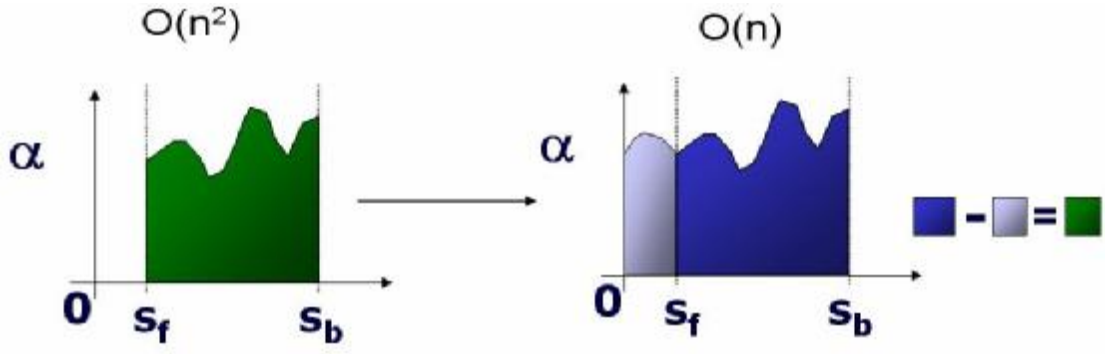
Generovanie palety pre pred integrovanú klasifikáciu

Vyčíslenie palety pre pred integrovanú klasifikáciu sa vykonáva na CPU ako predvýpočet. Pri zmene prenosovej funkcie, je nová paleta nahratá na grafickú kartu a použitá v zobrazovaní. Pri prenosovej funkcii, ktorá je definovaná ako lineárna lomená funkcia, stačí prepočítavať iba úseky, v ktorých nastala zmena. Prípadne prerátať len funkciu zodpovedajúcu danému farebnému kanálu, čím sa dá dosiahnuť isté urýchlenie. V rámci predspracovania sa generuje paleta, ktorá obsahuje všetky možné kombinácie integrovania s danými prenosovými funkciami. Pre názornosť na obrázku 7.3 je možné vidieť schému generovania palety. Priamočiare generovanie palety vedie k časovej

zložitosti $O(n^2)$ kde n je počet možných hustôt v datasete (najčastejšie 256, alebo 4096 hodnôt). Pri urýchlení generovania palety sa dá dosiahnuť časová zložitosť O(n), pomocou integrálnych funkcií (obrázok 7.4).



Obrátok 7.3: schéma generovania palety pre metódu pred integrovanej klasifikácie



Obrázok 7.4: využitie integrálnych funkcií pri generovaní palety pre metódu pred integrovanej klasifikácie

7.3 Zhodnotenie

Spomínané metódy aplikovania prenosovej funkcie sú vykonateľné interaktívne pri definovaní prenosovej funkcie s využitím grafického zariadenia. Pre jednotlivé metódy dostávame viditeľne rozdielne výsledky (obrázok 7.1 a 7.2) a to najmä pri veľkých zmenách prenosovej funkcie (vysoké frekvencie). Metóda po klasifikácie a metóda pred integrovanej klasifikácie sú v mapovaní prenosovej funkcie na rezy korektné. Pre prenosové funkcie s veľkou frekvenciou, so zachovaním dostatočnej kvality, vedie použitie

metódy po klasifikácie k veľkému počtu vzoriek, rezov (Nyquistove kritérium). Metóda pred integrovanej klasifikácie odstraňuje nutnosť použiť veľký počet rezov, blokov, pre prenosové funkcie s veľkými frekvenciami.

8 Fokus a kontext techniky

Objemové zobrazovanie zobrazuje celý dataset s rovnakými vlastnosťami pre rôzne smery pohľadu, čo je niekedy nedostačujúce. Používatelia sa pri niektorých aplikáciách potrebujú zamerať na významné miesta v dátach, ktoré sú potrebné pre ich výskum a potreby. Tieto algoritmy, ktoré umožňujú zvýrazniť významné oblasti v dátach, prípadne sa zamerať na dané miesto a ostatné potlačiť, spadajú do oblasti fokus a kontext techník. Tieto požiadavky na zvýraznenie istých oblastí v dátach je možné rozdeliť do určitých oblastí, kde každá oblasť pristupuje k tomuto problému inak.

Prenosové funkcie sú jednou z možností ako dosiahnuť potlačenie istých častí dát a zvýrazniť iné. Jednorozmerné prenosové funkcie sú rozširované o ďalšie dimenzie, aby sa dosiahla čo najpresnejšia špecifikácia zobrazenia potrebných častí [11, 13]. Problém prenosových funkcií je ten, že nie je jednoduché ich správne nadefinovať a aplikujú sa na celý objem.

Ďalšou oblasťou je **ilustratívne objemové zobrazovanie**, kde prvým prístupom bolo modifikovať priehľadnosť v závislosti na veľkosti gradientu, čím sa docieli zvýraznenie hraničných oblastí a potlačenie homogénnych oblastí [18]. Ďalšie prístupy sú prístupy, ktoré zvýrazňujú kontúry v dátach v závislosti od veľkosti gradientu a uhlom medzi smerom pohľadu a gradientom [4]. Medzi tieto metódy patrí aj dvoj úrovňové objemové zobrazovanie, kde sa používajú kombinácie rôznych zobrazovacích techník (MIP, zobrazovanie povrchov,…) na rôzne časti dát [10]. Tieto techniky sa nazývajú aj nefotorealistické objemové zobrazovanie (*non-photorealistic volume rendering*)

Orezávanie objemu rôzne definovanými orezávacími rovinami alebo inou geometriou spadá do ďalšej oblasti. Táto oblasť sa snaží dosiahnuť zobrazenie významných častí tým, že odreže menej potrebné časti, ktoré bránia vo výhľade na významné časti [26,30,31]. Medzi tieto techniky patrí aj zvyšovanie priehľadnosti dát, ktoré sa nachádzajú v oblasti medzi pozorovateľom a zameriavanou oblasťou. Oblasti v zameraní (fókus) sa zobrazujú so štandardnou priehľadnosťou (kontext)[2, 30].

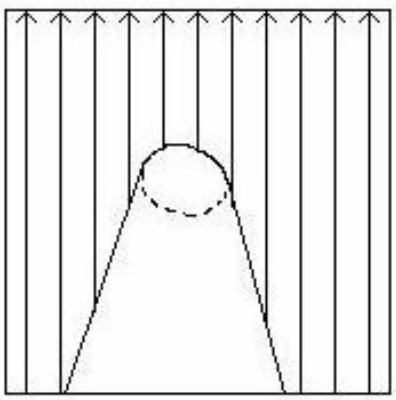
Niektoré z týchto techník pod oblasťou, ktorá je zobrazovaná ako významná zobrazujú dáta, ktoré poznajú zo segmentácie alebo iného procesu. Iné sa zameriavajú na oblasti, ktoré sa nachádzajú v určitom rozmedzí (pred pozorovateľom). Samozrejme, techniky, ktoré používajú segmentované dáta, umožňujú zobraziť priamo tieto dáta, čo

vedie k ich úplnému zobrazeniu. Toto zobrazenie ale stráca informáciu o umiestnení významných dát v ostatných dátach, stratil sa ich okolitý kontext.

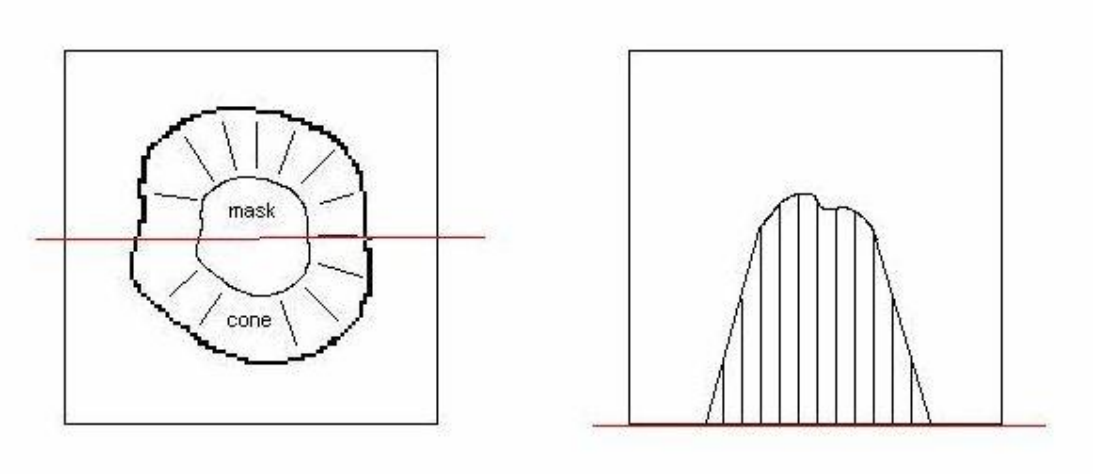
V ďalšej časti je navrhnutý a popísaný algoritmus ako dosiahnuť fokus a kontext techniku, ktorá spadá do oblasti orezávania objemu, pričom sa v zobrazovaní využívajú segmentované dáta. Uvedený návrh pracuje pre kolmé premietanie a využíva grafické akcelerátory na pomocné výpočty a aj na výsledné zobrazenie dát.

8.1 Základný návrh

Tento algoritmus spadá do algoritmov orezávajúcich objem, preto je potrebné vygenerovať nejakú geometrickú štruktúru, ktorá by bola použitá v ďalšom spracovaní na orezanie. V implementácii algoritmu sú využité možnosti grafického hardvéru a jeden z popísaných algoritmov v kapitole 6 na zobrazovanie objemových dát. Je to algoritmus vrhania lúča pre jeho vhodné možnosti rozšírenia pre potreby fokus a kontext algoritmu. Keďže algoritmus vrhania lúča nepracuje s početnou geometriou (hraničné strany datasetu), tak je snaha obísť generovanie dodatočnej geometrickej reprezentácie. V algoritme vrhania lúča sa dajú prispôsobiť štartovacie pozície jednotlivých lúčov pre potreby zobrazovania. Hlavná myšlienka popisovaného algoritmu je založená na posunutí štartovacích pozícii lúčov. Posunutie lúčov závisí na segmentovaných dátach, ktoré tvoria oblasť záujmu. Pri posunutí sa docieli zobrazenie významnej čsti a to tak, že štartovacie lúče sú posunuté do oblastí, kde nebránia viditeľnosti významnej časti. Toto posunutie lúčov vytvorí v dátach výrez, v ktorom je významná oblasť (obrázok 8.1). Tento výrez je tvorený odtlačkom segmentovaných dát (maska), kde pod odtlačkom rozumieme premietnutie masky do premietacej roviny pozdĺž pohľadového vektora (obrázok 8.2). Z tohto odtlačku postupným približovaním k pozorovateľovi a rozširovaním tohto odtlačku dostaneme pomyselný „výrezový kužeľ“, kde jeho vrchná podstava je tvorená obrysom daného odtlačku. Na reprezentáciu tohoto „kužeľa“ je použitá výšková mapa. Vo výškovej mape je uložená hodnota okoľko sa má posunúť štartovacia pozícia vrhaného lúča v jeho smere.



Obrázok 8.1: Zmena štartovacích pozícií vrhaných lúčov.



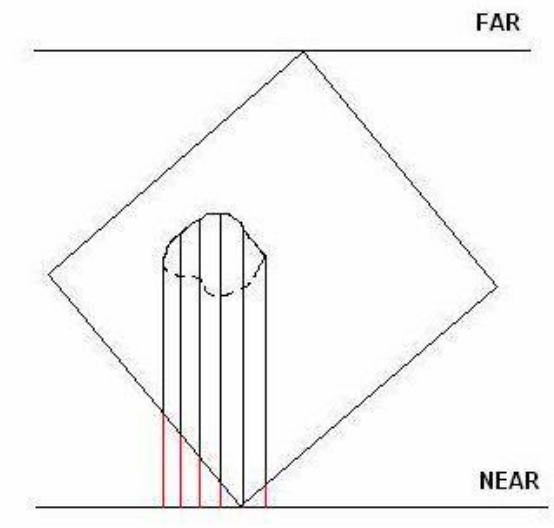
Obrázok 8.2: Výšková mapa reprezentujúca výrezový kužeľ (vľavo) a zobrazenie hodnôt z výškovej mapy (vpravo) na posun lúčov

8.2 Implementácia

Táto hlavná myšlienka je rozdelená do troch nadväzujúcich častí, ktoré vedú k požadovanému zobrazovaniu. Prvá časť sa zaoberá nájdením odtlačku masky – výškovej mapy. Nasledujúca časť rieši problém „rozširovania" výškovej mapy, aby bola dosiahnutá reprezentácia výrezového kužeľa. Poslednou časťou je výsledné zobrazovanie. Každá časť je popísaná v samostatnej pod kapitole a je pri nej navrhnutá aj implementácia na grafickej karte.

8.2.1 Hľadanie odtlačku masky

Pri hľadaní odtlačku masky je nepostačujúce nájsť iba jeho binárnu reprezentáciu, ale aj vzdialenosti od premietacej roviny, pre generovanie výškovej mapy. Premietacia rovina je kolmá na vektor pohľadu a prechádza cez najbližší bod datasetu (vo vzťahu k pozorovateľovi). Na získanie tohto odtlačku a jeho hodnôt použijeme rozšírený algoritmus vrhania lúča. Pre každý lúč a každý jeho krok sa testuje, či narazil na masku (hodnotu z dát väčšiu ako danú hraničnú hodnotu). Ak lúč narazil na masku, tak sa vyráta dĺžka tohto lúča. Pri ukončení krokovania lúča, pri opustení objemu, je známa vzdialenosť od začiatku krokovania lúča ku najvzdialenejšiemu bodu masky pozdĺž tohto lúča. Ak lúč nenarazil na masku, tak je táto vzdialenosť nulová. Táto vzdialenosť musí byť upravená ešte o vzdialenosť medzi premietacou rovinou a štartovacím bodom lúča na hranici datasetu (obrázok 8.3), aby sme predišli nekorektnej filtrácii popísanej v nasledujúcej časti.



Obrázok 8.3: Výsledná vzdialenosť uložená vo výškovej mape zložená z veľkosti vrhaného lúča (čierna) a zo vzdialenosti medzi priemetňou a hranicou datasetu (červená).

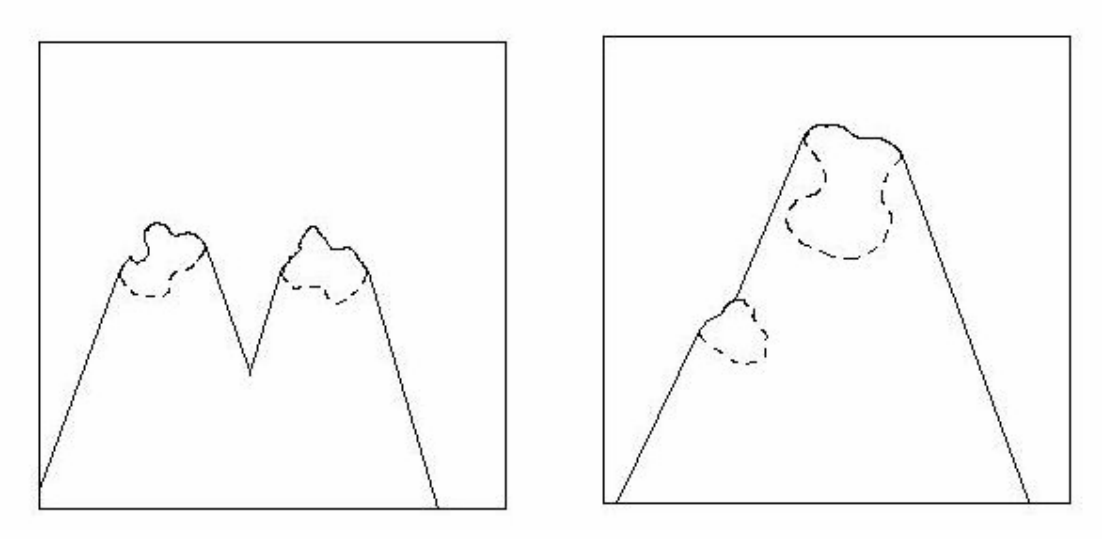
Fragment program pre hľadanie odtlačku masky:

```
struct input_data{
  float3 TexCoord0 : TEXCOORD0;
  float3 dirVect   : TEXCOORD1;
  float3 fragPos   : TEXCOORD2;
  float3 ws_dir    : TEXCOORD4;
  float  distance  : TEXCOORD5;
};
struct output_data{
  float4 color    : COLOR;
};
output_data main(input_data IN, uniform sampler3D mask,
                              uniform float slice_dist )
{
  output_data OUT;

  //korektúra štartovacej pozície lúča na nasledujúcu "pologuľu"
  float lastHSphere=floor(length(IN.dirVect) / slice_dist) * slice_dist;
  float addDist = slice_dist - (length(IN.dirVect) - lastHSphere);
  float3 shiftVect = normalize(IN.dirVect)*addDist;
  float3 tex_pos=IN.TexCoord0 + shiftVect;
  float3 tex_ray_step=normalize(IN.dirVect)*slice_dist;
  //pozícia a krokovací vektor vo svetových súradniciach
  float3 ws_ray_step=normalize(IN.cs_dir)*slice_dist;
  float3 ws_pos=IN.fragPos;

  float eps = 0.1;
  float result = 0.0;
  bool isHit = 0;

  //metóda vrhania lúča
  while (tex_pos.x > -eps && tex_pos.x < 1.0 + eps && tex_pos.y > -eps
     && tex_pos.y < 1 + eps && tex_pos.z > -eps && tex_pos.z < 1.0 + eps)
  {
    float4 col=tex3D(mask, tex_pos);
    if(col.a >= (0.0 + eps) ){
      result = length( ws_pos - IN.fragPos);
      isHit = 1;
    }
    tex_pos+=tex_ray_step;
```

```
    ws_pos+=ws_ray_step;
  }

  if(isHit){
    result = result + IN.distance;
  }
  OUT.color.rgba = result;
  OUT.color.a = 1.0;
  return OUT;
}
```

V prvej časti algoritmu je prevedená korektúra textúrovej súradnice (*tex_pos*), a výpočet krokovacieho vektora (*tex_ray_step*). Tieto hodnoty slúžia na korektné krokovanie v textúre (*mask*), ktoré prebieha v textúrovom priestore. Úprava štartovacej pozície vrhaného lúča je popísaná v kapitole 6.2.1. Na výpočet vzdialenosti od hranice datasetu po koniec vrhaného lúča slúžia parametre *ws_pos* a *ws_ray_step*, ktorých výpočet prebieha vo svetových súradniciach (*world space*). Ak lúč narazil na masku, tak sa po ukončení krokovania lúča k jeho vzdialenosti priráta ešte vzdialenosť medzi premietacou rovinou a hranicou datasetu (*IN.distance*). Táto vzdialenosť vstupuje do fragment programu z vertex programu. Táto výsledná vzdialenosť je uložená do RGB kanálov výstupnej farby a nastaví Alfa kanál na 1. Takto vygenerovaná farba je zapísaná do frame bufferu odkiaľ je skopírovaná do dvojrozmernej textúry príkazom *glCopyTexSubImage2D( ).* Pri každej zmene masky v priestore (rotácia, škálovanie, posunutie) je vypočítaná hodnota minimálnej a maximálnej z-tovej súradnice ohraničenia datasetu (b*ounding box*) a podľa týchto hodnôt je nastavená najbližšia a najvzdialenejšia orezávacia rovina pre pohľadový hranol (*vieving frustrum*). Týmto sa dosiahne maximálna presnosť výpočtu vzdialeností vo fragment programe a aj ich maximálna hodnota, ktorá je menšia rovná 1. Z toho vyplýva môžnosť uložiť vzdialenosť do RGB kanálov, bez toho aby bola orezaná a tým pádom znížená presnosť. Pri generovaní odtlačkov nie sú kladené žiadne obmedzujúce požiadavky na masku, napríklad počet častí masky (nespojité množiny), alebo jej konvexnosť (obrázok 8.4). V aplikácii tohto algoritmu je obmedzenie na rozlíšenie masky a to tak, že jej rozlíšenie musí byť rovnaké ako rozlíšenie kontextového datasetu. Avšak toto obmedzenie môže byť vynechané, ale potom treba robiť korektúry aby, datasety boli správne „napasované“.

Takto vygenerovaná výšková mapa (textúra) obsahuje odtlačok masky (obrázok 8.5a), ktorý sa bude v ďalšej časti rozširovať, a tak reprezentovať výrezový kužeľ.



Obrázok 8.4: Možné usporiadania masiek a reprezentácia ich výrezových kužeľov.

8.2.2 Rozširovanie výškovej mapy

Rozširovanie výškovej mapy z daného odtlačku je najpodstatnejšia časť celého algoritmu a aj najviac časovo náročná. Získaný odtlačok bude tvoriť základ výškovej mapy, reprezentáciu výrezového kužeľa (vrchnú podstavu), z ktorého je generovaná rozšírená výšková mapa. Na toto generovanie je využitý postup filtrovania výškovej mapy, pričom pod filtrovaním sa rozumie postupné rozširovanie odtlačku a zároveň aj znižovanie hodnôt (skracovanie vzdialeností) týchto rozšírených oblastí. Tento proces filtrovania je iteratívny proces, kde sa zakaždým filtruje textúra (výšková mapa) z predošlého filtrovania. Počet iterácií a hodnota o ktorú sa znižujú vzdialenosti pri každom filtrovaní závisí od rôznych parametrov.

Počet iterácií

Na získanie počtu iterácií potrebujeme poznať veľkosť uhla (obrázok 8.5), čo je voliteľný parameter, maximálnu hodnotu v textúre a rozmery hraničného objemu (*bounding box*) masky na obrazovke (*screen space*). Pod uhlom rozumieme uhol medzi pohľadovým vektorom a vektorom definovaným priamkou pozdĺž plášta pomyselného výrezového kužľa. Na získanie maximálnej hodnoty z textúry neposkytuje OpenGL žiadnu

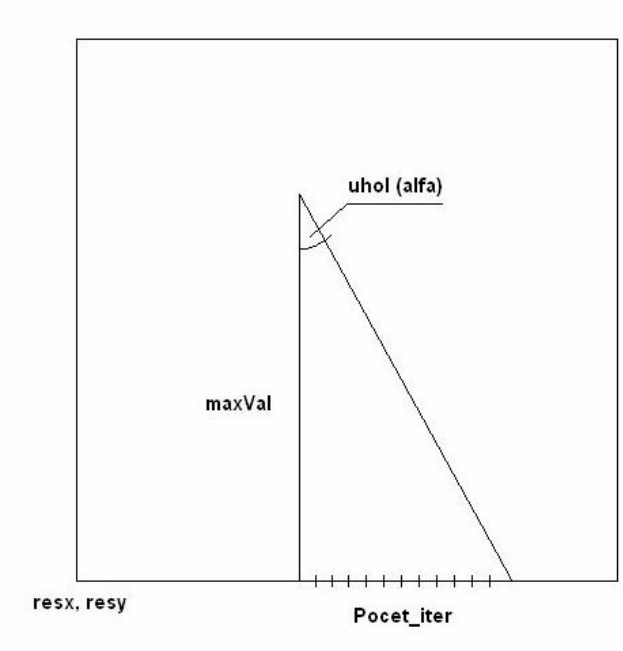
špeciálnu funkciu, a preto je potrebné skopírovať túto textúru do hlavnej pamäti počítača a tam nájsť maximálnu hodnotu. Na kopírovanie textúry do hlavnej pamäti bol použitý príkaz *glReadPixels( ).* Keď je už známa hodnota maximálnej vzdialenosti, tak je použitá na výpočet počtu iterácií. Táto hodnota je závislá na uhle a veľkosti datasetu na obrazovke. Na získanie pozícii bodov na obrazovke slúži príkaz *gluProject( ).* Zo získaných (premietnutých) bodov hraničného objemu na obrazovke je možné vyrátať veľkosti strán štvorca (*resx, resy*), v ktorom je premietnutý celý objem a ktoré budú použité v ďalšom výpočte. Vzorec na výpočet počtu iterácii je definovaný nasledovne:

$$Pocet\_iter = maxVal * \tan(alfa) * \max(resx, resy) \qquad (8.1)$$

Kde *maxVal* je maximálna hodnota z textúry, *alfa* je definovaný uhol a *resx*, *resy* sú rozmery datasetu na obrazovke. Samozrejme, úplne presný počet iterácii nie je jednoduché vypočítať a tento vzorec (8.1) slúži na horný odhad tohto výpočtu. Pokiaľ je textúra filtrovaná viac krát ako by mala byť, tak to výsledok neovplyvní, pretože prírastok na rozširovanie textúry bude nulový. Nevýhoda tohto horného odhadu je, že sa predĺži výpočtový čas. Z počtu iterácii vieme odvodiť hodnotu (dekrement) nasledovne:

$$dekrement = maxVal / Pocet\_iter \qquad (8.2)$$

O tento dekrement budú znižované hodnoty (vzdialeností) vo výškovej mape pri filtrovaní, aby sa dosiahla reprezentácia výrezového kužeľa (obrázok 8.2). Pokiaľ je počet iterácii väčší ako je maximum rozlíšení datasetu na obrazovke (veľký uhol), tak sa táto maximálna hodnota použije ako počet iterácií.



Obrázok 8.5: Znázornenie výpočtu počtu iterácií a uhla.

Filtrovanie

Filtrovanie je ďalší potrebný krok pri hľadaní výrezového kužeľa uloženého vo výškovej mape. Je vykonávané iteratívne v cykle s pevne daným počtom iterácií z predchádzajúceho výpočtu. Pri filtrovaní vstupuje do fragment programu výšková mapa s odtlačkom, na ktorej prebehne proces filtrovania, čím sa rozšíri odtlačok do svojho okolia a zníži o potrebnú hodnotu zodpovedajúcu reprezentovaniu veľkosti výrezového kužeľa. Výsledok z filtrovania je zapísaný do frame buffer-u, odkiaľ je následne skopírovaný do výškovej mapy (textúry) a následne použitý v ďalšej iterácii.

```
output_data main(input_data IN, uniform sampler2D texture,
                     uniform float decrement, uniform float resx,
                      uniform float resy)
{
 output_data OUT;
 float sqrt=1.414213562;
 float eps=0.0001;
 float2 stPoint = float2(0.0,0.0);
 stPoint =  IN.texcoord - float2( 1.0/resx, 1.0/resy );

 float4 color = tex2D(texture, IN.texcoord);
 color=max(tex2D(texture, stPoint)                    - sqrt*decrement,color);
```

```
 color=max(tex2D(texture, stPoint+float2(2.0/resx,0.0)) - sqrt*decrement, color);
 color=max(tex2D(texture, stPoint+float2(0.0,2.0/resy)) - sqrt*decrement, color);
 color=max(tex2D(texture, stPoint+float2(2.0/resx,2.0/resy)) - sqrt*decrement,
       color);
 color=max(tex2D(texture, stPoint+float2(1.0/resx,0.0))      -decrement, color);
 color=max(tex2D(texture, stPoint+float2(1.0/resx,2.0/resy)) -decrement, color);
 color=max(tex2D(texture, stPoint+float2(0.0,1.0/resy))      -decrement, color);
 color=max(tex2D(texture, stPoint+float2(2.0/resx,1.0/resy)) -decrement, color);

 OUT.color = color;
 OUT.color.a = 1.0;
 return OUT;
}
```

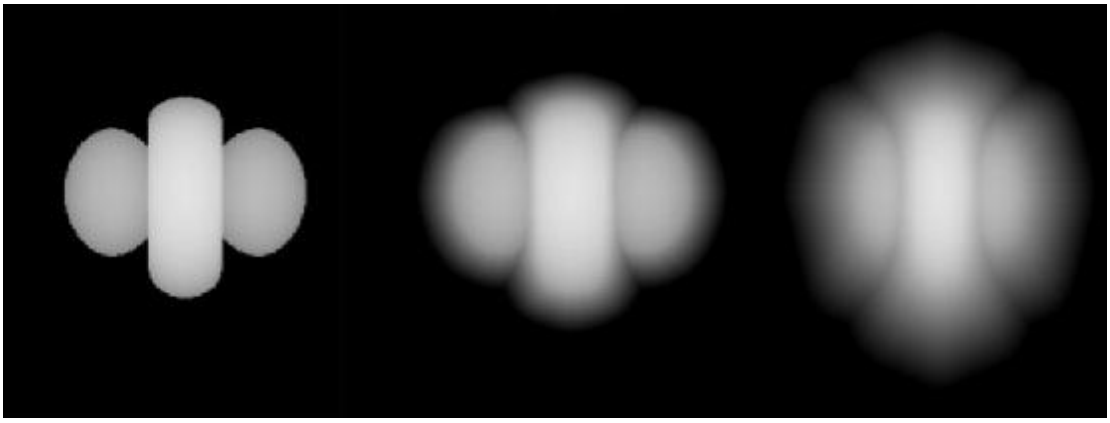
Hore uvedený fragment program, ktorý je zodpovedný za filtrovanie má vstupné parametre výškovú mapu (*texture*), dekrement (*decrement*) z výpočtu počtu iterácií a rozlíšenie výškovej mapy (*resx, resy*). Fragment program prechádza po každom fragmente a hľadá maximálnu hodnotu v jeho 3x3 okolí. Pri nájdení maximálnej hodnoty je táto hodnota znížená o dekrement v prípade, ak bola táto hodnota vpravo, vľavo, hore, dolu od daného fragmentu. Ak bola táto maximálna hodnota po diagonálach od daného fragmentu, tak je znížená o hodnotu √2*dekrement. Ak nie je nájdená hodnota väčšia ako je hodnota daného fragmentu, tak je použitá priamo táto hodnota bez zníženia. Daný fragment program uvažuje rozširovanie odtlačku rovnomerne vo všetkých smeroch, čomu zodpovedá použité jadro (8.3) pre znižovanie hodnoty fragmentu.

Dekrement *

-√2	-1	-√2
-1	0	-1
-√2	-1	-√2

(8.3)

Po skončení procesu filtrácie je vo výškovej mape uložená reprezentácia výrezového kužeľa (obrázok 8.6), ktorá bude použitá v ďalšej časti na zobrazovanie výsledného datasetu.



Obrázok 8.6: Zobrazenie výškovej mapy s reprezentáciou výrezového kužeľa pre rôzne veľkosti uhlov 0°, 15°, 30° (z ľava do prava).

8.2.3 Výsledné zobrazovanie

V tejto časti je popísané výsledné zobrazenie orezaného datasetu, pričom je použitý modifikovaný algoritmus vrhania lúča. Z predošlých častí popisovaného algoritmu je použitá vygenerovaná výšková mapa, v ktorej je uložená reprezentácia výrezového kužeľa na modifikovanie štartovacích pozícií vrhaných lúčov. Štartovacie pozície začínajú na hranici datasetu a preto nie je možné použiť priamo hodnotu z výškovej mapy, ktorá obsahuje vzdialenosti od premietacej roviny k bodu na reprezentovanom výrezovom kuželi. Túto hodnotu z textúry treba najskôr znížiť o vzdialenosť medzi premietacou rovinou a štartovacím bodom lúča pozdĺž pohľadového vektora. Je to opačný proces ako je vykonaný v časti hľadania odtlačku masky. Následne je vykonaný algoritmus vrhania lúča

```
struct input_data {
  float3 TexCoord0    : TEXCOORD0;
  float3 dirVect      : TEXCOORD1;
  float  distance     : TEXCOORD5;
};
struct output_data{
  float4 color        : COLOR;
};
output_data main(input_data IN, uniform sampler3D volume,
                 uniform sampler2D palette,
                 uniform sampler2D shiftTexture,
                 uniform float slice_dist,
```

```
                uniform float alphaThresh )
{
  output_data OUT;

  //korektúra štartovacej pozície lúča na nasledujúcu “pologuľu”
  float3 ray_step=normalize(IN.dirVect)*slice_dist;
  float lastHSphere=floor(length(IN.dirVect) / slice_dist) * slice_dist;
  float addDist = slice_dist - (length(IN.dirVect) - lastHSphere);
  float3 shiftVect = normalize(IN.dirVect)*addDist;

  //získanie potrebnej vzdialenosti z textúry s výrezovým kužeľom
  float distance=clamp( tex2D(shiftTexture,IN.TexCoord1).r -
      IN.distance, 0, 1);
  float3 tex_pos=IN.TexCoord0+shiftVect+normalize(IN.dirVect) * distance;

  float2 lookup;
  lookup.x = tex3D(volume, tex_pos).x;
  float4 res=float4(0,0,0,0);
  float alpha=1.0f;
  float eps = 0.0001;

  //algoritmus vrhania lúča
  while (alpha > 0.005 && tex_pos.x > -eps && tex_pos.x < 1.0 + eps &&
         tex_pos.y > -eps && tex_pos.y < 1 + eps && tex_pos.z > -eps &&
         tex_pos.z < 1.0 + eps)
  {
    tex_pos+=ray_step;
    lookup.y = tex3D(volume, tex_pos).x;
    float4 col=tex2D(palette, lookup.xy);
    if(col.a > alphaThresh){
      res+=col*col.a*alpha;
      alpha*=(1-col.a);
    }
    lookup.x = lookup.y;
  }

  OUT.color = res;
  OUT.color.a =1- alpha;
  return OUT;
}
```

V prvej časti fragment programu prebehne korektúra štartovacej pozície lúča (`tex_pos`), ktorá je uvedená v kapitole 6.2.1. Počas tejto korektúry sa textúrová súradnica (štartovacia pozícia) posunie pozdĺž pohľadového vektora o vzdialenosť (`distance`), ktorá zodpovedá vzdialenosti medzi hranicou datasetu a reprezentovaným výrezovým kužeľom. Vzdialenosť (`distance`) je získaná z výškovej mapy. Táto vzdialenosť je znížená o vzdialenosť medzi premietacou rovinou a hranicou datasetu, pričom môžeme dostať zápornú hodnotu, preto je ešte orezaná na interval <0,1>. Po tomto úvodnom výpočte je známa štartovacia pozícia lúča, ktorá sa nachádza na hranici reprezentovaného výrezového kužeľa a je poslaná do algoritmu vrhania lúča. Vo výpise tohto algoritmu je zakomponovaná aj pred integrovaná klasifikácia uložená v textúre `palette`. Je tu aj zakomponovaný alfa test, ktorý potláča akumulovanie hodnôt menších ako daná hraničná hodnota.

Pomocou tohto postupu sa zobrazí dataset s výrezom. Na zobrazenie masky je potrebné ju dodatočne zobraziť voliteľnou metódou objemového zobrazenia s tým , že sa nebude mazať frame buffer, ktorý obsahuje zobrazený orezaný dataset.

8.3 Výsledky

Testovanie časovej zložitosti vyššie popísaného algoritmu je znázornené a popísané v nižšie priloženej tabuľke (tabuľka 8.1). V tabuľke je možné vidieť časy výpočtu v milisekundách potrebné pre zobrazenie dát v závislosti od veľkosti výrezu (uhla). Algoritmus bol testovaný na viacerých dátach, pričom niektoré výsledky sú uvedené v tabuľke a výsledne zobrazenia na obrázku 8.7, alebo v prílohe A. Osobný počítač, ktorý slúžil na testovanie mal nasledovné parametre: Intel Pentium 4 3.2 MHz, 1GB RAM, GeForce 6600GT, operačný systém Windows XP.

	0	15	30	45	60
Harmonic 32x32x32	169ms	315ms	480ms	714ms	877ms
Tot 170x190x180	106ms	250ms	416ms	649ms	680ms
Knee 512x512x128	250ms	487ms	526ms	909ms	1.2s

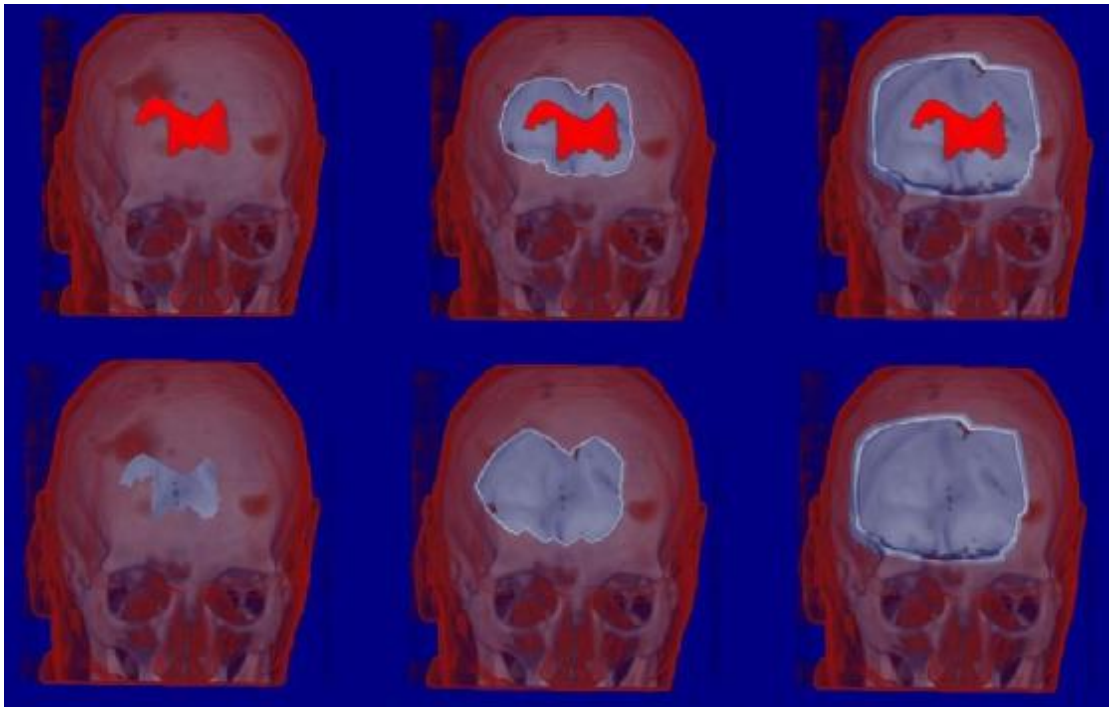
Tabuľka 8.1: prehľad časov potrebných na zobrazovanie v závislosti na veľkosti uhla.

8.4 Záver a ďalšia práca

Vyššie popísaný algoritmus zobrazuje dáta s výrezom, v ktorom je umiestnený významný objekt na ktorý sa zameriava používateľ (obrázok 8.7). Používateľ má možnosť vidieť objekt záujmu bez obmedzení, ktorý je umiestnený v originálnych dátach a tak sa nestráca informácia o priestorových usporiadaniach pozorovaného objektu. Jediným parametrom, ktorý sa dá meniť je veľkosť výrezu (uhol), ktorým je možné dosiahnuť potrebné vnímanie pozorovaného objektu v rámci dát (obrázok 8.7).

Grafické akcelerátory boli použité takmer na všetky časti výpočtu, pričom sa dá dosiahnuť interaktívne zobrazovanie.

V ďalšej práci by sme sa radi venovali optimalizácii algoritmu, v čom je pravdepodobnosť dosiahnuť zlepšenie. Hlavne sa jedná o nahradenie kopírovania dát z frame buffera do textúr priamym zápisom do textúry. Prípadne využiť *frame buffer objekt ,* ktorý by mohol urýchliť celkový čas zobrazovania. Ďalším potenciálnym urýchlením môže byť upravenie hľadania odtlačku masky, kde by sa nepostupovalo odpredu dozadu, ale naopak a pri prvom narazení na masku by sa ukončilo hľadanie vzdialenosti. Hlavnou častou by malo byť urýchlenie filtrácie, z dôvodu najpomalšej časti algoritmu. Možnosťou ako dosiahnuť vyššiu presnosť je použitie textúr s plávajúcou desatinou čiarkou. Pre lepšie vizuálne vnímanie zobrazených informácii, by bolo vhodné použiť rôzne zobrazovacie techniky pre zobrazovanie masky a datasetu, prípadne aplikovanie rôznych prenosových funkcií samostatne pre masku a samostatne pre dataset. Podstatným sa javí aj rozšírenie algoritmu pre stredové premietanie.



Obrázok 8.7: Výsledné zobrazenia dát v závislosti od veľkosti uhla 0°, 15°,30° (zľava do prava), zobrazenie s maskou (vrch) a bez masky (spodok).

9 f3dvr program

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

9.1 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 [42], ktorá je voľná, platformovo nezávislá pre tvorbu GUI. Na využívanie výkonu grafických akcelerátorov je použitá knižnica OpenGL [40]. V aplikácii sú použité Cg knižnice na priame programovanie grafických kariet [35]. Ďalšou knižnicou je knižnica f3dformat [37], ktorá slúži na jednoduchý popis objemových dát a poskytuje funkcie na načítanie a ukladanie týchto dát. Na kompresiu objemových dát slúži knižnica zlib [43]. 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 operačným systémom Windows XP.

9.2 Zobrazovacie algoritmy

Hlavnou podstatou aplikácie sú rôzne zobrazovacie algoritmy popísane vyššie, prípadne ich obmeny a doplnenie. Na jednotlivých zobrazovacích módoch je možné sledovať prípadné artefakty spomínané v predošlých častiach.

V prílohe A je možné vidieť obrázky, ktoré boli generované pomocou f3dvr aplikácie.

2D textúry - (2D Texture)

Zobrazovací algoritmus, ktorý bol popísaný v kapitole 6.1 a využíva 2D Textúry

3D textúry - (3D Texture OAS, 3D Texture VAS)

Algoritmus, ktorý bol popísaný v kapitole 6.3. Mód OAS je algoritmus založený na princípe 2D textúr, kde je objem rezaný rovnobežne so stenami objektu, ale nepoužívajú sa sady 2D textúr, ale jedna 3D textúra.

Metóda vrhania lúča – (Ray Casting)

Algoritmus vrhania lúča, ktorého základ bol popísaný v kapitole 6.4. Algoritmus implementovaný v programe je rozšírený o alfa test, ktorý musel byť implementovaný priamo vo fragment programe.

Zobrazenie gradientu – (3D Normal (texture), 3D Normal (fragment))

Tieto módy boli implementované na zobrazenie gradientov daného datasetu, ktoré sa využívajú pre módy používajúce osvetľovacie modely. Pre zobrazenie gradientu je potrebné poznať gradient (normálu) povrchu v danom bode. Tento gradient môže byť vyrátaný priamo z dát buď v rámci predspracovanie, alebo priamo za behu zobrazovania na grafickej karte. Mód texture zobrazuje gradient, ktorý je vypočítaný na CPU a uložený v 3D textúre. Mód fragment generuje gradient priamo za behu zobrazovania vo fragment programe a zobrazuje ho. Na generovanie gradientu bola použitá technika rozdielu susedných intenzít daného voxelu [28]. Tento gradient je mapovaný z intervalu <-1,1> na interval <0,1> a zobrazený ako farebné hodnoty. V nasledujúcom výpise je možné vidieť výpis programu Cg, ktorý generuje a zobrazuje gradient na GPU.

```
float3 normal;
float3 t0 = IN.texcoord1;
float3 t1 = IN.texcoord1;
t0.x-=tex_res.x;
t1.x+=tex_res.x;
normal.x = tex3D(volume,t1).r-tex3D(volume,t0).r;

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

t0 = IN.texcoord1;
```

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

OUT.color.rgb = normal*0.5+0.5;
OUT.color.a = tex3D(volume, IN.texcoord1).a;
```

Funkcia tex3D() definuje hodnotu z 3D textúry (`volume`), ktorá je určená textúrovými súradnicami (`t0 a t1`).

Tento výpočet je relatívne náročný (30 inštrukcií) oproti zobrazovaniu predvypočítaného gradientu, pretože uvedený výpočet sa vykonáva opakovane pre každý fragment. Pri použití ďalšej 3D textúry s predgenerovaným gradientom získaným v predspracovaní sa dosahuje rýchlejšie zobrazovanie avšak na úkor potrebnej pamäte.

Osvetľovacie modely (Light 2, Light (grad. mag.))

Štandardný zobrazovací proces v OpenGL využíva rozšírené Gouradovo tieňovanie pre polygóny. Avšak pre reálnejšie aplikácie je vhodné iné tieňovanie a to také, ktoré počíta osvetľovací model pre každý fragment. V Cg sa dajú veľmi jednoducho implementovať rôzne osvetľovacie modely. Mód Light 2 zobrazuje dáta založené na algoritme 3D textúr a pre každý voxel ráta zjednodušený osvetľovací model, odvodený od Phongovho osvetľovacieho modelu (rovnica 9.1) [8, 21, 32].

$$I(\lambda) = K_a(\lambda).I_a(\lambda) + K_d(\lambda).I_L(\lambda).(\mathbf{N.L}) + K_s.I_L(\lambda).(\mathbf{R.V})^k \qquad (9.1)$$

Tento osvetľovací model je možné zahrnúť aj do metódy vrhania lúča. Ďalším pridaním k tomuto módu je aj zakomponovanie prenosových funkcií do difúznej zložky a do alfa kanálu.

```
float4 color1 = tex3D(volume,IN.texcoord1);
float3 normal = tex3D(gradient,IN.texcoord1)-0.5;
float3 vecToEye   = normalize(posEye - IN.position);
float3 vecToLight = normalize(posLight - IN.position);

float3 vecHalf = normalize(vecToLight+vecToEye);
```

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

if(difuseLight <= 0) specLight = 0;

float2 lookup;
lookup.x = tex3D(volume, IN.texcoord1).x;
lookup.y = tex3D(volume, IN.texcoord2).x;

float3 difuse = tex2D(palette,lookup)*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 = tex2D(palette,tex3D(volume, IN.texcoord1).gb).a;
```

Tento mód 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.

Ďalším módom je objemové zobrazovanie s gradientovou moduláciou, kde sa používa gradient na modifikovanie priehľadnosti [18]. Veľkosti gradientu sú v predspracovaní normalizované na interval <0,1>, kde minimálna hodnota zodpovedá 0 a maximálna 1, a uložené do 3D textúry `grad_mag`.

```
float4 color1 = tex3D(volume,IN.texcoord1);
float3 normal = (tex3D(gradient,IN.texcoord1)-0.5)*2;
float3 vecToEye   = normalize(posEye - IN.position);
float3 vecToLight = normalize(posLight - IN.position);

float3 vecHalf = normalize(vecToLight+vecToEye);

float difuseLight = max(dot(normalize(normal),vecToLight),0);
float specularLight = pow(max(dot(normalize(normal),vecHalf),0), 45 );
float ambientLight = 0.5;

float shadedFactor = difuseLight + specularLight + ambientLight;

float2 lookup;
lookup.x = tex3D(volume, IN.texcoord1).x;
```

```
lookup.y = tex3D(volume, IN.texcoord2).x;

OUT.color.xyz = tex2D(palette,lookup)*shadedFactor;
OUT.color.a = tex2D(palette,color1.gb).a*tex3D(grad_mag, IN.texcoord1).a;
```

Fokus a kontext

V aplikácii je aj implementovaný algoritmus typu fokus a kontext popísaný v kapitole 8. Pre tento algoritmus je potrebne otvoriť nie len zobrazovaný dataset, ale aj masku, buď cez menu, alebo ako parameter programu (-d “meno datasetu” –m “meno masky”). Obmedzením pre masku je jej rozlíšenie, ktoré musí byť rovnaké ako rozlíšenie datasetu. V menu settings a záložke fokus and kontext je možné nastaviť veľkosť výrezového uhla a či sa má zobrazovať aj maska s datasetom, alebo len samotný dataset. V tomto nastavovacom menu je možné aj nastaviť zobrazovanie výškovej mapy.

9.3 Prenosové funkcie

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, beziérovú 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 všetky varianty klasifikácie ako sú popísané v kapitole 7 a je ich možné kombinovať takmer so všetkými zobrazovacími algoritmami.

9.4 Iné

Na skladanie farieb jednotlivých textúrových rezov je použité miešanie (blending), (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. Tieto orezávacie roviny možno nájsť v menu clipping

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.

10 Záver

Predkladaná rigorózna práca bola zameraná na prehľad rôznych techník a algoritmov pre objemové zobrazovanie s využitím grafických akcelerátorov. Techniky, ktoré boli popísané v práci umožňujú interaktívne zobrazovanie objemových dát. Poskytujú úpravu výsledných výstupných dát za účelom zlepšenia vizuálnej stránky, zvýraznenie, alebo potlačenie istých čŕt v dátach. Tieto techniky boli popísané v 3 kapitolách ktoré vedú k daným požiadavkám. Väčšina týchto techník bola implementovaná v programe, ktorý slúži na jednoduchý prehľad týchto techník. V práci bol podaný aj prehľad dnešného hardvéru, API a jazykov na programovanie grafického hardvéru. Čitateľ dostal hlavný prehľad o rôznych technikách, čo bolo cieľom tejto práce. Veľkým prínosom práce je navrhnutie nového algoritmu z oblasti fokus a kontext zobrazovania.

Literatúra

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

[2] S. Bruckner, S. Grimm, A. Kanistar, M. E. Gröller. *Illustrative Context-Preserving Volume Rendering.* In Proc. Eurographics, 2005

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

[4] B. Csébfalvi, L. Mroz, H. Hauser, A. König, and M. E. Gröller. *Fast visualization of object contours by non-photorealistic volume rendering.* In Proc. of EUROGRAPHICS, 2001.

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

[6] K. Engel, M. Kraus, T. Ertl. *High-Quality Pre-Integrated Volume Rendering Using hardware-Accelerated Pixel Shading.* Proceedings of the ACM SIGGRAPH / EUROGRAPHICS workshop on Graphics hardware , 2001

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

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

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

[10] H. Hauser, L. Mroz, G. Bischi, and M. E. Gröller. *Two-level volumerendering-fusing MIP and DVR.* In Proceedings of IEEE Visualization, 2000.

[11] J. Hladůvka, A. König, E. Gröller. *Curvature-Based Transfer Function for Direct Volume Rendering.* In Proc. SCCG, 2000

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

[13] J. Kniss, G. Kindlmann, Ch. Hansen. *Multidimensional Transfer Functions for Interactive Volume Rendering.* In Proc. IEEE Transactions on Visualization and Computer Graphics, 2002

[14] J. Krüger, R. Westermann. *Acceleration Techniques for GPU-based Volume Rendering* In Proc. IEEE Visualization, 2003

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

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

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

[18] M. Levoy. *Display of surfaces from volume data*. IEEE Computer Graphics and Applications, 1987

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

[20] 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

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

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

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

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

[25] S. Stegmaier, M. Strengert, T. Klein, T. Ertl. *A Simple and Flexible Volume rendering Framework for Graphics-Hardware-based Raycasting.* In Proc. Volume Graphics, 2005

[26] M. Straka, M. Červeňanský, A. La Cruz, A. Köchl, M. Šrámek, Eduard Gröller, and Dominik Fleischmann. *The VesselGlyph: Focus & context visualization in CT-angiography.* In Proceedings of IEEE Visualization, 2004

[27] P. Šereda, A. Vilanova Bartrolí, I.W.O. Serlie, and F.A. Gerritsen. *Visualization of Boundaries in Volumetric Data Sets Using LH Histograms.* In Proc. IEEE Transactions on Visualization and Computer Graphics, 2006

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

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

[30] I. Viola, A. Kanitsar, M. E. Gröller. *Importance-Driven Volume Rendering.* In Proc. IEEE Visualization, 2004

[31] D. Weiskopf, K. Engel, T. Ertl. *Volume clipping via per-fragment operations in texture-based volume visualization.* In Proc. IEEE Visualization, 2002

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

[33] 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

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

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

[36] The Official DirectX Website. http://msdn.microsoft.com/directx/

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

[38] The Official glsl Website. http://www.opengl.org/documentation/glsl/

[39] The Official General-Purpose Computation Using Graphics Hardware Website. http://www.gpgpu.org/

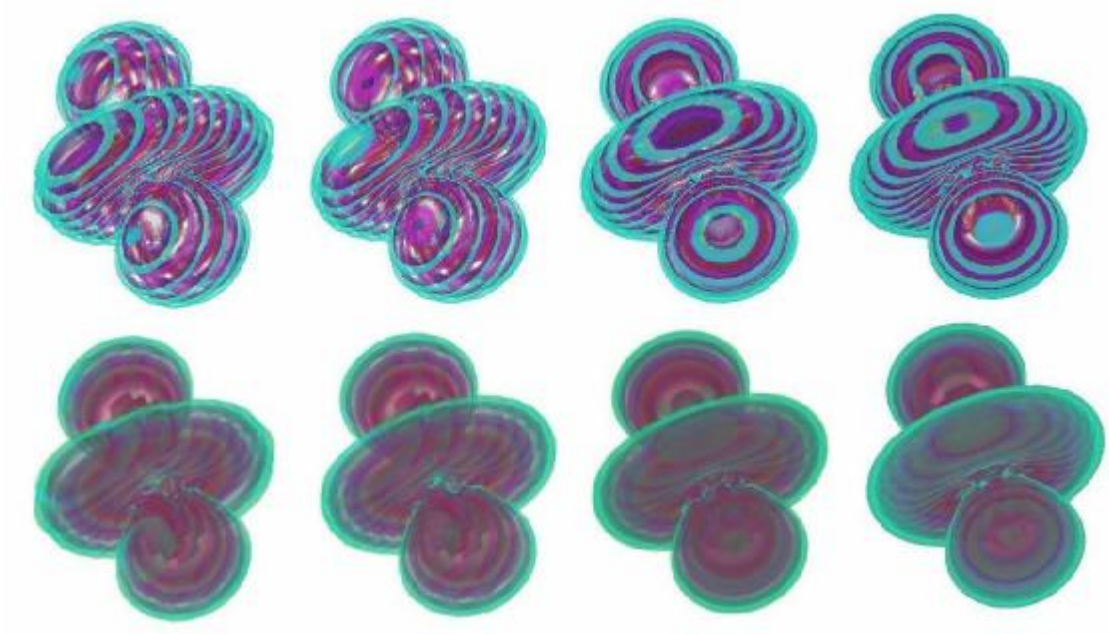
[40] The Official OpenGL Website. http://www.opengl.org

[41] The Official OpenGL extensions Website. http://oss.sgi.com/projects/ogl-sample/registry/

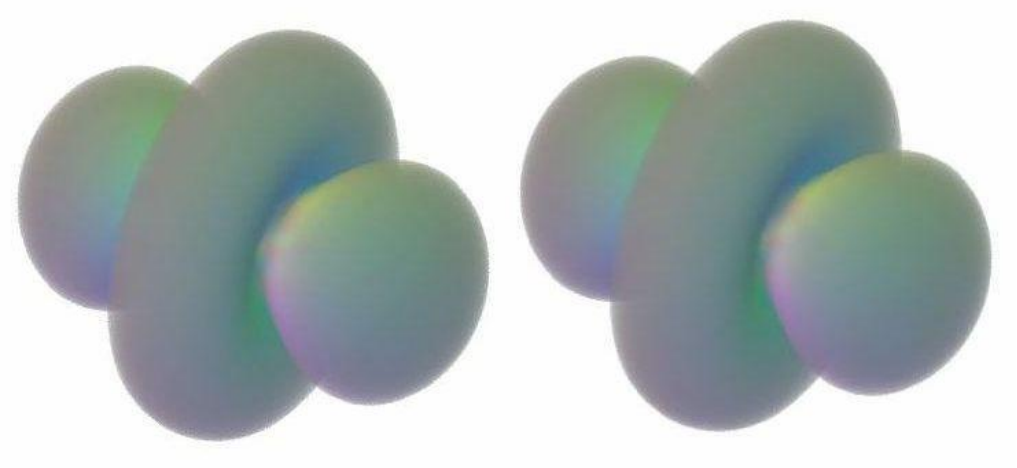
[42] The Official wxWidgets Website. http://www.wxWidgets.org

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

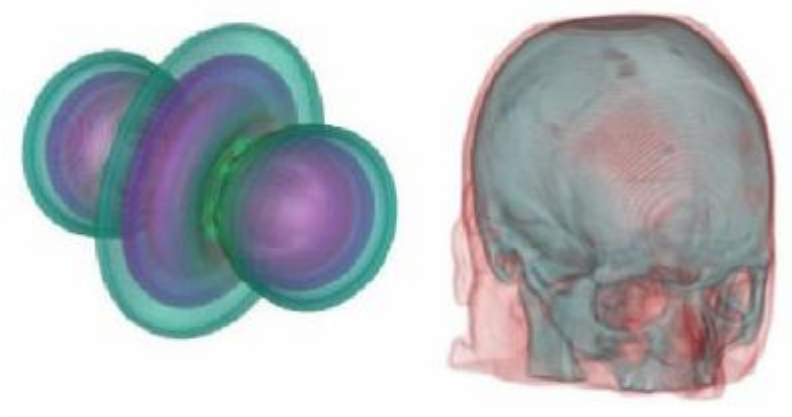
Príloha A



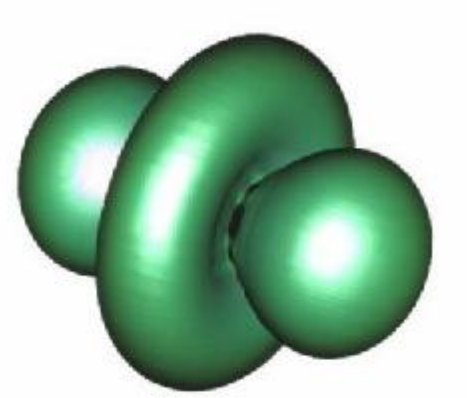
Obrázok A: Kombinácia rôznych zobrazocvacích módov a klasifikácií. Horný riadok po klasifikácia, spodný riadok pred integrovaná klasifikácia. Zobrazovacie algoritmy z ľava do prava: 2D textúra, 3D textúra OAS, 3D textúra VAS, Vrhanie lúča



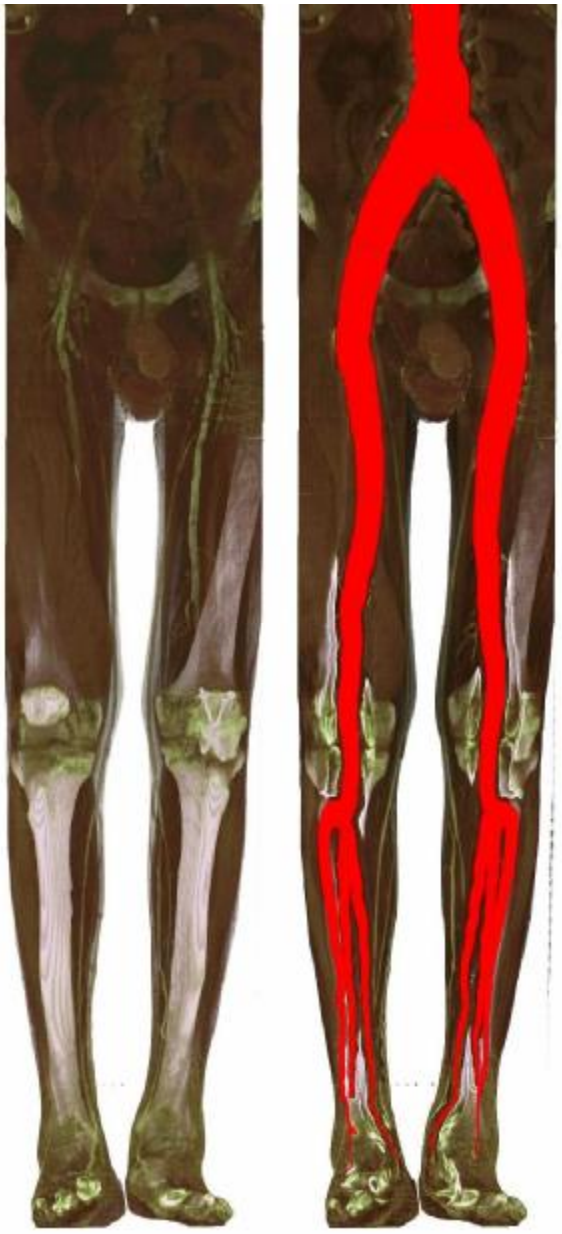
Obrázok B: Zobrazenie gradientu. Vľavo gradient predvypočítaný, vpravo gradient generovaný vo fragment programe.



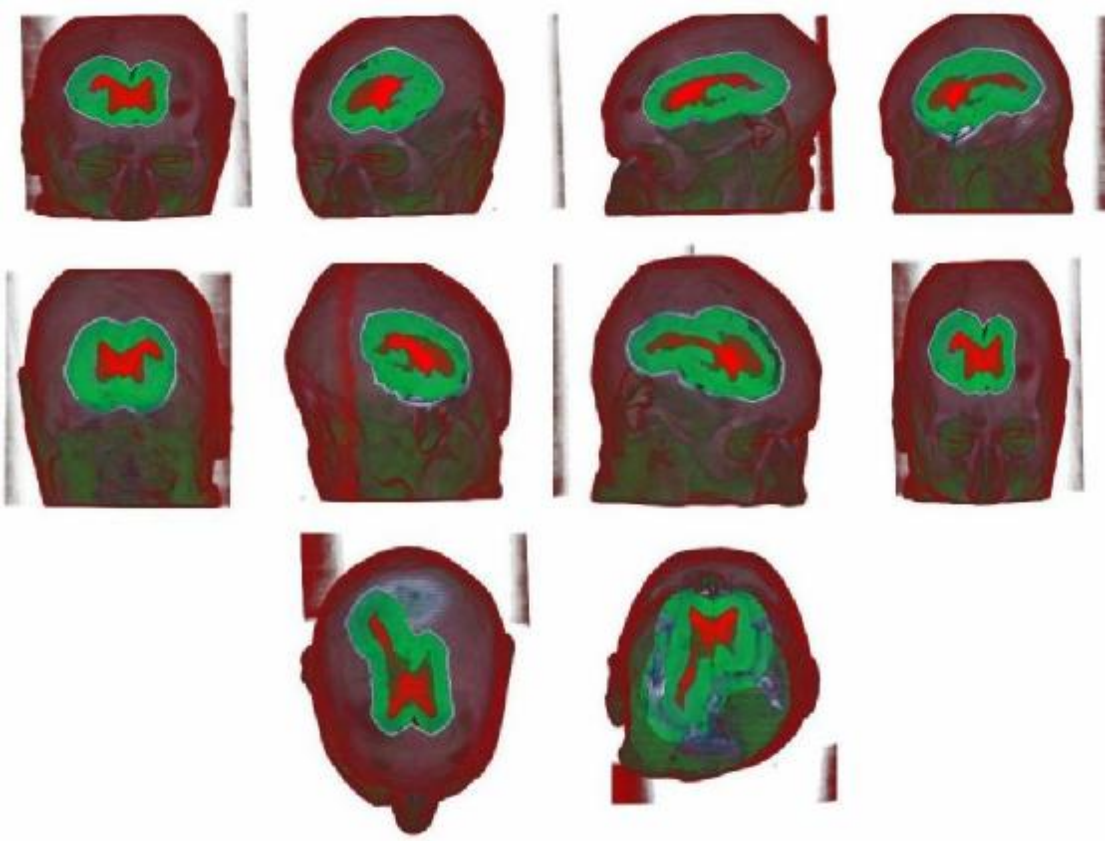
Obrázok C: Zobrazenie datasetov s použitím techniky gradienotvej modulácie.



Obrázok D: Použitie jednoduchého osvetlovacieho modelu pri zobrazovaní datasetu (hore).



Obrázok E: Zobrazenie datasetu z projektu AngioVis [34] s použitím techniky fokus a kontext (vpravo).



Obrázok E: Zobrazenie fokus a kontext techniky pre rôzne uhly pohľadu.

Príloha B

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

Zoznam:

- Text rigoróznej práce.
- Program f3dvr.
- Vstupné dáta.
- Zdrojový kód programu f3dvr.
- Testovacie prenosové funkcie.
- Knižnice