Fakulta matematiky, fyziky a informatiky
Univerzita Komenského, Bratislava

Rigorózna práca

Mgr. Michal Červeňanský 2006
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..................................................................................................................................................7

2 Vizualizácia objemových dát...........................................................................................................8
  2.1 Objemové dáta............................................................................................................................8
  2.2 Objemové zobrazovanie..............................................................................................................8

3 Algoritmy zobrazovania objemových dát......................................................................................10
  3.1 Nepriame metódy – povrchové zobrazovanie.........................................................................10
    3.1.1 Vyhľadávanie kontúr.............................................................................................................11
    3.1.2 Cuberille model.....................................................................................................................11
    3.1.3 Implicitné povrchové pokrývanie.........................................................................................12
  3.2 Priame metódy – objemové zobrazovanie..................................................................................12
    3.2.1 Objektovo orientované techniky.........................................................................................13
    3.2.2 Obrazovo orientované techniky............................................................................................13
  3.3 Interakcia svetla.........................................................................................................................13

4 Grafické akcelerátory......................................................................................................................15
  4.1 Minulosť.....................................................................................................................................15
  4.2 Súčasnosť.....................................................................................................................................15
  4.3 Budúcnosť.....................................................................................................................................16
  4.4 Grafické zobrazovanie – graphics pipeline...............................................................................16
    4.4.1 Spracovanie geometrie...........................................................................................................18
    4.4.2 Rasterizácia............................................................................................................................19
    4.4.3 Fragmentové operácie...........................................................................................................20

5 API (aplication programing interface)...............................................................................................22
  5.1 Direct3D......................................................................................................................................23
  5.2 OpenGL.......................................................................................................................................23
    5.2.1 OpenGL rozšírenia.................................................................................................................25
    5.2.2 Cg - C for graphics...............................................................................................................26
6 Objemové zobrazovanie s využitím grafických akcelerátorov.................................27
   6.1 Objemové zobrazovanie pomocou textúr..........................................................27
      6.1.1 Objemové zobrazovanie pomocou 2D textúr..............................................27
      6.1.2 Skladanie.......................................................................................................28
      6.1.3 Objemové zobrazovanie pomocou 3D textúr..............................................30
   6.2 Metóda vrhania lúča (ray casting)......................................................................31
      6.2.1 Implementácia..............................................................................................32
   6.3 Zhrnutie..............................................................................................................34
      6.3.1 2D textúry.....................................................................................................35
      6.3.2 3D textúry.....................................................................................................36
      6.3.3 Metóda vrhania lúča...................................................................................36

7 Prenosové funkcie....................................................................................................38
   7.1 Teória klasifikácie..............................................................................................38
   7.2 Implementácia....................................................................................................40
      7.2.1 Pred klasifikáciou........................................................................................40
         7.2.1.1 Pixel transfer (prenos dát)..................................................................40
         7.2.1.2 Textúry s paletami............................................................................41
      7.2.2 Po klasifikáciou..........................................................................................42
         7.2.2.1 Farebné palety.................................................................42
         7.2.2.2 Závislé textúry..............................................................................42
         7.2.2.3 Po klasifikáciu pomocou Cg.......................................................43
      7.2.3 Pred integrovaná klasifikácia..............................................................44
   7.3 Zhodnotenie.......................................................................................................46

8 Fokus a kontext techniky ........................................................................................48
   8.1 Základný návrh .................................................................................................49
   8.2 Implementácia ....................................................................................................50
      8.2.1 Hľadanie odštlačku masky.................................................................51
      8.2.2 Rozšírenie výškovej mapy.................................................................54
      8.2.3 Výsledné zobrazovanie ...........................................................................58
   8.3 Výsledky.............................................................................................................60
   8.4 Záver a ďalšia práca.........................................................................................61
1 Úvod

Počítačová grafika má veľké uplatnenie v mnohých vedných disciplínach, rovnako ako aj v zábavnom priemysle. Rozvoj počítačovej grafiky je hnaný používateliaťmi, ktorý žiadajú stále reálnejšie stvárnenie okolitého sveta pomocou počítača. Ako vznikajú nové a väčšinou náročnejšie algoritmy, tak spolu s týmito nárokom na výpočtový výkon sú vyvíjane aj rýchlejšie procesory a grafické karty, ktoré sú primárne určené na zobrazovanie grafiky. Výkon týchto grafických akcelerátorov narastá a v niektorých parametoch dokonca prekonávajú parametre hlavných procesorov osobných počítačov. Tieto grafické akcelerátoory sa v poslednej dobe stávajú nástrojom 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 hnaním motorom tejto oblasti a v iných oblastiach ako sú geológia, geografia, archeológia, rôzne simulácie prúdenia a podobne.

Kde'že 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 podobnej oblasti objemového zobrazovania s využitím tohto hardvéru. V práci sú uvedené techniky priamo súvisiacie 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 vnímanie 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 dátá

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

Objemové dátá 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 rovníč. Ak tento model opisuje trojrozmerný objekt hovoríme o voxelizácii.

2.2 Objemové zobrazovanie

Proces zobrazovania objemových dát je často popisaný ako visualization pipeline, na obrázku 2.1 je zobrazená jeho základná verzia. Vstupom do procesu zobrazovania sú nespracované dátá, z ktorých je zvyčajne len istá časť zaujímavá. Filtrovaním sú dátá analyzované a prevzorkované (interpolované) podľa používateľom definovaných kritérií. Mapovanie tvorí podstatnú časť, kde sú detegovane 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, priebehľadnosť). V zobrazovacom kroku sa použijú základné primitívy na zobrazenie
výsledného objektu (objemu). Časti týchto krokov môžu byť urýchlené pomocou grafických akcelerátorov.

*obrážok 2.1: Proces zobrazovania objemových dát (visualization pipeline).*
3 Algoritmy zobrazovania objemových 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 dáť 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 dosiahnutelňa použitím štandardných grafických akcelerátorov (pokiaľ trojhouholníkov nie je príliš veľa).
3.1.1 Vyhľadávanie kontúr

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

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 viest’ 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. Dalej predpokladajme, že objekt je pospájaný tak, že dva ľubovoľné voxel y 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 neprekryvajúce 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 priečinkov s hranami bunky a spájajú ich do mnohouholníkov, trojuholníkov. Priečinkov 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 krokom:
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 trojuholníkovovej 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 dátu priamo bez vytvárania pomocnej povrchovej reprezentácie. Tento prístup má výhodu v tom, že môžeme vizualizovať polo prieľadné materiály a štruktúry vnorené do iných štruktúr. Tieto techniky umožňujú zobrazovať rozhranie medzi materiálm a aj ich vnútro.

Pre objemové zobrazovanie sa používajú dva hlavné prístupy. Objektovo orientované alebo hybridné techniky, ktoré kombinujú prvé dve. Objektovo orientované zobrazovanie využíva mapovanie dát na zobrazovaci rovinu. V obrazovo
orientovaných algoritmov je pre každý pixel zo zobrazovacej roviny vrhaný lúč cez celé dát 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


3.2.2 Obrazovo orientované techniky

Tieto techniky posudzujú každý pixel vo výslednom obrazu samostatne. Pre každý pixel je počítaná výsledná hodnota z jednotlivých príspevkov celého objemu, ktoré zodpovedajú prechodu pohľadového lúča cez dátá. Typickým predstaviteľom tejto metódy je 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:

\[
\frac{dI(t)}{dt} = \rho(t)I(t) - k(t)\rho(t),
\]  

\[(3.1)\]
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 dostáť rovnicu pre diskrétny priestor [32], ktorá popisuje interakciu svetla s prostredím:

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

Táto rovnica je už prevedená na popisovanie metód pracujúcich zozadu dopredu, kde $v_k$ je prieľaďnosť 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 enorme 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 Minulost’


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 hlavné 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, CrossFire), 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 výkresťovanej 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 spracúvajúca vrcholy (vertex shader) a časť spracúvajúca fragmenty (fragment shader), každá tato 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 kolko jednotiek bude pridelených daným (vertex, fragment) častiam. Malo by to podporovať aplikácie, ktoré využívajú vo váčšej miere len jednu časť a druhá je nevyužitá. Programátor (ovládač) 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 rôznho 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 váčšinu 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 ...)


**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 poznáť proces grafického prechodu z trojrozmernej scény na rastrový obraz. Preto si ešte podrobnejšie vysvetlíme jednotlivé kroky.

![obrázok 4.1: grafické zobrazovanie – graphics pipeline](image-url)
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 matice 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 prebehné 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 nahradit' pevne danú postupnosť výpočtu programom (vertex program), ktorý je aplikovaný na každý príchádzajúci vrchol. Tento vertex program je aplikovaný vo vertex jednotke (vertex shader) a nahrádza prvé tri časti spracovania geometrie.

4.4.2 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 súradníck 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 prislúchojúca 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**


![Fragmentové operácie](image)

*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 prislúchojúcou 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.
Programovacie jazyky (C, C++, Java atd.), ktoré chcú využívať vymoženosti grafických kariet, musia vedieť komunikať 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 vyvíjании 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.

Efektivnosť: 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 prostrediac.

Pre programovanie grafických kariet boli vyvinuté samostatné programovacie jazyky, ktoré kooperujú s jednotlivými API. Pre Direct3D to je HLSL (high level shader language)[36] a pre OpenGL to je assembler podporovaný priamo v OpenGL rozšíreniách. Pre tento assembler 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 language)[38], ktorý je priamo zahnutý 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 obslusu 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 vysoké 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


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.


<table>
<thead>
<tr>
<th>Plusy a mínusy</th>
<th>OpenGL</th>
<th>Direct3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programovacie jazyky</td>
<td>C, C++, Fortran, Perl, Python, Java, Ada</td>
<td>Všetky, ktoré podporujú COM</td>
</tr>
<tr>
<td>Voľne šíriteľný štandard</td>
<td>Áno</td>
<td>Nie (vlastníctvo Microsoftu)</td>
</tr>
<tr>
<td>Rozšíriteľnosť</td>
<td>Áno (OpenGL rozšírenia)</td>
<td>Nie</td>
</tr>
</tbody>
</table>

*tabuľka 5.1: Porovnanie medzi OpenGL a Direct3D*
5.2.1 OpenGL rozšírenia

Aj keď základná knižnica OpenGL poskytuje dôležitú časť funkčností, pokrok grafického hardvéru rapídne rastie a nové zobrazovacie techniky nie sú pokryté v základnej knižnici. Našťastie rozšírovanie 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 verzíach 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 rôznom zdrojovým kódom. Pri využívaní multivendor rozšírení je zdrojový kód rovnaký pre grafické karty od rôznych výrobkov, ktoré podporujú dané rozšírenie. Preto je pre dobrú prenositeľnosť danej aplikácie vhodné využívať hlavne multivendor rozšírenia.

OpenGL rozšírenia sú definované svojim názvom, ktorý sa skladá s predponou, vo väčšine prípadov trojznakový 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 ...).

5.2.2 Cg – C for graphics

Cg is a programming language developed by nVidia Corporation in collaboration with Microsoft [7, 35]. Cg compiles using OpenGL extensions, which support programming of graphics cards in their own assembler, thereby eliminating the need to program in a lower-level language. Calls to the Cg language set the necessary parameters for OpenGL functions and use extensions that define the assembler of graphics cards. Cg uses the syntax of the programming language C and is compatible with the OpenGL API and Microsoft’s High-Level Shader Language (HLSL) for DirectX 9.0.

Cg program works with vertices and fragment data that enter the program and exit after certain transformations [7]. Cg allows replacing the parts of the rendering process performed by the program. Depending on where it is applied, we speak of a vertex program, which replaces the parts of the geometry processing, is executed in the vertex shader, and works on vertices. Fragment program, which works in the rasterization unit on fragments and executes fragment shader. Cg compiles the program into the assembler (set of instructions) defined in the OpenGL extensions. Compilation supports multiple extensions that specify different assembler instructions. Just like Cg is a creation of nVidia, it also supports multi-vendor extensions defining assembler of graphics cards, so programs based on these extensions should work on graphics cards from other firms.

26
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 voxelov sú väčšinou reprezentované ako 8 alebo 16 bitové čísla. Obvykle rozmery reálnych dát sú 512x512x512 a viac, menšie dátá sú väčšinou využívané len ako testovacie vzorky. Dáta z počítačovej tomografie mávajú rozlišenie 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 dátá.

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, povolojuje bi-lineárnú interpoláciu, ktorá je vo veľkej miere zodpovedná za výpočtovú zložitosť objemového zobrazovania. Preto je dobré využiť túto možnosť grafických kariet na urýchlenie zobrazovania.

![Obrázok 6.1: Objekt rozložený na sadu objektovo orientovaných rezov](image)

6.1.2 Skladanie

Jednotlivé zobrazené rezy musia byť nejakou vhodnou formou pospájané do výsledného obrazu tak, aby vyhovovali teoretickým predpokladom uvedeným v časti 3.3. Ďalšou úpravou rovnice (3.2) získaume rovnici [21,28,32]:

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

Použité 2D textúry môžeme reprezentovať na grafickej karte ako štvrť ploche 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 pozdlž 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 rovnici

\[ C'_{\text{dest}} = C_{\text{src}} A_{\text{src}} + C_{\text{dest}} (1-A_{\text{src}}) \]  \hspace{1cm} (6.2)

a použitím substitúcie nahrádzaču koeficienty pohlcovania a vyžarovania,

\[ C_{\text{src}} = I_{\text{emit}} \quad \text{a} \quad A_{\text{src}} = (1-v_k) \]  \hspace{1cm} (6.3)

dosiahneme rovnici (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 rovnici príkazom `glBlendEquationEXT(GL_MAX_EXT)`. Táto funkcia je zahrnutá v OpenGL rozšíreníach 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](image)

29
6.1.3 Objemové zobrazovanie pomocou 3D textúr


![Rezy, 3D textúra, Výsledný obraz]

*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.

30
obrázok 6.4: vzorkovanie pozdlž 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 dátaami, potom postupne prechádza dátaami a akumuluje hodnoty z dát pozdlž tohto lúča. Výsledná nakumulovaná hodnota je zobrazená na obraz. 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 pozdlž 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 výpočtové sú vkonávane priamo na GPU. Pre každý vrchol datasetu (kocky) sa do vertex programu na grafickú kartu posielá prislúchajúca texturová súradnica a potrebné maticie. 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.

```cpp
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 polohu lúča a v druhej sa už v cykle prechádza cez objem a akumulujú výsledné hodnoty. Grafická karta vygeneruje pre každý fragment textúrovú súradnicu, ktorú leží na hranici datasetu. Táto súradnica slúži ako počiatočný bod pre lúč, ktorý prechádza cez dátá. Počiatočnú polohu lúča treba posunúť...
v smere pohládového lúča k najblížšej „pologule“, aby sa dosiahol korektný algoritmus (obrázok 6.6), kde počet pologul’ (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 objem, alebo ak naakumulovaná hodnota priehľadnosti dosiahne určitú úroveň a ďalšie vzorky by už nezmenili aktuálnu hodnotu, prípadne len neaptrne.

```cpp
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 zhrneme hlavné výhody a nevýhody každého prístupu objemového  
obrazovania popísaného vyššie.

Tieto algoritmy sú založené na podpore grafických akcelerátorov, preto pre  
obrazované 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 hlavou pamäťou a video pamäťou pri prenose dát. Dnešné PCI Express zbernice umožňujú prenos teoreticky rýchlost'ou až 8GB/s. Tieto problémy ale v tejto práci nebudú rozoberané.

6.3.1 2D textúry


obrázok 6.7: vzorkovacie artefakty zapríčinené zmenou sady rezov
6.3.2 3D textúry


6.3.3 Metóda vrhania lúča


<table>
<thead>
<tr>
<th>2D textúry</th>
<th>Plusy</th>
<th>minusy</th>
</tr>
</thead>
<tbody>
<tr>
<td>vysoký výkon</td>
<td>vysoké pamäťové požiadavky</td>
<td></td>
</tr>
<tr>
<td>veľká dostupnosť</td>
<td>iba bi-lineárná interpolácia</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vzorkovacie artefakty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>prepínací efekt medzi sadami textúr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nekonzistentné vzorkovanie</td>
<td></td>
</tr>
</tbody>
</table>

*tabuľka 6.1: výhody a nevýhody zobrazovania pomocou 2D textúr*
### 3D textúry

<table>
<thead>
<tr>
<th>Plusy</th>
<th>mínusy</th>
</tr>
</thead>
<tbody>
<tr>
<td>vysoký výkon</td>
<td>rozmerové obmedzenie</td>
</tr>
<tr>
<td>trilineárna interpolácia</td>
<td>nekonzistentné vzorkovanie pre stredové premietanie</td>
</tr>
</tbody>
</table>

*tabuľka 6.2: výhody a nevýhody zobrazovania pomocou 3D textúr*

### Vrhanie lúča

<table>
<thead>
<tr>
<th>plusy</th>
<th>mínusy</th>
</tr>
</thead>
<tbody>
<tr>
<td>trilineárna interpolácia</td>
<td>rozmerové obmedzenie</td>
</tr>
<tr>
<td>korektný pre obe premietania</td>
<td>menší výkon a ako 2D/3D textúry</td>
</tr>
<tr>
<td>modulárny</td>
<td>podpora shader modelu 3.0c</td>
</tr>
</tbody>
</table>

*tabuľka 6.3: výhody a nevýhody metódy vrhania lúča*
7. Prenosové funkcie

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

\[ I_{\text{emit}} = T_{\text{emit}}(v) \quad \text{a} \quad v_k = T_{\text{abs}}(v), \]

kde \( I_{\text{emit}} \) reprezentuje koeфicient vyžarovania a \( v_k \) koeфicient 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 funkcí je manuálny a zdôlžavý proces, ktorý vyžaduje dôkladné znalosti a štruktúru, ktorú dáta reprezentujú. Iné výsledky sa obdržia pri použití tých istých prenosových funkcích, ale rôzných dátach rovnakého objektu (CT, MRI). Pre vhodné definovanie daných funkcií v procese klasifikácie je veľmi dôležitá viditeľná odozva používateľových operácií. Preto je žiaduše 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 (intenzíte) 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 pevnnej dĺžky. Koeфicient vyžarovania \( I_{\text{emit}} \) je zvyčajne reprezentovaný
ako RGB zložka, ktorá dovoľuje vyžarovanie farebného svetla, čím získavame farebné dátá. Koeficient pohlcovania je reprezentovaný ako skalárna veličina v rozmedzí 0 až 1, ktorý definuje priebeh časť danej vzorky. Tieto koeficienty môžeme skombinovať do RGBA hodnoty.

Objemové dátá 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 rekonštrukciou alebo až na spojitý signál po rekonštrukcii [23]. Obe metódy vedú k viditeľne rozdielnym výsledkom.

**Pred klasifikáciou** 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 klasifikáciu** 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 frekvencí v prenosových funkciiach, je treba zobrazovať veľa rezov (Nyquistovo kritérium)

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

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

Generovanie palety potrebné pre pred integrovanú klasifikáciu je časovo náročnejší proces ako generovanie paliet pre predošlé metódy. Predošlé metódy si vystačia s jednorozmernou paletou zloženou s RGBA hodnot, 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 využívanú techniku textúrového mapovania vysvetlené vyššie. Sú tu popísané techniky pred klasifikáciou, po klasifikácii a pred integrovanou klasifikáciou. Tieto techniky vedú pri zmenách transfer funkcie k interaktívnom zmenám zobrazovaných dát za predpokladu dostupnosti využívaných funkcií grafickým akcelerátorom.

7.2.1 **Pred klasifikácia**

Pred klasifikáciou vyžaduje zavedenie farebného 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 hlavnom pamäťou a video pamäťou, čím sa výrazne obmedzuje interaktívita 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


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 definování prenosovej funkcie. Súčasný grafický 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

Priamo 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 paleta 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 rozdielnom 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 klasifikácii 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 dátou použijeme ako index, textúrové súradnice, do textúry reprezentujúcu paletu. Na výpise 7.4 je uvedený fragment program napísaný v Cg.

```cpp
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;
}
```

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.
Vstupné premenné do frambgem 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 ($\text{tex3d()}$). Pomocou týchto skalárnych hodnôt sa získá 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 funkcií, ktorá je definovaná ako lineárná lomená funkcia, stačí prepočítať 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).

**Obrázok 7.3: schéma generovania palety pre metódu pred integrovanej klasifikáciou**

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

### 7.3 Zhodnotenie

Spomínané metódy aplikovania prenosovej funkcie sú vykonateľné interaktívne pri definovaní prenosovej funkcie s využitím grafického zariadenia. Pre jednotlivé metódy dostávame viditeľné rozdielne výsledky (obrázok 7.1 a 7.2) a to najmä pri veľkých zmenách prenosovej funkcie (vysoké frekvencie). Metóda po klasifikácii a metóda pred integrovanej klasifikáciou sú v mapovaní prenosovej funkcie na rezy korektné. Pre prenosové funkcie s veľkou frekvenciou, so zachovaním dostatočnej kvality, vedie použitie
metódy po klasifikácii k veľkému počtu vzoriek, rezov (Nyquistove kritérium). Metóda pred integrovanej klasifikácie odstraňuje nutnosť použiť veľký počet rezov, blokov, pre prenosové funkcie s veľkými frekvenciami.
Objemové zobrazovanie zobrazuje celý dataset s rovnakými vlastnosťami pre rôzne smery pohľadu, čo je niekedy nedostačujúce. Používateľia sa pri niektorých aplikáciách potrebuje zmerať na významné miesta v dátach, ktoré sú potrebné pre ich výskum a potreby. Tieto algoritmy, ktoré umožňujú zvýrazniť významné oblasti v dátach, prípadne sa zmerať na dané miesto a ostatné potlačíť, spadajú do oblasti fokus a kontext techník.

Tieto požiadavky na zvýraznenie istých oblastí v dátach je možné rozdeliť do určitých oblastí, kde každá oblast pristupuje k tomuto problému inak.

**Prenosové funkcie** sú jednou z možností ako dosiahnuť potlačenie istých častí dát a zvýrazniť iné. Jednorozmerné prenosové funkcie sú rozšírené 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 aplikovať na celý objem.

Álšou oblast'ou je **ilustratívne objemové zobrazovanie**, kde prvým prístupom bolo modifikovať priebeh'adnosť v závislosti na veľkosti gradientu, čím sa docieli zvýraznenie hranicných oblastí a potlačenie homogénnych oblastí [18]. Álšie prístupy sú prístupy, ktoré zvýrazňujú kontúry v dátach v závislosti od veľkosti gradientu a uhлом medzi smerom pohľadu a gradientom [4]. Medzi tieto metódy patrí aj dvojúrovňové 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 álšej oblasti. Táto oblast sa snaži 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 priebeh'adnosťi dát, ktoré sa nachádzajú v oblasti medzi pozorovateľom a zameriavanou oblast'ou. Oblasti v zameraní (fókus) sa zobrazujú so štandardnou priebeh'adnosťou (kontext)[2, 30].

Niektoré z týchto techník pod oblast'ou, ktorá je zobrazovaná ako významná zobrazujú dátu, 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 dátu, umožňujú zobrazit' priamo tieto dátu, 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 zobrazení 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

8.2 Implementácia

8.2.1 Hľadanie odtlačku masky

Pри 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 nekorektné filtrácií 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:

```glsl
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 štartovej 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 krokovaci 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 odkažená 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 (bounding box) a podľa týchto hodnôt je nastavená najbližšia a najvzdialenejšia orezávacia rovina pre pohľadový hranol (viewing 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 rozlišenie masky a to tak, že jej rozlišenie musí byť rovnaké ako rozlišenie kontextového datasetu. Avšak toto obmedzenie môže byť vynechané, ale potom treba robíť korektúry aby, datasets 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 poznáť veľkosť uhla (obrázok 8.5), čo je voliteľný parameter, maximálnu hodnotu v textúre a rozmery hraničného obaju (bounding box) masky na obrazovke (screen space). Pod uhlom rozumieme uhol medzi pohľadovým vektorom a vektorom definovaným priamkou pozdlž plášťa pomyselného výrezového kužeľa. 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 pozícií 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 iterácií je definovaný nasledovne:

\[
Pocet\_iter = \text{maxVal} \times \tan(alfa) \times \text{max}(\text{resx}, \text{resy}) \quad (8.1)
\]

Kde `maxVal` je maximálna hodnota z textúry, `alfa` je definovaný uhol a `resx, resy` sú rozmery datasetu na obrazovke. Samozrejme, úplne presný počet iterácií 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 iterácií vieme odvodiť hodnotu (dekrement) nasledovne:

\[
\text{dekrement} = \frac{\text{maxVal}}{\text{Pocet\_iter}} \quad (8.2)
\]

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

```cpp
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, stPoint) - sqrt*decrement, color);
```
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 $\sqrt{2}$*decrement. Ak nie je nájdená hodnota väčšia ako je hodnota daného fragmentu, tak je použitá priamo táto hodnota bez zníženia. Daný fragment program uvažuje rozširovanie odštlačku rovnomerne vo všetkých smeroch, čomu zodpovedá použité jadro (8.3) pre znižovanie hodnoty fragmentu.

$$
\begin{array}{ccc}
-\sqrt{2} & -1 & -\sqrt{2} \\
-1 & 0 & -1 \\
-\sqrt{2} & -1 & -\sqrt{2}
\end{array}
$$

(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 datatetu.
8.2.3 Výsledné zobrazovanie

V tejto časti je popísané výsledné zobrazenie orezaného datasetu, pričom je použitý modifikovaný algoritmus vrhania lúča. Z predošlých častí popísaného algoritmu je použitá vygenerovaná výšková mapa, v ktorej je uložená reprezentácia výrezového kužeľa na modifikované 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 kužeľu. 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

```c
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 "pologu"
    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 \((\text{tex\_pos})\), ktorá je uvedená v kapitole 6.2.1. Počas tejto korektúry sa textúrová súradnica (štartovacia pozícia) posunie pozdĺž pohľadového vektora o vzdialenosť \((\text{distance})\), ktorá zodpovedá vzdialenosti medzi hranicou datasetu a reprezentovaným výrezovým kužeľom. Vzdialenosť \((\text{distance})\) 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 \textit{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 orezovaný dataset.

### 8.3 Výsledky

Testovanie časovej zložitosti vyššie popísaného algoritmu je znázornené a popísané v nižšie priloženej tabuľke \((\text{tabuľka 8.1})\). 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.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>15</th>
<th>30</th>
<th>45</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harmonic 32x32x32</td>
<td>169ms</td>
<td>315ms</td>
<td>480ms</td>
<td>714ms</td>
<td>877ms</td>
</tr>
<tr>
<td>Tot 170x190x180</td>
<td>106ms</td>
<td>250ms</td>
<td>416ms</td>
<td>649ms</td>
<td>680ms</td>
</tr>
<tr>
<td>Knee 512x512x128</td>
<td>250ms</td>
<td>487ms</td>
<td>526ms</td>
<td>909ms</td>
<td>1.2s</td>
</tr>
</tbody>
</table>

\(\text{Tabuľka 8.1: prehľad časov potrebných na zobrazovanie v závislosti na veľkosti uhla.}\)
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 výzduchu (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 časťou 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 informácií, 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], ktorej 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 popísané 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 bolí 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 pri zobrazení osvetľovacích modelov. Pre zobrazenie gradientu je potrebné poznáť gradient (normálu) povrchu v danom bode. Tento gradient môže byť vyrátený priamo z dát buď v rámci predspracovania, 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.

```c
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;
```

64
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 opakované pre každý fragment. Pri použití ďalšej 3D textúry s predgenerovaným gradientom získaným v predspracovaní sa dosahuje rýchlejšie zobrazovanie avšak na úkor potrebné 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, odvoedený od Phongovho osvetľovacieho modelu (rovnica 9.1) [8, 21, 32].

\[
I(\lambda) = K_a(\lambda).I_a(\lambda) + K_d(\lambda).I_L(\lambda).(N.L) + K_s.I_L(\lambda).(R.V)^k
\]  

(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 diffuseLight = max(dot(normalize(normal),vecToLight),0);
float specularLight = pow(max(dot(normalize(normal),vecHalf),0), 15 );

if(diffuseLight <= 0) specularLight = 0;

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

float3 diffuse = tex2D(palette,lookup)*diffuseLight;
float3 specular = float3(1.0,1.0,1.0)*specularLight;

OUT.color.xyz = diffuse + 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 difúznego 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 diffuseLight = max(dot(normalize(normal),vecToLight),0);
float specularLight = pow(max(dot(normalize(normal),vecHalf),0), 45 );
float ambientLight = 0.5;

float shadedFactor = diffuseLight + 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 potrebné 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**


**9.4 Iné**

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

Špecifikácia OpenGL ponúka možnosť využiť orezávacie roviny, s ktorými je možné rotovať, posuvať 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.

10 Záver

Predkladaná rigorózná práca bola zamernána 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 črt 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


http://www.opengl.org


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


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

   http://www.gpgpu.org/


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


Obrázok A: Kombinácia rôznych zobrazovacích 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. Vlavo 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 uhol 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