Fakulta matematiky, fyziky a informatiky
Univerzita Komenského, Bratislava

Diplomová práca

Slavomír Hudák 2006

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
UNIVERZITA KOMENSKÉHO, BRATISLAVA
KATEDRA APLIKOVANEJ INFORMATIKY

Diplomová práca

Vizualizácia Dát z Konfokálneho Mikroskopu

Autor: Slavomír Hudák

Školiteľ: RNDr. Marek Zimányi

ŠTÚDIJNÝ ODBOR: INFORMATIKA

Bratislava 2006

Týmto vyhlasujem, že som diplomovú prácu vypracoval samostatne s odbornou pomocou školiteľa, čerpajúc z uvedenej literatúry a zdrojov dostupných na internete.

Bratislava, Júl 2006 Slavomír Hudák

Chcel by som sa poďakovať môjmu školiteľovi RNDr. Marekovi Zimányimu za odbornú pomoc, rady a usmerňovanie pri tvorbe tejto práce. Ďalej by som sa rád poďakoval RNDr. Antonovi Mateašíkovi, PhD. za rady a cennú pomoc s klastrom, priateľke a rodine za morálnu podporu.

Abstrakt

Táto práca sa zaoberá problémom vizualizácie dát z konfokálneho mikroskopu na paralelných architektúrach. Popisujeme viaceré možnosti vizualizácie objemových dát, ich optimalizáciu a implementáciu na paralelných architektúrach. Navrhli sme architektúru systému a implementovali sme algoritmy ray-casting a binary swap pre paralelnú objemovú vizualizáciu na šestnásť uzlovom PC klastri prepojenom vysokorýchlostnou sieťou Myrinet cez MPI. Používame viaceré techniky urýchlenia procesu renderovania vrátane preskakovania prázdnych úsekov a efektívne adresovanie v objemových dátach. Taktiež prezentujeme tipy pre optimálne nastavenie MPI a prekladačov. Systém zobrazuje konfokálne dáta interaktívne, t.j. niekoľko snímok za sekundu.

Kľúčové slová: Vrhanie lúča objemom, paralelné renderovanie, PC klastre, algoritmus binary swap.

Abstract

This thesis provides a presentation of a parallel volume data visualization system for data supplied from confocal laser scanning microscopy. It describes and compares various methods for volume visualization and optimalization and their implementations on parallel architectures. The parallel volume renderer we have developed is implemented on a 16 nodes PC cluster connected by a high-speed Myrinet network using MPI, ray-casting and binary-swap algorithm. To improve efficiency, several optimalizations for increasing the speed of the ray casting process are used, including empty regions skipping and an efficient bricking addressing scheme. We also suggest several methods of improving the performance when using MPI and compilers. The system is capable of running interactively (several frames per second) on a PC cluster.

Keywords: Volume ray-casting, parallel rendering, PC clusters, binary swap algorithm.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam skratiek

BTF	Back-To-Front
CLSM	Confocal Laser Scanning Microscopy
CPU	Central Processing Unit
CT	Computed Tomography
FTB	Front-To-Back
GNU	GNU's Not Unix (Free Software Fundation)
GPL	General Public Licence
GPU	Graphics Processor Unit
HPC	High Performance Clusters
IEEE	The Institute of Electrical and Electronics Engineers
ILC	International Laser Centre
ISO	International Standards Organization
I/O	Input/Output
LAN	Local Area Network
LSM	Laser Confocal Microscopy
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MLC	Medzinárodné Laserové Centrum
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
NUMA	Non-Uniform Memory Access
RADC	Ray Acceleration by Distance Coding
PGI	The Portland Group, Inc.
PSF	Point Spread Function
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SMP	Symmetrical Multi Processing
SPMD	Single Program, Multiple data
SSE	Streaming SIMD Extensions
SSH	Secure Shell
TIFF	Tagged Image File Format
1D	Jednorozmerný
2D	Dvojrozmerný
3D	Trojrozmerný

Kapitola 1

Úvod

Objemová vizualizácia je dobre známa vetva počítačovej grafiky (presnejšie oblasť vedeckotechnickej vizualizácie), ktorá umožňuje skúmanie vnútorných štruktúr a komplexného správania sa trojrozmerných objemových objektov. Najmä vďaka potrebám medicíny, korektne a rýchlo zobrazovať orgány ľudského tela na bežnom počítači, sa objemová vizualizácia rozvíja tak rýchlo. Nové techniky a dostupný hardvér dnes umožňujú lekárom vizualizovať objemové dáta dostatočne interaktívne. Výsledky dlhoročného bádania a vývoja vylepšili ako softvérovú, tak aj hardvérovú stránku problematiky. Sú známe a dobre preskúmané techniky zobrazujúce komplexné dátové štruktúry získané z rôznych vedných odborov. Oproti medicíne sú štruktúry na mikroskopickej úrovni ešte zložitejšie a mnoho z nich je stále nepreskúmaných.

Viacrozmerná vizualizácia sa v posledných rokoch na poli mikroskopie značne rozrástla. Mnohé techniky boli úspešne aplikované vo svetelnej alebo elektrónovej mikroskopii. Príchod konfokálnej laserovej rastrovacej mikroskopie (confocal laser scanning microscopy – CLSM) priniesol revolučné možnosti optického zobrazovania, ktoré viedli k rapídnemu nárastu využitia vizualizačných techník pre najrôznejšie typy vzoriek.

V *prvej* kapitole uvedieme problém interaktívnej vizualizácie, priblížime pojem konfokálna mikroskopia a charakterizujeme konfokálne dáta. Kapitola *druhá* ponúka prehľad metód objemovej vizualizácie, detailne popisuje princíp a fungovanie algoritmu sledovania lúča. *Tretia* kapitola uvádza čitateľa do problematiky tvorby paralelných programov a popisuje paralelné možnosti prístupu k objemovej vizualizácie. Kapitola číslo *štyri* prezentuje viaceré optimalizácie metód popísaných v predchádzajúcich kapitolách. Kapitola *päť* obsahuje všetky detaily návrhu a implementácie paralelného programu, zmieňujeme sa o problémoch, s ktorými sme sa počas vývoja stretli. Výsledky práce prezentujeme v kapitole *šesť*. Zhrnutie práce a jej možné rozpracovania sú obsahom *siedmej* kapitoly. Referencie a zoznam príloh uzatvárajú prácu.

1.1 Oblasť problematiky

Trojrozmerné spracovanie obrazu je časť počítačovej grafiky, ktorá na seba v posledných rokoch púta viac a viac pozornosti. Hlavným dôvodom je prudký nárast výkonu počítačov a dostupnosť nových technológii, vďaka ktorým sme schopní oveľa rýchlejšie spracovať obraz. Jej využitie postrehneme vo viacerých oblastiach života. Veľmi dôležitú úlohu zohráva v *medicíne*, kde pomáha doktorom analyzovať údaje zo zariadení ako magnetická rezonancia (MRI)

alebo počítačová tomografia (CT). Jej ďalšie využitie môžeme vidieť napríklad v *geografii* (zobrazovanie modelov ropy, plynov alebo vodných nádrží získaných zo seizmologických dát), *fyzike* (vizualizácia výsledkov simulácii fyzikálnych procesov), *priemysle* (analýza fraktúr časti materiálu), *námorníctve* (zobrazovanie signálov zo sonaru), alebo aj *biológii* (rekonštrukcia trojrozmerného modelu objektov zo série rezov získaných konfokálnym mikroskopom) (Bentum, 1996). V práci sa obmedzíme na dáta z konfokálneho mikroskopu, hoci algoritmy použité v tejto práci nie sú definované týmto typom údajov.

1.2 Motivácia a cieľ práce

Cieľom diplomovej práce bolo štúdium, porovnanie a výber najvhodnejších metód pre objemovú vizualizáciu konfokálnych dát na paralelných architektúrach. Navrhli a vytvorili sme nástroje pre vizualizáciu konfokálnych dát na paralelnom počítači – PC klastri. Hlavným cieľom použitia vizualizačných techník v oblasti konfokálnej mikroskopie je príprava ľahko interpretovateľných a, ak je to možné, aj realistických obrazov, bez artefaktov a s možnosťou zvýraznenia vybraných objektov záujmu prostredníctvom segmentačných a klasifikačných postupov. Hoci konfokálna mikroskopia poskytuje možnosti pre jednoduché a neinvazívne snímanie, vizualizácia výsledných dát je vcelku zložitý proces.

Na dosiahnutie tohto cieľa bolo potrebné detailne naštudovať problematiku objemovej vizualizácie a paralelného renderovania (kapitola 2 a 3). V týchto kapitolách sú uvedené dôležité techniky, ich vzájomné porovnania, ale aj problémy s nimi súvisiace. Objemové renderovanie je iba jedna z metód vizualizácie viacrozmerných dát. Porovnávame výhody a nevýhody volumetrického zobrazovania. Detailne popisujeme priamu objemovú metódu ray-casting (kapitola 2), ako aj tvorbu paralelného programu a ťažkosti s tým spojené (kapitola 3). V tejto kapitole popisujeme aj konštrukciu a zobrazenie obrazu na paralelných architektúrach. Pre vylepšenie celkovej rýchlosti programu sme v kapitole 4 vybrali viaceré optimalizácie softvéroveho ray-castingu, vrátane preskakovania nezaujímavých častí dát, efektívnu schému adresovania a vyhľadávacej tabuľky homogénnych buniek. Popis návrhu systému, implementácie algoritmov a ich urýchlení je uvedený v kapitole 5. Nachádza sa tu presná špecifikácia hardvérových možností, detaily ohľadom implementácií jednotlivých častí programu na paralelnom počítači a naše problémy spojené s vývojom softvéru. Výsledky, vo forme tabuliek a grafov, rýchlostných a kvalitatívnych porovnaní, sa nachádzajú v kapitole 6. Popisujeme nastavenia prekladačov pre dosiahnutie dobrých výsledkov, porovnávame ich a v tabuľkách uvádzame výsledky v kombinácii pri použití sietí Myrinet a Ethernet. Uvádzame porovnania základného a optimalizovaného algoritmu pri použití na jednom a viacerých procesoroch.

Hlavý prínos práce spočíva v paralelizovaní procesu zobrazovania CLSM dát priamymi objemovými metódami na PC klastri. Návrh jednotlivých súčastí programu sa snaží byť čo najvšeobecnejší v zmysle použitia na klasri. Implementácia tried a algoritmov zasa čo najportabilnejšia a najefektívnejšia. Výsledný čas zobrazovania obrázkov je interaktívny, t.j. niekoľko snímok za sekundu. Práca ponúka prehľad objemovej vizualizácie, tvorby paralelných programov a paralelnej objemovej vizualizácie.

Tvorba práce a vývoj softvéru prebiehala v spolupráci s Medzinárodným Laserovým Centrom (International Laser Centre)[1] sídliacim na Fakulte matematiky, fyziky a informatiky Univerzity Komenského v Bratislave.

Medzinárodné laserové centrum je organizácia, zameraná na vzdelávanie, výskum a vývoj v oblasti progresívnych metód a technológií fotoniky a ich aplikácií v rôznych oblastiach a na rôznych úrovniach národnej a medzinárodnej spolupráce. Na oddelení biofotoniky MLC je inštalovaný konfokálny mikroskop Zeiss LSM 510 Meta NLO vybavený spektrálnym detektorom a inverzný mikroskop Axiovert 200M s predprípravou vzoriek (vyhrievanie, perfúzia, elektrostimulácia). Toto zariadenie sa používa vo viacerých projektoch, ktoré sú zamerané, napríklad, na štúdium závislostí mitochondriálneho metabolizmu a modulácií excitácie a kontraktility v izolovaných kardiomyocytoch vo vzťahu k vývoju a priebehu hypertenzie, ďalej na štúdium polymérnych membrán a matríc vyrobených z polyelektrolytických komplexov a na štúdium zamerané na aplikácie rôznych fotosenzibilizátorov vo fotodynamickej terapii nádorov.

Spracovanie obrazových dát z konfokálneho mikroskopu je úzko zviazané s možnosťou využitia počítačového klastra IBM eServer 1350. V súčasnosti sa klaster využíva hlavne pri analýze kofokálných dát a overovaní modelov sledovaných dejov prebiehajúcich v živých systémoch. Avšak MLC predpokladá využitie klastra aj pre priame zobrazovanie – vizualizáciu rozsiahlych dát aplikáciami paralelných renderovacích techník.

V súčasnej dobe laserové centrum nedisponuje žiadnym vyhovujúcim nástrojom na interaktívne zobrazovanie nasnímaných preparátov. Momentálne riešenie predstavujú programy, umožňujúce vytvorenie dvojrozmerného obrazu alebo animácie (rotácia kamery okolo nasnímaného objektu) preparátu. Je dôležité poznamenať, že väčšina týchto programov pracuje na bežne dostupnom počítači. Tomu zodpovedá rýchlosť vytvorenia, a samozrejme aj kvalita výsledkov. Je potrebné dodať, že existujú aj komerčné riešenia požiadaviek laserového centra, ich cena je však príliš vysoká.

Boli realizované pokusy vizualizovať konfokálne dáta na grafických kartách pomocou programu `f3dvr` (Červeňanský, 2004), no výsledky boli neakceptovateľné. Prvým problémom bola veľkosť dát, ktoré je program schopný vizualizovať. Ako bude spomenuté neskôr, veľkosť konfokálnych dát presahuje aj 1 GB. Druhý problém opäť vyplýva z charakteristiky dát. Tie sú *veľmi* slabo navzorkované v z-tovom smere. Pri spracovaní grafickou kartou sa výsledok zobrazuje do istej roviny. Experimenty však ukázali stratu 3D vnemu skúmaného objektu a nižšiu kvalitu obrazu.

Výpočet na priestorových dátach je výpočtovo a pamäťovo veľmi náročný proces. S cieľom zobrazovať konfokálne dáta pokiaľ možno interaktívne, rozhodli sme sa zvoliť paralelný prístup s použitím PC klastra, ktorý sa nachádza v MLC.

1.3 Interaktívna vizualizácia priestorových dát

Ešte pred analýzou samotnej interaktívnej vizualizácie sa pozrime na význam niektorých pojmov. Nasledujúci odsek vychádza zo (Šrámek, 1998), (Bentum, 1996), (Žára, 1998). *Vizuali-*

[1] Medzinárodné laserové centrum: http://www.ilc.sk

záciu môžeme chápať ako grafickú reprezentáciu abstraktných údajov uložených väčšinou vo forme čísel alebo textu. Tabuľka s n vzorkami a im prislúchajúcimi hodnotami nám na prvý pohľad veľa nepovie. Ak však z týchto hodnôt vytvoríme graf, stratíme síce presnosť v podobe čísel, ale vieme oveľa viac povedať o rozmeroch a charakteristike údajov.

Volumetrickú (objemovú) vizualizáciu potom môžeme zadefinovať ako proces projekcie trojrozmerných dát na dvojrozmernú rovinu za účelom pochopenia štruktúry objektov obsiahnutých v dátach. V literatúre sa tiež môžeme stretnúť s pomenovaním *volumetrické renderovanie*.

Pod slovami *interaktívna vizualizácia* rozumieme vizualizáciu dát, ktorej parametre mení používateľ v reálnom čase. Požiadavka na interaktívnu vizualizáciu je pre nás kritická, nakoľko zobrazujeme objekty, o ktorých štruktúre často nemáme žiadne informácie. Zmeny by sa mali prejaviť okamžite, t.j. bez povšimnutia pozorovateľa. Pre plynulé zobrazovanie scény potrebujeme, aby náš program dokázal vykresľovať obraz rýchlosťou aspoň 25 snímok za sekundu (čo v praxi dosahujeme veľmi ťažko). Ako uvidíme neskôr, (interaktívna) volumetrická vizualizácia je výpočtovo veľmi náročná a ani paralelný prístup nevedie k uspokojivým výsledkom. Preto je potrebné používať špeciálne techniky a optimalizácie pri riešení tohto problému.

Volumetrická vizualizácia v reálnom čase má oproti klasickej statickej vizualizácii niekoľko výhod:

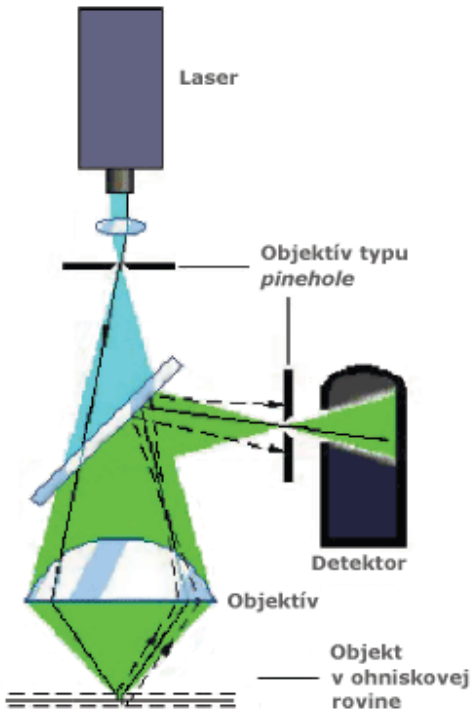
- **Ľubovoľný pohľad na scénu (pozícia kamery)**
  Používateľ môže ľubovoľne nastaviť pozíciu kamery a jej vzdialenosť od zobrazovaných objektov. Rotáciou objektu lepšie pochopíme tvar objektov v scéne.
- **Vyrezávanie**
  Nebudeme zobrazovať istú časť dát. Vďaka tomu sme schopní zobraziť vnútro objektov, resp. odstrániť tie oblasti, ktoré zakrývajú objekty nášho záujmu.
- **Modifikovanie vizualizačnej metódy**
  Používateľ môže počas behu programu meniť parametre projekčnej metódy.

1.4 Konfokálna mikroskopia

Základným princípom CLSM je osvetlenie vzorky laserovým lúčom, ktorý je zaostrený na jediný bod (Pawley, 1995). Následne sa vyžiarené svetlo (fluorescencia) premieta na apertúru, ktorá je umiestnená pred snímacím detektorom 1.1. Takto, bod po bode, konfokálna mikroskopia umožňuje objemové snímanie mikroštruktúr selektívnym zobrazovaním paralelných rezov. Snímky sa často získavajú pomocou špecifických fluorescenčných farbív, ktoré umožňujú zvýraznenie niektorých komponentov študovanej vzorky. Bodové osvetlenie a bodové snímanie vedú, v porovnaní s tradičnou mikroskopiou, k výrazne lepšiemu priestorovému rozlíšeniu. To znamená, že môžeme lepšie študovať rôzne hĺbkové vrstvy, pretože vo výslednom obrázku je výrazne potlačený vplyv hĺbkových vrstiev, ktoré sú mimo oblastí zaostrenia.

Najnovšie CLSM zariadenia umožňujú aj spektrálnu dekompozíciu emitovaného svetla prostredníctvom súčasného merania vo viacerých pásmach s nastaviteľnou frekvenciou (Dickinson a kol., 2001). Výsledný multispektrálny obraz takto predstavuje nielen priestorové (prípadne

aj časové) rozloženie intenzity, ale obsahuje informáciu aj o spektrálnej signatúre zobrazovaných objektov. Navyše, pokrok v oblasti fluorescenčného konfokálneho zobrazovania umožňuje aj súčasné meranie a identifikáciu dynamických zmien viacerých populácií fluorescenčnych molekúl v biologických vzorkách. Fluorescenčné značkovanie sa takto stáva základným nástrojom štúdia rôznych štrukturálnych alebo funkčných čŕt priamo v živých bunkách (Pawley, 1995). Viac informácii je dostupných na (http://www.microscopy.info/).

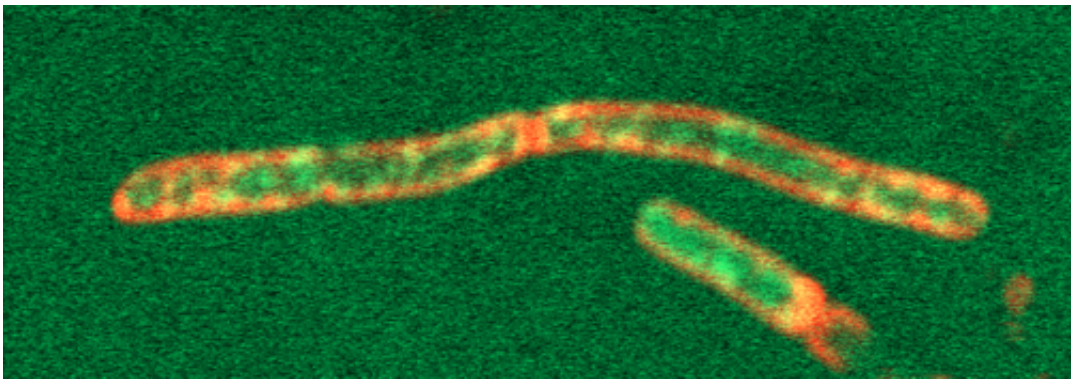


Obrázok 1.1: Schéma konfokálneho mikroskopu (Zdroj: Razdan a kol., 2001, s.3, modifikované).

1.4.1 Charakteristika CLSM dát

Cieľom vizualizácie biologických údajov je zobrazenie preparátu a jeho vnútornej štruktúry. *CLSM* dáta dostávame vo väčšine prípadov ako sériu dvojrozmerných obrázkov reprezentujúcich jednotlivé rezy preparátom. Takáto séria sa zvykne označovať ako *zásobník obrázkov*. Niekoľko málo rezov naskenovaného preparátu je zobrazených na obrázku 1.2. Pred jeho naplnením sa používateľ mikroskopu musí rozhodnúť, akú veľkosť obrázkov bude požadovať. Typické veľkosti obrázkov sú 256x256, 512x512, 1024x1024, no používateľ si môže nastaviť ľubovoľné rozmery. Obrázky v zásobníku sú zvyčajne zarovnané pozdĺž osí X,Y,Z. Farebná hĺbka je zvyčajne 8 alebo 12 bitov. Mikroskop dokáže produkovať nasnímané objekty až v 32 kanáloch, teda celkovo máme k dispozícii štvorrozmerné dáta. Tieto nastavenia ovplyvňujú celkovú veľkosť série (sérii v prípade viacerých kanálov. 2D obrázkov. Napríklad zásobník pozostávajúci z jednej série dvadsiatich 1024x1024 12-bitových obrázkov zaberie na diskovom poli približne 50 MB miesta. Hodnota pre daný pixel reprezentuje úroveň jasu, t.j. neobsahuje žiadne informácie o farebných zložkách (napríklad pre farebný model RGB). V praxi však pracujeme s oveľa väčšími dátami, ktorých veľkosť nezriedka presahuje 500 MB. Obrázky sú

hrubé, nespracované dáta, pričom sú všetky uložené na disku vo formáte LSM. Tento formát je v podstate viacstránkový TIFF formát (Adobe, 1992) s niekoľkými rozšíreniami. Viac informácii k tomuto formátu je možné nájsť v jeho špecifikácii (Zeiss). LSM súbor obsahujúci zásobník obrázkov bude vstupom pre náš program.



Obrázok 1.2: Príklad dvojkanálových konfokálnych dát.

1.5 Predspracovanie údajov

Ešte pred samotnou vizualizáciou dát je dôležité ich predspracovanie. Zásobník môže byť predspracovaný za účelom zlepšenia kvality obrázkov.

Jedna z najvyžadovanejších metód na zvýšenie kvality konfokálnych dát je *dekonvolúcia*. Jej aplikáciou dostávame obrázky s ostrejšou definíciou, lepším kontrastom a vylepšeným pomerom signál/šum. Dekonvolúcia je základom vysokých detailov pri zobrazovaní biologických štruktúr na bunkovej úrovni (Geert a kol, 1998). Dekonvolučné algoritmy sa používajú v tradičnej fluorescenčnej mikroskopii. V prípade konfokálnej mikroskopie je ich použitie stále náročné kvôli netriviálnemu určeniu prístrojovej funkcie (PSF) a obnovy obrázku pomocou získanej PSF. Do úvahy musíme brať aj veľké množstvo dát (2048x2048x256x32).

Pri CLSM dátach sem v obmedzenej miere spadá výpočet dodatočných rezov pomocou interpolácie vyššieho stupňa (rýchlosť v tomto kroku nie je dôležitá a celý výpočet sa uskutoční iba raz). Môžeme použiť rôzne filtre na celú množinu dát s cieľom odstrániť artefakty, vyhladiť alebo zostriť obraz, či upraviť kontrast a jas (Bentum, 1996). Je *dôležité* uvedomiť si, že aplikácia týchto filtrov radikálne ovplyvní priestorovú rekonštrukciu. Preto sa v niektorých prípadoch používajú iba za účelom zobrazenia a výpočty (kvantitatívne merania) sú aplikované na nespracované dáta.

Ešte pred 3D rekonštrukciou môžeme *vysegmentovať* objekty nášho skúmania – identifikovať zaujímavé voxely, ktoré neskôr zobrazíme. Pri zvyčajnej farebnej hĺbke 12 bitov môžeme zvyšné štyri bity z celkovo šestnástich bitov použiť na identifikáciu až šestnástich objektov v priestore (Bruckner, 2004). Túto dodatočnú informáciu využijeme počas generovania výsledného obrazu. Rôznym objektom priradíme rôznu farbu, aby sme zdôraznili oblasti nášho

záujmu (napr. telá článkov, ogranely). Dodávame, že v praxi je netriviálne stanoviť presné hranice medzi objektmi.

1.6 Problémy pri vizualizácii

V (Sakas a kol., 1996) autori zhrnuli do niekoľkých bodov charakteristiku CLSM dát a ťažkosti spojené s ich vizualizáciou.

1. Veľkosť dát. Kvôli veľkým rozmerom je nutné použiť efektívne metódy na spracovanie takýchto údajov.

2. Nízky kontrast, malé zmeny intenzít, zlý pomer signál/šum. Kontrast sa môže znižovať s rastúcou hĺbkou prenikajúceho svetla preparátom. Priamočiara segmentácia objektov záujmu od pozadia je prakticky nemožná, pretože všetky techniky (prahovanie, rozdiely farieb, homogénnosť regiónov) sú založené na *binárnom* rozhodnutí, bez ohľadu na to, či voxel do štruktúry patrí alebo nie.

3. Rôzne rozlíšenie v rovine XY a smere Z. Vizualizačná metóda musí pracovať s blokmi namiesto kubických voxelov. To prináša artefakty spojené s interpoláciou a zväčšenie dát v pamäti (tak, že ich nedokážeme spracovať na bežnom počítači).

4. Skúmame neznáme štruktúry. Vizualizačná metóda by mala potlačiť artefakty v čo najväčšom rozsahu. Pozorovateľ často nemá *skúsenosť* so skúmanou štruktúrou (čo je a čo nie je správne zobrazené), preto zobrazovanie artefaktov má veľmi zlý dopad na správne pochopenie tvaru objektu. Pomocou vhodne zvoleného osvetľovacieho modelu získame jasnú predstavu o objekte. Na mieste je tiež otázka celkovej rýchlosti vizualizácie. Jej parametre sa musia zisťovať interaktívne metódou *pokus-omyl*. Táto procedúra môže trvať veľmi dlhý čas, pretože na zobrazenie nového výsledku musí používateľ čakať aj niekoľko minút. Vo všeobecnosti prehliadka neznámeho objektu (potenciálne priestorovo veľmi komplikovaného) vyžaduje zmeny pozície a smeru kamery, osvetlenia a vizualizačných metód.

Za ostatné dôležité problémy vizualizácie konfokálnych dát spomeňme asymetrický tvar PSF nasledovaný poruchami 3D tvaru a útlm fluorescencie.

Kapitola 2

Volumetrická vizualizácia

Volumetrická vizualizácia je o pochopení komplexných viacrozmerných dát. (Bentum, 1996) vo svojej práci rozdelil vo všeobecnosti proces volumetrickej vizualizácie do troch krokov.

Prvým krokom predstavuje predspracovanie údajov.

Druhý krok závisí na použitej technike. Dvojrozmerné obrázky (rezy) sú namapované na trojrozmernú mriežku. Týmto vytvoríme 3D dáta.

- **Algoritmy zobrazujúce povrchy** (surface based techniques) vytvárajú pomocnú geometrickú reprezentáciu povrchu. V trojrozmerných dátach sa hľadajú hrany a body povrchu. Z nich sa interpoluje povrch dvojrozmernými záplatami. Ide o nepriamu metódu.
- **Objemové algoritmy** (volume based techniques) renderujú objemové dáta priamo – využívajú plnú priestorovú informáciu. Namiesto pomocnej geometrickej reprezentácie povrchu vieme každej vzorke priradiť farbu a nepriehľadnosť.

Tretím krokom je vygenerovanie výsledného obrazu. Pomocný povrch sa zobrazuje tradičnými metódami počítačovej grafiky, zatiaľ čo reprezentácia pomocou farieb a nepriehľadnosti sa zobrazuje pravým objemovým renderovaním (napríklad pomocou metódy sledovania lúča).

2.1 Vizualizačné metódy

Existuje mnoho vizualizačných metód, ktoré sa v praxi používajú s rôznym úspechom. Môžeme ich rozdeliť podľa viacerých kritérií (Elvins, 1992). Jedno delenie je však spoločné, a to rozdelenie na *algoritmy zobrazujúce povrchy* a *objemové algoritmy*.

2.1.1 Algoritmy zobrazujúce povrchy

Tieto metódy sa snažia z objemových dát aproximovať povrch pomocou geometrických primitív a tie zobrazovať pomocou dobre známych metód počítačovej grafiky. Najznámejšie algoritmy sú:

- sledovanie obrysov (Keppel, 1975)
- pochodujúce kocky (marching cubes) (Lorensen a kol., 1987)

- marching tetrahedra (Shirley a kol., 1990)
- rozdelenie kociek (dividing cubes) (Cline a kol., 1988)
- povrchové kocky (opaque cubes, cuberille) (Herman a kol., 1979)

Prehľad s popisom algoritmov možno nájsť v (Šrámek, 1998) a (Elvins, 1992). Algoritmy vytvárajú pomocnú geometrickú reprezentáciu povrchu, ktorú potom vieme rýchlo, vďaka redukcii dát, zobraziť štandardnými grafickými akcelerátormi. Informácie o vnútrajšku objektu sa však stratia. Vo všeobecnosti sa v týchto metódach robí pre každú vzorku test príslušnosti k povrchu objektu. Pri slabo navzorkovaných dátach alebo dátach obsahujúcich amorfné objekty (hmla a iné) preto vznikajú chyby. Tie sa vyskytujú buď vo forme dier, ktoré neexistujú v pôvodnom objekte, alebo vo forme falošných stien.

2.1.2 Objemové algoritmy

Oproti predchádzajúcim algoritmom, objemové algoritmy využívajú plnú priestorovú informáciu na vygenerovanie výsledného obrazu a nezávisia od zložitosti scény. Každej vzorke je priradená farba a nepriehľadnosť. Tieto údaje sú neskôr zložené do výslednej farby obrazu. Na výpočet používajú celú trojrozmernú mriežku dát, sú teda veľmi náročné na pamäť a procesor počítača. Každé vyrenderovanie výsledného obrazu si vyžaduje traverzovanie objemom dát. Dokážeme však zobraziť ľubovoľný vnútorný detail a amorfné objekty. Celkovo tieto algoritmy poskytujú viac informácií ako algoritmy používajúce pomocnú reprezentáciu povrchu.

Kvôli lepšej interaktivite boli predstavené mnohé urýchlenia a optimalizácie. Náhodné vzorkovanie dát alebo prechod v nízkom rozlíšení sa niekedy používa na rýchle vygenerovanie obrázkov v zlej kvalite. Ak používateľ nemení parametre zobrazovacej metódy, znovu vygenerujeme obraz, tento krát vo vyššom rozlíšení. Tento proces postupného zvyšovania rozlíšenia a kvality výsledného obrazu sa nazýva *postupné zlepšovanie* (progressive refinement). Iná možnosť je použitie špeciálneho hardware. Prehľad možných urýchlení spomíname v časti 2.4 alebo konkrétne v kapitole 4.

Objemové algoritmy môžeme rozdeliť podľa funkcie, ktorou robíme klasifikáciu dát na (Šrámek, 1998):

- *binárne*
- *pravdepodobnostné*

Binárne algoritmy pokrývajú každý voxel (nejakým objektom) buď úplne, alebo vôbec. Sú to povrchovo orientované algoritmy.

Pravdepodobnostné algoritmy (tiež známe pod názvom polopriehľadné renderovacie algoritmy) priraďujú voxelom percentuálny podiel nejakého objektu. Sú založené na nahromaďovaní príspevkov od všetkých vzoriek pozdĺž lúča do jedného obrazového pixela. Veľa autorov sa pri objemovom zobrazovaní obmedzuje iba na túto skupinu algoritmov. Pravdepodobnostné algoritmy patria pod objemovo orientované techniky.

Ďalšie dôležité rozdelenie je podľa poradia, v akom sú voxely postupne spracované pri tvorbe obrazu. Inými slovami, je to delenie podľa domény algoritmu (Zuiderveld, 1995).

Algoritmy pracujúce v obrazovom priestore

Pre každý pixel výsledného obrazu sa hľadajú voxely v objektovom priestore, ktoré prispievajú do výslednej farby bodu. Nakoľko pozície týchto voxelov obyčajne padnú mimo vrcholy mriežky, ich hodnotu musíme zisťovať interpoláciou. Do tejto skupiny môžeme zaradiť nasledujúce algoritmy:

- Trasovanie lúčov (ray-casting, ray-tracing) (Levoy, 1988)
- Sábelová metóda (Sabella, 1988)

Algoritmy pracujúce v objektovom priestore

Pre každý voxel objektového priestoru sa hľadajú pixely výsledného obrazu, ktoré daný voxel ovplyvní. Splatting je technika traverzujúca objektový priestor. Na každý voxel je aplikovaná konvolúcia s 3D rekonštrukčným filtrom. Príspevok filtrovaných bodov sa nahromaďuje do obrazového priestoru. Fungovanie algoritmu si môžeme predstaviť ako hádzanie snehovej gule na sklenenú plochu. Príspevok snehu v strede dopadu bude najvyšší a postupne bude klesať s rastúcou vzdialenosťou od miesta dopadu. Algoritmy:

- V-buffer (Upson, 1989)
- Splatting (Westover, 1990)

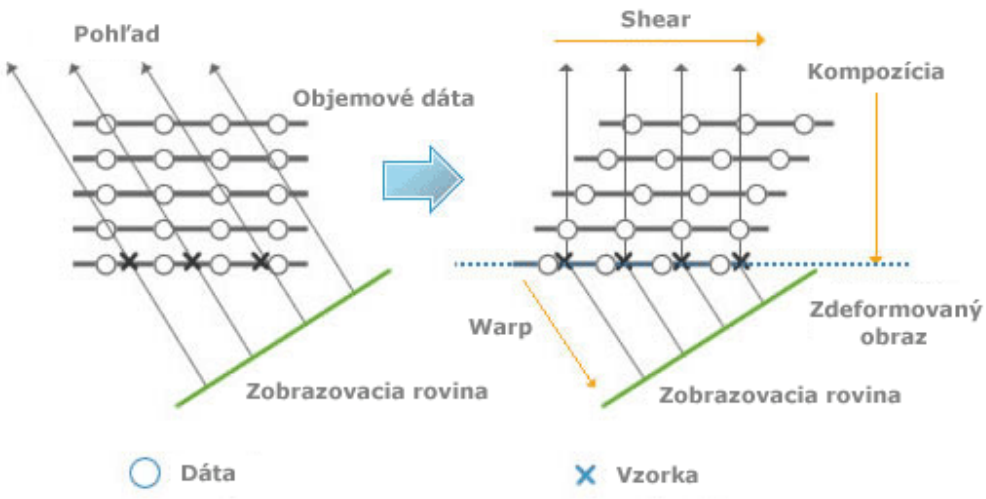
Algoritmy pracujúce na hybridnom princípe

Existujú aj *hybridné* algoritmy, ktoré kombinujú výhody oboch predchádzajúcich prístupov. Príkladom je algoritmus *Shear-warp – Posuň a pokriv* (Lacroute a kol., 1994), ktorý je považovaný za najrýchlejší, čisto softvérový renderovací algoritmus (Bruckner, 2004). Myšlienka algoritmu spočíva v *posunutí* (shear) jednotlivých rezov v objektovom priestore tak, aby ich namapovanie (kompozícia) na 2D zobrazovací priestor bolo jednoduché a rýchle – vzorky pozdĺž lúčov budú ležať presne v rovinách rezov. Traverzovanie dátami vyžaduje jednoduché adresovanie a 2D prevzorkovací filter. Po projekcii zdeformovaný obraz *pokrivíme* naspäť (warp). Jednotlivé kroky algoritmu lepšie ilustruje obrázok 2.1.

Algoritmus je veľmi rýchly aj na bežne dostupných počítačoch. Na druhej strane však produkuje artefakty, je pohľadovo závislý a kvalita výsledného obrazu je nízka. Nižšia kvalita výsledného obrazu je spôsobená (Bentum, 1996):

- Algoritmus obsahuje až dva kroky prevzorkovania. Viacnásobné prevzorkovanie môže spôsobiť rozmazanie a stratu detailných informácii.
- Rekonštrukčný filter je iba dvojrozmerný. V rezoch sa používa iba bilineárna interpolácia, medzi rezmi iba interpolácia najbližším susedom. Tento bod je hlavnou nevýhodou algoritmu shear-warp.
- Počet lúčov je rovný počtu voxelov jedného rezu, algoritmus produkuje alias kvôli podvzorkovaniu.

Boli predstavené niektoré optimalizácie (napr. použitie min-max oktálových stromov), ale výsledná kvalita obrazu ešte stále zaostáva napr. za algoritmom ray-casting.



Obrázok 2.1: Postupnosť krokov algoritmu shear-warp (Zdroj: MEViSYS. [online]. Dostupné na: <http://www.mevisys.com/html/3d/over view.html>, modifikované).

2.1.3 Porovnanie vizualizačných metód

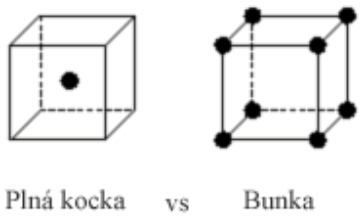
V dnešnej dobe je snaha vyhnúť sa algoritmom zobrazujúcim povrchy a používať objemové algoritmy. Tieto vieme vyladiť, aby výsledok bol rovnaký ako pri algoritmoch používajúcich pomocnú reprezentáciu povrchu. Rýchle algoritmy ako Shear-warp produkujú obrazy v nízkej kvalite. Binárne objemové algoritmy sú ľahko implementovateľné a nepotrebujú veľa pamäte v porovnaní s pravdepodobnostnými. Trpia však nedostatkami podobne ako algoritmy zobrazujúce povrchy. Pri binárnej klasifikácii, resp. testovaní, či vzorka patrí povrchu, môžu vzniknúť falošné povrchy alebo neexistujúce diery. Ray-casting a splatting dávajú rovnako kvalitné výsledky. Čas vyrenderovania výsledného obrazu závisí od typu dát a klasifikácie (prechodovej funkcie). Implementácia ray-castingu je priamočiara, ale zahŕňa zdĺhavé prevzorkovanie. Na algoritmy pracujúce v objektovom priestore je často potrebné aplikovať antialiasingové techniky. Túto činnosť vieme ľahšie implementovať pri ray-castingu. Jeho schopnosť zobrazovať s celým objemom dát, priamočiara implementácia, urýchlenia a ďalšie dôvody, ktoré budú spomenuté neskôr, (jednoduchá paralelizovateľnosť) nás utvrdili v rozhodnutí implementovať tento algoritmus v našom programe. Dobré zhrnutie a prehľad algoritmov pre volumetrickú vizualizáciu možno nájsť v (Elvins, 1992).

2.2 Základy

V tejto časti sú obsiahnuté niektoré informácie týkajúce sa súradnicových systémov, klasifikácie, tieňovania, princípov prevzorkovania, interpolácie a kompozície. Tieto techniky sú základom metódy *vrhania lúča* (ray-casting). Nasledujúce časti vychádzajú z (Žára, 1998), (Bentum, 1996), (Elvins, 1992).

2.2.1 Viacrozmerná reprezentácia dát

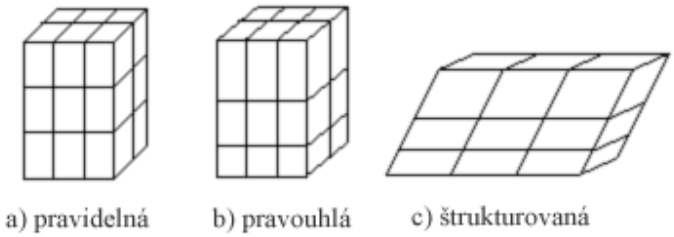
Ako už bolo spomenuté vyššie, údaje (sada vzoriek), s ktorými pracujeme, chápeme ako zásobník rezov. Každý rez je v skutočnosti dvojrozmerné pole čísel reprezentujúce istú hodnotu (napr. intenzitu). Celý zásobník predstavuje trojrozmerné pole elementárnych objemových jednotiek, spoločne nazývané *voxely* (voxel = objemový element, analógia dvojrozmerného pixelu). Má štruktúrovanú podobu a je reprezentovaný pravidelnou alebo nepravidelnou *mriežkou*.



Obrázok 2.2: Rôzna interpretácia voxelov.

Ako interpretovať voxely? Môžeme ich chápať ako plnú kocku konštantnej hodnoty alebo body mriežky (obrázok 2.2). V ďalšom texte budeme používať druhú interpretáciu, t.j. hodnoty poznáme iba vo vrcholoch mriežky. Obrázky vygenerované týmto prístupom sú *hladšie* ako pri použití prvého spôsobu. Osem vzoriek vytvára *bunku* (cell). Vnútorné hodnoty bunky určujeme pomocou interpolácie medzi jej vrcholmi. Najčastejšie sa používa interpolácia prvého rádu (napr. trilineárna inerpolácia), menej častejšie interpolácia vyššieho rádu (napr. lagrangová interpolácia).

Objemové dáta nemusia byť nutne definované iba ako karteziánska mriežka. Speray a Kennon (Speray a kol., 1990) rozdelili všetky mriežky podľa ich geometrického tvaru do siedmich tried : *karteziánska, pravidelná, pravouhlá, štruktúrovaná, neštruktúrovaná, blokovo-štruktúrovaná a hybridná mriežka.*
Príklady niektorých dátových mriežok sú na obrázku 2.3. V ďalšom texte sa budeme zaoberať prevažne pravidelnými mriežkami.



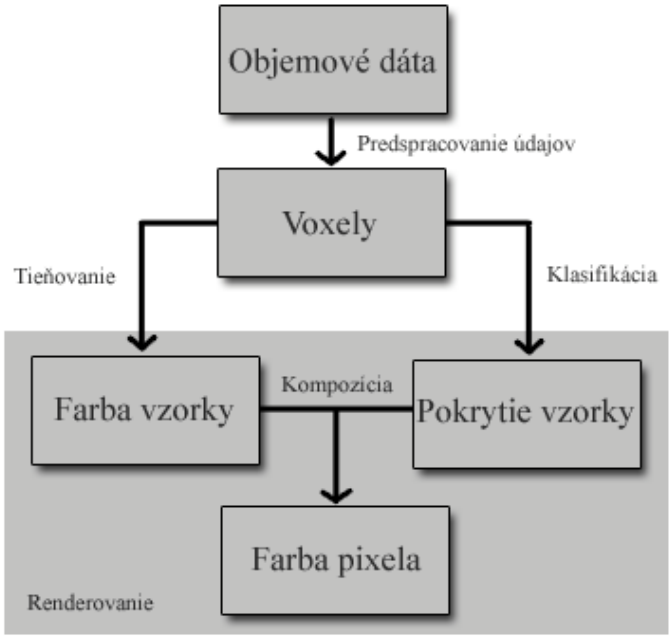
Obrázok 2.3: Príklady dátových mriežok.

Na údaje v trojrozmernom priestore a hodnoty pixelov na obrazovke sa budeme odkazovať súradnicovými systémami. Predpokladajme, že voxely ležia vo vrcholoch trojrozmernej mriežky v pravotočivej súradnicovej sústave, označovanej ako *objektový priestor*. Voxel sa nachádza

na súradniciach (x, y, z). Súradnicový systém obrazu, ktorý bude vygenerovaný, označujeme ako *zobrazovací priestor*.

Pre lepšie pochopenie samotnej vizualizácie sa pozrime trochu bližšie na *renderovaciu pipeline*. Na tomto mieste je nutné poznamenať, že tento jednoduchý popis pipeline sa vzťahuje na algoritmus ray-casting, ktorý spomenieme neskôr. Vstupom pre náš program je trojrozmerné pole voxelov, reprezentované 3D mriežkou. Keďže poloha kamery môže byť v ľubovoľnom bode priestoru, sme nútení urobiť *prevzorkovanie* (vygenerovanie adresy + interpolácia), pretože voxely, ktoré budú prispievať do výsledného obrazu, nemusia ležať presne vo vrcholoch mriežky. Vygenerovanie adresy chápeme ako určenie pozície v objektovom priestore práve odoberanej vzorky pozdĺž vrhnutého lúča do scény. Počas kompozície priradíme každému pixelu výsledného obrazu hodnotu zloženú z farby a nepriehľadnosti príslušných voxelov (presnejšie tých, ktoré pretne lúč na svojej ceste). Výsledný obraz namapujeme (otočíme, posunieme, zaškálujeme) na obrazovku.

Na obrázku 2.4 je blokovo znázornený algoritmus ray-casting. V ďalších častiach budeme podrobne hovoriť o operáciách ako klasifikácia, tieňovanie a kompozícia. Okrem toho spomenieme prevzorkovanie a traverzovanie objemových dát. Predspracovanie údajov bolo spomenuté už v prvej kapitole.



Obrázok 2.4: Diagram algoritmu ray-casting.

2.2.2 Klasifikácia

Klasifikácia je proces priradenia farby a nepriehľadnosti zrekonštruovaným vzorkám. Na tento účel sa obyčajne používa *transfér funkcia*. Vo väčšine prípadov býva implementovaná pomocou tabuliek alebo napr. ako Bezierova krivka. Pri interaktívnej vizualizácii požadujeme možnosť zmeny transfér funkcie počas vykonávania programu. Na jej rozumné nastavenie má vplyv niekoľko faktorov: používateľova znalosť materiálu, pozícia a množstvo elementov v objektovom priestore. Za najťažšie sa považujú nastavenia, ktoré musí vykonať používateľ. Ak je oboznámený s materiálom, špecifikácia transfér funkcie je relatívne jednoduchá a rýchla. V opačnom prípade tento krok zahŕňa rozsiahle skúmanie materiálu, prípadne konzultácie s osobou, ktorá sa v charakteristike materiálu vyzná.

Predbežná a dodatočná klasifikácia

Objemové algoritmy sa tiež odlišujú spôsobom, akým vyčísľujú farbu a nepriehľadnosť vzorky. Hodnotu vzorky vo vnútri bunky vieme určiť interpoláciou. Poradie interpolácie a aplikácie transfér funkcie nám definuje nasledujúce rozdelenie renderovacích techník.

Predbežná klasifikácia je aplikácia transfér funkcie ešte *pred* interpoláciou, t.j. farba a nepriehľadnosť sa vypočítajú vo vrcholoch mriežky počas fázy predspracovania údajov. Počas renderovania sa farba a nepriehľadnosť vzorky určí interpoláciou týchto údajov. *Dodatočnú klasifikáciu* chápeme ako aplikáciu transfér funkcie na hodnotu vzorky *po* jej interpolácii z ôsmich hodnôt bunky.

Predbežná a dodatočná klasifikácia vo väčšine prípadov vyprodukuje rôzne výsledky. Dodatočná klasifikácia je *správna* v zmysle aplikovania transfér funkcie na spojitý priestor skalárnych hodnôt reprezentovaný mriežkou (Engel a kol., 2001). Predbežná klasifikácia sa od nej neodlišuje iba v prípade, keď je transfér funkcia konštantá alebo identita.

2.2.3 Traverzovanie

Ako bolo spomenuté vyššie, algoritmus ray-casting spadá do skupiny *image-order* algoritmov, t.j. algoritmov pracujúcich v obrazovom priestore. Postupne prechádzame všetky body (pixely) zobrazovacieho priestoru (obrazovky) v usporiadaní *rozkladových riadkov (scan-line order)*[1].

2.2.4 Tieňovanie

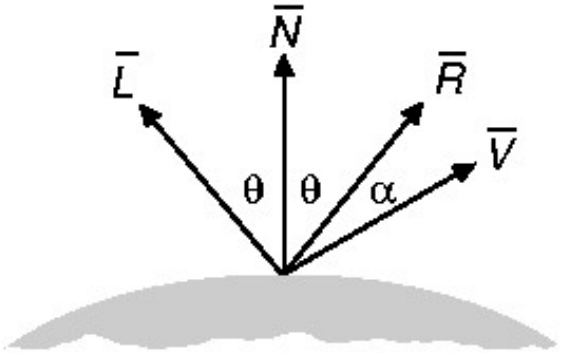
Tieňovanie je dôležitá súčasť volumetrickej vizualizácie. Umožňuje vyprodukovať výzorovo realistické výsledky. Svetlo na ceste od zdroja mení svoje farebné zloženie (pri odraze resp. lome). Po dopade na povrch telesa sa svetlo rozptýli do všetkých smerov. Matematická funkcia, ktorá vyjadruje intenzitu lúča rozptýleného svetla v závislosti od jeho smeru, intenzity a vlnovej dĺžky dopadajúceho svetla, sa nazýva *odrazová funkcia* a je základom *osvetľovacích modelov*. Čím máme lepší osvetľovací model na popis chovania svetla, tým realistickejšie výsledky dostávame. V počítačovej grafike sa používajú dva prístupy.

Prvý prístup vychádza z fyzikálnej podstaty šírenia svetla a takýto model nazývame *globálny osvetľovací model*. Berie do úvahy vzájomné vzťahy medzi objektmi v priestore a dáva

[1] preklad slova scan-line podľa J.Žára - Moderní počítačová grafika

kvalitné výsledky. V počítačovej grafike však patrí medzi výpočtovo najnáročnejšie problémy.

Druhý prístup sa snaží tento fyzikálny dej obísť (presnejšie priblížiť) pomocou umelo vytvorených osvetľovacích modelov nesúcich spoločný názov *lokálne osvetľovacie modely*. Tieto modely dávajú dobré výsledky, ktoré sa približujú skutočnému chovaniu svetla a sú výpočtovo oveľa jednoduchšie ako globálne osvetľovacie metódy. Ich výpočet je založený na normále povrchu v bode dopadu svetla, pozície svetelného zdroja a pozorovateľa. Vo väčšine algoritmov



Obrázok 2.5: Základné vektory a uhly použité v lokálnom osvetľovacom modely.

zobrazujúcich objem sa nepoužívajú globálne modely. Hlavným dôvodom je potreba rýchlo, resp. interaktívne zobrazovať skúmaný objekt. Niekedy použitie globálneho osvetľovacieho modelu môže mať negatívny efekt vo forme nesprávnej interpretácie zobrazeného objektu. Preto sa v tejto práci budeme ďalej zaoberať iba lokálnymi osvetľovacími modelmi.

Pre väčšinu aplikácii postačuje známy Phongov osvetľovací model (Phong, 1975). Odrazené svetlo pozostáva z troch zložiek:

- *Ambientná zložka* I_A reprezentuje okolité svetlo v danej oblasti prichádzajúce zo všetkých smerov, ktoré je príliš zložité modelovať. Táto konštantná zložka závisí od intenzity okolitého svetla I_a a konštanty k_a, ktorá vyjadruje schopnosť povrchu odrážať okolité svetlo ($0 \leq k_a \leq 1$).

- *Difúzna zložka* I_D reprezentuje odrazené svetlo. Nezávisí na polohe pozorovateľa a pravdepodobnosť nového smeru lúča je rovnaká pre všetky smery. Veľkosť difúznej zložky závisí na uhle dopadu a k_d koeficiente difúzneho odrazu ($0 \leq k_d \leq 1$). Difúzne svetlo prináša informácie o farbe povrchu.

- *Zrkadlová zložka* I_S reprezentuje odlesk na povrchu telesa. Závisí od polohy pozorovateľa, vektora R symetrického k vektoru dopadu podľa normály a exponentu n, vyjadrujúceho ostrosť zrkadlového odrazu. Odlesk môže mať inú farbu ako povrch telesa. (Schlick, 1994) prišiel s výbornou a rýchlou aproximáciou zrkadlovej zložky pre nízky exponent n. Pri výpočte sa používa iba jedno sčítanie, odčítanie, násobenie a delenie.

Celkové svetlo vnímané pozorovateľom teraz môžeme vyjadriť ako

$$I = I_A + I_D + I_S \tag{2.1}$$

Nech I_l je jednofarebné svetlo. Odrazené svetlo vypočítame upraveným Phongovým osvetľovacím modelom pomocou Schlickovej aproximácie (Zuiderveld, 1995):

$$I = I_a k_a + I_l\Big(k_d MAX(L \cdot N, 0) + k_s \frac{t}{n - nt + t}\Big) \tag{2.2}$$

Koeficient t vyrátame ako $t = MAX(N \cdot H, 0)$ kde $H = \frac{(L+V)}{|L+V|}$, $n \in [1, \infty]$. Z predchádzajúceho textu vieme ako určiť farbu bodu osvetlenej plochy. Výpočet pre všetky body plochy by bolo veľmi časovo náročne, preto vznikli metódy na urýchlenie tohto výpočtu. Tieto metódy sa nazývajú *tieňovanie*. Pomocou osvetľovacieho modelu určíme farbu iba v niektorých bodoch plochy a zvyšné odvodíme. V praxi sa zvyčajne používa *Gouraudovo* a *Phongovo* tieňovanie. Gouraudovo tieňovanie je založené na interpolácii farieb (Gouraud, 1971). Phong navrhol interpolovať normálové vektory. Toto tieňovanie dáva lepšie výsledky ako Gouraudovo, je však výpočtovo náročnejšie.

Normálu imaginárneho povrchu prechádzajúcim bodom bunky aproximujeme pomocou *gradientu*.

$$N(x_i, y_j, z_k) = \frac{\nabla f(x_i, y_j, z_k)}{|\nabla f(x_i, y_j, z_k)|} \tag{2.3}$$

Výpočet gradientu $\nabla f(x_i, y_j, z_k)$ je časovo náročná operácia. Na jeho určenie sa často zvykne používať vzorec *centrálnej diferencie*:

$$\nabla f(x_i, y_j, z_k) \approx \tag{2.4}$$
$$\frac{1}{2}\Big(f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k),$$
$$f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k),$$
$$f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})\Big).$$

Tento výpočet prvého stupňa (6 susedov) je citlivý na šum a alias artefakty. Použitie výpočtu druhého stupňa (26 susedov) dáva lepšie výsledky. S cieľom dosahovať interaktívne rýchlosti sa používajú aj analytické metódy určenia gradientu, kde meníme kvalitu výsledku za rýchlosť výpočtu.

2.2.5 Prevzorkovanie a interpolácia

Prevzorkovanie (resampling) je proces transformácie diskrétneho signálu (napr. obrázku) z jednej sústavy súradníc do druhej. Jeho hlavné použitie je pri navzorkovaní trojrozmerných dát za účelom ich projekcie na dvojrozmernú rovinu. Prevzorkovanie pozostáva z dvoch krokov – *vygenerovania adresy* a *interpolácie*. Vygenerovanie adresy nám určuje pozíciu vzorky v mriežke (dátach). Interpolácia je nutná operácia, pretože pozície, odkiaľ chceme získať hodnotu väčšinou, nepadnú do rohov mriežky (voxelov). Spomeňme jeden poznatok z teórie signálov. Frekvenčne obmedzená funkcia, ktorá spĺňa *Nyquistovo kritérium*, môže byť presne zrekonštruovaná použitím ideálneho rekonštrukčného filtra $sinc(x)$. Funkcia *sinc* je definovaná na nekonečnom intervale, preto potrebujeme nekonečne veľa čísel na zrekonštruovanie medzivzorky. Obraz má ale konečne veľa bodov, takže na rekonštrukciu musíme použiť iné funkcie:

- Najbližší sused
- Lineárna interpolácia
- Interpolácia vyššieho stupňa (orezaný sinc, kubické splajny)

Interpolácia vyššieho stupňa dáva lepšie výsledky za cenu pomalšieho výpočtu. Najjednoduchšia interpolačná funkcia je interpolácia nultého stupňa, známa pod menom interpolácia hodnotou najbližšieho suseda. Hodnota vzorky je hodnotou najbližšieho vrchola bunky (voxelu) vzhľadom na jej pozíciu. Interpolácia prvého stupňa – trojrozmerná lineárna interpolácia (*trilineárna interpolácia*) pri výpočte berie do úvahy hodnoty všetkých ôsmich susedov vzorky v bunke. Nech pozícia bodu b v bunke je $(x, y, z)^T$. Označme $v0, \ldots, v7$ hodnoty ôsmich vrcholov bunky. Potom hodnotu bodu b zrekonštruujeme pomocou vrcholov bunky nasledujúcim vzťahom :

$$\begin{aligned} b \quad = \quad & v0(1-x)(1-y)(1-z)+ \\ & v1(x)(1-y)(1-z)+ \\ & v2(1-x)(y)(1-z)+ \\ & v3(1-x)(1-y)(z)+ \\ & v4(x)(1-y)(z)+ \\ & v5(1-x)(y)(z)+ \\ & v6(x)(y)(1-z)+ \\ & v7(x)(y)(z) \end{aligned}$$

2.2.6 Kompozícia

Po prevzorkovaní obsahujú všetky vzorky interpolovanú hodnotu z ôsmich voxelov bunky, a teda každej vzorke vieme priradiť farbu a nepriehľadnosť. Vysoká hodnota nepriehľadnosti je priradená jasne viditeľným objektom, malé hodnoty patria priehľadným materiálom. Skombinovaním týchto hodnôt získame výslednú farbu pixelu. Inak povedané, *kompozíciou* vyprodukujeme dvojrozmernú projekciu takýchto údajov. Intenzita svetla prechádzajúceho priehľadnými materiálmi je postupne tlmená. Objekt môže emitovať svetlo určitej intenzity. Volumetrickú kompozíciu môžeme implementovať dvoma spôsobmi.

Odzadu-dopredu (back-to-front, BTF)
Kompozícia začína najvzdialenejšími vzorkami a postupuje smerom k pozorovateľovi. Predpokladajme, že máme postupnosť objektov o intenzite I_i a priehľadnosti P_i označených vo vzrastajúcom poradí od najvzdialenejších objektov k najbližším vzhľadom na pozorovateľa. Intenzita pozadia je $I_0 = N_0$. N_i reprezentuje nahromadenú intenzitu počas kompozície. Jeden krok v metóde BTF môžeme teraz zapísať ako (Bentum, 1996):

$$N_i = P_i N_{i-1} + I_i \tag{2.5}$$

Hodnota N_n výsledného obrazu je daná vzťahom:

$$N_n = \sum_{k=0}^{n} I_k \prod_{l=k+1}^{n} P_l \tag{2.6}$$

(Levoy, 1988) podal iný pohľad na kompozíciu (BTF), princíp však zostáva ten istý. Označme symbolom α priehľadnosť a F farbu. Ďalej označme:

$$I_i = \alpha_i F_i, P_i = 1 - \alpha_i \text{ a } N_i = F_{out,i} = F_{in,i+1} \tag{2.7}$$

Z 2.7 a 2.5 dostávame:

$$F_{out,i} = (1 - \alpha_i)F_{in,i} + \alpha_i F_i \tag{2.8}$$

Odpredu-dozadu (front-to-back, FTB)
Kompozícia pracuje v opačnom poradí ako pri technike BTF. Postupne prechádzame objekty smerom od pozorovateľa a počítame nahromadenú intenzitu. Pri tomto postupe si musíme udržiavať v pamäti nahromadenú nepriehľadnosť:

$$\begin{aligned} \alpha_n &= \alpha_n + \alpha_v(1 - \alpha_n) \\ F_n &= F_n + F_v\alpha_v(1 - \alpha_n) \end{aligned} \tag{2.9}$$

F_n a α_n reprezentujú nahromadenú farbu, resp. nepriehľadnosť, F_v a α_v farbu a nepriehľadnosť príslušnej vzorky.

Kompozíciu robíme pozdĺž lúča prechádzajúceho objektovým priestorom. Výsledná farba je priradená pixelu na obrazovke. Ak $\alpha(I, J, K)$ je priehľadnosť vzorky so súradnicami (I, J, K) a $F(I, J, K)$ je farba, potom farbu v obraze na pozícii (I, J) vieme vyrátať ako:

$$F(I, J) = \sum_{k=1}^{n} \alpha(I, J, k)F(I, J, k) \prod_{l=1}^{k-1}(1 - \alpha(I, J, l)) \tag{2.10}$$

Lúč prechádza objektmi a akumuluje farbu dokiaľ neopustí dáta alebo nahromadená nepriehľadnosť dosiahla takú úroveň, pri ktorej sa farba lúča stabilizovala. Druhá podmienka má za následok značné urýchlenie renderovania, pretože odoberanie vzoriek pozdĺž lúča sa ukončí, ako náhle nahromadená nepriehľadnosť dosiahla istú prahovú hranicu. Táto technika sa označuje ako *predčasné ukončenie lúča* (adaptive alebo early ray termination). Levoy označil prahovú hodnotu ako 0.95 (príspevky ďalších vzoriek sú už zanedbateľné vzhľadom na výslednú kvalitu obrazu) (Levoy, 1990).

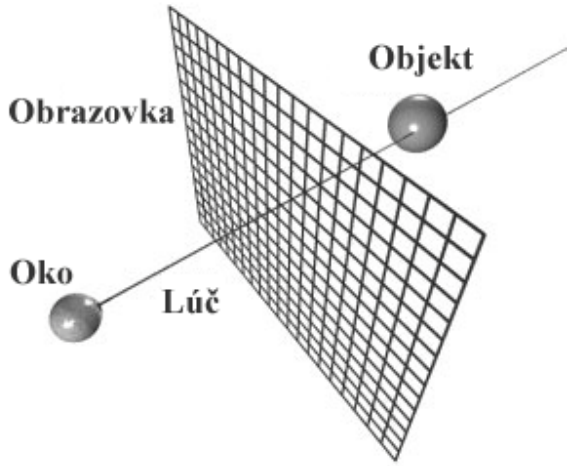
Všetky spomenuté vzťahy platia iba v prípade odoberania vzoriek z rovnako vzdialených pozícií o dĺžke 1. Ak zmeníme vzdialenosť medzi odoberanými vzorkami, musíme zmeniť aj nepriehľadnosť vzoriek. Ak by sme nechali pôvodnú nepriehľadnosť, zoberme prípad $vzdialenosti < 1$, zobrazený objekt by bol menej priehľadný (viac vzoriek prispeje k výslednej nepriehľadnosti bodu). Kompozícia je nelineárny proces, preto si nevystačíme s vydelením hodnoty nepriehľadnosti navzorkovaným pomerom. Predpokladajme rovnaké vzdialenosti medzi vzorkami. Úpravu nepriehľadnosti dosiahneme vzťahom:

$$\alpha' = 1 - (1 - \alpha)^{\Delta s} \tag{2.11}$$

kde α' je opravená nepriehľadnosť, α je pôvodná nepriehľadnosť a Δs je vzdialenosť medzi vzorkami. Avšak tento vzťah platí iba v homogénnych materiáloch, no ako bude spomenuté neskôr (3.2), vďaka korektnosti kompozície časti lúčov je všetko v poriadku – nehomogénne materiály vieme rozdeliť na niekoľko homogénnych a skombinovať ich.

2.3 Sledovanie lúča – Raycasting

Metóda sledovania lúča je najčastejšie používaný algoritmus pre generovanie vysoko kvalitných obrázkov pracujúci v obrazovom priestore (obrázok 2.6). Pre každý pixel zobrazovacieho pries-



Obrázok 2.6: Metóda sledovania lúča (Zdroj: The Recursive Ray Tracing Algorithm. [online]. Dostupné na: <http://www.geocities.com/jamisbuck/ raytracing.html>, modifikované).

toru vrháme do objektového priestoru lúč. Pozdĺž lúča zbierame vzorky. Pre každú pozíciu vzorky vypočítame farbu a nepriehľadnosť pomocou tieňovania, resp. klasifikácie. Výsledná farba pixelu je potom určená kompozíciou vzoriek pozdĺž príslušného lúča. Popis základného algoritmu možno nájsť v (Levoy, 1988). Ray-casting zdedil väčšinu výhod a nevýhod objemových algoritmov (direct volume rendering methods). Je veľmi náročný na pamäť a procesor počítača, no obrázky sú kvalitné a zobrazujú celý objem dát, nie len množinu tenkých povrchov ako v prípade algoritmov zobrazujúcich povrchy (surface-fitting algorithms). Ray-casting je paralelizovateľný na úrovni pixelov (lúče sú navzájom nezávislé) a dát (rozdeliť dáta na menšie časti, aplikovať algoritmus a výsledky skombinovať). Druhý prístup sa zvykne označovať ako *kompozícia čiastkových lúčov* (partial ray composition).

Gradient ∇f, farbu F a nepriehľadnosť α vieme vypočítať vo vrcholoch trojrozmernej mriežky reprezentujúcej objemové dáta už vo fáze predspracovania. Potom údaje na pozícii vzorky vieme určiť interpoláciou hodnôt v rohoch príslušnej bunky. Tento postup vedie k horšej kvalite obrazu v porovnaní s priamym počítaním týchto hodnôt počas renderovania. Taktiež je nezanedbateľná veľkosť a čas napĺňania týchto pomocných dátových štruktúr. Nároky na pamäť môžu stúpnuť až niekoľkonásobne. Náš program by mal umožňovať interaktívnu zmenu polohy zdroja svetla a transfér funkcie. Použitie predrátaných hodnôt v pamäti počítača by znamenalo ich kompletné prepočítanie pri každej zmene polohy, resp. funkcie, čo je v protiklade s požiadavkou na interaktívne ovládanie.

2.4 Zvyšovanie kvality obrazu a rýchlosti

Základná priamočiara implementácia ray-castingu je výpočtovo náročná. Stále rýchlejší hardvér, a dokonca ani paralelizmus spomenutý nižšie, neprináša uspokojivé výsledky – zatiaľ. V tejto časti spomenieme niektoré techniky urýchľujúce algoritmus.

2.4.1 Urýchlenie ray-castingu

Stratégie urýchľujúce algoritmus ray-casting sú založené napríklad na predpočítaní parametrov dát ešte pred samotnou kompozíciou, súvislých oblastiach v objemových dátach alebo výslednom obrázku (adaptívne vzorkovanie). Prípadne *vymeníme* kvalitu vygenerovaného obrázku za rýchlosť. V nasledujúcich bodoch spomenieme najdôležitejšie prístupy.

1. **Predspracovanie údajov**
   Niekoľko krokov renderovacieho algoritmu môže byť vypočítaných vo fáze predspracovania. Napríklad (Levoy, 1988) vytieňoval všetky voxely mriežky pred samotným algoritmom ray-casting. Predspracovanie so sebou väčšinou prináša drobné artefakty, najčastejšie sa týkajú nesprávneho tieňovania.

2. **Predčasné ukončenie lúča** (adaptive ray terminantion).
   Túto techniku možno použiť iba pri kompozícii typu odpredu-dozadu. Ak nepriehľadnosť prekročí prahovú hodnotu, žiadne ďalšie vzorky pozdĺž lúča neodoberáme, ich príspevok do výslednej farby je zanedbateľný. Ako bolo spomenuté vyššie, podľa Levoya je 0.95 primeraná hraničná hodnota.

3. **Zjednodušenie vizualizačného algoritmu**
   Tieto urýchlenia zahŕňajú zanedbanie (napr. pri tieňovaní) alebo zjednodušenie (trilineárna vs. interpolácia najbližším susedom) niektorých výpočtov. Patria sem aj rôzne implementačné *"finty"* (hacks), medzi ktoré patrí napr. použitie vyhľadávacích tabuliek (look-up tables) pri tieňovaní alebo použitie celočíselných výpočtov namiesto počítania s reálnymi číslami. Sú zvyčajne prispôsobené na konkrétnu architektúru počítača (berieme do úvahy množstvo dostupnej pamäte a cenu počítania s číslami s pohyblivou desatinnou čiarkou).

4. **Progresívne zlepšovanie kvality obrazu**
   Rýchlo vyrenderujeme obrázky nízkej kvality, pokiaľ používateľ mení parametre zobrazovacej metódy. Inak postupne renderujeme obrázky vo vyššej kvalite. Podvzorkovaním dokážeme ray-castingom vygenerovať obrázky nízkej kvality relatívne rýchlo.

5. **Funkcia diaľky**
   Každé reálne objemové dáta obsahujú súvislé oblasti nezaujímavých voxelov. Pomocou tejto funkcie ich vieme preskakovať. Rýchlosť algoritmu zvyšujeme napr. použitím oktálového stromu alebo inej pomocnej štruktúry – dokážeme rýchlo preskočiť prázdne, resp. nezaujímavé časti objemu, ktoré neprispievajú do výsledného obrazu. Tieto techniky zo sebou prinášajú aj niekoľko nevýhod: traverzovanie a testovanie vzoriek v takejto štruktúre môže byť dosť zložité, resp. pomocná dátová štruktúra môže byť nezanedbateľne veľká.

Všetky spomenuté metódy závisia od vstupných údajov.

2.4.2 Zvyšovanie kvality obrazu

Kvalitu obrazu dokážeme zlepšiť vrhnutím viacerých lúčov jedným pixelom zobrazovacieho priestoru alebo zvýšením počtu odoberaných vzoriek pozdĺž každého lúča. Hlavné zvýšenie kvality obrazu stojí na zvýšení kvality prevzorkovania (použitie interpolácie vyššieho stupňa) a odhadu *gradientu* používaného pri klasifikácii a tieňovaní.

Typ interpolácie a techniky odhadu lokálneho gradientu výrazne ovplyvňujú kvalitu výsledného obrazu.

Kapitola 3

Paralelné renderovanie

Vizualizácia viacrozmerných dát je výpočtovo veľmi náročný proces. Ak sú vstupné údaje veľké, pamäť a výpočtová sila sekvenčných procesorov brzdí interaktivitu celého programu. Paralelné architektúry sú preto častou voľbou pre spracovanie takýchto dát. Vo všeobecnosti ich rozdeľujeme do troch kategórii: Architektúry

- so *zdieľanou pamäťou* (shared-memory),
- s *distribuovanou pamäťou* (distributed memory),
- s *distribuovanou zdieľanou pamäťou* (distributed shared-memory).

Každá z týchto architektúr môže podporovať jeden alebo viacero programovacích modelov: *Single Instruction, Multiple Data* (SIMD), *Single Program, Multiple Data* (SPMD) a *Multiple Instruction, Multiple Data* (MIMD) (Flynn, 1966). V nasledujúcich častiach kapitoly sa bližšie pozrieme na problematiku písania paralelného programu pracujúceho na spomenutých architektúrach.

3.1 Problém dekompozície

Prvým krokom pri tvorbe paralelného programu je rozdeliť daný *problém* na menšie problémy. Tieto problémy sú priradené procesorom, ktoré na nich pracujú simultánne. V praxi sa vo väčšine prípadov implementujú dva typy dekompozície (PACS, 2001):

- *Priestorová dekompozícia* (Domain decomposition)
- *Funkčná dekompozícia* (Functional decomposition)

V počítačovej grafike sa zvykne uvádzať ešte jeden typ paralelného prístupu, a to *dočasný paralelizmus* (temporal parallelism). Tento paralelizmus sa využíva prevažne v počítačovej animácii, preto sa o ňom nebudeme ďalej zmieňovať.

3.1.1 Priestorová dekompozícia

Princíp tohto prístupu, autormi často označovaného aj ako *paralelizmus dát* (data parallelism), spočíva v rozdelení dát na približne rovnaké časti. Tieto sú potom rozdelené medzi procesory a každý pracuje iba na svojej časti dát. Procesory, v prípade potreby získania údajov z iného procesora, môžu medzi sebou komunikovať. Paralelné programy napísané touto technikou

pozostávajú zo sekvencie základných inštrukcií, ktoré sú aplikované na dáta. SPMD[1] sleduje tento model, kde každý procesor obsahuje rovnaký program. Paralelizmus dát je limitovaný ekonomickými a technickými obmedzeniami, napr. počet prepojených procesorov v jednom systéme. Pri výbere paralelného algoritmu rozhoduje aj komunikačná sieť prepájajúca procesory medzi sebou a *vice versa*.

Algoritmy založené na priestorovej dekompozícii môžeme rozdeliť do dvoch tried podľa toho, ako rozdeľujú a distribuujú dáta medzi procesory (Crockett, 1995).

Obrazový paralelizmus (image-space partitioning) rozdeľuje zobrazovací priestor na malé časti a následne jedna alebo viacero častí je priradená procesoru. Medzi hlavné výhody obrazového paralelizmu patrí rovnomerné rozloženie výpočtu medzi procesory a nízke nároky na komunikáciu. Malé časti sa neprekrývajú, a teda pre vytvorenie výsledného obrazu nie je potrebná žiadna kompozícia. Aby sa procesory vyhli častým re-distribúciam údajov medzi sebou (kvôli vygenerovaniu obrazu z ľubovoľnej pozície), potrebujú mať prístup k *celým* dátam – čo je vďaka súčasným hardvérovým možnostiam v praxi realizovateľné len pre veľmi malé dáta. Zatiaľ.

Na druhej strane, *objektový paralelizmus* (object-space partitioning) rozdeľuje dáta na menšie časti a každý procesor je zodpovedný za niekoľko takýchto menších objemových dát. Veľkou výhodou tejto triedy je jej *pamäťová škálovateľnosť*, t.j., že so zvyšujúcim sa počtom procesorov vieme do pamäte paralelného počítača uložiť oveľa viac dát. Nakoľko do výslednej farby pixela generovaného obrazu môže prispievať niekoľko farieb z dátových blokov priradených procesorom, poskladanie konečného obrazu vyžaduje kompozíciu čiastkových obrázkov z rôznych procesorov. Tento krok vyúsťuje do vysokých nárokov na šírku komunikačného pásma. Kompozícia sa môže stať *úzkym miestom* (bottleneck) celého paralelného programu.

V (Garcia, 2002) autori prezentujú *hybridný* algoritmus kombinujúci výhody oboch spomínaných prístupov. Procesory rozdelili do skupín a nie každému procesoru, ale každej skupine priradili časť objemových dát – zaručenie pamäťovej škálovateľnosti a nižších komunikačných nákladov vo fáze kompozície. V každej skupine sú pixely obrazového priestoru *prekladané* (interleaving) medzi jednotlivé procesory – zaručenie rovnomerného rozdelenia výpočtu.

3.1.2 Funkčná dekompozícia

Rozdeliť proces vygenerovania výsledného obrazu na niekoľko rôznych funkcií – *paralelizmus úloh* (task parallelism) – je inou cestou dosiahnutia paralelizmu. Celý problém je rozdelený na veľa menších úloh. Ako náhle sa procesor stane voľným, je mu hneď priradená jedna z úloh čakajúca na spracovanie. Celková rýchlosť programu je limitovaná rýchlosťou najpomalšieho procesu, zvyšné voľné procesory nerobia žiadnu užitočnú prácu. Veľmi často tento prístup *nevedie* k najefektívnejšiemu algoritmu pre paralelný program (PACS, 2001). Volíme ho vtedy, ak časy spracovania jednotlivých časti dát sú veľmi rozdielne.

Tento typ algoritmov býva implementovaný ako klient-server aplikácia. Šéf (master processor) prideľuje úlohy sluhom (slave processors), pritom môže niektoré úlohy vykonať sám. Funkčný paralelizmus pracuje dobre najmä pri algoritmoch renderujúcich polygóny a povrchy.

[1]SPMD a ďalšie modely budú vysvetlené v nasledujúcich kapitolách.

3.2 Zdieľaná verzus distribuovaná pamäť

Ako bolo spomenuté na začiatku kapitoly, väčšina paralelných architektúr spadá do dvoch kategórii: architektúra so *zdieľanou* a *distribuovanou pamäťou*.

Počítač so **zdieľanou pamäťou** umožňuje procesorom pristupovať do spoločnej pamäte vďaka vysokorýchlostnej pamäťovej zbernici. Pomocou zdieľanej pamäte si vedia procesory medzi sebou relatívne rýchlo a efektívne zdieľať a vymieňať údaje. Množstvo spracovaných údajov je obmedzené šírkou pásma zbernice, preto sa počet procesorov v takejto architektúre pohybuje okolo 2 – 16, t.j. pamäťová škálovateľnosť je značne limitovaná. V snahe minimalizovať toto obmedzenie, algoritmy musia byť napísané tak, aby minimalizovali synchronizáciu a častý prístup na rovnaké miesta pamäte.

Paralelný počítač s **distribuovanou pamäťou** je zjednodušene povedané niekoľko navzájom prepojených sériových počítačov, uzlov (nodes), pracujúcich spoločne na zadanom probléme. Uzly majú veľmi rýchly prístup do vlastnej pamäte, no neexistuje tu žiadna globálna zdieľaná pamäť. Komunikácia medzi procesormi sa dosahuje pomocou komunikačnej siete. Obyčajne sa jedná o *proprietárnu* vysokorýchlostnú sieť. Procesory si dáta v takejto sieti posielajú vo forme *správ*. Takáto architektúra ponúka oveľa väčšie možnosti škálovania za cenu pomalšieho prístupu do pamäte jednotlivých uzlov. Renderovací proces zvyčajne produkuje veľké množstvo dočasných dát, ktoré musia byť dynamicky mapované z objektového priestoru na zobrazovací. Programy preto musia obzvlášť dbať na fázy komunikácie medzi procesormi. Globálne operácie, synchronizácia a presun údajov sú obyčajne veľmi *drahé*, preto sa programy napísané na tomto type architektúr prikláňajú k statickému priradeniu úloh a dát jednotlivým uzlom.

Posledná generácia paralelných počítačov používa *mixovaný* prístup so zdieľanou/distribuovanou pamäťou. Každý uzol pozostáva zo skupiny 2–16 procesorov prepojených lokálnou zdieľanou pamäťou a viacprocesorové uzly sú medzi sebou prepojené vysokorýchlostnou sieťou (PACS, 2001).

3.3 Výpočtové modely paralelných architektúr

Flynn v roku 1966 zaviedol klasifikáciu počítačových architektúr založenú na počte inštrukcií a dátových prúdov v systéme. "Akýkoľvek počítač, sekvenčný alebo paralelný, vykonáva inštrukcie na dátach." (Flynn, 1966)

Prúd inštrukcií (algoritmus) hovorí počítaču *čo robiť*. Prúd dát (vstup programu) je *ovplyvnený* týmito inštrukciami. Podľa počtu prúdov inštrukcií, resp. dát rozlišujeme nasledujúce modely:

SISD Počítač
Jedná sa o štandardný sekvenčný počítač. Jedna procesorová jednotka má ako vstup jeden prúd inštrukcií, ktoré sa vykonávajú na jednom prúde dát. Nie je tu obsiahnutý žiadny paralelizmus, počítač obsahuje iba jeden procesor.

Jeden prúd inštrukcií, Jeden prúd dát	Single Instruction Stream, Single Data Stream	SISD
Viac prúdov inštrukcií, Jeden prúd dát	Multiple Instruction Stream, Single Data Stream	MISD
Jeden prúd inštrukcií, Viac prúdov dát	Single Instruction Stream, Multiple Data Stream	SIMD
Viac prúdov inštrukcií, Viac prúdov dát	Multiple Instruction Stream, Multiple Data Stream	MIMD

Tabuľka 3.1: Výpočtové modely paralelných architektúr

MISD Počítač
N procesorov zdieľa spoločnú pamäť. V počítači je prítomných N prúdov inštrukcií a jeden prúd dát. Procesory vykonávajú rôzne operácie v rovnakom čase na rovnakých dátach. Tento model je užitočný, ak na rovnaký vstup potrebujeme aplikovať niekoľko rôznych operácii.

SIMD Počítač
Všetkých N identických procesorov pracuje pod kontrolou jediného toku dát. Zjednodušene – každý procesor obsahuje rovnaký program. Procesor pracuje na jednom z N vstupných prúdov dát (rôzne dáta pre rôzne procesory). Procesory pracujú zosynchronizovane pomocou globálnych hodín, t.j. každý procesor v jednom *tiku* vykoná rovnakú inštrukciu na rôznych dátach[2]. SIMD počítače sa používajú, ak majú vstupné dáta regulárnu štruktúru, t.j. rovnaké inštrukcie môžeme aplikovať na podmnožiny vstupu.

MIMD Počítače
Najsilnejší a najvšeobecnejší model klasifikácie. Máme N procesorov, N prúdov inštrukcií a N dátových prúdov. Každý procesor je schopný vykonávať vlastný program na rôznych dátach. To znamená, že procesory pracujú asynchrónne.

SPMD – Jeden program, viac prúdov dát
SIMD paradigma je príklad synchrónneho paralelizmu – každý procesor pracuje v tikoch. Asynchrónna verzia tohto modelu sa označuje ako SPMD (Single Program, Multiple Data). Procesory vykonávajú rovnaký program na MIMD počítači.

3.4 Klaster

Klaster je skupina prepojených počítačov, ktoré pracujú spolu tak *blízko*, že v mnohých prípadoch sa na takéto zoskupenie môžeme pozerať ako keby to bol jeden počítač. Jednotlivé počítače, zvyknú sa označovať ako uzly[3], sú väčšinou prepojené vysokorýchlostnou miestnou počítačovou sieťou (LAN). Klastre rozdeľujeme do troch kategórii: *vysoko dostupné klastre* (high-availability clusters), *rovnomerne zaťažené klastre* (load balancing clusters) a *vysoko výkonné klastre* (high-performance clusters HPC). Našu prácu sme vytvárali na posledne menovanom klastri. Viac informácii na (http://www.beowulf.org/).

[2]Ako bude spomenuté neskôr, SPMD pracuje asynchrónne – rovnaký program pracuje na rôznych dátach za použitia modelu MIMD.

[3]Uzlom môže byť aj osobný digitálny asistent – PDA alebo iné sieťové zariadenie.

3.4.1 Vysoko výkonné klastre

Tieto klastre sú primárne vytvorené pre zvýšenie výkonu rozdelením úloh medzi veľa uzlov paralelného počítača. Väčšinou sa používajú na vedecko-technické výpočty, ako fyzikálne simulácie a výpočtovo náročné renderovanie v počítačovej grafike. Tento typ klastra je najpopulárnejší v konfigurácii, kde uzly pracujú pod operačným systémom GNU/Linux a paralelizmus je implementovaný vďaka slobodnému softvéru (free software). V publikáciách sa táto konfigurácia zvykne často označovať ako *Beowulfov klaster*. Mnoho autorov implementuje paralelné algoritmy v niektorom populárnom programovacom jazyku (väčšinou ide o C/C++) s použitím knižníc ako napr. *MPI* (Message Passing Interface), ktoré sú špeciálne navrhnuté pre programovanie vedecko-technických aplikácií na HPC klastroch.

3.4.2 Myrinet a Amdahlovo pravidlo

Sieť prepájajúca uzly klastra hrá veľmi dôležitú úlohu pri dosahovaní vysokého výkonu viacprocesorového počítača. Jej výkon sa odráža v celkovom výkone a rýchlosti programu pracujúceho na paralelnom počítači. Ak komunikačná sieť nedokáže poskytnúť patričný výkon, počas behu programu budú procesory nútené čakať na prichádzajúce údaje. Myrinet[4] je vysokorýchlostná miestna počítačová sieť prepájajúca viacero počítačov, ktoré vytvárajú klaster (http://www.myri.com/). Má nižšie režijné náklady oproti štandardom ako*Ethernet*, lepšiu priepustnosť, menšie rušenie a latenciu. Myrinet sa obyčajne používa priamo programami, ktoré o nej *vedia*. Fyzicky sa táto sieť skladá z dvoch optických vlákien (upstream, downstream). Prvá generácia poskytovala pásmo o šírke 512 Mbit/s v oboch smeroch, neskoršie verzie 1.28 Gbit/s a 2 Gbit/s. V čase písania tejto práce je najnovšia štvrtá generácia podporujúca prenos 10 Gbit/s a je schopná spolupracovať s 10 Gigabitovým Ethernetom na fyzickej vrstve (káble, konektory, atď.). Priepustnosť Myrinetu sa blíži k teoretickému maximu fyzickej vrstvy. Pre naše výpočty je nízka latencia dôležitejšia ako priepustnosť. *Amdahlovo pravidlo* hovorí, že najpomalší sekvenčný proces spôsobuje úzke miesto výkonného paralelného systému, čo je v našom prípade latencia prenosu správ sieťou (Amdahl, 1967).

3.5 Tvorba paralelného programu

Pri použití počítačového klastra existujú dva prístupy tvorby paralelného programu (PACS, 2001):

1. použitie *príkazových jazykov s paralelným spracovaním údajov* (directives-based data-parallel language),
2. explicitné *posielanie správ* pomocou funkcií z knižnice v štandardnom programovacom jazyku.

V prvom prípade sériový kód prepíšeme na paralelný pridaním značiek, ktoré hovoria kompilátoru ako distribuovať dáta a výpočet medzi procesory. Tieto značky sa tvária ako komentáre v sériovom kóde. Prekladač za programátora vyrieši všetky detaily ako distribuovať dáta, robiť výpočet a komunikáciu. Zvyčajne sa používajú na architektúre so zdieľanou pamäťou, pretože globálna pamäť veľmi zjednodušuje napísanie samotného kompilátora.

[4] ANSI/VITA 26-1998, dizajn od spoločnosti Myricom

Druhý prípad necháva všetku prácu na programátorovi. On sa rozhodne, ako rozdeliť dáta a prácu medzi procesory, spravovať celú komunikáciu a podobne. Jedná sa o veľmi flexibilný prístup.

3.6 Ťažkosti pri dizajne kódu

Pri písaní paralelného programu sa už vopred musíme zaoberať niektorými otázkami, ak chceme dostať z paralelného počítača maximálny výkon.

- Ako rovnomerne rozložiť prácu medzi dostupné procesory?
- Ako minimalizovať komunikáciu?
- Ako prekryť prácu a výpočet?

Nastolené otázky bližšie rozoberieme v nasledujúcich častiach.

3.6.1 Rovnomerné rozloženie práce

Stojíme pred problémom, ako približne rovnomerne rozdeliť prácu medzi procesory. V prípade SIMD/SPMD modelu je riešenie relatívne jednoduché oproti prípadu, keď čas spracovania dát závisí od ich obsahu. V tom prípade je riešenie netriviálne a zrejme by bolo lepšie zvoliť inú metódu pre vyriešenie úlohy.

V prípade paralelného renderovania je niekoľko faktorov, ktoré sťažujú dosiahnutie rovnomerného rozloženia záťaže medzi procesory. Ak rozdelíme dáta na rovnaké časti medzi procesory, niektoré bloky môžu byť úplne prázdne, iné presne naopak. Ak sa zameriame konkrétne na algoritmus ray-casting, uhol pohľadu, optimalizácie ako predčasné ukončenie lúča prispievajú k nerovnomernému rozloženiu záťaže.

3.6.2 Minimalizovať komunikáciu

Pri paralelných programoch sa zameriavame na celkovú dobu trvania výpočtu z dôvodu porovnávania s inými sekvenčnými/paralelnými programami. Celkový čas trvania výpočtu sa skladá z času:

1. na *výpočet*
2. *čakania* na údaje/prácu (Idle)
3. na *komunikáciu*

Čas na výpočet predstavuje čas, ktorý procesory spotrebujú na vyriešenie úloh. V ideálnom prípade očakávame čas výpočtu $\frac{1}{N}T$, ak na probléme pracuje N procesorov a T je pôvodný čas sekvenčného programu.

Časom čakania označujeme dobu, počas ktorej procesor nevykonáva žiadnu inštrukciu na dátach, nerobí nič užitočné. Napríklad spracovanie vstupu a výstupu v paralelnom programe (jeden procesor spracováva vstup/výstup, ostatné sú v stave idle).

Komunikačný čas je čas, ktorý zaberie posielanie a prijímanie správ medzi procesormi. Celkovú dĺžku odhadujeme pomocou termínov *latencia* a *šírka pásma*. Latencia je čas potrebný na vytvorenie obálky pre dáta, ktoré sa budú sieťou prenášať. Šírka pásma je rýchlosť prenosu (bity za jednotku času). Nakoľko sériové programy neobsahujú komunikáciu medzi uzlami, je veľká snaha autorov paralelných algoritmov minimalizovať práve tento čas.

3.6.3 Prekrytie komunikácie a výpočtu

Pri návrhu paralelného programu sme neradi, ak v systéme sú v isté časové okamihy procesory, ktoré nerobia žiadnu užitočnú prácu. Jednou z možností ako minimalizovať tento idle čas procesorov je nechať daný procesor spracovávať ďalšiu úlohu pokiaľ procesor čaká na ukončenie komunikácie (použitie neblokovanej komunikácie). V praxi sa tento bod dosahuje veľmi ťažko.

3.7 Základy modelu preposielania správ

V nasledujúcej časti kvôli jednoduchosti predpokladajme, že rôzne procesy bežia na rôznych procesoroch. Preto termíny *proces* a *procesor* môžeme medzi sebou zameniť. Paralelný výpočet pozostáva z niekoľkých procesov, každý pracuje na lokálnych dátach. Každý proces má iba vlastnú pamäť a neexistuje priamy prístup do pamäte iného procesu. Zdieľanie údajov medzi procesmi sa uskutočňuje pomocou preposielania správ – posielanie a prijímanie údajov medzi procesmi.

Hlavnou výhodou tohto prístupu je jeho všeobecnosť, modelom preposielania správ vieme naprogramovať skoro všetky paralelné algoritmy. Navyše je implementovaný na rôznych platformách, dokonca aj na bežných jedno procesorových počítačoch. Umožňuje veľkú kontrolu nad tokom a umiestnením dát. Pri písaní paralelného programu nám ide na prvom mieste o výkon – a vyšší výkon v porovnaní z inými modelmi získame práve modelom preposielania správ. Preto sa tento model asi nikdy nevytratí zo sveta paralelného programovania (PACS, 2001).

3.7.1 Čo je to MPI?

MPI (Message Passing Interface) je špecifikácia pre vývojárov a používateľov knižníc na preposielanie správ (http://www-unix.mcs.anl.gov/mpi/). MPI samo o sebe nie je knižnica ale skôr špecifikácia, o čom by takáto knižnica mala byť (špecifikácia rozhrania). Vývoj MPI prebiehal na MPI Fóre (http://www.mpi-forum.org/). Cieľom MPI je poskytnúť štandard pre tvorbu programov používajúcich knižnice preposielania správ. Pri vývoji rozhrania sa dbalo na tieto hlavné body:

1. efektivita
2. portabilita
3. použiteľnosť
4. flexibilita

Špecifikácia rozhrania (funkcií) je napísaná pre jazyky C/C++ a Fortran (vyše 115 funkcií). MPI nie je IEEE alebo ISO štandard, no môžeme ho zaň *de facto* považovať.

Hlavnou výhodou MPI je jej výborná portabilita. Je efektívne implementovaná na rôznych platformách. Podporuje aj heterogénne paralelné architektúry. Funkcie ako ladenie alebo paralelný vstup/výstup (I/O) sú mimo MPI.

MPI-1 štandard bol navrhnutý a implementovaný už v roku 1994[5]. V roku 1996 bol predstavený štandard MPI-2. V súčasnosti sú implementácie MPI kombináciou MPI-1 a MPI-2. Len veľmi málo z nich obsahuje celú funkcionalitu oboch špecifikácii. MPI bolo pôvodne určené pre architektúry s distribuovanou pamäťou, no popularita systémov so zdieľanou pamäťou dala za vznik implementáciám pre takéto architektúry (SMP/NUMA). V dnešnej dobe sa MPI používa skoro na akejkoľvek paralelnej architektúre. Paralelizmus dosahujeme explicitne, t.j. programátor je zodpovedný za korektnú implementáciu paralelných algoritmov za použitia MPI funkcií. Počet úloh pridelených paralelnému programu je statický.

3.8 Paralelný ray-casting

Vysoká výpočtová zložitosť algoritmov založených na trasovaní lúča viedla k vytvoreniu rôznych urýchľovacích techník. Výpočtová sila bežne dostupného počítača nie je dostačujúca na interaktívne zobrazovanie veľkých objemových dát. Paralelizmus je jednou z možných ciest. Ray-casting je algoritmus, ktorý ponúka mnoho možností na vylepšenie. Modifikujeme pôvodnú metódu tak, aby bola implementovateľná na paralelnom počítači.

Podľa formy paralelizmu použitej v algoritme dostávame nasledujúce rozdelenie prístupov k paralelnému ray-castingu (Pfister a kol., 1996):

- *Paralelizmus lúča (ray-parallel)*
- *Zväzky lúčov (beam-parallel)*
- *Paralelné rezy (slice-parallel)*

V prvom prípade sú všetky vzorky pozdĺž lúča spracované naraz. Algoritmus prechádza lúč za lúčom. Spracovanie všetkých voxelov jedného lúča naraz si vyžaduje globálnu komunikáciu medzi procesmi. Takáto komunikácia výrazne obmedzuje výkon a rozšíriteľnosť programu, pretože potrebujeme dostatočne široké pásmo na prenos údajov.

Alternatívny pristup spočíva v spracovaní vzoriek niekoľkých lúčov naraz. Prístup *zväzky lúčov* pracuje naraz na jednej úrovni lúčov. Pri načítaní dát z hlavnej pamäte do L2 cache pamäte procesora sa totiž nenačíta iba osem voxelov bunky, ale viac — isté okolie bunky (pamäte)[6]. Pri tomto prístupe sa snažíme čo najviac načítaných údajov spracovať, aby sme ich nemuseli opäť zdĺhavo načítavať. Pri technike *paralelné rezy* sa spracúvajú naraz celé rezy objemových dát. V jednom kroku spracujeme (skoro) všetky lúče. Tento postup je obsiahnutý napríklad v známom algoritme shear-warp alebo architektúre Cube-4 (Pfister a kol., 1996).

[5]MPI-1 Štandard: http://www.mcs.anl.gov/Projects/mpi/standard.html

[6]Jedno z riešení optimalizačného problému dômyselného načítania okolia voxelov sa nazýva *Bricking* a bude spomenuté v ďalšej časti práce.

3.8.1 Kompozícia častí lúčov

Mohlo by sa zdať, že rovnica 2.9 z kapitoly kompozícia je počítaná sériovo. V skutočnosti je asociatívna. Kompozícia farby a nepriehľadnosti je založená na operátore *nad* (Porter a kol., 1984) a s pomocou algebry nie je ťažké overiť, že

$$a \textbf{ nad } (b \textbf{ nad } c) = (a \textbf{ nad } b) \textbf{ nad } c. \tag{3.1}$$

Táto asociativita nám umožňuje rozdeliť lúč na segmenty, vykonať odber vzoriek a kompozíciu nezávisle na segmente a v konečnom kroku kompozície (partial ray composition) skombinovať výsledky z každého segmentu (Hsu, 1993). Na tejto znalosti je založených mnoho paralelných algoritmov sledovania lúča.

Lema 3.8.1 *Farbu F(I,J) v 2.10 vieme z farieb* $F_{\frac{n}{2}}(I,J)$ *a* $F_n(I,J)$ *zložiť nasledujúcou rovnosťou:*

$$F(I,J) = F_{\frac{n}{2}}(I,J) + (1 - \alpha_{\frac{n}{2}})F_n(I,J) \tag{3.2}$$

kde:

$$\alpha_{\frac{n}{2}} = \sum_{k=1}^{\frac{n}{2}} \alpha(I,J,k) \prod_{l=1}^{k-1} (1 - \alpha(I,J,l)) \tag{3.3}$$

Vzťah 3.2 sa rovná vykonaniu ďalšieho kroku kompozície. Viac informácii a dôkaz lemy možno nájsť v (Bentum, 1996).

3.9 Z objektového do zobrazovacieho priestoru

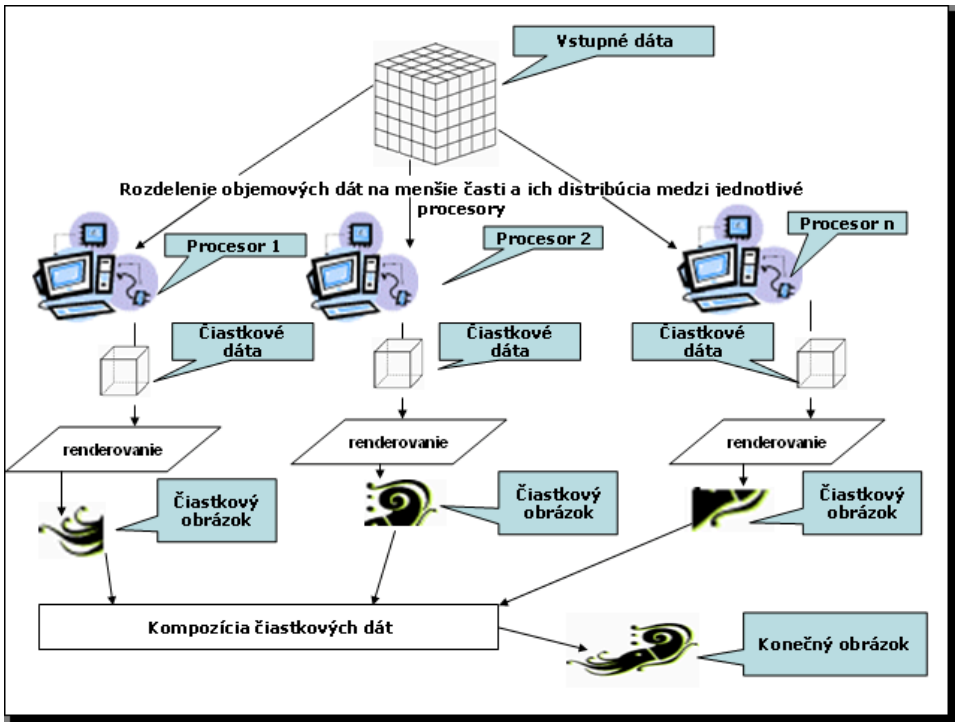
Zobrazovanie z objektového do zobrazovacieho priestoru je komplexný a dynamický proces. Komunikácia v ňom obsiahnutá je jednou z hlavných otázok, ktoré riešime v paralelných algoritmoch na systémoch s distribuovanou pamäťou. (Molnar a kol., 1994) zatriedil všetky algoritmy do troch skupín podľa *miesta*, kde sa v algoritme uskutočňuje zobrazovanie (komunikácia, triedenie) z objektového do zobrazovacieho priestoru. Výsledkom ich štúdie o výpočtovej a komunikačnej zložitosti je tvrdenie, že žiadny prístup nie je značne lepší vo všetkých prípadoch.

Utrieď na začiatku (Sort-first)
Namapovanie sa vykoná ešte na začiatku renderovacieho procesu vo fáze predspracovania – objekty sa priradia príslušným procesorom. To so sebou prináša nerovnomerné rozloženie záťaže, kvôli rozdielnym distribúciám objektov v obraze. Na druhej strane má tento prístup nižšie komunikačné náklady oproti ostatným prístupom, pretože vo fáze renderovania sú už dáta priradené správnym procesorom a nepotrebujeme ich presúvať na ďalšie spracovanie.

Utrieď v strede (Sort-middle)
Implementácia komunikácie v algoritmoch tejto kategórie je priamočiara, pretože sa uskutočňuje na prirodzenom mieste – medzi transformačným a rasterizačným krokom. Oproti prvej kategórii je tento prístup menej citlivý na nerovnomerné rozloženie záťaže, pretože istá časť práce je vykonaná pred zobrazením dát do zobrazovacieho priestoru.

Utrieď na konci (Sort-last)
Posledná kategória algoritmov je ešte menej citlivá na rozdelenie objektov v obraze. Skoro celý

výpočet sa uskutočnil s pôvodným rozdelením objektového priestoru (obrázok 3.1). Komunikácia sa uskutočňuje na úrovni pixelov, preto sú nároky na šírku pásma vysoké. Napriek tomu sú algoritmy tohto typu implementované vo viacerých zobrazovacích systémoch.



Obrázok 3.1: Schéma algoritmov typu "utrieď na konci". Algoritmus binárna výmena (binary swap) je tiež tohto typu.

3.10 Konštrukcia a zobrazenie obrazu

Vysokovýkonné renderovacie systémy produkujú na výstup veľké množstvo dát – obrázkov (obrázkový prúd). Plynulé zobrazenie na celej obrazovke pri plných farbách (rozlíšenie 1280x1024 pixelov, farebná hĺbka 24 bitov/pixel, 30 obrázkov/sekundu) vyžaduje šírku pásma 120MB/s (Crockett, 1995). Ako sme spomenuli vyššie, náš prístup kombinuje výsledky z procesorov do výsledného obrazu. Ak by sme nedokázali tento proces robiť dostatočne rýchlo, stane sa úzkym miestom paralelného renderovacieho programu.

Autori sa snažia túto fázu algoritmu urýchliť hardvérovo, napr. oddeliť renderer od zásobníka snímok (frame buffer), a prepojiť ich vysokorýchlostnou sieťou. Príkladom je Pixel-plane 5 systém (Fuchs a kol., 1989) používajúci 640MB/s sieť typu *token ring* prepájajúcu jednotlivé komponenty systému. Inou možnosťou je zhotoviť obraz v paralelnom systéme a poslať ho preč na zobrazenie (paralelné počítače väčšinou nemajú vhodné zariadenia pre zobrazovanie výstupu). Zo súčasných systémov spomenieme Intel Paragon a Cray T3D, ktorých komunikačné siete zvládnu prenos cez 100MB/s.

Poznáme niekoľko algoritmických prístupov pre kompozíciu obrazu na klastri. Jednou z možností je *vyhradiť* jeden uzol klastra, ktorý bude zbierať vyrenderované časti obrázkov od jednotlivých procesorov (Camahort a kol., 1993). Tento algoritmus je vhodný len pre veľmi malé počty procesorov paralelného systému. V ostatných prípadoch sa prijímajúci uzol stáva úzkym miestom. Hsu predstavil techniku paralelného ray-castingu zvanú *segmentovaný ray-casting* (segmented ray-casting) (Hsu, 1993). Program implementoval na SIMD počítači MP-1. Objemové dáta rozdelil do blokov a každému procesoru priradil niekoľko blokov. Pixely vyrenderovaných obrázkov – segmentov – sa posielali procesoru zodpovednému za dané pixely vo výslednom obraze. Keď mal procesor k dispozícii všetky segmenty lúča, bola uskutočnená kompozícia výslednej farby pixela. Ma vo svojej práci navrhol algoritmus *binárna výmena* (binary swap) na kompozíciu obrázkov pracujúci v logaritmickom čase, pričom vyťažuje všetky procesory systému (Ma a kol., 1994). Tento algoritmus používame aj v našej práci, a preto ho v nasledujúcej časti rozoberieme podrobnejšie. Stompel prezentoval nový optimalizovaný algoritmus zvaný *plánovaná lineárna kompozícia*, ktorý redukuje komunikáciu pomocou *kompozičného plánu* vypočítaného za behu programu (Stompel a kol., 2003). Autori vymenili časť komunikácie za ďalší (relatívne krátky) výpočet. Ten závisí iba od rozlíšenia obrazu a počtu procesorov (ich počet nemusí byť mocnina dvojky ako v prípade algoritmu binary swap). Pre väčší počet procesorov (a väčšie obrázky) dáva tento algoritmus jednoznačne lepšie výsledky. Tento algoritmus bol navrhnutý pre pomalšie siete.

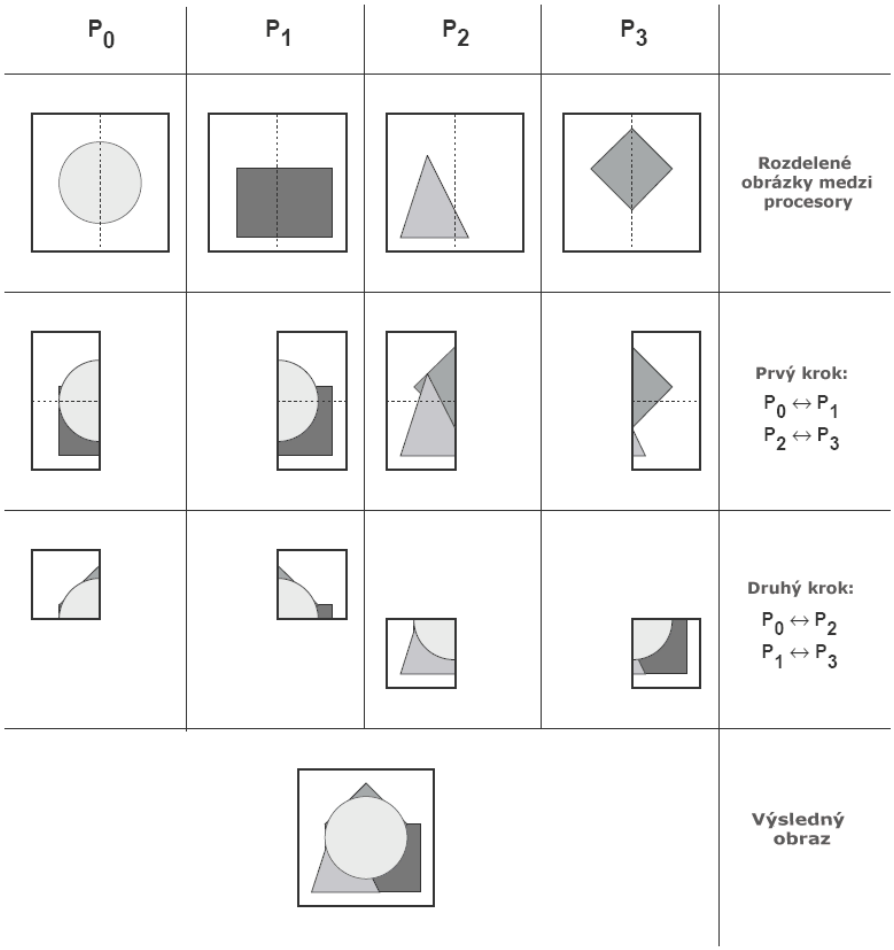
3.10.1 Binary swap objemové renderovanie

Algoritmus binárna výmena (binary swap) (Ma a kol., 1994) je založený na myšlienke rozdeľuj a panuj. Dáta sú rozdelené medzi procesory, vyrenderované oddelene a nakoniec skombinované do výsledného obrazu. Počet procesorov musí byť mocnina dvojky. Dáta sú rozdelené pomocou štruktúry zvanej k-D strom – na každej úrovni sú dáta striedavo rozdelené rovinami rovnobežnými s jednotlivými osami súradníc. Celý objem je rovnomerne rozdelený vo všetkých troch rovinách. Aby sme sa vyhli posielaniu dát medzi procesormi (interpolácia, výpočet gradientu), hraničné údaje zreplikujeme medzi príslušné procesory. Po vyrenderovaní *lokálnych* dát nasleduje krok kompozície. Obrázky kombinujeme odpredu-dozadu a vďaka štruktúre k-D strom vieme ľahko zistiť správne poradie. Rekurzívne prechádzame strom, pričom navštívime vždy *predného syna* skôr ako *zadného*. Hlavná myšlienka tejto fázy je rozdeliť obraz, pričom dvojica procesorov vždy pracuje na rôznej časti, vid. obrázok 3.2. Na začiatku je procesor zodpovedný za veľkú časť obrazu, avšak postupne sa táto časť zmenšuje. V logaritmickom čase je celá kompozícia hotová. Označme P ako počet procesorov, A veľkosť obrázku v pixeloch ($A^{1/2} \times A^{1/2}$). Potom celkový počet prenesených pixelov po dokončení fázy kompozície je $P \times \sum_{k=1}^{logP} \frac{A}{2^k}$ a každý pixel pozostáva z intenzity a nepriehľadnosti (Yang a kol., 2001). Každý pixel je reprezentovaný šestnástimi bajtmi, a preto je pre každý procesor lokálny čas výpočtu a komunikačný čas v kompozícii algoritmu binárna výmena nasledovný:

$$T_{vyp}(BS) = \sum_{k=1}^{logP} \left(T_o \times \frac{A}{2^k}\right), \tag{3.4}$$

$$T_{kom} = \sum_{k=1}^{logP} \left(T_s + \left(16 \cdot \frac{A}{2^k}\right) \times T_c\right), \tag{3.5}$$

kde T_o je čas miešania dvojice pixelov, T_s je počiatočný čas v komunikačnom kanáli a T_c je čas prenosu dát vzhľadom na bajt.



Obrázok 3.2: Kompozícia algoritmu binárna výmena. V každom kroku je obraz rozdelený a procesor je zodpovedný za kompozíciu príslušných dvoch polovíc (Zdroj: Crockett, 1995, s.30, modifikované).

Kapitola 4

Urýchľovacie techniky

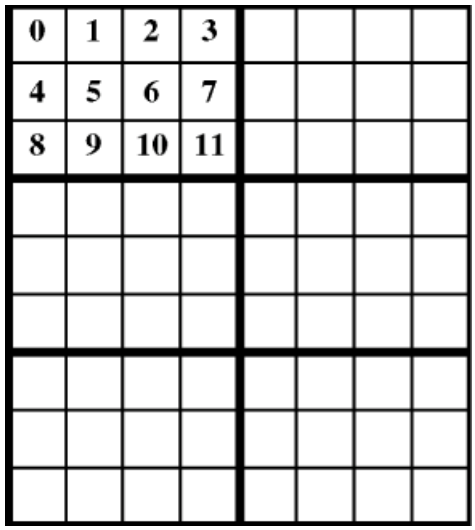
Napriek rapídnemu nárastu výkonu počítačov, interaktívna vizualizácia musí stále pracovať na veľkých objemoch dát. V snahe o dosiahnutie čo najlepších výsledkov (počet obrázkov za sekundu a kvalita obrazu) používame viaceré urýchlenia základného renderovacieho algoritmu. V časti 2.4.1 boli v niekoľkých bodoch všeobecne zhrnuté techniky urýchlenia (nie len) algoritmu vrhania lúča. V nasledujúcich odsekoch sa pozrieme bližšie na niektoré dôležité urýchlenia. Podrobne sa už nebudeme zaoberať predčasným ukončením sledovania lúča, detaily môže čitateľ nájsť v kapitolách 2.2.6 a 2.4.1.

4.1 Bricking

Rozdiel vo výkone medzi procesorom a pamäťou je kritický pre aplikácie interaktívne renderujúce veľké množstvo dát. V tomto prípade sa prístup do pamäte môže stať úzkym miestom programu. (Bruckner, 2004) vo svojej práci prezentuje jeden z možných prístupov k riešeniu daného problému – *bricking*. Obvykle sú celé viacrozmerné dáta uložené v pamäti počítača *lineárne* (linear layout) v usporiadaní XYZ. Reprezentácia objemu pomocou 1D poľa je vhodná len pre veľmi malé veľkosti dát. Na konkrétne dáta sa vieme v takomto poli ľahko odkazovať, avšak táto schéma má v prípade algoritmov sledovania lúča nevýhody. *Cache* pamäť má veľmi limitovanú veľkosť a pri načítaní jedného voxelu sa načítajú aj okolité voxely (uložené lineárne). Do objemu vrháme lúče jeden za druhým. Ak lúč traverzuje objem v rovnakom smere ako sú voxely fyzicky uložené v pamäti, ďalší navštívený voxel je už načítaný v cache pamäti. V ostatných prípadoch sa stránky cache pamäte musia načítavať znova a znova, rovnaké dáta sú načítané do pamäte veľa krát a zbytočne. Zbernica počítača je veľmi vyťažená a celkový čas renderovacieho procesu vzrastá. Na odstránenie tohto problému vo všeobecnosti poznáme dva prístupy (Razdan a kol., 2001). Buď traverzujeme objektový priestor, alebo zmeníme spôsob adresovania sa do 1D poľa tak, aby susedné voxely boli uložené v rovnakej pamäťovej stránke (obrázok 4.1[1]). V našej aplikácii implementujeme algoritmus vrhania lúčov, preto používame druhý prístup. Prvý prístup vedie k algoritmom podobným ako *splatting*.

Použitie *tehličiek* (bricks, bricking layout) zahŕňa rozdelenie objemu na menšie bloky dát konštantnej veľkosti. Veľkosť celého objemu musí byť v jednotlivých dimenziách rovný 2^n. Každý blok je uložený lineárne v usporiadaní XYZ. Snažíme sa zvoliť veľkosť bloku tak, aby sa celý vošiel do rýchlej cache pamäte ($L2$) architektúry. Komplikované a časovo náročné je

[1] obrázok z Interactive RayTracing



Obrázok 4.1: Aby sme susedné voxely dostali do jednej pamäťovej stránky, dáta sú organizované do tehličiek. Čísla v prvej tehličke vyjadrujú usporiadanie voxelov v pamäti.

adresovanie voxelov v rámci rozdeleného objemu. Aby sme sa dostali k jednému voxelu, potrebujeme poznať adresu bloku a adresu voxelu v rámci bloku. Lineárnu schému môžeme brať ako jeden veľký blok. V renderovacom algoritme je vstupom trilineárnej interpolácie osem voxelov, pričom v najhoršom prípade môže každý patriť inému bloku. V snahe vyhnúť sa zdĺhavému výpočtu adries bolo predstavených niekoľko riešení.

(Sakas a kol., 1996) implementovali riešenie pomocou troch vyhľadávacích tabuliek. Objemové dáta sú rozdelené do menších kociek o veľkosti $size^3$, kde $size$ je veľkosť strany kocky. Súčasné cache pamäte majú obyčajne veľkosť 512, 1024, 2048 kB (najnovšie procesory už aj 8192kB). Preto je vhodná veľkosť $size$ medzi 8 a 12. Označme rozmery objemových dát ako `data.x`, `data.y`, resp. `data.z`. Adresu voxelu na pozícii (i, j, k) určíme vzťahom:

```
adresa(i,j,k) = zakl_adr + Gadr[i] + Hadr[j] + Iadr[k]
```
(4.1)

kde:

`zakl_adr` je počiatočná adresa jednorozmerného poľa dát,

$$\begin{aligned} \texttt{Gadr[i]} &= \frac{i}{size}\, size^3 + (i \,\%\, size), \\ \texttt{Hadr[i]} &= \frac{i}{size}\frac{data.x}{size}\, size^3 + (i \,\%\, size) size, \\ \texttt{Iadr[i]} &= \frac{i}{size}\frac{data.x}{size}\frac{data.y}{size}\, size^3 + (i \,\%\, size) size^2. \end{aligned} \tag{4.2}$$

`Gadr`, `Hadr` a `Iadr` sú vyhľadávacie tabuľky o veľkosti `data.x`, `data.y`, resp. `data.z`. Analogické riešenie bolo prezentované aj v (Parker a kol., 1999).

Bruckner predstavil riešenie pomocou vyhľadávacej tabuľky o veľkosti 224 bytov. Pozícia prvého voxelu bloku sa určí podľa vzťahu:

$$\texttt{VratOffset(i,j,k)} \quad = \quad BlokOffset_{i,j,k} \cdot (BD_x \cdot BD_y \cdot BD_z) + OffsetVnutriBloku_{i,j,k}$$

BD_x, BD_y, BD_z sú rozmery tehličky (brick dimension) a obe funkcie rátajú štandardným spôsobom presne to, čo ich názov napovedá. *Offsety* ostatných voxelov sa určia pomocou adresy prvého voxelu a offsetu z vyhľadávacej tabuľky. Index do vyhľadávacej tabuľky vieme určiť rýchlo iba pomocou operácii sčítania, bitového posunu a logického sčítania. Úspech závisí od vhodného zvolenia rozmerov bloku. V porovnaní s priamočiarym riešením autori dosiahli urýchlenie okolo 30 %. Ďalšie informácie a technické detaily môže čitateľ nájsť v článku (Grimm a kol., 2004).

4.2 Homogénne bunky

Ak všetky voxely bunky obsahujú rovnakú hodnotu, t.j. celá bunka je *homogénna*, výsledok trilineárnej interpolácie môžeme nahradiť hodnotou ľubovoľného z nich. Vo fáze predspracovania môžeme každej bunke priradiť jednobitovú značku určujúcu jej homogénnosť. Takúto bitovú masku uložíme do 1D poľa o veľkosti $(data.x-1)*(data.y-1)*(data.z-1)/8$ bajtov. Počas vykonávania algoritmu ray-casting potom testujeme príslušný bit. Ak je bunka homogénna, vynechávame trilineárnu interpoláciu. Spomenuté urýchlenie bolo implementované v (Razdan a kol., 2001).

4.3 Rýchle určenie normály

Pri tieňovaní za behu programu potrebujeme okrem iného určiť aj normálu objektu v bode dopadu lúča. Normálu určíme pomocou gradientu (2.3), ktorý sa obyčajne ráta metódou centrálnej diferencie (2.4). Gradient vo vnútri bunky môžeme aproximovať trilineárnou interpoláciou gradientov ôsmich voxelov bunky, no tento výpočet je veľmi náročný. Výpočet gradientov vo vrcholoch mriežky dát a ich následné uloženie do pomocnej dátovej štruktúry je častým riešením v mnohých softvérových aj hardvérových implementáciách algoritmov sledovania lúča. Na druhej strane však toto urýchlenie vyžaduje oveľa viac dostupnej pamäte. Aj keby sme disponovali dostatočnou pamäťou, tri trilineárne interpolácie predstavujú dvadsaťštyri násobení, ktoré len ťažko umožňujú interaktívne zobrazovať objemové dáta.

Preto pre aproximáciu gradientu v našej implementácii používame rýchlu metódu výpočtu analytického gradientu. Celý vzorec obsahuje iba dvanásť násobení a nepotrebujeme rátať gradient, pokiaľ je bunka homogénna. Tento prístup použili autori v (Razdan a kol., 2001) a podľa ich výsledkov je degradácia obrazu na prijateľnej úrovni. Gradient v bunke na pozícii (x_i, y_j, z_k) vieme vypočítať zo vzťahu:

$$\nabla f(x_i, y_j, z_k) = (s, t, u) \tag{4.3}$$

kde:

$$s = (1-y_j)(v1 - v0 + z_k(v4 - v3 + v0 - v1)) + (y_j)(v6 - v2 + z_k(v7 - v5 + v2 - v6)),$$

$$t = (1-x_i)(v2 - v0 + z_k(v5 - v3 + v0 - v2)) + (x_i)(v6 - v1 + z_k(v7 - v4 + v1 - v6)),$$

$$u = (1 - x_i)(v3 - v0 + y_j(v5 - v2 + v0 - v3)) + (x_i)(v4 - v1 + y_j(v7 - v6 + v1 - v4)),$$

s, t, u sú parciálne derivácie trilineárnej funkcie vzhľadom na x_i, y_j, z_k.

4.4 Preskakovanie prázdnych úsekov

S veľkosťou dát lineárne narastá výsledný čas generovaného obrázku, nakoľko sa na výsledku podieľajú všetky voxely objemu – pri priamočiarej implementácii metódy sledovania lúča sa jedná o *zaujímavé* a *nezaujímavé* voxely. Vo väčšine prípadov volumetrické dáta obsahujú relatívne veľké množstvo nezaujímavých voxelov, ktoré neprispievajú do výslednej farby. (Zuiderveld, 1995, s.95) vo svojej práci s CT (Computed Tomography) a MRA (Magnetic Resonance Angiography) dátami uvádza, že tieto údaje obsahujú iba 35 % resp. 5.7 % zaujímavých voxelov. Dáta z konfokálneho mikroskopu nie sú výnimkou. Preskočenie nezaujímavých voxelov na ceste lúča má za následok výrazné urýchlenie výsledného času renderovania. Túto techniku môžeme vo všeobecnosti nazvať *preskakovanie prázdnych úsekov* a jej aplikovanie v softvérových implementáciách trasovania lúča (vzhľadom na interaktivitu) je priam nutnosťou.

4.4.1 Efektívne trasovanie lúča objemovými dátami

Rýchlosť, resp. zvýšenie rýchlosti, závisí od použitých prídavných dátových štruktúr. Levoy vo svojom článku (Levoy, 1990) prezentoval dnes veľmi populárnu techniku urýchlenia štandardného algoritmu trasovania lúča. Princíp spočíva v hierarchickom zakódovaní prázdnych miest (buniek) dát do stromovej štruktúry zvanej *oktálový strom* (octree). Pri traverzovaní lúča sa posúvame v *pyramíde* smerom nahor a opačne podľa pozície odoberanej vzorky. Táto technika bola úspešne implementovaná v mnohých projektoch, avšak trpí (pre aplikácie bežiace v reálnom čase) jednou závažnou nevýhodou: prechod touto dátovou štruktúrou a testovanie na nepriehľadnosť bunky môže byť zložité.

4.4.2 Metóda RADC

Urýchlenie lúča zakódovaním vzdialeností (Ray Acceleration by Distance Coding, RADC) je alternatívny prístup, ktorý sa zbavuje režijných nákladov spojených s traverzovaním hierarchickej dátovej štruktúry. Bol predstavený v (Zuiderveld, 1995). Pre každý voxel objemových dát určíme 3D vzdialenosť k najbližšiemu zaujímavému voxelu. Výsledkom tohto procesu je nové pole nazývané *objem vzdialeností* (distance volume). Jeho rozmery sú identické s rozmermi vstupných dát.

Objem vzdialeností určuje pozíciu najbližšej zaujímavej (neprázdnej) vzorky. Hodnota garantuje, že v danom okruhu (sfére) neexistuje žiadna dôležitá vzorka. Zuiderveld vo svojej práci priamočiaro upravil známy *3D-DDA* algoritmus (3D Digital Differential Analyzer). Podľa hodnoty voxela sa ľahko určí vektor, o ktorý posunieme súčasnú pozíciu vzorky. Fakt, že sa vzorka nachádza v zaujímavom regióne[2], nám určuje nulová hodnota bunky vo vyhľadávacej tabuľke. V ostatných prípadoch sa nachádzame v nezaujímavom regióne. Navyše poznáme maximálnu vzdialenosť, ktorú môžeme bez obáv preskočiť. Toto číslo je aproximácia euklidovskej vzdialenosti k hranici zaujímavej bunky.

[2]Bunka, ktorá obsahuje aspoň jeden neprázdny voxel.

Zuiderveld na základe vlastných experimentov uvádza redukciu počtu odobratých vzoriek pozdĺž lúča o faktor medzi 5 a 20 v závislosti od množstva odoberaných vzoriek (sampling rate). V porovnaní s *tradičnými* metódami (napr. zakódovanie do pyramídy) RADC ponúka lepší výkon. Redukcia počtu odobratých vzoriek je úzko spojená s charakteristikou objemových dát. Na základe Zuiderveldových výsledkov porovnania vlastností typických pre CLSM dáta a súčasného porovnania RADC metódy s Levoyovou metódou, mal RADC lepšie výsledky čo sa týka počtu asociovaných operácii. Podľa Zuiderveldových výsledkov porovnania RADC metódy s Levoyovou a na základe vlastností typických pre CLSM dáta, ktoré boli popísané v časti 1.4 má, RADC lepšie výsledky čo sa týka počtu asociovaných operácii. Navyše:

- Modifikovaný 3D-DDA algoritmus je rýchlejší ako pohyb v pyramíde a testy na prienik s bunkou.
- Najlepšie výsledky pri použití hierarchických dátových štruktúr sa dosahujú po zistení optimálneho nastavenia algoritmu. V praxi sú tieto nastavenia fixované, čo vedie k horším časom.
- Implementácia RADC je oveľa jednoduchšia. Najťažšia časť je predspracovanie objemových dát (vzdialenostná transformácia).
- Rozšírenie štandardného RADC algoritmu dokáže eliminovať lúče, ktoré netrafia objekt[3], čo vedie k ďalšiemu zníženiu výpočtovej zložitosti.

Hlavnou nevýhodou RADC je použitie veľkého množstva pamäte na uloženie objemu vzdialeností. Vieme, že prístup do pamäte je drahý. Existujú však techniky, ktoré toto veľké množstvo dokážu zredukovať. Jednu z nich prezentuje už niekoľko krát spomínaný (Zuiderveld, 1995).

4.4.3 Výpočet 3D vzdialenosti

Aproximáciu vzdialenosti k najbližšej potenciálne zaujímavej bunke vieme vypočítať technikou *vzdialenostná transformácia* (distance transform). Existuje niekoľko algoritmov na jej výpočet. (Herman a kol., 1992) vo svojej práci predstavil algoritmus používajúci bázické *skosené vektory* (chamfer vectors) a celočíselnú aproximáciu ich dĺžok. Skosenú metriku a vektory detailne vysvetľuje (Borgefors, 1986). Algoritmus v dvoch prechodoch dátami aproximuje riadok po riadku vzdialenosť od hraníc buniek za použitia šablón 3x3x3, resp. 5x5x5. Výsledok je kombináciou oboch prechodov — zhora dole a naopak. Výpočtová zložitosť algoritmu závisí od počtu prvkov šablóny. V 3D prípade je výpočet vzdialenosti časovo náročný.

[3]Rozšírenie je založené na nasledujúcej myšlienke: minimálna vzdialenosť, ktorú lúč stretne na svojej ceste, je rovná vzdialenosti od najbližšieho zaujímavého lúča.

Kapitola 5

Návrh a implementácia

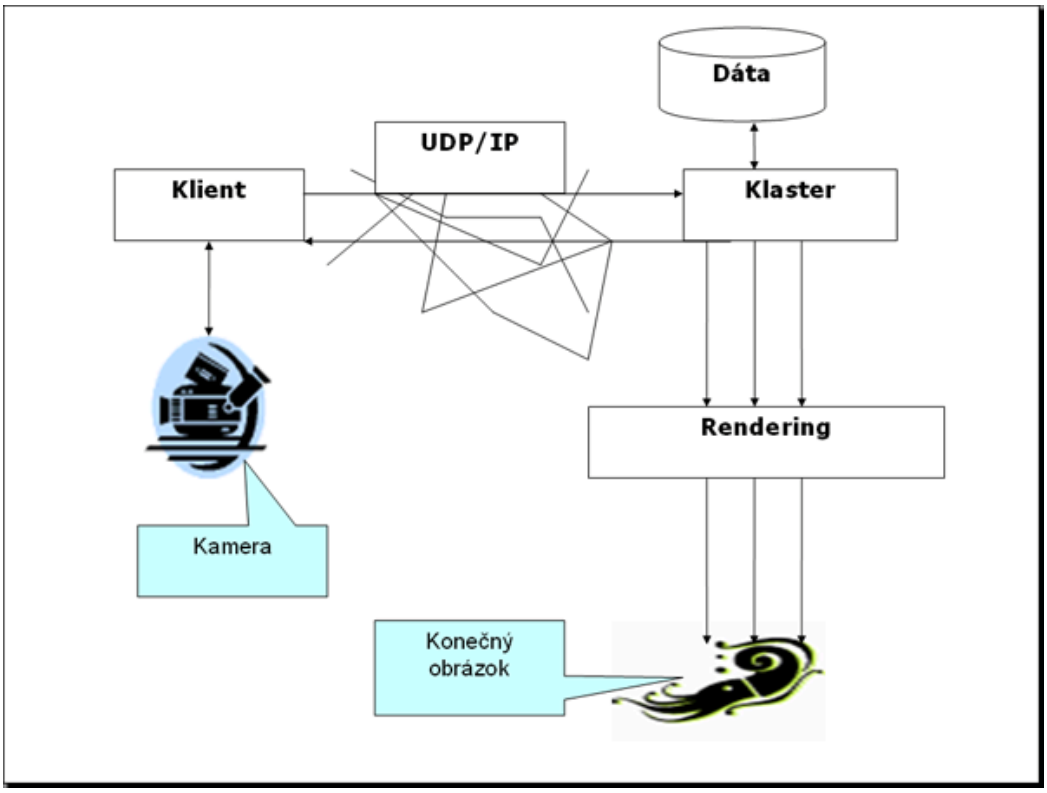
Vývoj nástrojov pre vizualizáciu dát z konfokálnej mikroskopie prechádzal niekoľkými etapami. Kvalitný návrh rozsiahlej a navzájom poprepájanej aplikácie bol jedným z kľúčových bodov projektu. Pri tvorbe jednotlivých *modulov* (programov, tried) sme museli mať na pamäti prepojenia vstupov a výstupov funkcií, ako čitateľ uvidí v ďalšom texte.

5.1 Návrh

Návrh architektúry systému je zobrazený na obrázku 5.1. Celá aplikácia je typu *klient-server*. Klient, program bežiaci na bežnom počítači, pošle serveru požiadavku vyrenderovať vstupné objemové dáta. Server, homogénny počítačový klaster požiadavku spracuje a spustí proces renderovania na určenom počte procesorov. Výsledný obraz posiela spať klientovi. Samotnú serverovú časť môžeme opäť rozdeliť na dve časti - *lokálnu renderovaciu* a *paralelnú kompozičnú*. Beh programu je stručne popísaný v nasledujúcej časti.

Na začiatku renderovania klient pošle hlavnému uzlu klastra (master node) informácie *ako* a *čo* chce renderovať. Pôvodný plán zahŕňal posielanie vstupných objemových dát z klienta na server, no kvôli rýchlosti a efektivite práce používateľa sme sa rozhodli posielať iba cestu k dátam. V našom prípade sa tak či tak nachádzajú na serveri (klastri). Implementáciu vstupu z klienta spomíname ako ďalšie možné rozpracovanie projektu. Hlavný uzol načíta dáta a rozdelí ich medzi procesory. Každý procesor si v tomto okamihu inicializuje všetky interné štruktúry a nastavenia (inicializácia). Začína samotné renderovanie, kde procesory jednotlivé dáta spracujú algoritmom vrhania lúča (renderovanie), výsledné obrázky si medzi sebou vymieňajú (binary-swap kompozícia), finálne obrázky sú pozbierané hlavným uzlom (zber obrázkov) a výsledný obraz je poslaný po sieti klientovi (poslanie a zobrazenie).

Sieť, ktorou sa posielajú výsledné obrázky klientovi, môže byť aj internet. Náš návrh umožňuje aj takéto riešenie, ktoré je však zatiaľ realizovateľné len v obmedzenej miere. Väčšina pripojení do siete internet zatiaľ nedisponuje dostatočnou šírkou pásma. Firmy, nemocnice a pod. si dnes nemôžu (zrejme ani v blízkej budúcnosti) dovoliť financovať drahé paralelné stroje. Na druhej strane sa na Slovensku rozmáha rýchly internet a pripojenie optickým vláknom do budovy firmy či bytu. Ceny týchto komerčných služieb klesli už na úroveň dostupnú pre značne veľkú časť firiem. Podobne ako v prípade iných pripojení (modem, ISDN, DSL) sa aj v tomto prípade očakáva ďalšie znižovanie cien a zvyšovanie rýchlosti. Mnoho autorov vo



Obrázok 5.1: Návrh architektúry systému. Dáta sú uložené na diskovom poli hlavného uzla klastra. Klient mení pozíciu kamery a nastavenia paralelného renderera. Klaster vygenerované obrázky posiela protokolmi TPC/IP alebo UDP/IP klientovi.

svojich prácach poukazuje na fakt, že v ich krajine je dostatok paralelných počítačov, z ktorých je väčšina v stave *idle* počas celého dňa. Pri tvorbe tejto práce sme sa zamýšľali aj nad možnosťou využitia týchto výkonných počítačov v medicíne alebo biológii, pričom by organizácia/inštitút ani nemusela výkonný klaster vlastniť. Existencia softvéru schopného pracovať so vzdialeným klastrom by tento problém riešila. Používateľovi (napr. lekár alebo vedec) by sa stačilo prihlásiť na *dostupný* paralelný počítač, poslať mu potrebné dáta a ako odpoveď by dostal zobrazený trojrozmerný model dát, s ktorým by mohol ďalej manipulovať. Ak by aj veľká inštitúcia ako nemocnica vlastnila takýto výkonný počítač, určite by sa (z prevádzkových a iných dôvodov) nemohol nachádzať v jednej miestnosti s lekármi, či dokonca pacientmi. Vizualizácia dát na vzdialenom počítači by umožnila uzavrieť počítač vo vhodnej miestnosti, ktorá by bola napojená vysokorýchlostnou sieťou na pracovné stanice budovy. Preto jeden z návrhov ďalšieho možného uberania sa tejto práce by mohla byť implementácia a nasadenie vzdialenej vizualizácie do inštitúcií ako nemocnice a pod.

Jednotlivé časti programu a fázy fungovania budú detailne popísané v ďalších častiach tejto kapitoly.

5.2 Hardvér a softvér klastra

Vývoj paralelného programu prebiehal na homogénnom 16 uzlovom PC klastri IBM Cluster 1350[1], ktorý sa nachádza v ILC na FMFI UK v Bratislave. Klaster je zložený z ôsmich počítačov, z ktorých sedem sú výpočtové uzly (compute nodes) a posledný je uzol pre manažment (management node, master node). Prepojenie je realizované cez gigabitový Ethernet prepojený UTP káblami a optickou sieťou Myrinet 2000. Táto sieť má prenos 2+2 Gigabit/s pri full-duplex móde. Latencia MPI je v rozmedzí 2.6μ až 3.2μ. Jednosmerná rýchlosť MX prenosu údajov je 247 MB/s resp. 495 MB/s, rýchlosť TCP/IP prenosu je 1.98 Gbit/s, resp. 3.95 GBit/s. Záleží od použitia jedno- alebo dvojportovej sieťovej karty. Viac informácií o technológiách, komponentoch a softvéri možno nájsť na oficiálnej stránke (http://www.myri.com/). Hlavný hardvér jednotlivých uzlov je zobrazený v tabuľke. Na každom počítači klastra je

Komponenta	**Typ**
CPU Master	2xIntel Xeon, 2.40GHz, L2 512kB
CPU Slaves	2x Intel Xeon, 2.80GHz, L2 512kB
Pamäť	4x512MB PC2100 CL2.5 ECC DDR SDRAM RDIMM
HDD	6x 74.3GB 10K-RPM ULTRA 160 SCSI Hot-Swap SL
RAID	IBM Netfinity ServeRAID controller
Eth karty	Intel Corp. 82546EB Gigabit Ethernet Controller
Eth prepínače	Switch CISCO Catalyst 3550
Myr karty	Myrinet-2000-Fiber 64/32-Bit, 66/33MHz adapter
Myr prepínače	Myrinet-2000 switch

Tabuľka 5.1: Parametre klastra IBM Cluster 1350.

nainštalovaný Red Hat Linux 9.0, kernel verzia: 2.4.20-8smp. Každý procesor má na základe našich testov k dispozícii približne 800 MB RAM. Ďalší nainštalovaný softvér predstavuje: PGI CDK 5.1 (Portland Cluster Development Kit), MPICH 1.2.5 (Ethernet), MPICH-GM-2.0.10 (Myrinet). Program a jeho moduly boli kompilované/linkované pomocou kompilátorov GNU `g++` (ver. 3.2.2) a PGI `pgCC` (ver. 5.1).

5.3 Implementácia

Triedy, funkcie a programy sme vytvárali skoro od *začiatku*. K tomu nás viedlo niekoľko skutočností. Existuje veľké množstvo prác a experimentov na poli paralelnej volumetrickej vizualizácie, no len veľmi málo z nich má sprístupnené zdrojové kódy svojich implementácii. Ak aj sú k dispozícii, prichádzajú do cesty problémy s licenciou a copyrightom (na rozdiel napr. od open-source projektov). Alebo sú zdrojové kódy (triedy) veľmi zviazané z danou aplikáciou (ich štúdium a úprava by boli možno ešte viac časovo náročnejšie ako tvorba nového kódu), resp. sú komplikované a zaťažené rôznymi, pre nás nepotrebnými nákladmi navyše. Nanešťastie je väčšina kombináciou predchádzajúcich dvoch. Výsledok našej implementácie sú triedy (a funkcie), ktoré sú jednoduché, a pritom dostatočne silné na to, aby sa pomocou nich vybudoval celý paralelný program. Projekt je otvorený, aby jeho výsledky so zdrojovými kódmi mohli bez problémov použiť ďalší vývojári.

[1]IBM Cluster 1350: http://www-03.ibm.com/systems/clusters/hardware/1350.html

Stará múdrosť vraví: "Elegancia a rýchlosť sa nedajú dosiahnuť súčasne". (Zuiderveld, 1995) implementoval svoj program pomocou objektovo orientovaných metód (OOP). Naše riešenie sa práve naopak snaží vyhnúť ďalším nákladom spojených s OOP návrhom v kritických miestach programu. Všetky triedy použité v blokoch programu, ktoré bežia real-time, obsahujú iba dátové typy a preťažené operátory. Všetky ostatné funkcie sú mimo tried. Takýto návrh tried umožňuje ich ľahké ďalšie rozšírenie a modifikáciu pri ich budúcom použití. Nepoužívame žiadne techniky OOP návrhu programov (zapuzdrenie, polymorfizmus, dedenie). Jedna z ďalších optimalizácii kódu predstavuje použitie *inline* funkcií. Podľa (Bruckner, 2004) je táto technika jedna z najhodnotnejších stratégií pre manuálnu optimalizáciu kódu. Za ostatné menšie optimalizácie spomenieme redukciu počtu náročných aritmetických operácií, ich zámenu za bitový posun a transformácie (časti) funkcií za rýchlejšie. Práca prekladača, linkera a *shell* skriptu `mpirun` bola optimalizovaná voliteľnými parametrami pre dosiahnutie čo najrýchlejších výsledkov a fungovania programu.

Programy boli implementované v jazyku C++. Vývoj prebiehal pod operačným systémom Linux za použitia viacerých nástrojov spomenutých v závere kapitoly. Aby sme dosiahli dobrú portabilitu medzi rôznymi architektúrami a OS, bolo snahou vytvoriť zdrojové kódy v súlade s ANSI C/C++ štandardom. Taktiež sme vytvorili viacero implementácii funkcie, ktorá sa inak vytvára pod rôznymi OS. Vďaka konštrukcii `#if ! defined WIN32 ... #endif ... #else ... #endif` dokážeme jeden zdrojový kód kompilovať tak pod OS Linux, ako aj pod OS Windows.

Je potrebné poznamenať, že program *nebol* vytváraný konkrétne pre daný klaster. Našou snahou bolo vytvoriť čo najvšeobecnejšiu implementáciu algoritmov aplikovateľnú v najrôznejších architektúrach.

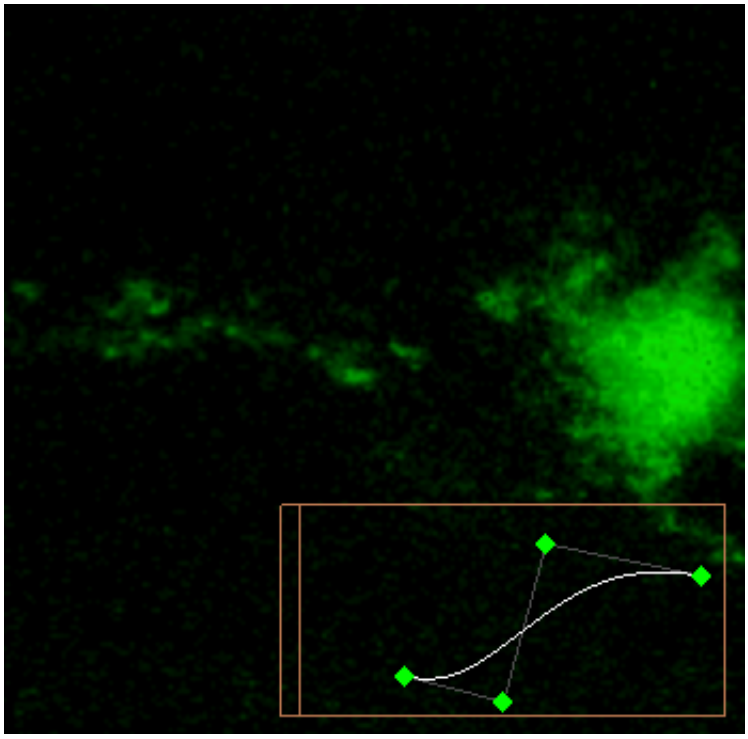
Popis projektu sme rozdelili do troch častí. V prvej sa venujeme aplikácii na strane klienta a spojením klient-server. V druhej podrobnejšie rozoberáme lokálne renderovanie objemových dát procesorom a v tretej časti je popísaná implementácia kompozície a inicializácie renderovania.

5.3.1 Klient a komunikácia so serverom

Klient je navrhnutý ako jednoduchá aplikácia, ktorá slúži ako vrstva medzi používateľom a renderovacou stanicou - klastrom. Obsahuje jednoduché rozhranie nenáročné na ovládanie. Používateľ pohybom ukazovacieho zariadenia rotuje kameru ako keby po sfére okolo objektu, ktorý sa nachádza v počiatku scény (0,0,0). Pohyb kamery sme ale nijako neobmedzili, a tak je ďalej možné pomocou klávesnice upravovať jej pozíciu a pohyb všetkými smermi. Používateľ môže posúvať v priestore aj polohu samotných dát.

Aplikácia posiela sieťou klastru nastavenia (pozícia kamery, spôsob renderovania), prijíma a zobrazuje vygenerované obrázky. Zobrazovanie sme vyriešili použitím knižnice OpenGL/GLUT. Z prijatých dát sa vygeneruje textúra, ktorá je namapovaná na zadanú veľkosť obrazovky.

Pre aplikáciu klienta sme vytvorili *editor prechodovej funkcie*, ktorý umožňuje používateľovi interaktívne meniť transfér funkciu (obrázok 5.2). Pri CLSM dátach takýto editor potrebujeme. Ako bolo spomenuté v časti 1.4.2, často skúmame neznáme vnútorné štruktúry a kompletná binárna klasifikácia nie je možná. Transfér funkciu sme implementovali vo forme vyhľadávacej tabuľky (look-up table). Viac informácii o hodnotách CLSM dát môže čitateľ nájsť v časti 1.4. Editor prechodovej funkcie bol implementovaný ako Bezierova krivka. Používateľ môže



Obrázok 5.2: Editor prechodovej funkcie implementovaný ako Bezierova krivka.

interaktívne pridávať, odoberať riadiace body, meniť ich pozíciu a stupeň krivky. Výška krivky je namapovaná na interval $\langle 0, 1\rangle$ a určuje nepriehľadnosť pre daný bod. Používateľ dokáže jediným kliknutím zarovnať všetky riadiace body do jednej horizontálnej priamky. K tomuto kroku je vyhradený malý vertikálny panel v ľavej časti editora. Pri akejkoľvek zmene krivky používateľom sa nanovo generuje vyhľadávacia tabuľka. Na základe našich experimentov je použitie tabuľky badateľne rýchlejšie ako priamy výpočet pri modifikáciách krivky. Teóriu Bezierových kriviek môže čitateľ nájsť napríklad v (Žára, 1998).

Spojenie klient-server

Pre komunikáciu medzi aplikáciami klienta a servera sme navrhli použiť protokol UDP/IP oproti TCP/IP. Protokol TCP/IP ponúka spojené a spoľahlivé služby. Tak kvalitný prenos pri zobrazovaní obrazu v reálnom čase nepotrebujeme, dôležitá je rýchlosť prenosu. Nespojené a nespoľahlivé spojenie cez datagramy UDP protokolu nám umožňuje prenos väčšieho množstva údajov, keďže sa sieťou neposielajú potvrdenia. Výborný kurz k soketom a TCP/UDP/IP protokolom napísal (Dostál, 2003). Spojenie klient-server sme začali implementovať protoko-

Trieda (Class)	Popis
`CVektor3`	Vektor reálnych čísel a operácie s vektorom
`template TVektor`	Šablóna vektora
`CSvetlo`	Typ, pozícia a parametre svetla
`CFarba`	Model RGBA, operácie, farieb. modely
`CShader`	Dátová štruktúra materiál a Phongovo tieňovanie
`CRay`	Lúč a základné operácie
`CMatica`	Matica a rôzne funkcie pre prácu s maticami
`CKamera`	Kamera, rotácie, škálovania, pohyb a iné operácie
`CKvader`	Kvader v priestore, operácie, prienik lúča a kvádra
`CData`	Dáta, adresné schémy, Bricking, homogénne dáta
`CBezKrivka`	Bezierova krivka, výpočet krivky, vykresľovanie
`CRaycaster`	Raycasting renderer

Tabuľka 5.2: Popis C++ tried lokálneho renderovania

lom TCP/IP. Upravili sme voľne dostupné knižnice `libtcp++-0.1.2` pre naše potreby, avšak serverová časť bola zbytočne zložitá. Upustili sme od tohto riešenia a rozhodli sa pre protokol UDP/IP. Spojenie bude implementované v ďalšom rozpracovaní diplomovej práce.

5.3.2 Server - Lokálne renderovanie

Implementovali sme algoritmus ray-casting popísaný v kapitole 2. Hlavnými dôvodmi bola jeho flexibilita (oproti renderovaniu za použitia hárdveru, napr. GPU), ktorú pri vizualizácii dát z konfokálnej mikroskopie potrebujeme. Okruh algoritmov sledovania lúčov je dobre preštudovaný a ponúka veľa možných urýchlení. Aktuálna verzia pracuje s jednokanálovými CLSM dátami. Vytvorili a implementovali sme nasledujúce triedy: Dátová štruktúra `CData` reprezentuje laser konfokálneho mikroskopu. Trieda drží väčšinu informácii o objemových dátach vo forme rôznych dátových štruktúr. Konkrétne obsahuje:

- Objemové dáta
- Vyhľadávaciu tabuľku homogenity buniek
- Objem vzdialeností
- Dáta pri inej adresovanej schéme (Bricking) + vyhľadávacie tabuľky
- Objekt LSM súbor

Trieda `CRaycaster` obsahuje všetky triedy potrebné pre proces renderovania. Sú to nastavenia obrazovky, lúčov, Bezierovej krivky, vstupné údaje pre editor prechodovej funkcie, svetla, farby, tieňovania, vyhľadávacích tabuliek pre algoritmus RADC, textúru a rotáciu kamery.

Proces renderovania

Každým pixelom obrazovky vrháme do priestoru s objemovými dátami lúč. Ak lúč trafí dáta (`CKvader`), začneme pozdĺž neho odoberať vzorky, ktoré kombinujeme do výslednej farby a nepriehľadnosti. Srdce algoritmu ray-casting sa skladá z nasledujúcich 4 častí: posun a odobratie

ďalšej vzorky, interpolácia, tieňovanie, kompozícia.

Pre každý lúč si uchovávame smer a počiatočnú pozíciu (`CRay`). V literatúre sa tento spôsob zvykne označovať ako šablóna lúča (ray template). Veľkosť kroku je fixovaná (určí ju napr. používateľ), t.j. vzorky odoberáme v pravidelných intervaloch. Pri zmene vzdialenosti medzi odoberanými vzorkami sa musí brať do úvahy aj zmena nepriehľadnosti. Preto sme do implementácie zahrnuli aj vzťah 2.12.

Na výpočet hodnoty vo vnútri bunky sme implementovali vzťah (2.5) kapitoly 2.2.5 - trilineárna interpolácia. Pomocou danej hodnoty a vyhľadávacej tabuľky potom určíme nepriehľadnosť danej vzorky.

Tieňovanie vylepšuje realizmus. Implementovali sme Phongov osvetľovací model (`CShader`), ktorý sme popísali v kapitole 2.2.4. Osvetľovací model používame s jedným zdrojom bieleho svetla (`CSvetlo`), ktoré sa nachádza v rovnakom bode ako pozorovateľ (`CKamera`). Dosahujeme tak efektu pohybujúceho sa svetla (Minerova lampa). Tento prístup bol použitý aj v práci (Razdan a kol., 2001), ako vhodné riešenie cieľov interaktívnej navigácie a skúmania objektov v priestore.

Kompozíciu vzoriek sme implementovali odpredu-dozadu ako je uvedené v časti 2.2.6. Predčasné ukončenie lúča bol hlavný dôvod použitia vzťahu 2.10 v renderovacej funkcii. Výsledné farby a nepriehľadnosti pixelov sa uložia do poľa `textura` triedy `CRaycaster`. Tieto dáta sa budú posielať klientovi na zobrazenie.

Typické CLSM dáta sú veľmi slabo navzorkované v z-tovej súradnici. Konfokálny mikroskop nedokáže nasnímať toľko rovín ako je rozlíšenie jedného rezu preparátu. Inak povedané, vzdialenosti medzi vzorkami v XY rovine sú oveľa menšie ako vzdialenosti v robine YZ. Preto sme implementovali používateľom nastaviteľnú veľkosť bunky, a tak dosiahli vernejšiu vizualizáciu skutočných rozmerov preparátu.

5.3.3 Optimalizácia rýchlosti algoritmu sledovania lúča

Základný algoritmus ray-casting je v čisto softvérovom prevedení pomalý. Od jeho uvedenia do sveta počítačovej grafiky bolo vymyslených mnoho dômyselných urýchlení a vylepšení. Náš program implementuje hneď niekoľko z nich.

Ak počas sledovania lúča odoberieme vzorku z homogénnej bunky, t.j. všetky voxely majú rovnakú hodnotu, jej hodnota bude konštanta. Snažíme sa vyhnúť zdĺhavej trilineárnej interpolácii. Vstupné dáta spracujeme a vytvoríme pomocné pole (udržiavame ho v `CData`), v ktorom každý bit označuje homogenitu bunky. Ďalšie detaily sa nachádzajú v sekcii 4.2.

Rozdiel v rýchlosti súčasných procesorov oproti pamätiam je skutočne veľký. Dáta z hlavnej pamäte nestíhajú prichádzať na vstup dostatočne rýchlo a tento fakt môže limitovať náš algoritmus. Preto používame rôzne adresné schémy (sekcia 4.1) pre rýchlejší prístup do pamäte. Presnejšie, pre uchovanie okolitých a následne spracovávaných dát v L2 cache pamäti procesora. Prvé verzie programu implementovali riešenie (Sakas a kol., 1996). Ako bolo spomenuté, ich riešenie vyžadovalo použitie troch relatívne veľkých vyhľadávacích tabuliek. Neskôr

sme implementovali oveľa lepšie riešenie Brucknera. Úložný priestor vyhľadávacích tabuliek sa zmenšil iba na 224 bytov nezávisle od veľkosti dát. Autori testovali rovnaké procesory (P4 Xeon, 512kB L2 cache), aké sa nachádzajú v počítačoch nášho klastra. Na základe testov uvádzajú optimálnu veľkosť tehličky 64kB (32x32x32) pre zrýchlenie o faktor okolo 2.8. Pre zistenie pozície voxelov v bunke vyvinuli veľmi efektívne riešenie, ktoré sme tiež implementovali. Ak poznáme rozmery tehličky, výpočet indexu efektívne zapíšeme do zdrojového kódu. Samozrejme, naša implementácia uvažuje aj s univerzálnym riešením. V našom prípade dostávame nasledovnú funkciu:

```
void getIndexLut(int i, int j, int k, CData& d, UINT32* v) {
    // dimenzia tehlicky (brick dim)
    int _i= (i % bd_x) << 2;
    int _j= (j % bd_y) << 1;
    int _k=(k % bd_z);

    index=((((_i&0x7F)+4)&0x80)+(((_j&0x3F)+2)&0xC0)
    +(((_k&0x1F)+1)&0xE0)) >> 5;

    v[0]=getIndexBrick(i,j,k,d);
    v[1]=v[0]+d.lut[index][0]; // 100
    v[2]=v[0]+d.lut[index][1]; // 010
    v[3]=v[0]+d.lut[index][2]; // 110
    v[4]=v[0]+d.lut[index][3]; // 001
    v[5]=v[0]+d.lut[index][4]; // 101
    v[6]=v[0]+d.lut[index][5]; // 011
    v[7]=v[0]+d.lut[index][6]; // 111
}
```

Súčasná implementácia programu podporuje obe adresné schémy, celkovo teda tri (ak rátame aj základný lineárny layout pamäte). Pomocné dátové štruktúry sú uložené v triede `CData` a sú naplnené vo fáze inicializácie, o ktorej ešte budeme hovoriť.

V snahe o čo maximálnu rýchlosť generovania obrázkov bolo samozrejmosťou implementovať algoritmus preskakujúci prázdne úseky objemu. Už pri načítaní CLSM dát je vykonaná najjednoduchšia segmentácia - prahovaním. Farba pozadia je definovaná nulovou hodnotou. Pôvodný plán predstavoval implementáciu Levoyovho algoritmu hierarchického zakódovania dát pomocou oktálového stromu. Začali sme implementovať algoritmus spomínaný v časti 4.4.1, no po opätovnom preskúmaní výhod a nevýhod Levoyovho Riešenia, a braní do úvahy možnosti klastra (v porovnaní s inými riešeniami), sme od implementácie upustili a vydali sa inou cestou. Za konečné riešenie sme zvolili metódu RADC popísanú v časti 4.4.2. Jednotlivé uzly klastra disponujú dostatočne veľkou pamäťou (aj pri alokovaných ďalších pomocných dátových štruktúrach), a preto nie sme veľmi limitovaný. Výhody a vhodnejšie riešenie oproti Levoyovi bolo popísane v kapitole 4. Zdôvodnenie dostatočne veľkej pamäte bude spomenuté v nasledujúcej časti (paralelizmus dát). K metóde RADC sme implementovali techniku vzdialenostná transformácia pomocou 3D šablón, o ktorej sa zmieňujeme v časti 4.4.3. Táto technika je v trojrozmernom priestore časovo náročná. V našom prípade objektového paralelizmu dát medzi až šestnásť procesorov je proces predspracovania údajov stále *použiteľne* časovo náročný. Namiesto reálnych čísel používame v našej implementácii algoritmu celé čísla kvôli dosiahnutiu

lepších časov vo fáze predspracovania. Pre minimalizovanie výpočtových nákladov používame v implementácii modifikovaného 3D-DDA algoritmu dve vyhľadávacie tabuľky. Podľa experimentov Zuidervedla, každý krok trvá okolo $1150ns$, čo je približne $2x$ viac ako štandardný 3D-DDA algoritmus.

5.3.4 Server - binary swap kompozícia

Pri návrhu celej aplikácie bolo našou snahou dosiahnuť čo najväčšiu nezávislosť jednotlivých tried a použitých algoritmov. Preto sme nezávisle na renderovacom algoritme implementovali algoritmus binary swap (popísaný v časti 3.10). Vďaka tomu možno renderovaciu časť ľahko nahradiť s inou (napr. využívajúcu dnes populárne GPU) doplnením príslušným knižníc a menšou úpravou zdrojového kódu.

Náš klaster obsahuje kvalitnú a rýchlu sieť Myrinet a gigabitový Ethernet, preto sme sa rozhodli nemeniť ďalší výpočet za menšie množstvo komunikácie ako v prípade (Stompel a kol., 2003). Ďalším dôvodom bol fakt, že pre malé množstvo procesorov (náš klaster obsahuje šestnásť) sa tento algoritmus správa približne rovnako rýchlo ako binárna výmena.

Ako bolo spomenuté v časti 3.9, pri návrhu paralelného programu sme mali na výber v podstate dve možnosti – použiť:

- obrazový paralelizmus
- objektový paralelizmus

Usilovali sme sa o dosiahnutie čo najvyššej rýchlosti, ako prvé v poradí prišlo do úvahy prvé riešenie. Každý zo šestnástich procesorov disponuje voľnou pamäťou o veľkosti cca. 800MB RAM. Objemové dáta by museli byť zreplikované medzi všetky procesory, no vyhli by sme sa kompozícii finálneho obrazu. Všetky pomocné dátové štruktúry zaberajú nezanedbateľné miesto pamäte, čo by značne limitovalo možnosti vizualizácie. To by bolo zlé riešenie, a preto sme sa priklonili k druhej možnosti. V literatúre sa zvykne označovať aj ako *paralelizmus dát*. Po rozdelení objemu na menšie časti nám ostáva dostatočné množstvo pamäte pre pomocné dátové štruktúry. Pamäť prestala byť problémom. Náš projekt sme vytvorili na počítači s disribuovanou pamäťou (klastri). Jednotlivé uzly sú prepojené vysokorýchlostnou sieťou Myrinet 2000, resp. Ethernet. Časti objemových dát sme rozdelili medzi procesory a jednotlivé obrazové príspevky skladáme pomocou algoritmu typu *utrieď na konci*. Tento prístup je podľa nás najvhodnejší pre paralelné objemové renderovanie na klastri. Viac detailov v nasledujúcich častiach a v kapitolách 3.9 a 3.10.

Inicializácia a plán výmeny

Základom tejto časti programu sú 2 triedy, ktoré sme implementovali: `class KDTree` a `class Procesor`.

Trieda `KDTree` je implementáciou stromovej štruktúry k-D strom pre 3D priestor. Používame ju pre rozdelenie hlavných dát a výpočet plánu kompozície. Vďaka nemu vieme jednoducho určiť procesory, ktoré sa majú medzi sebou kombinovať a v akom poradí. Podrobnejší popis a odkazy nájde čitateľ v časti 3.10.1. Okrem toho k-D strom udržiava pozíciu čiastkových dát v priestore pre jednotlivé procesory a používame ho pri počiatočnej distribúcii dát.

Trieda `Procesor` je základom celého programu bežiaceho na každom procesore. Obsahuje nasledujúce dátové štruktúry:

- plán kompozície (s kým a ako kombinujeme prijaté dáta)
- k-D strom (rozdelenie dát, máme prehľad, ktorý procesor drží ktoré dáta a kde)
- vygenerovaný obrázok
- buffre na prijatie a posielanie obrázkov vo fáze kompozície
- renderovací algoritmus, v našom prípade CRaycaster

Pred začiatkom renderovania a kompozície musia byť inicializované niektoré dátové štruktúry. Procesor obdrží od klienta informáciu, ktoré dáta si želá používateľ vizualizovať. Po úspešnej inicializácii MPI zistí svoje ID a počet procesorov zapojených do výpočtu (`MPI_Comm_rank()`, `MPI_Comm_size()`). Nasleduje inicializácia renderovacej triedy - `CRaycaster`. Alokuje sa pamäť pre Bezierovu krivku, vyhľadávacie tabuľky, výslednú textúru (čiastkový obrázok), nastavujú sa rozmery editora prechodovej funkcie. Hlavný uzol načíta parametre vstupných dát a pomocou MPI *broadcast* funkcie `MPI_Bcast()` ich pošle všetkým aktívnym procesorom. Procesor po obdržaní informácií vytvorí k-D strom a pomocou neho *plán kompozície*. Ten sa skladá z ID procesora a binárnej hodnoty, ktorá udáva poradie skombinovania čiastkových obrázkov. Hlavný uzol potom prechádza strom, načíta a pošle príslušnú časť dát pre procesor i (nachádzajúci sa v liste). Pri posielaní a prijímaní správ používa MPI vlastné buffre. Ich dostupnosť je základom úspešného fungovania programu. V snahe vyhnúť sa ich preplneniu pri distribúcii dát medzi procesory sme implementovali vlastné funkcie `posli_data()` a `prijmi_data()`, ktoré obsahujú vlastný malý buffer. Veľké dáta sa posielajú cez tento buffer, takže prijímajúci procesor načítava dáta ešte počas ich posielania zo zdroja. Pri vývoji programov používajúcich MPI je potrebné venovať zvýšenú pozornosť preplneniu buffrov, a teda organizácie komunikácie (PACS, 2001). Veľmi ľahko totiž môžeme dospieť k *uviaznutiu* (deadlock), ako autori v práci demonštrovali na jednoduchých príkladoch. Aj preto v našej implementácii používame namiesto dvojice funkcií `MPI_Send()` a `MPI_Recv()` funkciu `MPI_Sendrecv()` všade tam, kde je to možné.

Načítaním dát zo súboru alebo prijatím dát zo siete začína proces inicializácie ray-casting algoritmu. Dáta sú uložené v jednorozmernom poli triedy `Procesor->CRaycaster->CData`. Podľa rozmerov a pozície dát prispôsobíme pozíciu kamery (`CRaycaster->CKamera`). Z dát vytvoríme pomocnú štruktúru objem vzdialeností. Tento proces je časovo najnáročnejší spomedzi všetkých inicializácii (objemom prechádzame 3D maskou). Nasleduje vygenerovanie objemových dát uložených do tehličiek, vytvorenie vyhľadávacej tabuľky pre homogénne bunky a overenie nastavení. Konečne môžeme vstupné dáta vygenerovať. Výsledok je uložený do poľa. Generovanie na každom procesore prebieha bez akejkoľvek komunikácie s ostatnými procesormi.

Kompozícia obrázkov

Implementácia tejto časti algoritmu je jednoduchým prechodom dvoma poľami (vlastný a prijatý obrázok) dát. Každý pixel je reprezentovaný troma farebnými zložkami + alfa kanál

(RGBA) uložený v šestnástich bajtoch (4x `float`). Dvojica pixelov je zmiešaná (blending) dokopy podľa *alfa* zložky a plánu kompozície.

Fázy algoritmu binary swap

Algoritmus binárnej výmeny vyžaduje 2^n procesorov a pracuje v n etapách, pričom do výpočtu sú zapojené *všetky* procesory počas celej kompozície. V každej etape si dvojica procesorov vymení polovice čiastočného obrázku pomocou funkcie MPI `MPI_Sendrecv()`. Táto blokovaná operácia kombinuje v jednom volaní poslanie a prijatie dát od jedného zdroja, čo je presne náš prípad. Posledná etapa algoritmu zahŕňa pozbieranie $(2^n - 1)$ obrázkov hlavným uzlom (funkcia `MPI_Recv()`), všetky ostatné uzly posielajú časť konečného obrázka o veľkosti $1/2^n$ funkciou `MPI_Send()`.

Urýchlená verzia algoritmu binárna výmena

Reprezentácia hodnoty RGBA pomocou 16 bajtov vyžaduje vo fáze kompozície dostatočne rýchlu sieť. Upravili sme preto algoritmus binárna výmena nasledovne. V rámci výpočtov na procesore ďalej pracujeme s hodnotou RGBA ako s 4×4 bajtami, no pri posielaní správ sieťou hodnotu konvertujeme na 4×1 bajty. Implementovali sme oba algoritmy z dôvodov dosiahnutia lepšej interaktivity aj na pomalšom Ethernete a testovania sietí.

5.3.5 Vývoj softvéru nástrojmi GNU

Domnievame sa, že voľba slobodného softvéru je najlepšou cestou pre zdieľanie nášho projektu a zdrojových kódov. Aby sme umožnili ostatným slobodne používať, modifikovať a redistribuovať náš softvér, rozhodli sme sa naň aplikovať *GNU GPL licenciu* (GNU General Public Licence)[2]. Táto licencia garantuje všetky práva, ktoré sú nutné na to, aby sme náš program urobili slobodným a umožňuje použitie našich zdrojových kódov vo všetkom slobodnom softvéri. "Slobodný softvér dáva jeho používateľovi možnosť kopírovať, distribuovať, študovať, meniť a zlepšovať ho" (Richard Stallman). GPL je jedna z možností ako implementovať *copyleft*. A my sme ju aplikovali. Takýto softvér totiž poskytuje najlepší spôsob jeho rozšírenia medzi verejnosť, ľudí, vývojárov.

Pri tvorbe nášho programu sme použili GNU systém pre vývoj programov (The GNU build system)[3]. Tento systém má 2 hlavné ciele: vývoj portabilných programov a zjednodušenie tvorby programov dodávaných vo forme zdrojových kódov. Skladá sa z nasledujúcich balíčkov: *Autoconf, Automake, Libtool, Autotools*. Výsledkom použitia GNU systému sú skripty, ktoré na jednej strane umožnia jednoducho vytvoriť a nainštalovať program, na strane druhej umožňujú vývojárom vytvárať verzie softvéru, distribúcie a viacadresárové softvérové balíčky. Dobrý tutorial k GNU Autotools, informácie a filozofiu slobodného softvéru môže čitateľ nájsť na (Gkioulekas, 1999).

[2]GNU GPL licencia: http://www.gnu.org/copyleft/gpl.html
[3]GNU systém pre vývoj programov: http://sourceware.org/autobook/

5.4 Naše problémy spojené s vývojom softvéru

Na tomto mieste by sme sa radi zmienili o skúsenostiach a niektorých problémoch, ktoré sme (nevy)riešili počas práce na klastri.

Vývoj na vzdialenom počítači

Pri špecifikácii a návrhu diela je potrebné mať na pamäti nasledovné. Vývoj na vzdialenom počítači nie je taký rýchly a pohodlný v porovnaní s prácou na lokálnom počítači. K dispozícii je väčšinou iba Linuxová konzola, ktorá niekedy trpí pomalšou odozvou. Počas našej práce na klastri v ILC sme *pravdepodobne* narazili na problém zlého nastavenia *prepínačov* alebo *driverov* sieťových zariadení. "Pravdepodobne" preto, lebo sa nám problém nepodarilo presne definovať a odstrániť. Odozva konzoly bola vyše päť sekúnd v čase, keď sa z hlavného uzla sťahovali veľké dáta (výsledky experimentov). V blízkej budúcnosti sa plánuje prepojenie ILC a FMFI UK optickou linkou a tento problém by už nemal viac existovať.

Tvorba zdrojového kódu

Na klastri sme väčšinu času používali editor Vim[4]. ILC má zakúpený a nainštalovaný aj vývojové štúdio a vizuálne nástroje CodeWarrior[5] umožňujúce krokovanie, analýzu a vývoj programov a nástroje PGI Tools (prekladač, debuger a iné). Linuxový oknový systém X[6] (X Window System) a protokol SSH podporuje preposielanie X11 spojení smerom na klienta. Inými slovami, ak v konzole spustíme X11 program, spojenie bude prebiehať cez zakódovaný kanál a s skutočné pripojenie k X serveru bude vykonané na strane klienta[7]. Túto príjemnú vlastnosť sme v neskorších fázach projektu využívali za použitia prostredia CodeWarrior na lokálnom PC, nakoľko počet zdrojových súborov projektu presiahol číslo 40. Vývoj softvéru sa značne urýchlil a zjednodušil. Z rovnakých dôvodov sme triedy, ktoré boli nezávislé od klastra a knižnice na ňom nainštalované implementovali na lokálnom PC. Aj preto sme zdrojové kódy písali v súlade v ANSI C/C++ štandardom. Použili sme pri tom editory Vim a open-source vývojové prostredie KDevelop[8]. Renderovacia časť programu (`CRaycaster`) bola implementovaná lokálne, a preto je k dispozícii jej verzia pre bežné počítače vo verziách pre Linux aj MS Windows. Zdrojové kódy sú súčasťou balíčkov a je možné si ich stiahnuť na webovej stránke projektu alebo z priloženého CD-ROMu.

Pripomíname aj náročnosť krokovania (`debug`) paralelného programu na viacerých procesoroch. Hľadanie chyby sa niekedy stáva veľmi tvrdým orieškom, pretože jej výstupom je často iba typ (napr. pri preposielaní správ cez MPI). Situácia v plne asynchrónnom modely je ešte komplikovanejšia, odporúčame prejsť do blokovaného režimu a postupne vypisovať/overovať obsah správ.

V neposlednom rade by sme chceli upozorniť na celkový návrh časového plánu projektu. Poznatky softvérového inžinierstva odporúčajú určenú dĺžku vynásobiť číslom dva. Je potreb-

[4]Oficiálna stránka editoru Vim: http://www.vim.org/
[5]http://www.codewarrior.com/
[6]X Window System: http://www.xfree86.org/, http://www.x.org/
[7]Viac informácii napr. v manuálových stránkach: man ssh
[8]KDevelop Projekt: http://www.kdevelop.org/

né pozorne preskúmať a zhodnotiť špecifikáciu projektu, vedomosti a skúsenosti vývojového tímu v oblastiach ako programovanie pod OS Linux, programovanie paralelných architektúr, skúsenosti so systémami na preposielanie správ (MPI), skúsenosti so sieťami, znalosť programovania soketov a skúsenosti z počítačovej (objemovej) grafiky. Nesprávny odhad vlastných vedomosti a času mal negatívny vplyv aj na náš projekt. Celková dĺžka vývoja sa predĺžila o niekoľko mesiacov.

Problémy s implementáciami MPI

(Cavin a kol., 2003) pri svojich experimentoch s MPI uvádzajú vyššiu rýchlosť (4x rýchlejšie vykonanie funkcie `MPI_Sendrecv()`) a spoľahlivosť implementácie LAM/MPI[9] oproti MPICH. V ich práci uvádzajú aj ďalší projekt s rovnakými zisteniami. Pri štúdiu rôznych paralelných implementácii sme v (Hansen a kol., 1995) narazili aj na problém neefektívneho broadcastu pri použití PVM (`pvm_recv()`). Autori museli tento problém riešiť implementáciou vlastnej funkcie. Odporúčame preto aktualizovať (nie len) implementáciu MPI, prípadne nainštalovať viacero rôznych (napr. MPICH a LAM/MPI) naraz a porovnať rýchlosti a spoľahlivosť. Autori nenašli vysvetlenie pre tento fenomén, no efektívnosť LAM/MPI bola preukázaná aj inými autormi. Ďalšie informácie a linky možno nájsť v spomenutom článku. Na záver by sme radi upozornili na jednu implementáciu MPI – Open MPI[10]. Ide o implementáciu MPI nasledujúcej generácie. Projekt sa snaží kombinovať technológie a zdroje viacerých existujúcich projektov (FT-MPI, LA-MPI, LAM/MPI, a PACX-MPI) v snahe vytvoriť najlepšiu implementáciu. Obsahuje plnú podporu MPI-2, použitie vlákien, podporu 64-bitových architektúr. Kompletný zoznam je k dispozícii na oficiálnych stránkach projektu.

[9]LAM/MPI projekt: http://www.lam-mpi.org/
[10]Vysoko výkonná implementácia Open MPI: http://www.open-mpi.org/

Kapitola 6

Výsledky

Neskorší vývoj a testovania sprevádzali technické problémy. Nefunkčné sieťové karty a prepínače nám znemožnili testovať zapojenie Myrinet na viac ako dvoch procesoroch hlavného uzla. Preto výsledky Myrinetu neodrážajú jeho rýchlosť medzi dvoma *rôznymi* počítačmi klastra. Okrem toho boli počítače prepojené iba 100 Mbitovým Ethernetom. V najbližšej dobe plánuje MLC prepojenie uzlov gigabitovým Ethernetom a blízkej budúcnosti aj modernizáciu siete Myrinet.

Zdrojové kódy sme preložili prekladačmi `mpiCC` (GNU `g++`) a PGI `pgCC`. Prekladač `pgCC` sme používali s implementáciou MPICH 1.2.5, prekladač `g++` s implementáciami MPICH 1.2.7 a MPICH-GM. Naše testy ukázali 5-10 % vyšší výkon verzie 1.2.7 oproti 1.2.5. Obe verzie boli napojené na sieť Ethernet, implementácia MPICH-GM bola napojená na sieť Myrinet.

Dobrou optimalizáciou prekladu zdrojového kódu sme dosiahli výrazne vyšší výkon. Najmä v prípade lokálneho renderovania na procesore závisí paralelná kompozícia vo veľkej miere od rýchlosti siete. Experimentovali sme s viacerými atribútmi kompilátorov s rôznymi výsledkami. PGI kompilátor sme testovali kombináciami nasledujúcich atribútov: `-Mcache_align`, `-tp p7`, `-fastsse`, `-O3`, `-O4`, `-fastsse`, `-Minline`, `-Mvect=sse`, `-s`, `-fast`. Aj napriek tomu, že PGI prekladač obsahuje viaceré funkcie a vylepšenia navyše oproti `g++`, výsledky boli prekvapujúce v zlom slova zmysle. Nielenže optimalizácia prekladača výkon zvýšila skutočne minimálne, v niektorých prekvapujúcich situáciách (napr. pri použití samotného parametra `-O3`, resp. `-O4`) boli časy ešte horšie. Vedci a vývojári poukazujú na lepšie výsledky `pgCC` oproti `g++` (Ballabio a kol, 2005). V našom prípade však bola situácia opačná. Prekladač PGI sa dodáva iba v binárnom tvare, nakoľko ide o komerčný produkt. Domnievame sa, že príčinou výsledného pomalého kódu je nevhodná konfigurácia C/C++ prekladača, pravdepodobne v `debug` režime. Do budúcnosti odporúčame preinštalovať/aktualizovať softvér s optimalizáciami vzhľadom na rýchlosť. "Najlepšie" výsledky sme dosiahli pri použití optimalizácii: `-O3 -s -fast`.

Z výsledkov našich testov rýchlosti kódu prekladača GNU `g++` odporúčame na klastri použiť nasledujúce optimalizácie:
`-march=pentium4 -O3 -pipe -fomit-frame-pointer -fmerge-all-constants -ffast-math -mmmx -msse -msse2 -fssa`.

Vykonali sme experimenty lokálnej renderovacej časti a paralelnej binary-swap kompozície na klastri v MLC (technické detaily, viď. kapitola 5.2) za použitia hore uvedených prekladačov

a rôznych kombinácii optimalizácii prekladu zdrojového kódu. Taktiež sme testovali konkrétnu konfiguráciu pre rozlíšeniach obrazovky 64x64, 128x128, 256x256, 512x512, 1024x1024. Testovali sme reálne údaje z konfokálneho mikroskopu, ako aj syntetické dáta.

Lokálne renderovanie

Táto časť nevyžaduje žiadnu medziprocesorovú komunikáciu. Prahovanie (segmentácia) bolo nastavené na hodnotu 500 (12bit dáta). Použili sme najlepšie optimalizácie prekladačov uvedených na začiatku kapitoly. Testy prebehli na hlavnom uzle klastra.

Časy vo výsledkoch sme získali pomocou funkcie `gettimeofday()`. Oproti `MPI_WTIME()` nenesie so sebou ďalšie *MPI* náklady na čas a prácu. V tabuľkách 6.1 a 6.2 sú výsledky rende-

Prekladač	64x64	128x128	256x256	512x512	1024x1024
$A1$	41.8154	10.8711	2.80671	0.724252	0.183216
$B1$	31.9456	8.30381	2.10061	0.545197	0.134749
$C1$	27.6747	7.52639	1.9530	0.490780	0.123512
$A2$	42.1496	11.0870	2.87302	0.729961	0.183840
$B2$	32.2031	8.35701	2.14038	0.545711	0.135932
$C2$	29.0393	7.55738	1.97034	0.491747	0.123928

Tabuľka 6.1: CLSM dáta 1024x1024x32. Porovnanie prekladačov a optimalizácii. Hodnoty reprezentujú počet snímok za sekundu (Frame Per Second – FPS).

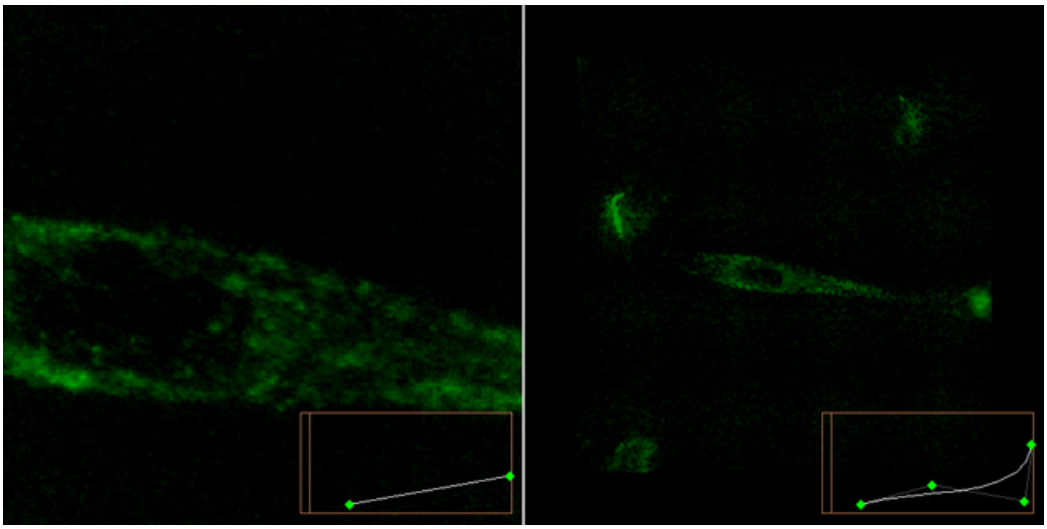
rovania jednokanálových dát rozmerov 1024x1024x32 pri nastavení zoom: 0, resp. zoom: -5 lokálnym procesorom. $A1$, $C1$ označujú použitie *optimalizovaného*, resp. *neoptimalizovaného* prekladu `g++` kompilátorom. Výsledky prekladača `pgCC` boli v optimalizovanej a neoptimalizovanej verzii približne *rovnaké*. Preto uvádzame iba prvú možnosť pod označením $B1$. Znak 1 označuje priemerný počet snímok za sekundu, 2 maximálny. Výsledky ukazujú dôležitosť

Prekladač	64x64	128x128	256x256	512x512	1024x1024
$A1$	90.0662	23.7287	6.05493	1.358620	0.352404
$B1$	58.9926	15.2396	3.94879	0.989825	0.267798
$A2$	90.5303	23.8697	6.24856	1.360880	0.353135
$B2$	59.3650	15.5788	3.97665	0.995876	0.268692

Tabuľka 6.2: CLSM dáta 1024x1024x32, kamera zoom: -5, porovnanie prekladačov a optimalizácii. Priemerné a maximálne hodnoty: počet snímok/sekunda.

správneho nastavenia parametrov prekladu zdrojového kódu. Prekladač `g++` dosť výrazne prekonáva `pgCC` vo všetkých testovaniach. Kód neoptimalizovaného GNU prekladača jednoznačne zaostáva za optimalizovaným. Dáta sú vizualizované na obrázku 6.1. Pre porovnanie sme kameru vzdialili natoľko, aby sme dosiahli celkový pohľad na dáta.

Testovali sme výkon *optimalizovaného* procesu renderovania oproti *prvotnej* verzii a výsledky porovnávame s výkonom pri optimalizovanom *tieňovaní*, viď. tabuľka 6.3. Testy sme uskutočnili s prekladačom GNU `g++`. Použité parametre optimalizácii boli rovnaké ako v úvo-



Obrázok 6.1: CLSM dáta 1024x1024x32, prahovanie: 500.

de kapitoly, veľkosť dát bola 1024x1024x16. V tabuľke sú uvedené priemerné hodnoty počet obrázkov za sekundu.

Typ	64x64	128x128	256x256	512x512	1024x1024
Neopt.	40.9948	9.67256	2.68359	0.681401	0.170546
Optimal.	57.9721	13.5258	3.85650	0.982315	0.246799
Phong	23.8970	5.83125	1.52084	0.381794	0.095832

Tabuľka 6.3: Porovnanie výkonu optimalizovaného, neoptimalizovaného a tieňovaného renderovania na lokálnom procesore (počet snímok za sekundu).

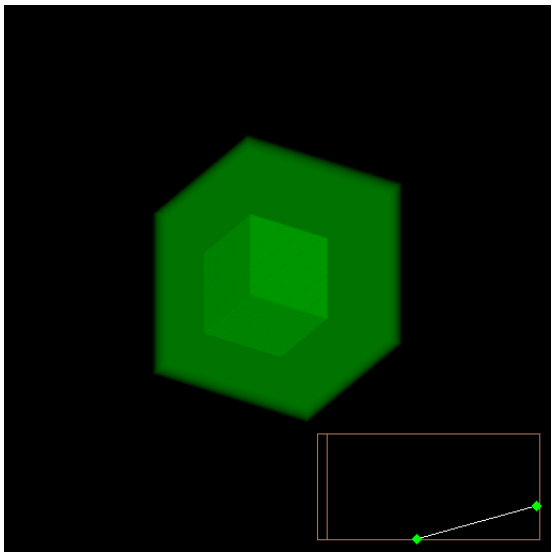
Rýchlosť lokálneho renderera sme testovali aj na syntetických dátach zobrazených (obrázok 6.2). Výsledky sú uvedené v tabuľke 6.4.

Prekladač	64x64	128x128	256x256	512x512	1024x1024
$A1$	255.068	70.8249	18.3722	4.69897	1.051860
$B1$	185.642	51.4800	13.3210	3.40340	0.828289
$A2$	258.534	71.4643	18.5657	4.73048	1.058850
$B2$	192.086	51.8675	13.4216	3.43038	0.832511

Tabuľka 6.4: Syntetické dáta a porovnanie rýchlosti prekladačov.

Načítanie dát a inicializácia pomocných štruktúr

Tabuľka 6.5 detailne zobrazuje časy inicializácie lokálneho renderera. Testy boli vykonané na hlavnom uzle klastra. Použité dáta: 1024x1024x32, 16bit. Označenie "rc init" inicializuje



Obrázok 6.2: Syntetické dáta "kocka v kocke". Veľkosť 256x256x256, 12 bit farebná hĺbka.

triedu `CRaycaster`, editor prechodovej funkcie a pamäť pre textúru, "data load" je čas načítania CLSM dát a vykonanie segmentácie, "dvolume init" reprezentuje čas výpočtu vzdialenostnej transformácie na vstupných dátach, "brick init" udáva čas transformácie adresovania dát do tehličkovej podoby (bricking) a "homog init" predstavuje inicializáciu pomocnej dátovej štruktúry homogénnych buniek. Inicializácie a overenia trvajúce menej ako $0.00001s$ sme nebrali do úvahy. A označuje optimalizovaný preklad kompilátorom `g++`, B označuje neoptimalizovaný a C reprezentuje `pgCC`.

funkcia	A	C	B
rc init	0.00138	0.00177	0.00139
data load	6.79445	6.46718	6.06448
dvolume init	21.3043	46.5929	34.9688
brick init	3.20087	4.6557	4.43892
homog init	2.20332	6.11953	4.56155
celkovo	33.5043	63.8372	50.0352

Tabuľka 6.5: Načítanie CLSM dát a inicializácia procesorov (v sekundách).

V budúcnosti MLC plánuje vytvoriť na klastri výkonné diskové pole zdieľané všetkými procesormi. Časti objemových dát by potom nebolo nutné posielať sieťou. Ak vezmeme do úvahy rozdelenie dát medzi šestnásť procesorov, celková doba inicializácie paralelného renderera by sa znížila na veľmi nízku dobu.

Binary-swap kompozícia

Algoritmus binárna výmena pozostáva z dvoch častí: výmeny *čiastkových* obrázkov medzi procesormi a pozbieranie výsledných častí *finálneho* obrázka hlavným uzlom. Algoritmus sme testovali na dvoch implementáciách MPI – MPICH 1.2.7 (pripojená na Ethernet) a MPICH-GM (pripojená na Myrinet).

Technické problémy v MLC nás pri vývoji a testovaní limitovali. Pre porovnanie siete Myrinet 2000 s Ethernetom sme mali k dispozícii iba 2 procesory hlavného uzla. Uzly boli Ethernetom prepojené linkou s rýchlosťou 100Mbit/s. Tabuľka 6.6 porovnáva rýchlosti algoritmu binárna výmena na *dvoch* procesoroch za použitia implementácií MPICH 1.2.5, MPICH 1.2.7 a MPICH-GM. Celkový čas sa skladá z časov preposielania si obrázkov medzi procesormi, ich miešanie a pozbieranie obrázkov hlavným uzlom. Pixel sa skladá z hodnôt RGBA, ktoré sú uložené v šestnástich bajtoch (4×4 bajty). Údaje označujú počet snímok za sekundu. Výsledky testov ukázali, že dva procesy bežia rýchlejšie na jednom počítači ako na dvoch rôznych počítačoch.

MPI	64x64	128x128	256x256	512x512	1024x1024
MPICH-GM	3497.85	390.737	78.5041	19.4650	4.89496
MPICH 1.2.7	1634.91	242.975	52.8245	13.2328	3.32691
MPICH 1.2.5	1136.15	235.006	52.6802	13.1345	3.30792

Tabuľka 6.6: Ethernet vs Myrinet (2 procesory). Testovanie algoritmu binárna výmena, priemerný počet snímok za sekundu.

MPICH skript `mpirun` obsahuje parameter `-machine`, ktorý umožňuje špecifikovať typ procesorov. V našom prípade sme testovali rýchlosť po spustení paralelného programu skriptom `mpirun -machine p4` pre procesory Intel Xeon. Avšak podobne ako v prípade optimalizácii PGI prekladača `pgCC` sme nezaznamenali žiadne zlepšenia vo výkone. Na dvoch procesoroch

# procesorov	64x64	128x128	256x256	512x512	1024x1024
2	116.075	28.0363	6.96695	1.74405	0.434638
4	83.9540	19.3743	4.75140	1.18333	0.294344
8	75.7397	16.5332	4.08904	1.01312	0.253506
16	49.6492	8.25763	2.52935	0.62031	0.153627

Tabuľka 6.7: Algoritmus binárna výmena (hodnoty: priemerný počet snímok/sekunda) pri použití implementácie MPICH 1.2.7 (Ethernet).

je Myrinet oproti Ethernetu lepší o približne 50 %.

Tabuľka 6.7 zobrazuje počet snímok/sekundu algoritmu binary-swap na viacerých procesoroch. Testovanie prebehlo za použitia MPI implementácie MPICH 1.2.7. Výsledky urýchlenej implementácie algoritmu binárna výmena (hodnotu RGBA reprezentujú 4×8 bitov = 4 bajty) môže čitateľ nájsť v tabuľke 6.8. Algoritmus je približne 3.5-krát rýchlejší.

# procesorov	64x64	128x128	256x256	512x512	1024x1024
2	422.551	105.609	24.5468	6.24830	1.560550
4	285.592	83.6211	17.3648	4.18995	1.075930
8	236.069	73.5214	15.2594	3.62562	0.920516
16	158.633	49.7796	8.05665	2.34520	0.577426

Tabuľka 6.8: Zrýchlený algoritmu binárna výmena (hodnoty: priemerný počet snímok/sekunda) pri použití implementácie MPICH 1.2.7.

Paralelné renderovanie bez fázy kompozície

So zvyšujúcim sa počtom procesorov zapojených do výpočtu sa zmenšuje objem dát pripadajúci na jeden proces renderovania. V tabuľke 6.9 môžeme vidieť rýchlosť (počet snímok za sekundu) samostatnej renderovacej časti pracujúcej na CLSM dátach rozmerov 1024x1024x16. Kompozičnú časť (binary swap) sme úplne vynechali.

# procesorov	64x64	128x128	256x256	512x512	1024x1024
2	111.85	28.513	7.4229	1.8682	0.4713
4	169.39	55.428	14.120	3.5648	0.8941
8	367.30	96.267	24.147	6.1130	1.5381
16	2571.5	670.23	168.81	42.218	10.569

Tabuľka 6.9: Paralelné renderovanie bez fázy kompozície (hodnoty: priemerný počet snímok/sekunda). Použitý prekladač g++ a MPICH 1.2.7.

Dočasné riešenie pomocou Ethernetu (100 Mbit/s) predstavuje *úzke miesto* paralelného programu. Porovnaním tabuliek 6.9 a 6.7, resp. 6.8 vidíme dôležitú skutočnosť. Ani urýchlena verzia kompozície rýchlostne nestíha za fázou lokálneho renderovania. Algoritmus binárnej výmeny je možné ďalej optimalizovať a tak získať menšie nároky na šírku pásma pri volaní `MPI_Sendrecv()` (kapitola 7.1). Najlepšie výsledky však bude možno získať až po obnove Myrinetu v MLC. Kompozícia by potom nemala limitovať interaktivitu celého systému.

Celková rýchlosť paralelného renderera

V tabuľke 6.10 sú hodnoty (počet snímok za sekundu) celkového generovania objemových dát na *dvoch* procesoroch cez Myrinet a 2 – 16 procesoroch prepojených cez Ethernet. Proces zahŕňa načítanie dát hlavným uzlom, distribúciu dát medzi všetky zostávajúce uzly zapojené do výpočtu, inicializáciu (renderer, k-D strom, plán kompozície, pomocné dátové štruktúry), vygenerovanie čiastkových obrázkov, ich kompozíciu medzi procesormi a pozbieranie výsledkov hlavným uzlom. Testované dáta boli rozmerov 1024x1024x16.

Rýchlosť v prípade Myrinetu brzdí rýchlosť *lokálne renderovanie*. Zapojenie na viac procesoroch sme otestovať nemohli. V prípade Ethernetu brdzí celkovú rýchlosť *paralelná kompozícia* (pomalá sieť, neoptimalizovaný algoritmus). Paralelizmus spojený s optimalizáciami algoritmu ray-casting priniesol urýchlenie cca $3-4$ krát oproti počiatočnej verzii na jednom procesore. Čitateľ bude mať lepšiu predstavu o urýchleniach po zhliadnutí tabuľky 6.11. Označme znakom

# procesorov	64x64	128x128	256x256	512x512	1024x1024
Myrinet 2	104.435	25.8817	6.36063	1.63172	0.40404
Ethernet 2	95.9161	22.7871	6.15001	1.53497	0.37753
4	74.4146	15.1626	3.40447	0.84801	0.20691
8	61.1132	13.1370	3.45547	0.85779	0.20153
16	53.4461	9.78893	2.35510	0.57218	0.14179

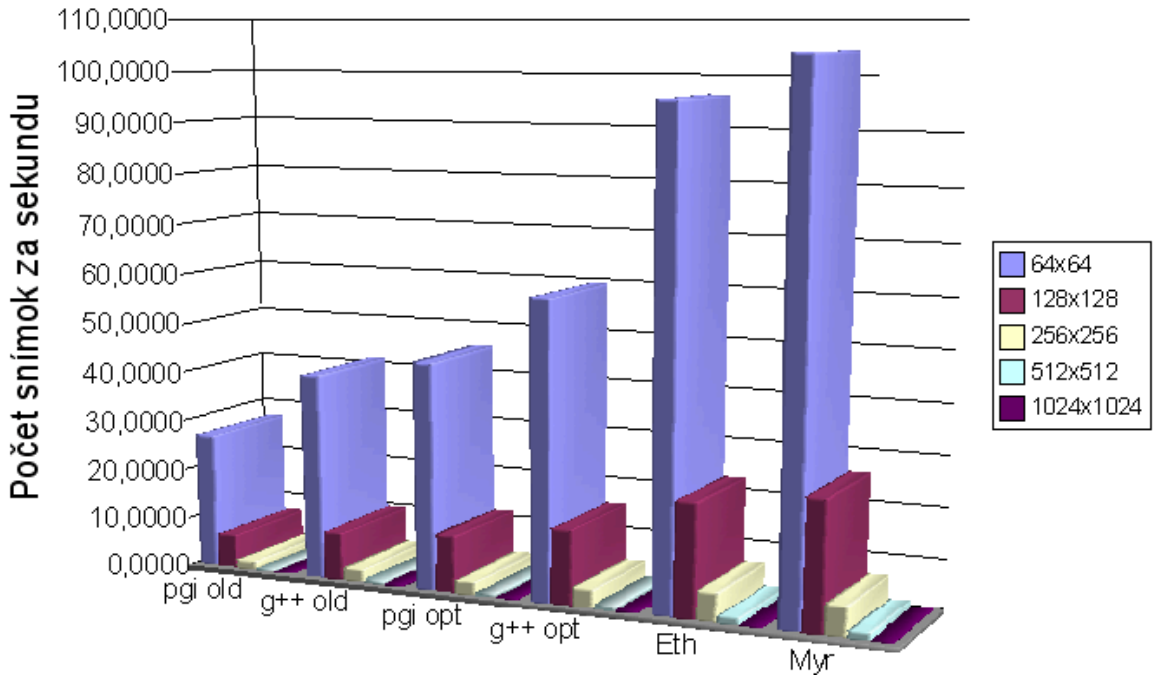
Tabuľka 6.10: Celková rýchlosť (priemerný počet snímok/sekunda) paralelnej objemovej vizualizácie. Porovnanie implementácii MPI (MPICH-GM, MPICH 1.2.7, g++).

A pôvodnú implementáciu algoritmu ray-casting na jednom procesore, znakom *B* implementáciu so všetkými urýchleniami a znakom *C* paralelnú implementáciu na dvoch procesoroch. Znaky 1 a 2 predstavujú použitie prekladača `pgCC` a `g++`. Hodnoty predstavujú počet snímok za sekundu.

Tabuľka odráža vývoj paralelného programu, od začiatkov a implementovanej neoptimalizovanej verzie renderera. Urýchľovacie techniky nás posunuli o krok dopredu a paralelizmus pridal ku zlepšeniu najviac. Každú z troch prezentovaných fáz sme kompilovali pod prekladačom `pgCC` a `g++`. Výsledky sú zoradené od prvých (najslabších) po konečné (najrýchlejšie). Graficky sú výsledky vizualizované na obrázku 6.3. Úzke miesto paralelného programu je práve časť kompozície (algoritmus binárna výmena). Pre dosiahnutie interaktívnych časov je vysoko-rýchlostná sieť v prípade paralelnej objemovej vizualizácie nutnosťou.

# MPI	64x64	128x128	256x256	512x512	1024x1024
A1	27.0157	6.85875	1.72242	0.436279	0.109282
A2	40.9948	9.67256	2.68359	0.681401	0.170546
B1	45.1151	11.5151	2.84394	0.740287	0.184866
B2	59.2794	15.2375	3.93188	0.991588	0.244981
Ethernet C1	95.9161	22.7871	6.15001	1.53497	0.37753
Myrinet C2	104.435	25.8817	6.36063	1.63172	0.404048

Tabuľka 6.11: Porovnanie dosiahnutých výsledkov (FPS) na CLSM dátach 1024x1024x16.



Obrázok 6.3: Postupný vývoj rýchlosti programu a porovnanie prekladačov.

Kapitola 7

Záver

7.1 Prínos práce a záver

Preštudovali sme rôzne metódy pre objemovú vizualizáciu a paralelné renderovanie. Vybrali sme najvhodnejšie riešenia vzhľadom na podmienky (CLSM dáta, klaster). Práca ponúka prehľad problematiky vizualizácie CLSM dát na paralelných počítačoch a problémov s ňou spojených. Rozoberá a porovnáva jednotlivé techniky a algoritmy. Pre vybrané metódy detailne popisujeme dôvody ich zvolenia ako primárne. Navrhli sme triedy a paralelný program typu klient-server pre vizualizáciu CLSM dát. Návrh sa snaží byť maximálne všeobecný, nezávislý od algoritmov a klastra pre ich prípadné vylepšenia a modifikácie. Zdrojové kódy sú v čo najväčšej miere portabilné. Výsledkom nášho úsilia je prototyp paralelného objemového renderera pracujúci na linuxovskom klastri. Serverová časť (na klastri) je zložená z dvoch nezávislých častí: lokálnej renderovacej a paralelnej kompozičnej. S cieľom dosiahnuť čo najvyššiu rýchlosť sme implementovali viacero optimalizácií renderovacieho algoritmu. Vykonali sme rozsiahle testy lokálneho renderera a paralelnej kompozicie až na šesnástich procesoroch. Otestovali a porovnali sme tri implementácie MPI a dva prekladače zdrojových kódov. Program renderuje konfokálne dáta interaktívne (niekoľko snímok za sekundu). Vďaka snahe o čo najväčšiu nezávislosť je k dispozícii aj renderovací program objemových CLSM dát pre bežné počítače (lokálna renderovacia časť) . Aby sme zaručili maximálne rozšírenie projektu medzi verejnosť, vydávame softvér pod licenciou GNU GPL. Uvádzame možné rozšírenia a modifikácie implementovaných algoritmov a zdrojových kódov za účelom rozšírenia funkcionality a zvýšenia výkonu. A to nie len pre súčasnú konfiguráciu, ale aj všeobecne, pre iné architektúry a hardvér. V práci sme sa zaoberali možnosťou vzdialenej vizualizácie ako riešením pre inštitúcie bez potrebného hardvérového vybavenia. Uvádzame tiež odporúčania a skúsenosti pri vývoji paralelných programov na distribuovaných architektúrach.

7.2 Možné rozpracovania a modifikácie práce

Implementovali sme viacero algoritmov a tried, ktoré spolu tvoria paralelný objemový rendererovací program. Oblasť paralelných výpočtov, ako aj objemovej vizualizácie, je dobre preštudovaná a ponúka mnoho urýchlení. V tejto časti spomíname niektoré možné pokračovania práce, ktoré sa tykajú urýchlenia systému, ako aj zvýšenia funkcionality. Hardvér klastra ponúka preskúmanie inštrukčných sád (SSE, SSE2) typických pre Intel Xeon. Experimenty by mohli prebehnúť aj s použitím vkladaného asembléru. Ďalšou a dôležitou optimalizáciou

by mohla byť implementácia optimalizovanej verzie algoritmu binárna výmena, ktorý prezentujú vo svojich prácach (Takeuchi a kol., 2003), (Yang a kol., 2001). Pre dosiahnutie lepších výsledkov na úrovni lokálneho softvérového renderera by bolo dobré do programu zakomponovať ďalšie optimalizácie algoritmu ray-casting, spomenuté napríklad v (Bruckner, 2004). Pre architektúry s podporou vlákien, viacprocesorové počítače, resp. najnovšie viacjadrové procesory by bolo možné paralelizovať niektoré časti programu. Napríklad pomocou rozhrania OpenMP pridaním jediného príkazu kompilácie do zdrojového kódu: `#pragma omp parallel for`. Pre vrhanie lúčov do scény alebo paralelné miešanie pixelov v algoritme binárna výmena by táto optimalizácia určite priniesla svoje ovocie. Zvýšenie kvality vizualizácie CLSM dát by prinieslo ich predspracovanie príslušným filtrom v snahe znížiť napr. šum (medián) (Razdan a kol., 2001). Neimplementovali sme podporu viackanálových CLSM dát. Dôležitým rozšírením funkcionality renderera by bola podpora viackanálových CLSM dát a spektrálnych dát. Pre väčšie pohodlie používateľa softvéru by bolo potrebné implementovať univerzálnejšie načítavanie vstupných dát a ich príjem od klienta (aplikácie). V časti 3.1.1 spomíname zaujímavý hybridný prístup paralelizmu dát (Garcia, 2002). Autori práce rozdelili procesory do skupín a každej skupine (nie procesoru) priradili časť objemových dát. V rámci skupiny sa robí obrazový paralelizmus, zatiaľ čo na úrovni skupín objektový. Tento prístup maximálne využíva dostupnú pamäť a znižuje komunikáciu vo fáze kompozície. Miešame obrázky, ktorých počet je rovný počtu skupín. Aby rozdelili prácu rovnomernejšie a tým dosiahli vyšší výkon, v rámci skupiny prekladajú pixely medzi všetky procesory. Klaster v ILC disponuje dostatočnou pamäťou pre implementáciu spomenutého prístupu. Posledným možným rozpracovaním, ktoré uvedieme, je progresívne zlepšovanie kvality. Pri modifikáciách pozície kamery, tranfér funkcie a iných nastavení by klaster generoval obrázky v nízkom rozlíšení pri vysokom pomere počet obrázkov/sekunda. Ak by klient neposlal serveru žiadne nové *zmeny*, server by automaticky generoval obrázky postupne vo vyššom rozlíšení a posielal ich klientovi (ktorý by ich samozrejme očakával). Až do maximálneho rozlíšenia. Neimplementovali sme spojenie sieťou protokolom UDP/IP, táto časť bude implementovaná v pokračovaní práce.

Referencie

ADOBE Developers Association. 1992. TIFF Revision 6.0 Final. Dostupné na internete: <http://www.adobe.com/Support/TechNotes.html>

AMDAHL, G. 1967. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, AFIPS Conference Proceedings, (30), pp. 483-485.

BALLABIO, G., BOSCHI B., CALONACI, C., CAVAZZONI, C., EMERSON, A., GHELLER, C., GORI, R., TARSI, A. 2005. High Performance Systems User Guide. Supercomputing Group, High Performance Systems Department. CINECA.

BENTUM, M.J. 1996. Interactive Visualization of Volume Data. Ph.D. Thesis, University of Twente, Enschwede, The Netherlands.

BEOWULF.org. 2004. Beowulf.org: The Beowulf Cluster Site. Dostupné na internete: <http://www.beowulf.org>

BORGEFORS, G. 1986. Distance Transformations in digital images. Computer Vision, Graphics, and Image Processing, 34(3):344-371.

BRUCKNER, S. 2004. Efficient Volume Visualization of Large Medical Datasets. Master's Thesis, Vienna University of Technology. Dostupné na internete: <http://www.cg.tuwien.ac.at/research/publications/2004/bruckne r-2004-EVV>

CAMAHORT, E., CHAKRAVARTY, I. 1993. Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures. IEEE, Parallel Rendering Symposium, pp:89-96.

CAVIN, X., HARTNER, M., HANSEN, C. 2003. Implementation and Evaluation of Binary Swap Volume Rendering on a Commodity–Based Visualization Cluster. Technical Paper. SCI Institute, University of Utah.

CLINE, H.E., LUDKE, S., LORENSEN, W.E., TEETER, B.C. 1988. A 3D Medical Imaging Research Workstation. Volume Visualization Algorithms and Architectures, ACM SIGGRAPH '90 Course Notes, Course Number 11, ACM Press.

CROCKETT, W.T. 1995. ICASE Parallel Rendering. Institute for Computer Applications in Science and Engineering, NASA Langley Research Center.

ČERVEŇANSKÝ, M. 2004. Využitie komerčných grafických akcelerátorov pre vizualizáciu a spracovanie objemových dat. Diplomová práca. Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, Slovensko.

DICKINSON, M.E., BEARMAN, G., TILLE, S., LANSFORD, R., FRASER, S.E. 2001. Multispectral imaging na d linear unmixing add a whole new dimension to laser scanning fluorescence microscopy. Biotechniques, 31(3):1272–1278.

DOSTÁL, R. 2003. Sokety a C/C++. Dostupné na internete:
<http://www.builder.cz/serial147.html>, <http://www.root.cz/serialy/sokety-a-cc/>

ELVINS, T.T. 1992. A Survey of Algorithms for Volume Visualization. Computer Graphics, 22(4):194-201.

ENGEL, K., KLAUS, M., ERTL, T. 2001. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pp.9-16, Los Angeles, California, United States.

FLYNN, M.J. 1966. Very High Speed Computing Systems. IEEE, 54:1901-1909.

FUCHS, H., POULTON, J., EYLES, J., GREER, T., GOLDFEATHER, J., ELLSWORTH, D., MOLNAR, S., TURK, G., TEEBS, B., ISRAEL, L. 1989. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. Computer Graphics, 23(3):79-88.

GARCIA, A., SHEN, H. 2002. An Interleaved Parallel Volume Renderer With PC-clusters. Fourth Eurographics Workshop on Parallel Graphics and Visualization, pp:51-60.

GKIOULEKAS, E. 1999. Developing software with GNU - An introduction to the GNU development tools. University of Washington. Dostupné na internete:
<http://www.amath.washington.edu/ lf/tutorials/autoconf/toolsmanual.html>

GOURAUD, H. 1971. Continous Shading of Curved Surfaces. IEEE Transaction of Computers, 20(6):623-629.

GRIMM, S., BRUCKNER, S., KANITSAR, A., GROELLER, M.E. 2004. A Refined Data Addressing and Processing Scheme to Accelerate Volume Raycasting. Computers Graphics, 28(5):719-729.

HANSEN, C. D., KROGH, M., PAINTER, J., de VERDIERE, G.C., TROUTMAN, R. 1995. Binary-swap volumetric rendering on the T3D. Cray Users Group Conference, (Denver, Co.). Lawrence Livermore National Lab, Livermore.

HERMAN, G.T., LIU, H.K. 1979. Three-dimensional display of Human Organs from Computed Tomograms. Computer Graphics and Image Processing. 9(1):1-21.

HERMAN, G.T., ZHENG, J., BUCHOLTZ, C.A. 1992. Shape-based interpolation. IEEE Computer Graphics and Applications, 12(3):69-79.

HSU, W. 1993. Segmented Ray Casting for data Parallel Volume Rendering. IEEE, Parallel Rendering Symposium, ACM Press.

KEPPEL, E. 1975. Approximating Complex Surfaces by Triangulation of Contour Lines. IBM Journal of Research and Development, 19(1):2-11.

LACROUTE, P., LEVOY, M. 1994. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. Computer Graphics, 28(4):451-458.

LEVOY, M. 1988. Display of Surfaces from Volume Data. IEEE Comuter graphics and Applications, 8(3):29-37.

LEVOY, M. 1990. Efficient Ray Tracing of Volume Data. ACM Transactions on Graphics, 9(3):245-261.

LORENSEN, W.E., CLINE, H.E. 1987. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. Computer Graphics, 21(4): 163-169.

MA, K.-L., PAINTER, J.S., HANSEN, C.D., KROGH, M.F. 1994. Parallel Volume Rendering Using Binary Swap Image Composition. IEEE Computer Graphics and Applications, 14(4):59-68.

MC Services. 2006. Confocal Laser Scanning Microscopy. Dostupné na internete: <http://microscopy.info/>

MOLNAR, S., COX, M., ELLSWORTH, D., FUCHS, H. 1994. A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications, 14(4):23-32.

MYRICOM, Inc. 2006. Myricom Home Page. Dostupné na internete: <http://www.myri.com/>

PACS Training Group. 2001. Introduction to MPI. NCSA Access. University of Illinois.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, CH., SHIRLEY, P. 1999. Interactive Ray Tracing for Volume Visualization.IEEE Transactions on Visualization and Computer Graphics, 5(3):238-250.

PAWLEY, J.B. 1995. Handbook of biological confocal microscopy. Plenum Press, New York.

PFISTER, H., KAUFMAN, A. 1996. Cube-4: A Scalable Architecture for Real-Time Volume Rendering. ACM/IEEE Symposium on Volume Visualization, pp:47-54.

PHONG, B.T. 1975. Illumination for Computer Generated Pictures. Communications of the ACM, 18(6):311-317.

PORTER, T., DUFF, T. 1984. Compositing digital images. Computer Graphics (SIGGRAPH '84 Proceedings), 18(3):253-259.

RAZDAN, A., PATEL, K., FARIN, G.E., CAPCO, D.G. 2001. Volume Visualization of Multicolor Laser Confocal Microscope data. Computers and Graphics, 25(3):371-382.

SABELLA, P. 1988. A rendering Algorithm for Visualising 3D Scalar Fields. Computer Graphics, 22(4):51-58.

SAKAS, G., VICKER, G., PLATH, J. October 1996. Case Study: Visualization of Laser Confocal Microscopy Datasets. Proceedings IEEE Visualization '96, pp. 375-380, San Francisco.

SCHLICK, CH. 1994. A fast alternative to Phong's specular model. Graphics Gems IV, Academic Press, Boston, pp:385-387.

SHIRLEY, P., TUCKMAN, A. 1990. A polygonal Approximation to Direct Scalar Volume Rendering. Computer Graphics, 24(5): 63-70.

SPERAY, D., KENNON, S. 1990. Volume Probes: Interactive Data Exploration on Arbitrary Grids. Computer Graphics, 24(5):1-12.

ŠRÁMEK, M. 1998. Visualization of Volumetric Data by Ray Tracing. Dissertation, Technischen Universität Wien, Oesterreich.

STOMPEL, A., MA, K.-L., LUM, E.B. 2003. SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering. IEEE Symposium on Parallel and Large-Data Visualization and Graphics (2003), pp:33-40.

TAKEUSCHI, A., INO, F., HAGIHARA, K. 2003. An improvement on binary-swap compositing for sort-last parallel rendering. SAC '03: Proceedings of the 2003 ACM symposium on Applied computing. ACM Press, pp:996–1002.

THE MESSAGE PASSING INTERFACE (MPI) Forum Home Page. 2006. Dostupné na internete: <http://www.mpi-forum.org/>

THE MESSAGE PASSING INTERFACE (MPI) Standard. 2006. Dostupné na internete: <http://www-unix.mcs.anl.gov/mpi/>

UPSON, C. (ed.), 1989. The V-Buffer: Visible Volume Rendering. Computer Graphics, 22(4):59-64.

van KEMPEN, G.M.P. 1998. Image Restoration in Fluorescence Microscopy. Delf Iniversity Press, ISBN 90-407-1792-3.

WESTOVER, L. 1990. Footprint Evaluation for Volume Rendering. Computer Graphics, 24(4):367-376.

YANG, D.-L., YU, J.-C., CHUNG, Y.-C. 2001. Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. The Journal of Supercomputing 18(2) (February), pp:201–220.

ZEISS Company. LSM 5xx File Formats, Description - Release 2.0. Dostupné na internete: <http://ibb.gsf.de/homepage/karsten.rodenacker/IDL/Lsmfile.doc>

ZUIDERVELD, K. J. March 1995. Visualization of Multimodality Medical Volume Data using Object-Oriented Methods. Ph.D. thesis, University of Utrecht.

ŽÁRA, J., BENEŠ. B., FELKEL, P. 1998. Moderní Pocítacová Grafika. Praha, Computer Press, ISBN 80-7226-049-9

Zoznam príloh

CD-ROM obsahujúci:

- Zdrojové kódy všetkých tried použitích v implementácií, lokálneho renderera vo verziách pre GNU/Linux a MS Windows, paralelného renderera pre linuxový PC klaster
- Preloženú verziu našich implementácií pre GNU-Linux, resp. MS Windows
- Vstupné CLSM dáta
- Elektronickú verziu textu diplomovej práce