Platformovo nezávislý engine na renderovanie objemových dát

Michal Hučko

2008

Platformovo nezávislý engine na renderovanie objemových dát

Diplomová práca

Michal Hučko

Univerzita Komenského
Fakulta matematiky, fyziky a informatiky
Katedra aplikovanej informatiky

Školiteľ: RNDr. Michal Červeňanský

Bratislava 2008

Poďakovanie

Chcel by som sa poďakovať môjmu školiteľovi RNDr. Michalovi Červeňanskému, vedúcemu diplomového seminára doc. RNDr. Andrejovi Ferkovi, PhD., doc. Ing. Milošovi Šrámkovi, PhD. za odbornú pomoc a cenné rady. Ďalej by som chcel poďakovať Mgr. Jurajovi Starinskému a Pavlovi Ostertagovi za pomoc pri implementácií a testovaní ako aj všetkým členom seminára o objemovej grafike za ich pripomienky.

Abstrakt

HUČKO, Michal: Platformovo nezávislý engine na renderovanie objemových dát. [Diplomová práca]. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra aplikovanej informatiky. Školiteľ: RNDr. Michal Červeňanský. Stupeň odbornej kvalifikácie: Magister. Bratislava, 2008. 54 s.

Cieľom práce je vytvoriť prostredie vhodné na rýchly vývoj vizualizačných algoritmov alebo ich implementácií s dôrazom na vysokú znovupoužiteľnosť kódu a platformovú nezávislosť. Prostredie má byť založené na modeli grafu scény a má umožniť použitie s ľubovoľným užívateľským rozhraním. V práci sme sa zamerali na oddelenie priamo nesúvisiacich operácií a vytvorenie komunikačného kanálu medzi nimi. Navrhli sme originálnu architektúru využívajúcu samostatne kompilovateľné pluginy rozširujúce funkčnosť enginu. Architektúra pozostáva z troch funkčných blokov zameraných na prácu s dátami, elementami scény a vizualizačnými pluginmi. Navrhnuté prostredie je kompletne bezpečné voči súčasnému použitiu z viacerých vlákien a umožňuje vykonávať požadované operácie asynchrónne. Implementovali sme aj serverovú verziu enginu umožňujúcu vzdialenú vizualizáciu. Výsledok práce umožní znížiť náročnosť implementácie vizualizačných algoritmov, ako aj tieto zdielať v rámci vývojových skupín, či mimo nich.

Kľúčové slová objemová vizualizácia, vizualizačný engine, vzdialená vizualizácia

Abstract

HUČKO, Michal: Platform independent engine for volume visualization. [Master thesis]. Commenius University in Bratislava. Faculty of mathematics, physics and informatics; Department of applied informatics. Supervisor: RNDr. Michal Červeňanský. Bratislava, 2008. 54 pages

Aim of the thesis is to develop framework for volume visualization that would enable rapid development of visualization algorithms and its implementations. Key element is high reusability of the visualization code and engine's platform independence. Engine should be based on scene-based model. We have separated operations not directly related to each other and designed communication link between them. We have designed unique architecture extendable with plugins. Engine consists of three functional blocks for data, scene and plugin management. Developed library is thread-safe and can perform operations asynchronously. We have implemented also server version of the engine which can be used for remote visualization. Created engine will make visualization algorithms development easier and will enable to share them among research groups or broader public.

Keywords volume visualization, visualization engine, remote visualization

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam skratiek

CORBA	Common object request broker architecture
FPS	Frames per second
POSIX	Portable operating system interface
TCP/IP	Transmission control protocol / internet protocol
XML	Extensible markup language

Úvod

Objemová vizualizácia ako časť počítačovej grafiky ťaží z rapídneho vývoja grafických akcelerátorov. Tento vývoj, hoci spôsobený hlavne herným priemyslom, umožňuje vykonávať objemovú vizualizáciu aj na bežnom hardvéri. S dostupnosťou hardvéru prichádza aj potrebný softvér, pričom existuje veľké množstvo vizualizačných nástrojov. Tieto však väčšinou tvoria kompaktný celok, ktorý nie je jednoduché rozšíriť. Na druhej strane rozšíriteľné riešenia bývajú príliš zložité pre používateľa, keďže sú zväčša zamerané všeobecnejšie na vedeckú vizualizáciu. Snahou diplomovej práce je poskytnúť prostredie, ktoré je jednoducho rozšíriteľné o nové vizualizačné algoritmy, resp. ich implementácie a zároveň poskytuje rozhranie, ktoré je jednoduché pre používateľa. Táto aplikácia má mať využitie hlavne pri vývoji a testovaní nových implementácií algoritmov.

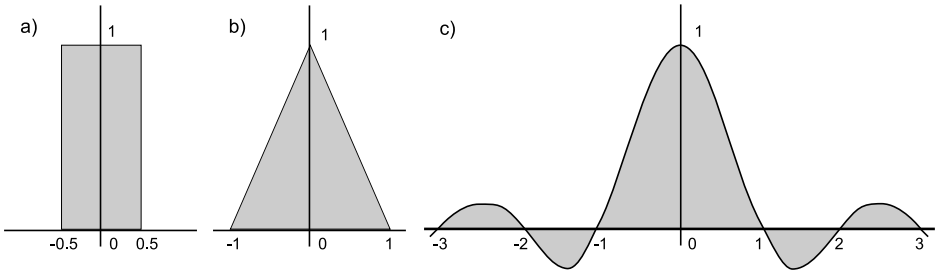
Kapitola 1

Objemová vizualizácia

Objemová vizualizácia je časť počítačovej grafiky zaoberajúca sa vizualizáciou objemových dát [24, 8] . Tieto dáta sú zväčša produkované medicínskymi diagnostickými zariadeniami, ale používajú sa aj v iných vedných odboroch. Pre rôzne účely sa používajú rôzne vizualizačné algoritmy, pričom neustále vznikajú ich nové implementácie. S narastajúcim výkonom grafických akcelerátorov a intenzitou ich vývoja, je čoraz viacej implementácií zameraných práve na ne. Vzhľadom na neexistenciu jednotného rozhrania, či už používateľského alebo programátorského, je častokrát nemožné použiť niektorú časť vizualizačného balíka samostatne. To spôsobuje nutnosť opakovaného programovania toho istého kódu a znižuje efektivitu práce.

1.1 Objemové dáta

V počítačovej grafike existujú rôzne reprezentácie trojrozmerných objektov. Bežné sú povrchové reprezentácie, ktoré popisujú objekty pomocou dvojrozmerných primitív. Najčastejšie sa jedná o súbor polygónov alebo špeciálne trojuholníkov tvoriacich povrch nejakého objektu. Objemové dáta obsahujú informácie aj o vnútorných štruktúrach objektov, príkladom môžu byť dáta z tomografu. Vizualizácia takýchto dát sa líši od vizualizácie spomenutých geometrických primitív, keďže neobsahujú explicitnú informáciu o povrchu. Objemové dáta sa zvyčajne vyskytujú v medicíne, kde vznikajú pri procesoch, ako počítačová tomografia alebo magnetická rezonancia, či iných. Majú však využitie aj v iných odboroch, akými sú napríklad fyzika či archeológia. Používajú sa aj syntetické dáta, ktoré sú produktom rôznych matematických modelov. Na počítačoch bývajú dáta vzorkované v diskrétnych bodoch a kvantizované do konečného počtu úrovní.



Obrázok 1.1: Rozličné rekonštrukčné filtre: a) pravoúhle okno, b) trojuholníkové okno, c) sinc filter

1.2 Vzorkovanie a rekonštrukcia

Vzorkovanie predstavuje proces konvertovania spojitej funkcie na diskrétnu [20]. Hodnoty funkcie sú zaznamenávané v diskrétnych, zväčša rovnomerne vzdialených bodoch. Keďže na počítači sú dostupné iba číselné typy obmedzených rozsahov a pôvodná funkcia môže nadobúdať prakticky neobmedzené hodnoty, je nutné hodnoty jednotlivých vzoriek kvantizovať. Obor hodnôt spojitej funkcie sa uniformne rozdelí na intervaly, ktorých počet zodpovedá počtu kvantizačných úrovní. Hodnoty spadajúce do jedného intervalu budú reprezentované rovnakou kvantizačnou hodnotou. Bežne sú používané celočíselné typy o rozsahu jedného, či dvoch bajtov a čísla s plávajúcou desatinnou čiarkou v jednoduchej aj dvojitej presnosti.

Pri vizualizácií zväčša potrebujeme poznať hodnotu funkcie aj mimo vzorkovaných bodov a teda rekonštruovať pôvodnú spojitú funkciu z jej diskrétnej verzie. Rekonštrukcia sa vykonáva konvolúciou jadra rekonštrukčného filtra s diskrétnou funkciou. Používajú sa rôzne filtre, najjednoduchším je pravouhlé okno (box filter, obr. 1.1(a)), ktorý predstavuje interpoláciu najbližšieho suseda. Často používaným filtrom je trojuholníkové okno (tent filter, obr. 1.1(b)), ktoré zodpovedá lineárnej interpolácii a je priamo implementované v dnešných grafických kartách. Na dosiahnutie úplnej rekonštrukcie musí pre pôvodnú funkciu platiť, že neobsahuje frekvencie vyššie ako je nejaká hranica a zároveň pri vzorkovaní musí byť použitá aspoň dvojnásobná frekvencia (Nyquist frequency) [20]. Ak dáta spĺňajú tieto požiadavky, úplná rekonštrukcia sa dá teoreticky docieliť použitím sinc filtra ($sinc(x) = \frac{sin(\pi x)}{\pi x}$, obr. 1.1(c)). Tento filter má však neohraničený rozsah a prakticky je nemožné ho použiť. Dajú sa však docieliť lepšie výsledky ako pri použití trojuholníkového okna, ak v ohraničenom rozsahu aproximujeme sinc filter. Používajú sa napríklad bikubické alebo iné splajny. Vo všeobecnosti sa dá na dnešných

grafických kartách použiť ľubovoľný filter, pri cene vyšších časových nárokov na renderovanie [9].

Kapitola 2

Postupy vizualizácie objemových dát

Proces zobrazovania objemových dát na výstupné dvojrozmerné médium sa nazýva objemovou vizualizáciou. Keďže dáta neobsahujú explicitnú informáciu o povrchu, neexistuje priamočiary spôsob ako tieto dáta zobraziť. Existujú rôzne prístupy riešenia tohto problému a vo všeobecnosti ich možno rozdeliť na priame a nepriame [24]. Nepriame metódy sa snažia extrahovať z objemových dát informáciu o povrchu a ten následne renderovať klasickými metódami počítačovej grafiky. Priama vizualizácia pracuje bez explicitného skúmania povrchu a zobrazuje každý objemový element (voxel).

2.1 Nepriama vizualizácia

Pri nepriamych metódach dochádza k vytváraniu povrchu z oblastí s rovnakými alebo podobnými vlastnosťami a k jeho následnému vykresleniu bežnými metódami. Nevýhodou tohto prístupu sú relatívne nekvalitné výsledky pri použití nedostatočného počtu geometrických primitív. Ďalej sa využíva iba časť dát a zvyšná informácia sa ignoruje, čo môže spôsobiť vynechanie dôležitých častí dát pri zobrazovaní. Pre objekty, pri ktorých je ťažké hovoriť o povrchu (dym, oblaky), metódy nedávajú uspokojivé výsledky. Medzi najčastejšie používané metódy patrí vyhľadávanie kontúr a implicitné povrchové pokrývanie (marching cubes) [14].

2.1.1 Vyhľadávanie kontúr

V prípade, že objemové dáta sú reprezentované ekvidistančnou alebo uniformnou mriežkou, dá sa na ne prirodzene nazerať ako na sadu rezov. Me-

dicínske zariadenia dáta v rezoch v skutočnosti aj produkujú. Prirodzenou metódou extrakcie povrchu je rozšírenie z dvojrozmerného prípadu do trojrozmerného. V každom reze sa identifikuje hranica objektu pomocou segmentácie. Hranice v susedných rezoch sa následne spájajú plošnými primitívami, obvykle trojuholníkmi. Výsledkom je množina primitív tvoriaca povrch objektu medzi jednotlivými rezmi. Keďže neexistuje univerzálne pravidlo pre spájanie kontúr v susedných rezoch, používajú sa rôzne prístupy. Maximalizuje sa ohraničený objem [10], minimalizuje sa vytváraný povrch [7], či sa vyberajú také priečky, ktoré majú podobnú orientáciu alebo najkratšiu dĺžku [3]. Žiadna z automatických metód však nevytvára zaručene správny povrchový model a tak zásah operátora býva častokrát nevyhnutný.

2.1.2 Implicitné povrchové pokrývanie

Algoritmy aproximujú povrch objektu geometrickými primitívami. Delia priestor na množinu neprekrývajúcich sa buniek, najčastejšie kociek. Každý vrchol je označený ako vnútorný alebo vonkajší na základe hodnoty funkcie definujúcej objem. V bunkách, ktoré obsahujú vrcholy oboch typov sa musí nachádzať povrch objektu. V týchto bunkách sú identifikované hrany tvorené prienikom povrchu so stenami bunky a následne je z nich vytvorená plošná primitíva (zväčša trojuholník). Hrany sa hľadajú buď interpoláciou hodnôt vo vrcholoch, alebo ak to funkcia definujúca objem dovoľuje, tak aj priamo z nej.

Jedným z najpoužívanejších algoritmov je algoritmus marching cubes [14], ktorý delí priestor na sieť kociek. Algoritmus bol pôvodne určený na vytváranie trojuholníkových modelov trojrozmerných medicínskych dát. Algoritmus uvažuje 256 možných spôsobov usporiadania povrchu v rámci bunky, pričom pri použití symetrie sa toto číslo redukuje na 14. Pri vytváraní trojuholníkov sa vyrátava aj normála, ktorá je následne použitá na tieňovanie. V niektorých prípadoch môže algoritmus dávať lokálne nesprávne výsledky. Na ich odstránenie existuje niekoľko modifikácií algoritmu [2, 19, 18].

2.2 Priama vizualizácia

Na rozdiel od nepriamych, priame metódy explicitne neextrahujú informácie o povrchu objektu, ale priamo vizualizujú jednotlivé objemové elementy (voxle) objemu. Vo všeobecnosti sú na jednotlivé voxle mapované optické vlastnosti, ktoré určujú, ako sa príslušný voxel prejaví vo výslednom obraze.

2.2.1 Optické vlastnosti

Zariadenia produkujúce objemové dáta pracujú na rôznych fyzikálnych princípoch. V závislosti na použitom zariadení a použitej metóde snímania potom môžu hodnoty v dátach reprezentovať rôzne fyzikálne veličiny, či vlastnosti objemu. Keďže vizualizácia pracuje s optickými vlastnosťami, tieto musia byť na dáta mapované. Zväčša sa pracuje s farbou a intenzitou emitovaného svetla, priehľadnosťou, či vlastnosťami súvisiacimi s rozptylom svetla. Spôsob, akým tieto optické vlastnosti ovplyvňujú výsledný obraz je určený použitým optickým modelom. Medzi najbežnejšie patria podľa [17] nasledovné:

Absorpcia Objem je tvorený perfektne čiernymi telesami, ktoré pohlcujú svetlo. Žiadne svetlo neemitujú ani nerozptyľujú. Model môže byť použitý na simuláciu použitia röntgenu.

Emisia Teleso obsahuje len častice, ktoré emitujú svetlo. Nedochádza k žiadnej absorpcii. Model je použiteľný pre svetlo emitujúce častice, ktorých rozmer je zanedbateľne malý. Príkladom môžu byť žiariace plyny.

Absorpcia a emisia Častice emitujú svetlo a zároveň pohlcujú prichádzajúce svetlo. Nedochádza k rozptylu.

Rozptyl a tieňovanie alebo tienenie Svetlo prichádzajúce mimo voxla je rozptýlené a buď sa s ním narába ako s ničím neovplyvneným svetlom z nejakého vzdialeného zdroja alebo sa berú do úvahy aj voxle cez ktoré prechádza.

Viacnásobný rozptyl V tomto modeli sa berie do úvahy aj svetlo, ktoré už bolo rozptýlené viacerými časticami.

Na dnešnom hardvéri je už možné v reálnom čase používať aj niektoré zo zložitejších modelov spomedzi uvedených, avšak najpoužívanejším modelom v objemovej vizualizácií je absorpcia a emisia. Poskytuje dostatočne kvalitné výsledky pri relatívne nízkych nárokoch na výpočtový čas. Pri tomto modeli prebieha zobrazovanie v smere pohľadu a kombinuje optické vlastnosti voxlov. Tento proces sa dá popísať pomocou renderovacieho integrálu 2.1.

$$C = \int_0^D c(s(\vec{x}(t)))e^{-\int_0^t \tau(s(\vec{x}(t')))dt'}dt \tag{2.1}$$

Na získanie výslednej farby C pixla, je potrebné integrovať lúč, ktorý prechádza objemom a dopadá kolmo na priemetňu v danom pixli. Integrovať stačí

do hĺbky, kde sa ešte nachádza zobrazovaný objem (vzdialenosť D). Funkcia $\vec{x}(t)$ predstavuje hodnoty objemu a $s(\vec{x}(t))$ ich skalárne vzorky. Keďže používame emisno-absorpčný model, $\tau(s(\vec{x}(t)))$ predstavuje absorpčnú zložku bodu v objeme a $c(s(\vec{x}(t)))$ svetelnú (farebnú) zložku. Integruje sa emisia bodu, pričom je pohlcovaná objemom ležiacim medzi aktuálnym bodom a priemetňou ($e^{-\int_0^t \tau(s(\vec{x}(t')))dt'}$). V praxi je integrál vyčíslovaný numericky a to buď spredu dozadu (front-to-back) alebo zozadu dopredu (back-to-front).

2.2.2 Klasifikácia

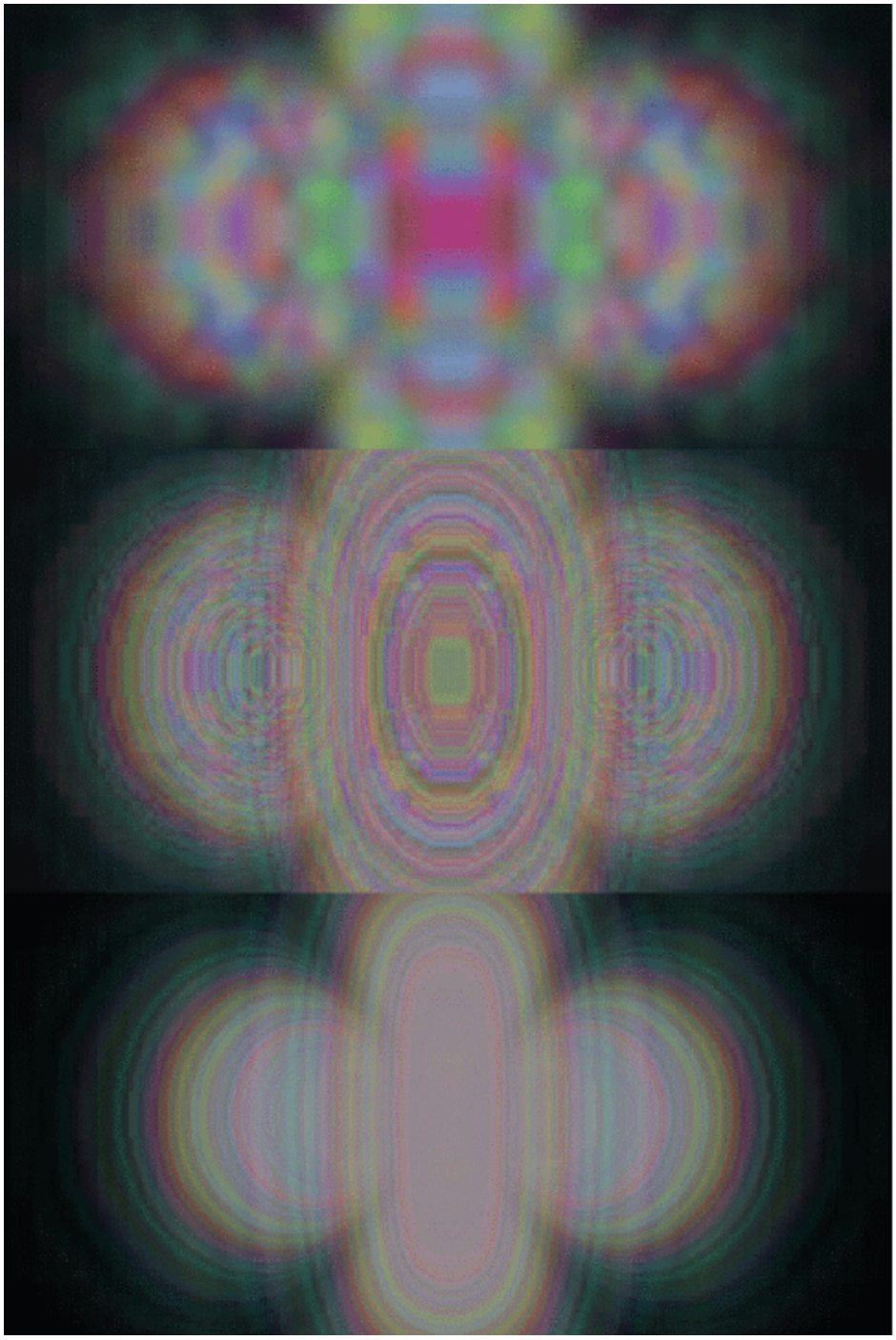
Proces mapovania optických vlastností na dáta sa označuje ako proces klasifikácie. Mapovanie je zväčša určené prenosovou funkciou vo forme mapovacej tabuľky. Pri vizualizácií je potrebné diskrétne dáta prevzorkovať a až následne vykresliť. Momentálne sa používajú tri odlišné prístupy, ako mapovať na dáta optické vlastnosti. Pred-klasifikácia mapuje optické vlastnosti priamo na vzorky a prevzorkovanie pracuje už s príslušnými optickými vlastnosťami. Po-klasifikácia robí prevzorkovanie na pôvodných dátach a optické vlastnosti mapuje až následne. Pred-integrácia rieši problém nelineárnych, vysokofrekvenčných prenosových funkcií. Udržiava predrátanú tabuľku optických vlastností pre všetky kombinácie hodnôt vzoriek v susedných rezoch, pričom optické vlastnosti sú medzi rezmi integrované [5].

Výsledky po-klasifikácie sú kvalitnejšie ako výsledky pred-klasifikácie, špeciálne v prípade, že prenosová funkcia obsahuje prudké zmeny. Pred-integrácia je porovnateľná s po-klasifikáciou a v prípade vysokofrekvenčných prenosových funkcií podáva ešte kvalitnejšie výsledky. Porovnanie výsledkov jednotlivých metód je na obrázku 2.1. Všetky prístupy sú dostupné na bežných grafických kartách.

2.2.3 Metódy

Vo všeobecnosti možno priame metódy vizualizácie rozdeliť do dvoch skupín podľa spôsobu prechádzania dát. Obrazovo orientované techniky pracujú v priestore výsledného dvojrozmerného obrazu a pre každý obrazový element (pixel) vypočítavajú výslednú farbu. V objeme sú prechádzané len voxle ovplyvňujúce práve spracovávaný pixel. Typickým príkladom obrazovo orientovanej metódy je algoritmus vrhania lúča (ray casting) [24].

Druhou skupinou sú objektovo orientované algoritmy, ktoré postupne prechádzajú objem a počítajú príspevok spracovávaného voxla vo výslednom obraze. Používajú sa dva prístupy - spredu dozadu (front-to-back) a zozadu dopredu (back-to-front). Prvý prechádza objem postupne od voxlov, ktoré sú najbližšie k pozorovateľovi až po najvzdialenejšie voxle. Pri výpočte



Obrázok 2.1: Výsledky klasifikácie sférickej harmonickej funkcie priamou metódou za použitia náhodnej prenosovej funkcie. Použité spôsoby klasifikácie: pred-klasifikácia (hore), po-klasifikácia (v strede), pred-integrácia (dole) (zdroj [8]).

príspevku vzdialenejších voxlov sa používajú doteraz vypočítané prírastky bližších voxlov. Druhý prístup pracuje v opačnom smere, pričom treba riešiť problém ovplyvňovania prírastkov vzdialenejších voxlov, ešte nespracovanými, bližšími voxlami. Medzi objemovo orientované techniky patria napríklad prístupy renderujúce objem po rezoch [23].

Nie všetky metódy sa dajú jednoznačne zadeliť do niektorej skupiny. Metódy, ktoré využívajú oba prístupy nazývame hybridné. Nasleduje popis niektorých najčastejšie používaných metód.

Vrhanie lúča

Sledovanie lúča (ray casting) predstavuje priamočiaru numerickú evaluáciu renderovacieho integrálu. Tento prístup môžeme zaradiť medzi obrazovo orientované, keďže algoritmus postupne vykresľuje jednotlivé body výsledného obrazu. Pre každý obrazový bod sa vyšle lúč v smere pohľadu do objemu. Pozdĺž tohto lúča sa v rovnomerne vzdialených bodoch rekonštruuje hodnota z diskrétnych vzoriek. Vzdialenosť bodov zodpovedá vzorkovacej vzdialenosti. Na rekonštrukciu sa obvykle používa trilineárna interpolácia [13]. Po získaní interpolovanej hodnoty a jej prislúchajúcich optických vlastností je aproximovaný renderovací integrál. Nahradením integrovania sumáciou dostávame diskrétnu reprezentáciu procesu. Ak namiesto sumácie v exponente e použijeme súčin členov, rovnica 2.1 dostane tvar:

$$C \doteq \sum_{i=0}^{\lfloor D/d \rfloor} c(s(\vec{x}(id)))d \prod_{j=0}^{i-1} e^{-\tau(s(\vec{x}(jd)))d} \tag{2.2}$$

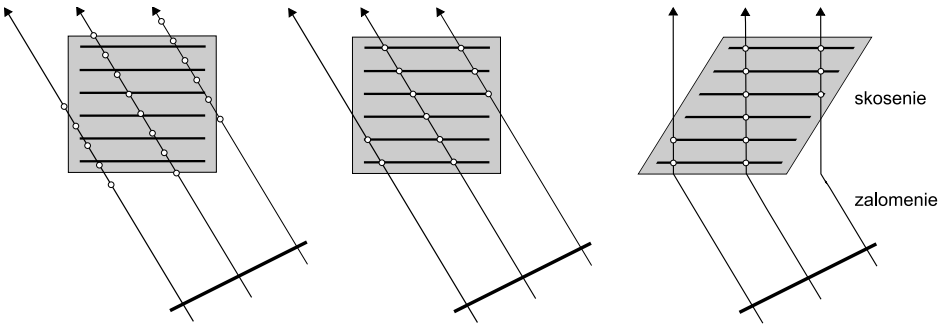
V rovnici sa vyskytujúce d predstavuje vzorkovaciu vzdialenosť. Ak označíme $C_i = c(s(\vec{x}(id)))d$ a $A_i = 1 - e^{-\tau(s(\vec{x}(id)))d}$, dostaneme vzorec v tvare

$$C \doteq \sum_{i=0}^{\lfloor D/d \rfloor} C_i \prod_{j=0}^{i-1} (1 - A_j) \tag{2.3}$$

Rovnica 2.3 sa dá vyčísliť iteratívne pomocou alfa zmiešavania (alpha-blending). Pri prechode zozadu dopredu je aktuálna akumulovaná hodnota C'_i vypočítaná z farby C_i a nepriehľadnosti A_i na aktuálnej pozícii i a z akumulovanej farby C'_{i+1} doteraz prejdeného objemu.

$$C'_i = C_i + (1 - A_i)C'_{i+1} \tag{2.4}$$

Začiatočná podmienka je $C'_{\lfloor D/d \rfloor} = 0$.



Obrázok 2.2: Algoritmus shear-warp pre ortogonálnu projekciu

Pri prechode spredu dozadu je treba udržiavať aj akumulovanú alfa hodnotu.

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \quad (2.5)$$
$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \quad (2.6)$$

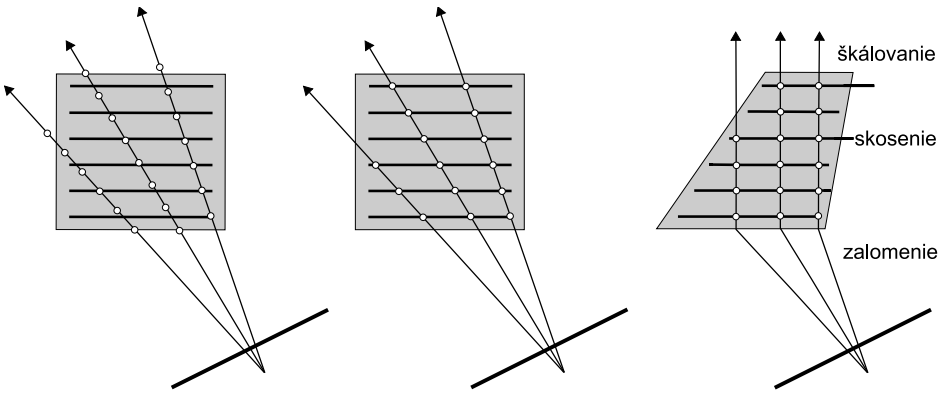
Začiatočné podmienky sú $C'_0 = A'_0 = 0$. Výpočet akumulovanej alfa hodnoty predstavuje úkon navyše, avšak umožňuje prechod predčasne ukončiť, ak akumulovaná alfa hodnota dosiahne 1.

Shear-warp algoritmus[12]

Shear (skosenie) a warp (zalomenie) sú úkony, ktoré umožňujú zarovnať rezy objemu voči pozorovateľovi tak, aby ich bolo možné jednoducho vykresliť. Algoritmus pracuje v objektovom priestore a postupne kombinuje susedné rezy. Najskôr vypočíta vektor posunutia jednotlivých rezov (shear) tak, aby pohľadový lúč prechádzajúci objemom bol kolmý na jednotlivé rezy. Rezy sa zobrazujú na priemetňu rovnobežnú s rezmi, ktorá sa následne premietne na priemetňu zodpovedajúcu vektoru pohľadu (warp). Obrázok 2.2 ilustruje prácu algoritmu pri ortogonálnej projekcii. V prípade perspektívnej projekcie je nutné okrem posunutia rezov tieto ešte škálovať. Postup je zobrazený na obrázku 2.3. Algoritmus bol navrhnutý pre softvérové renderovanie, avšak renderovanie na grafických kartách za použitia textúr pracuje na podobnom princípe. V tomto prípade sú jednotlivé transformácie vykonávané automaticky grafickou kartou.

Zobrazovanie pomocou textúr

Texturovacia funkcionalita dostupná na dnešných grafických kartách implementuje proces prevzorkovania, ktorý je možné využiť pri objemovej vizu-

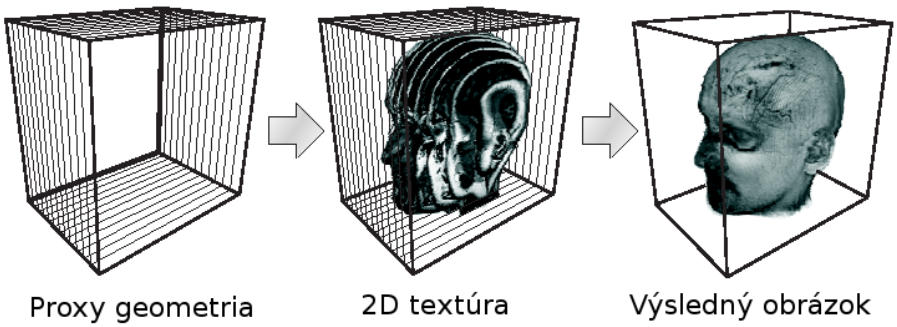


Obrázok 2.3: Algoritmus shear-warp pre perspektívnu projekciu

alizácií. Na kombinovanie rôznych vzoriek sa používa hardvérové alfa zmiešavanie. Prístup pracuje v objektovom priestore a prechádza dáta, ktoré sú uložené vo forme sady 2D textúr alebo jednej 3D textúry [23, 4]. Aby bolo možné tieto dáta vykresliť, je nutné textúry mapovať na nejaké geometrické primitívy. Tieto geometrické objekty sa nazývajú proxy geometria a zväčša sa jedná o obdĺžniky. V prípade použitia proxy geometrie zodpovedajúcej rezom objemu, dostávame analogický výpočet ako v prípade algoritmu shear-warp.

Dnešné grafické karty umožňujú špecifikovať vlastné programy, ktoré sa vykonávajú pri spracovaní geometrických primitív a každého fragmentu vzniknutého rasterizáciou. Tieto programy sa označujú ako shadre (nom. shader). Pomocou shadrov je možné vykonávať mapovanie optických, či iných vlastností a vo všeobecnosti upravovať spôsob, akým sa bude narábať s fragmentom. Shadre sa preto často používajú pri vizualizácii pomocou textúr, kde umožňujú rozšíriteľnosť v rámci príslušnej metódy.

2D textúry Pri tejto metóde sú dáta uložené ako súbor 2D textúr, ktoré predstavujú obdĺžnikové rezy objemu. Tieto rezy je možné na grafických kartách priamo renderovať a využiť hardvérovú podporu pre textúrovanie, čím dosiahneme bilineárnu interpoláciu. Aby pri rôznych pohľadových uhloch nedochádzalo k pohľadu pomedzi rezy, používajú sa tri sady textúr, každá reprezentujúca objem v niektorej osovej orientácii. Sada použitá pre dané vykreslenie sa zvolí na základe veľkosti pohľadového uhla s normálou rezov danej sady. Obrázok 2.4 ilustruje usporiadanie rezov pri použití 2D textúr. Rezy sú vykresľované od najvzdialenejšieho po najbližší. Emisia je reprezentovaná RGB farbou a alfa hodnota reprezentuje priehľadnosť. Pri



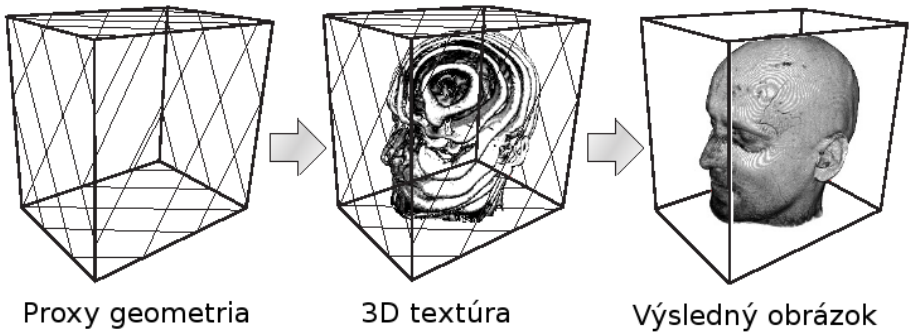
Obrázok 2.4: Osovo orientované rezy pri použití 2D textúr (zdroj [8])

vykresľovaní je použitý požadovaný hardvérovo podporovaný mód pre zmiešavanie.

Výhodou použitia 2D textúr je vysoká výkonnosť vďaka použitej hardvérovej podpore bilineárnej interpolácie pri texturovaní ako aj dostupnosť aj na starších grafických kartách. Medzi nevýhody patrí vysoká pamäťová náročnosť spôsobená nutnosťou udržiavať tri sady textúr, ďalej nepresnosť bilineárnej interpolácie, ako aj viditeľné prechody pri niektorých zmenách pohľadového uhľa, spôsobených prepnutím používanej sady textúr. Navyše fixná orientácia rezov má za príčinu nerovnomernú vzorkovaciu frekvenciu pri rôznych pohľadových uhloch.

3D textúry Pri použití 3D textúry sú dáta udržiavané v celku v jednej objemovej textúre. Namiesto bilineárnej sa používa trilineárna interpolácia a renderované rezy nemusia byť orientované podľa osí objemu. Používajú sa rezy kolmé na pohľadový vektor, čím je odbúraný problém premenlivej vzorkovacej frekvencie vyskytujúci sa pri 2D textúrach. Keďže sa používa trilineárna interpolácia, je možné jednoducho upravovať počet rezov, resp. vzdialenosť dvoch susedných rezov. Orientácia rezov je ilustrovaná na obrázku 2.5.

Výhodami tejto metódy je kvalita výstupu spôsobená trilineárnou interpoláciou, ako aj pamäťová náročnosť, keďže dáta sú udržiavané len v jednej kópii. Nevýhodou je nižšia rýchlosť trilineárnej interpolácie ako aj dostupnosť podpory pre 3D textúry na starších grafických kartách.



Obrázok 2.5: Pohľadovo orientované rezy pri použití 3D textúr (zdroj [8])

2.3 Bricking

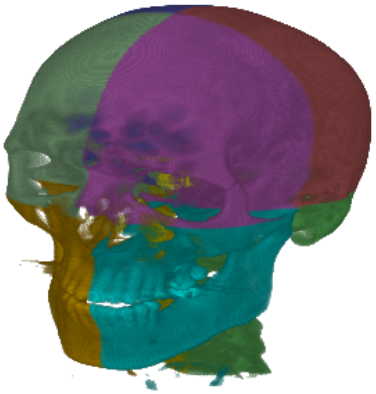
Bricking znamená rozdelenie objemu na menšie bloky. Umožňuje renderovať dáta veľkých rozmerov, ktoré by sa normálne nezmestili celé do pamäte. Ďalšie využitie má pri optimalizácií výpočtového času, resp. pri eliminácií výkonových rozdielov v závislosti od pohľadového uhlu za použitia 3D textúr [22].

Pri brickingu sa jednotlivé bloky postupne načítavajú do pamäte a následne renderujú. Aby nebolo potrebné pri renderovaní bloku pristupovať k ostatným, je potrebné, aby sa susedné bloky prekrývali. Pri textúrovaní totiž dochádza k interpolácii, ktorá v závislosti na použitom rekonštrukčnom filtri potrebuje pristupovať k okoliu bodu. V prípade, ak by sa bloky neprekrývali, na hranici by sme dostávali nekorektné výsledky. Bloky sa musia navzájom prekrývať o $w-1$ pixlov, kde w je šírka rekonštrukčného filtra [16]. Na obrázku 2.6 je naznačené rozdelenie objemu na 8 blokov.

2.4 Vizualizačné riešenia

Vo všeobecnosti možno dostupné vizualizačné nástroje rozdeliť podľa prístupu k dátam. Aplikácie využívajúce model toku dát (data-flow) používajú sieť filtrov, v ktorej dáta tečú z jedného filtra do ďalšieho. Scénovo orientované aplikácie obsahujú nejakú reprezentáciu virtuálnej scény, v ktorej sa nachádza vizualizovaný objem.

Model toku dát Model toku dát je založený na použití samostatných modulov reprezentujúcich rôzne operácie na dátach. Každý modul má svoju

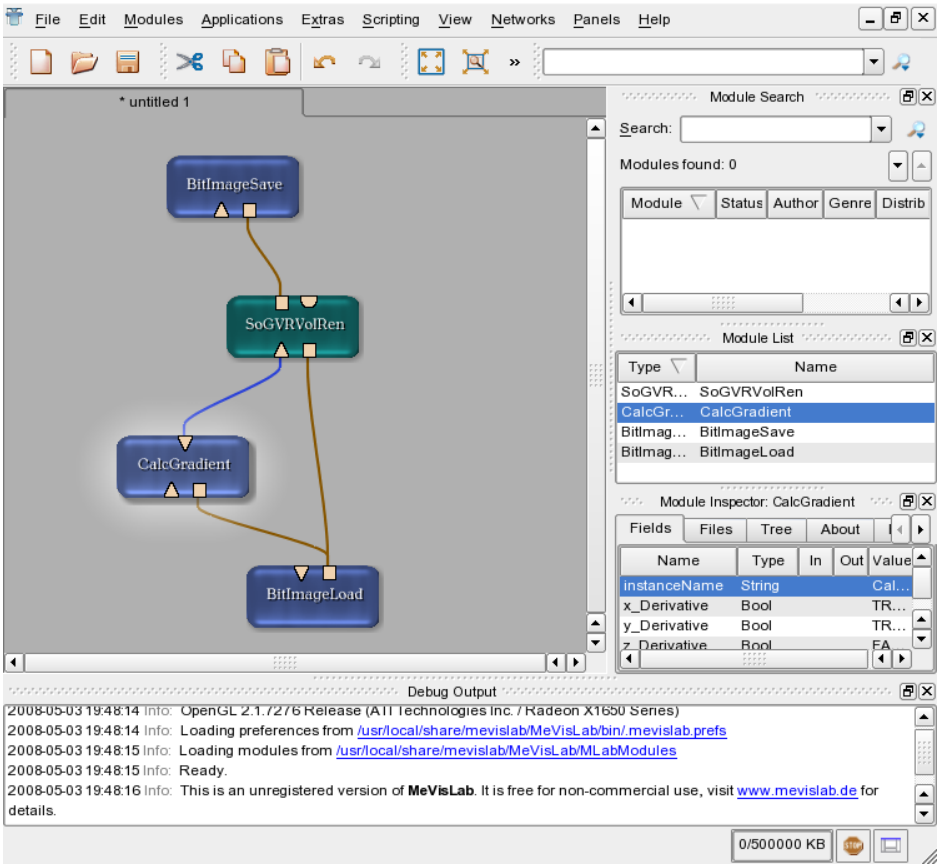


Obrázok 2.6: Príklad rozdelenia objemu na bloky

vstupno-výstupnú charakteristiku, na základe ktorej je možné moduly spájať. Takto sa vytvorí acyklický orientovaný graf, reprezentujúci postupnosť vykonávania operácií na dátach. Tieto putujú z jedného modulu do druhého, až kým nedosiahnu terminálny modul, ktorý dáta zobrazí, alebo uloží na disk. Väčšina používaných operácií spadá do kategórie predspracovania a samotná objemová vizualizácia je väčšinou reprezentovaná jedným samostatným modulom. Medzi aplikácie využívajúce model toku dát patrí napríklad SciRUN[1] univerzity v Utahu alebo program MeVisLab[2] (obr. 2.7).

Graf scény Pojem grafu scény nemá exaktnú definíciu a môže reprezentovať veľa rôznych štruktúr. Vo všeobecnosti sa jedná o nejakú hierarchickú reprezentáciu virtuálnej scény. Prostredia ako OpenSceneGraph[3], Open Inventor[4] a iné obsahujú graf scény, ktorý je hierarchicky prechádzaný a vizualizovaný. Hlavným stavebným kameňom tohto grafu sú geometrické primitívy. Vo vizualizačných aplikáciach sa častokrát pracuje len s kolekciou objektov ako napríklad kamera, rezové roviny a iné, ktoré síce nemusia byť reprezentované hierarchicky, ale reprezentujú virtuálnu scénu. V ďalšom texte budeme pod pojmom scéna/graf scény rozumieť ľubovoľnú reprezentáciu virtuálnej scény. Algoritmy využívajúce graf scény budú potom reprezentovať všetky algoritmy, ktoré vytvárajú model virtuálnej scény, použiteľný bez špecifikovania operácií, ktoré sa s ním majú vykonávať.

[1] http://software.sci.utah.edu/scirun.html
[2] http://www.mevislab.de
[3] http://www.openscenegraph.org
[4] http://oss.sgi.com/projects/inventor/



Obrázok 2.7: Prostredie programu MeVisLab

Keďže knižnice pre graf scény sú orientované na geometrické objekty, zväčša neobsahujú podporu pre vizualizáciu objemových dát. Je však možné tieto knižnice rozšíriť. Existuje napríklad rozšírenie prostredia OpenSG[5] o funkcionalitu priamej objemovej vizualizácie za použitia textúr [11]. Toto rozšírenie obsahuje autonómny objekt vložiteľný do scény, ktorý zabezpečuje zobrazovanie objemových dát. Spôsob vizualizácie je ovplyvňovaný zvolenou prenosovou funkciou a shaderom, ktorý špecifikuje algoritmus zmiešavania. Rozšíriteľnosť spočíva v možnosti použitia iných shaderov, bez nutnosti zasahovania do zvyšku aplikácie.

Ďalšie delenie vizualizačných aplikácií čiastočne súvisí s použitým modelom. Model toku dát umožňuje väčšiu modularitu, aj keď skôr v oblasti predspracovania. Graf scény je skôr použiteľný v monolitických aplikáciach.

Kompletné aplikácie Bežne sú dostupné kompletné aplikácie integrujúce užívateľské rozhranie s vybranou sadou vizualizačných algoritmov. Aj keď tieto aplikácie nemusia byť postavené na monolitickom prístupe, rozhranie jednotlivých algoritmov je úzko naviazané na použité oknové knižnice a tak nie je jednoduché použiť či už užívateľské rozhranie alebo vizualizačné algoritmy samostatne. Rozšíriteľnosť sa zväčša obmedzuje na dostupné shadre[6].

Vizualizačné knižnice Okrem kompletných aplikácií sú dostupné aj čisto vizualizačné knižnice. Problém s použitím knižníc je v nejednotnosti ich programovacieho rozhrania, čo sťažuje použitie viacerých knižníc v rámci jednej aplikácie.

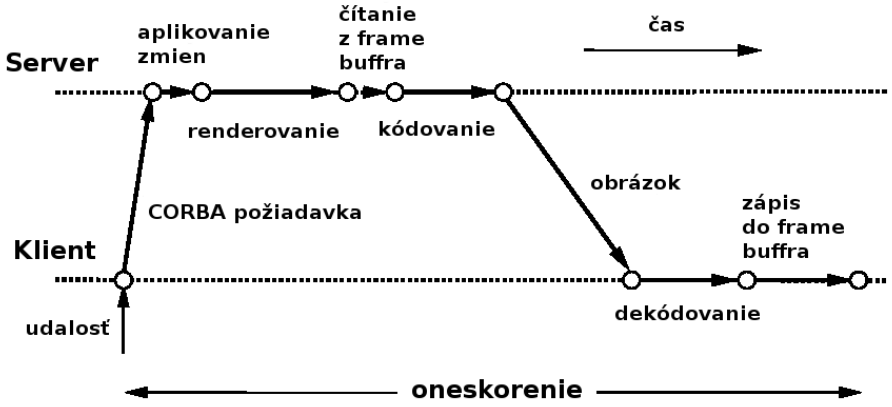
Komplexné prostredia Najvšeobecnejšou skupinou sú prostredia pre vedeckú vizualizáciu. Prostredia ako SciRUN, či MeVisLab pracujú s modelom toku dát, čo síce umožňuje vyššiu voľnosť v narábaní s dátami, ale zároveň prináša istú zložitosť do ovládania.

2.5 Vzdialená vizualizácia

Vizualizácia objemových dát má relatívne vysoké hardvérové nároky. Aj keď na trhu sa nachádzajú výkonovo dostatočné komponenty akými sú napríklad viacjadrové procesory, či moderné grafické karty, ich cena nie je zanedbateľná.

[5] http://opensg.vrsource.org/trac

[6] Príkladom môže byť aplikácia OpenQVis, ktorá napríklad dovoľuje použitie shaderov len pre grafické karty NVidia - http://openqvis.sourceforge.net



Obrázok 2.8: Časové oneskorenie spôsobené vzdialenou vizualizáciou (zdroj [6])

Zároveň však dochádza k prudkému rozvoju komunikačných sietí, umožňujúcich za prijateľné ceny prenášať pomerne vysoké objemy dát. To umožňuje oddeliť od seba výpočtovo náročnú fázu renderovania od užívateľského rozhrania a vizualizáciu prevádzkovať na výkonných sieťovo dostupných výpočtových staniciach.

Klaus Engel a kol. [6] predstavili architektúru pre vzdialenú vizualizáciu aplikácií, založených na Open Inventor a Cosmo3D[7] prostrediach. Ľubovoľné vizualizačné riešenie postavené na spomínaných prostrediach sa dá upraviť na serverovú verziu, ktorá je používaná pri vizualizácii. Komponenty klientskej verzie, zabezpečujúce komunikáciu so serverom a zobrazovanie renderovaných dát, sú implementované v prostredí Java2[8]. Navrhovaný prístup umožňuje dva spôsoby použitia.

- Klientská verzia zbiera vstupy od užívateľa a pomocou rozhrania CORBA[9] ich zasiela serveru. Serverová verzia, pracujúca bez rozhrania, prijíma vstupy, reaguje na ne a odpovedá opäť cez rozhranie CORBA. V prípade, že je potrebné vyrenderovať objemové dáta, tieto sú renderované do mimoobrazovkového buffra a pomocou socketov je výsledný obraz zaslaný klientskej aplikácii. Obrázok 2.8 ilustruje oneskorenie spôsobené vzdialenou vizualizáciou.

[7] Cosmo3D už nie je vyvíjaný
[8] http://java.sun.com/javase/
[9] Common Object Request Broker Architecture, http://www.corba.org/

- Užívateľ pracuje so serverovou verziou, ktorá poskytuje aj užívateľské rozhranie. Dáta sú renderované priamo na obrazovku a zároveň sú rozosielané všetkým pripojeným klientom, ktorí môžu sledovať interakciu operátora.

Na komunikáciu je použité rozhranie CORBA, ktoré umožňuje kooperovať objektom pri použití rôznych programovacích jazykov. Cieľom je umožniť upraviť ľubovoľnú aplikáciu, nezávisle od programovacieho jazyku, v ktorom je napísaná. Takisto prostredie Java2 je dostupné na rôznych platformách.

Za účelom zníženia objemu prenášaných dát, tieto je možné komprimovať. Ďalšej úspory je dosiahnuté zmenšovaním rozlíšenia prenášaného obrázku, pri rotácií scény. Pokiaľ užívateľ narába so scénou, dáta sú prenášané iba v polovičnej, či štvrtinovej veľkosti a následne sú zväčšené na pôvodnú veľkosť. V okamihu, keď užívateľ prestane manipulovať so scénou, obraz je prenesený v plnej veľkosti. Tieto postupy umožňujú dosiahnuť postačujúcu obnovovaciu frekvenciu aj pri pripojení využívajúcom 56k modem.

Kapitola 3

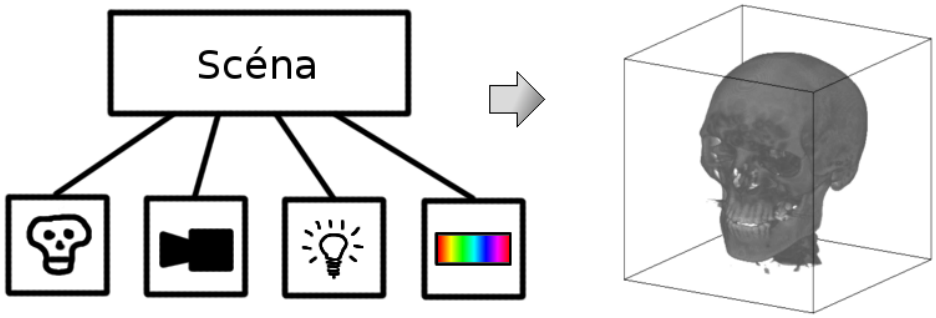
Architektúra

Pri vytváraní enginu sme vychádzali z analýzy požiadaviek. Keďže niektoré požiadavky sú prirodzene protichodné, nie všetky mohli byť splnené kompletne. Pretože existuje veľké množstvo dostupných vizualizačných riešení, ktorých kód je viac či menej optimalizovaný, zamerali sme sa na chýbajúci prvok - širokú rozšíriteľnosť. V rámci možností sme sa snažili zachovať aj vysoký výkon, výsledky výkonnostných meraní sa nachádzajú v kapitole 5.

3.1 Rozšíriteľnosť

Primárnou požiadavkou kladenou na engine je možnosť rozšíriteľnosti o nové vizualizačné algoritmy, resp. ich implementácie. Aby bola táto požiadavka uspokojivo splnená, musí dôjsť k oddeleniu kódu pre vizualizáciu od samotného enginu. Rozšíriť engine musí byť možné bez nutnosti zasahovať do kódu samotnej aplikácie. Toto sa dá dosiahnuť presunutím kódu vizualizačných algoritmov do samostatne kompilovateľných modulov alebo pluginov. Tieto bude možné ideálne pridávať za behu a bude tak umožnené pracovať s ľubovoľným, v budúcnosti vytvoreným pluginom.

Keďže engine má byť rozšíriteľný pomocou samostatne kompilovateľných modulov, je nevyhnutné, aby rozhranie, ktorým bude engine komunikovať s modulom bolo nemenné. Keďže pluginy majú byť vytvárané priebežne, toto rozhranie musí byť dostatočne flexibilné, aby umožnilo prácu ľubovoľného algoritmu. Plugin nemusí nutne využívať všetku funkcionalitu poskytovanú rozhraním a naopak nemusí implementovať všetky špecifické funkcie. O skutočných potrebách a možnostiach pluginu by mal byť engine informovaný pomocou dohodnutého spôsobu. Engine potom umožní užívateľovi používať iba implementované funkcie pluginu.



Obrázok 3.1: Scéna ako kolekcia objektov a nastavení

3.2 Scéna

Jednou z požiadaviek na engine je využitie grafu scény. Keďže nechceme klásť žiadne obmedzenia pre vizualizačné algoritmy, je žiadúce, aby formát scény bol čo najvšeobecnejší. Napríklad rozšírenie [11] je implementované pomocou komponent prostredia OpenSG. Je preto možné ho použiť len v spomenutom prostredí a navyše umožňuje len rozšíriteľnosť za použitia shadrov. Našou snahou je vytvoriť scénu, ktorá nie je priamo naviazaná na nejaké prostredie. Z tohto dôvodu sme navrhli scénu, ktorá pozostáva len z kolekcie vybraných elementov.

Požiadavka na rozšíriteľnosť enginu predstavuje nutnosť zvolenia fixnej sady elementov scény. Užívateľské rozhranie totiž nemá možnosť poznať špeciálne parametre budúcich pluginov a naopak renderovací plugin nemôže vedieť pracovať s elementami scény zavedenými nejakým užívateľským rozhraním. Aby zvolená sada elementov scény umožňovala prácu čo najširšieho spektra renderovacích modulov, musí zahŕňať najčastejšie sa vyskytujúce parametre viažuce sa na samotnú virtuálnu scénu, ako aj k prípadnému renderovaciemu algoritmu. Zo strany scény sme zvolili tieto elementy: kamera, svetelné zdroje, orezávacia geometria. Medzi elementy viažuce sa na vizualizačný algoritmus patrí 1D a 2D prenosová funkcia. Keďže nie všetky renderovacie pluginy musia nutne používať všetky vymenované elementy a keďže napríklad prenosová funkcia je špecifická iba pre niektoré postupy, plugin informuje o podporovaných elementoch a ostatné pri renderovaní ignoruje.

Na umožnenie špecifikovania ďalších hodnôt, ktoré môžu byť nevyhnutné pre prácu vizualizačného algoritmu, sme navrhli triedy dodatočných parametrov. Tieto umožňujú zadávať bežné typy, akými sú celé čísla, čísla s

pohyblivou rádovou čiarkou, booleovské hodnoty, reťazce, záznamy a polia číselných typov. Každý z týchto dodatočných parametrov obsahuje názov a popis, ktoré umožňujú ich správnu interpretáciu používateľom. Dodatočné parametre sú vytvárané renderovacím pluginom pri inicializácií. Keďže ich triedy sú vopred známe, užívateľské rozhranie má možnosť sprostredkovať užívateľovi tieto špecifické parametre algoritmu. Ďalšou možnosťou by bolo umožniť renderovacím modulom vytvárať aj inštancie vlastných tried, pričom tieto by bolo možné použiť len s užívateľským rozhraním poznajúcim tieto triedy. Túto možnosť sme ponechali pre prípadné ďalšie verzie enginu.

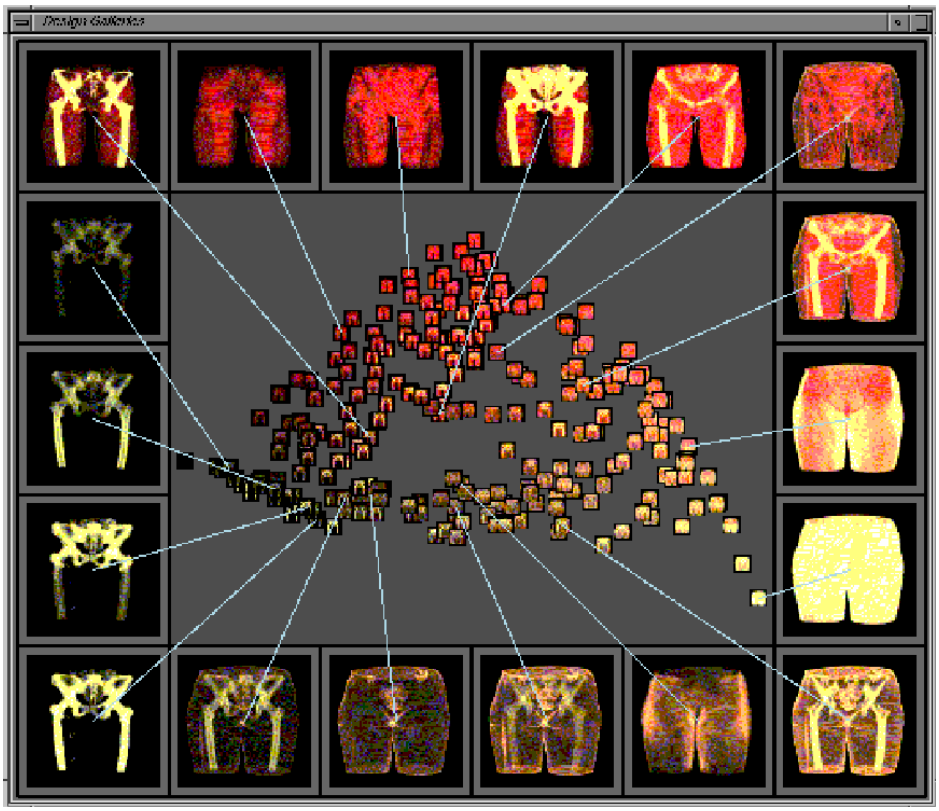
3.3 Podpora viacerých scén

Bežne sa vo vizualizačných aplikáciach poskytujúcich grafické užívateľské rozhranie pracuje naraz iba s jednou scénou. Pri vývoji nových algoritmov však môže byť užitočné paralelne vizuálne porovnávať výsledky vyvíjaného pluginu s nejakým existujúcim. Takisto môže mať význam zobrazovať objemové dáta súčasne z rôznych uhlov, či pracovať s rôznymi prenosovými funkciami zároveň (napríklad pri tzv. design gallery [15], pozri obr. 3.2). Možné sú dva spôsoby riešenia tejto požiadavky - použiť viacero inštancií enginu alebo umožniť enginu spravovať viacero scén. My sme sa z dôvodu jednoduchšieho použitia užívateľom, rozhodli pre druhý. Ak viacero scén používa rovnaké dáta v rovnakom formáte, tieto je možné zdielať v pamäti. Pri modeli jednej inštancie enginu podporujúceho viacero scén, sa užívateľ nemusí zaoberať zabezpečením komunikačného spojenia medzi viacerými inštanciami enginu pri modeli podporujúcom iba jednu scénu.

3.4 Pluginy

V engine používame okrem renderovacích aj ďalšie dva druhy pluginov. Jedná sa o pluginy zodpovedné za čítanie súborových formátov dát (v našom názvosloví loader) a potom o pluginy obstarávajúce vytváranie internej reprezentácie dát (builder). Využitie loaderu slúži na jednoduchú rozšíriteľnosť podpory nových súborových formátov. Jeho úlohou je extrahovať meta-dáta o súbore a poskytnúť ich užívateľovi ako aj ďalším komponentom enginu a hlavne dekódovať príslušný súbor. Loader umožňuje pristupovať k jednotlivým voxlom objemu, resp. ich čítať blokovo v nekomprimovanej forme.

Použitie builderov súvisí s podporou viacerých scén. V prípade, že by sme umožnili existenciu iba jednej scény, za vytváranie internej reprezentácie dát by mohol byť zodpovedný aj renderovací plugin. Pri viacerých scénach, však



Obrázok 3.2: Design gallery pre prenosové funkcie s rôznymi hodnotami priehľadnosti (zdroj [15])

tieto môžu zdielať dáta rovnakého formátu a teda je nutné túto funkcionalitu vyňať mimo plugin. Navyše sa tak uľahčí vytváranie renderovacích modulov, keďže v prípade použitia nejakej bežnej reprezentácie dát nebude treba programovať jej vytváranie. Builder po získaní dekódovaných dát z loaderu, vytvorí ich internú reprezentáciu a zároveň ju uskladní. Renderovacie pluginy potom získavajú referenciu buď na samotný builder, alebo na uskladnené dáta.

3.5 Vlákna a asynchrónne operácie

Podpora viacerých scén nabáda pracovať s enginom z viacerých vlákien, čo umožňuje prevádzať viacero operácií súčasne. Viacvláknový prístup však ohrozuje integritu interných premenných a tie je teda potrebné chrániť. Na tieto účely slúžia objekty operačného systému, ktoré však spôsobujú určité režijné náklady. Vzhľadom na to, že majoritný je čas potrebný na samotné renderovanie dát, tieto režijné náklady nepredstavujú neúnosné spomalenie operácií. Analýza režijných nákladov spojených s ochránením integrity dát sa nachádza v časti 4.6.

Pri vizualizácií v interaktívnom prostredí ovplyvňuje čas potrebný na vyrenderovanie dát mieru interaktivity. Vzhľadom na častokrát veľké rozmery objemových dát, použitie synchrónnych operácií by viedlo k strate interaktivity. Konkrétne v operačnom systéme Windows je fronta správ vykonávaná iba jedným vláknom. Aby sme odbúrali nutnosť vytvárať užívateľom vlákna vykonávajúce zdĺhavé operácie, rozhodli sme sa do enginu zaradiť podporu asynchrónnych volaní. Takéto volania sú v skutočnosti vykonávané separátnym vláknom a teda volajúce vlákno nemusí čakať na dokončenie operácie. Existujú v princípe dva prístupy, ako informovať užívateľa o skončení výpočtu. Buď musí sám užívateľ kontrolovať, či vykonávanie už skončilo alebo poskytne asynchrónnej operácií prostriedok, pomocou ktorého potom bude informovaný o ukončení operácie. Vzhľadom na to, že prvý prístup obsahuje prvky činného čakania, čiže jalového využívania procesorového času na kontrolu stavu operácie, rozhodli sme sa pre druhý prístup. Užívateľ má možnosť špecifikovať funkciu pre spätné volanie (tzv. callback), ktorá bude po skončení asynchrónnej operácie zavolaná.

3.6 Klient/server

Navrhnutá architektúra poskytujúca možnosť použitia viacerých scén a prístupu z viacerých vlákien súčasne, umožňuje priamočiare rozšírenie na ap-

likáciu typu klient/server. Na strane serveru treba zabezpečiť načúvaniu príchodzím sieťovým spojeniam, následne vykonávať požiadavky zasielané po sieti a posielať späť výsledky. Na klientskej strane musí najprv dôjsť k nadviazaniu spojenia so serverom a následne môžu byť požiadavky preposielané po sieti. Existujú prostredia umožňujúce použitie objektov, ktorých implementácia sa nachádza na vzdialenom počítači ako napríklad CORBA, tie sú však príliš zložité a trpia nedostatkami - napríklad používaním rovnakého komunikačného prístupu nezávisle na tom, či sa implementácia nachádza na lokálnom počítači alebo nie. Z uvedených dôvodov sme sa sa rozhodli namiesto zakomponovania vzdialeného prístupu do celej architektúry, vytvoriť separátny kód zabezpečujúci komunikáciu a následne na strane serveru používať lokálnu verziu enginu.

Toto riešenie umožní prácu s objemovými dátami aj na nedostatočne výkonnom hardvéri, keďže skutočné renderovanie sa deje na serveri. Ak je tento server dostatočne výkonný, umožní súčasné vykonávanie renderovania pre viacero klientov. V prípade interaktívnych aplikácií sa renderovanie vykonáva len pri zmene nastavení scény, ktorých frekvencia je limitovaná reakčným časom používateľa. V prípade dávkových operácií sú tieto vykonávané v nepretržitom slede, avšak nie je pri nich nutná interaktívna odozva a tak prípadné spomalenie nepredstavuje prílišný problém. Ďalšou výhodou tohto riešenia je aj šetrenie diskových kapacít, keď dátové objemy môžu byť skladované centrálne na serveri. To má špeciálne význam pri veľmi veľkých objemoch, rádovo v desiatkach GB.

Kapitola 4

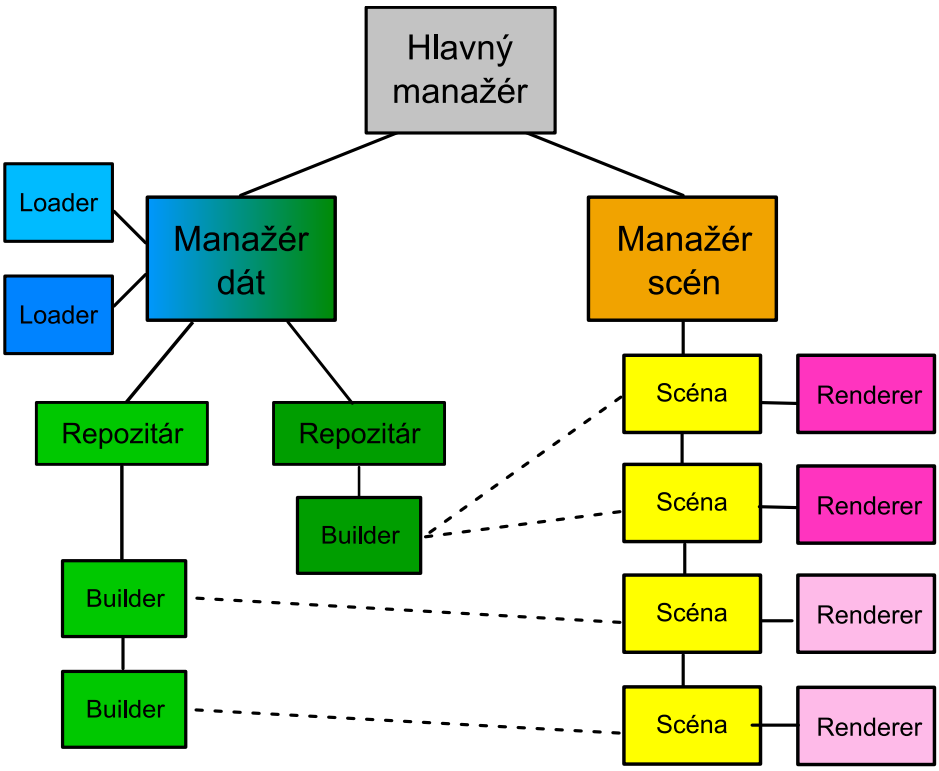
Implementácia

Pri implementácii sme sledovali niekoľko cieľov. Za prvé, vytváraná knižnica by nemala byť limitovaná na nejaký konkrétny operačný systém. Túto požiadavku sme splnili čiastočne, keď podporujeme operačné systémy Microsoft Windows a Linux. Využívame absolútne minimum funkcií špecifických pre jednotlivé operačné systémy a tak rozšírenie podpory o ďalšie, nepredstavuje vážny problém. Za druhé, naším cieľom bolo neobmedziť výkonnosť implementovaných vizualizačných algoritmov výberom technológií použitých na implementáciu enginu. To sme dosiahli výberom výkonného jazyka C++.

4.1 Manažéry

Funkcionalita pokrývaná enginom sa dá rozdeliť do viacerých kategórií ako je správa scén či dátových objemov. Aby nemusel užívateľ obstarávať komunikáciu medzi jednotlivými komponentami, rozhodli sme sa engine postaviť na centralizovanom prístupe. Engine pozostáva z tried tzv. manažérov, ktoré sú zastrešené triedou hlavného manažéru. Táto trieda reprezentuje samotný engine, pričom všetka komunikácia užívateľa s enginom je vykonávaná prostredníctvom hlavného manažéru. Ten obsahuje manažér dát a manažér scén, ktoré sú zodpovedné za jednotlivé funkcie enginu. Obrázok 4.1 ilustruje organizáciu pluginov a scén v rámci manažérov.

Manažér dát zodpovedá za načítavanie dátových súborov, čiže obhospodaruje jednotlivé dostupné loader pluginy. Tieto sú pri vytvorení manažéru automaticky nájdené a používajú sa na prechádzanie súborového systému. Každý plugin je schopný identifikovať súbory jemu známeho typu a vracať ich identifikátory enginu. Manažér ďalej obsahuje repozitár pre každý interný formát dát, v ktorom sú nejaké dáta načítané. Manažér dát udržiava počítadlá referencií a na ich základe podľa potreby číta dátové súbory z disku



Obrázok 4.1: Organizácia pluginov a scén v engine

alebo dáta uvoľňuje. Týmto spôsobom je zabezpečené, že konkrétne dáta v konkrétnom formáte sú uložené v engine iba raz.

Manažér scén udržiava scény, ich elementy, priradené renderovacie pluginy ako aj identifikátory používaných dát. Pri zmene dát komunikuje s dátovým manažérom, ktorý uvoľní staré dáta (ak sa nepoužívajú žiadnou inou scénou) a načíta nové (ak už nie sú). Manažér scén ďalej umožňuje prístup k elementom scény, ktoré chráni proti prístupu z viacerých vlákien a umožňuje vykonávať renderovacie operácie na scéne, či už synchrónne alebo asynchrónne. Každá scéna má priradený jedinečný identifikátor, ktorý sa používa pri všetkých volaniach manažéru. Užívateľ nemá možnosť pristupovať k samotnej scéne, ale iba k elementom scény, pomocou volaní má možnosť meniť niektoré nastavenia scény a vykonávať na scéne operácie. Toto riešenie bolo zvolené z nutnosti ochrániť štruktúry scény pred súčasným prístupom z viacerých vlákien.

Samotná trieda hlavného manažéru integruje manažér dát aj scén. Umožňuje prístup k manažéru scén a poskytuje rozhranie pre časť funkcionality manažéru dát, ktorý je v engine privátny. Hlavný manažér ďalej slúži na zisťovanie dostupnosti renderovacích pluginov.

4.2 Programovací jazyk

Pri výbere programovacieho jazyku sme zvažovali podporovaný objektový model a všeobecnú výkonnosť programov napísaných v tomto programovacom jazyku. Implementáciu sme chceli založiť na objektovo orientovanom prístupe a teda sme zvažovali jazyky podporujúce objekty. Z tých sme najprv zamietli jazyky interpretované, resp. kompilované na požiadanie, akými sú JAVA alebo jazyky pre prostredie .NET. Aj keď tieto poskytujú vyspelý objektový model a množstvo rôznej funkcionality, výsledné programy vykazujú relevantné spomalenie oproti jazykom ako C++, či Pascal/Delphi.

Ďalej sme sa rozhodovali medzi už spomenutými jazykmi C++ a Delphi. Na rozdiel od prostredí JAVA a .NET, tieto jazyky sú kompilované priamo do strojového kódu a teda výsledné aplikácie nie je možné spustiť na ľubovoľnom systéme. Jazyky sú však dostatočne rozšírené a na všetkých relevantných operačných systémoch existujú pre ne kompilátory. Aj keď Delphi poskytuje modernejší objektový model, rozhodli sme sa pre C++ hlavne z dôvodu jeho rozšírenosti a z dôvodu existencie veľkého množstva rôznych knižníc napísaných v tomto jazyku, ktoré tak bude možno bez úprav použiť v engine alebo jeho pluginoch. Navyše aplikácie v jazyku C++ vo všeobecnosti vykazujú oproti aplikáciam v jazyku Delphi vyšší výkon.

4.3 Použité knižnice

Pri implementácii enginu sme využili niektoré ďalšie knižnice, ktoré tak musia byť dostupné pri kompilovaní. Na výslednú architektúru vplýva aj zvolená knižnica pre prácu s grafickými kartami, preto bolo treba už na začiatku niektorú vybrať. Najbežnejšie sú knižnice OpenGL[1][21] a DirectX[2][1]. Engine sme navrhovali pre prácu s knižnicou OpenGL, keďže je dostupná na veľkom množstve platforiem. Knižnica DirectX je obmedzená iba na použitie v operačnom systéme Windows.

Pri výbere knižnice pre vlákna a synchronizačné objekty sme sa snažili minimalizovať nutnosť vetviť kód podľa použitého operačného systému. Keďže POSIX vlákna (Pthreads), dostupné na všetkých POSIX systémoch (Linux, UNIX) sú dostupné aj v prostredí Windows[3], vybrali sme ich pre použitie v engine.

Na komunikáciu po sieti sme vybrali sockety, keďže umožňujú posielať dáta medzi procesmi na lokálnom počítači ako aj po sieti. Navyše ich použitie nevyžaduje zložitú implementáciu. Používame sockety s aktívnym spojením, komunikujúce pomocou TCP/IP, ktoré zabezpečuje bezchybný prenos dát. Samotné posielané dáta sú z dôvodu ľahkej rozšíriteľnosti kódované pomocou XML, v prípade binárnych dát (vyrenderované obrázky) dáta posielame skomprimované. Na prácu s XML kódom využívame knižnicu TinyXML[4] a na kompresiu knižnicu bzip2[5]. Obe knižnice sú voľne dostupné. Na narábanie so súbormi, reťazcami a dynamickými poľami využívame objekty štandardných C++ knižníc.

4.4 Operácie enginu

Užívateľ pracuje s enginom prostredníctvom inštancie triedy hlavného manažéru. Tá okrem prístupu k manažéru scén poskytuje aj časť funkcionality manažéru dát. Vo fáze inicializácie hlavného manažéru dôjde k vytvoreniu manažéru scén a dát. Manažér dát vytvorí interný OpenGL kontext, ktorý bude tvoriť globálny priestor identifikátorov textúr. Aby mohli byť dáta vo forme textúr zdieľané medzi scénami, každý ďalší kontext, ktorý k nim potrebuje pristupovať bude zdieľať tento priestor identifikátorov. Pri vytvá-

[1] http://www.opengl.org/
[2] http://msdn.microsoft.com/directX
[3] http://sourceware.org/pthreads-win32/
[4] http://www.grinninglizard.com/tinyxml/
[5] http://www.bzip.org/

raní manažéru dát ešte dôjde k automatickej lokalizácií všetkých dostupných loader pluginov a k ich načítaniu.

Hlavný manažér poskytuje rozhranie pre identifikáciu dostupných renderovacích pluginov. Vrátený je zoznam identifikátorov, ktoré je možné použiť neskôr pri priraďovaní pluginov scéne. Okrem toho manažér umožňuje prechádzať súborový systém. Okrem adresárov identifikuje za pomoci dostupných loader pluginov dátové súbory známych typov. Pri lokálnej verzii enginu táto funkcionalita umožňuje filtrovať z dostupných súborov len dátové, pri sieťovej verzii je nevyhnutná pre prechádzanie súborového systému serveru.

Manažér scén poskytuje funkcie pre pridanie a zrušenie scény. Pri vytvorení novej scény je táto uložená v rámci manažéru a užívateľovi je vrátený unikátny identifikátor scény. Tento identifikátor potom jednoznačne určuje scénu, na ktorej sa majú vykonávať operácie. Keďže k scénam môže naraz pristupovať viacero vlákien, je nevyhnutné ochrániť interné štruktúry scény. Na to slúži dvojica zamykacích volaní, ktoré volajúcemu vláknu vytvoria zámok na scéne pre čítanie alebo zapisovanie. V jednom momente môže elementy scény čítať viacero vlákien, avšak pri zapisovaní nemôže k scéne pristupovať žiadne iné vlákno. Po skončení práce s elementami scény je nutné túto odomknúť a tak umožniť ďalším vláknam získať zámok.

Priradenie renderovacieho pluginu a dátového objemu sa vykonáva pomocou príslušných volaní. Pri zmene renderovacieho pluginu dôjde k uvoľneniu dát, vytvoreniu inštancie nového pluginu a k opätovnému načítaniu dát v správnom formáte. Dátový objem je možné taktiež meniť, pričom interný formát dát je automaticky zistený z priradeného renderovacieho pluginu. Manažér dát v skutočnosti nevykonáva zakaždým načítanie súboru z disku, ale počíta počet referencií na dáta. Ak sa požadované dáta nenachádzajú v príslušnom repozitári, dôjde k ich načítaniu zo súboru. Každá ďalšia požiadavka na načítanie toho istého súboru v tom istom formáte však už iba vráti referenciu na uložené dáta. Skutočné uvoľňovanie dát sa opäť deje až keď počítadlo referencií klesne na 0. Priraďovanie dátového objemu scéne je implementované ako synchrónna aj asynchrónna operácia. V prípade synchrónnych operácií volajúce vlákno čaká, kým sa vykonávanie neskončí. Asynchrónne volania iba spustia operáciu, ktoré je vykonávaná separátnym vláknom a okamžite sa vrátia užívateľovi. Ten je informovaný o skončení asynchrónnej operácie pomocou funkcie spätného volania.

Samotné renderovanie scény je možné vykonávať buď "do pamäte", čiže do pamäťového buffra alebo do užívateľom špecifikovaného OpenGL kontextu. Renderovacie pluginy musia implementovať aspoň jednu z týchto možností. V prípade, že užívateľ vyžaduje renderovaciu operáciu, ktorá nie je v plugine implementovaná, engine sa automaticky postará o konverziu. V

prípade renderovania do pamäte sa jedná o extrakciu dát z OpenGL kontextu, v prípade renderovania pomocou OpenGL dôjde k vytvoreniu dočasnej textúry z pamäťových dát a k vyrenderovaniu otexturovaného obdĺžnika pokrývajúceho celý rozsah kontextu. Renderovacie operácie sú implementované ako synchrónne aj asynchrónne.

Operácie modifikujúce scénu (priradenie dátového súboru, nastavenie renderovacieho pluginu, ...) majú exkluzívny prístup k scéne a teda neumožňujú súčasný beh žiadnej inej operácie ani žiaden ďalší aktívny zámok na scéne. Medzi zvyšné operácie patrí renderovanie, ktorému stačí k scéne pristupovať iba na čítanie. Pri zámku na čítanie je síce umožnené iným vláknam čítať scénu, ale súčasný beh viacerých procesov renderovania nemá význam, keďže by nutne poskytli rovnaký výsledok. Pri renderovaní pomocou OpenGL navyše vzniká problém vlastníctva OpenGL kontextu. Z tohto dôvodu sme umožnili naraz prevádzať len jednu operáciu na scéne, či už synchrónnu, alebo asynchrónnu.

4.4.1 OpenGL

Ako sme už spomenuli, engine umožňuje renderovať do pamäte alebo do OpenGL kontextu. Engine takisto automaticky vráti vyrenderované dáta žiadaným spôsobom, aj keď renderer toto neumožňuje. To znamená, že musí existovať interný OpenGL kontext, do ktorého sa renderuje pluginom, ktorý nevie renderovať priamo do pamäte. Keďže na rôznych scénach môže prebiehať renderovanie súčasne, tento interný kontext musí existovať pre každú scénu zvlášť. Okrem renderovania sa využíva aj pri nastavení nového pluginu, či aplikovaní zmien na elementoch scény, keďže renderovací plugin môže udržiavať štruktúry na grafickej karte. V prípade interného aj užívateľského kontextu vzniká problém s vlastníctvom. Ak by sme vyžadovali, aby sa príslušný kontext nastavil ako aktuálny vždy na začiatku operácie a na konci sa toto nastavenie opäť zrušilo, výrazne by sme ovplyvnili výkon enginu. Funkcie systému ako wglMakeCurrent a glXMakeCurrent[6] totiž vyžadujú čas, ktorý sa stáva signifikantným pri opakovanom renderovaní niekoľko krát za sekundu. Z tohto dôvodu je žiaduce obmedziť množstvo týchto operácií na minimum. Keďže všetky volania pracujúce buď s jedným, alebo s druhým kontextom sú buď výlučné alebo je umožnené aj tak bežať iba jednej, vytvorili sme pre každú scénu dve špeciálne vlákna. Každé vlastní príslušný kontext a vykonáva všetky operácie na ňom prebiehajúce. V prípade, že by užívateľ chcel sám používať kontext, ktorý poskytol enginu, toto spojenie je možné zrušiť a kontext použiť vrámci klientskej aplikácie. V takomto prípade

[6] Špecifikácie wgl a glX sú dostupné na stránke http://www.opengl.org/

nie je umožnené renderovať do OpenGL kontextu asynchrónne a synchrónne len za použitia aktívneho kontextu (engine predpokladá, že bol nastavený aktívny kontext). Neskôr je možné kontext priradiť späť enginu.

4.4.2 Pluginy

V prípade pluginov potrebujeme dosiahnuť, aby jednotlivé moduly boli zameniteľné a v budúcnosti rozširovateľné. To znamená, že potrebujeme univerzálne rozhranie, pomocou ktorého bude engine komunikovať s pluginom. Keďže používame objektový prístup, pluginy sú reprezentované triedou implementujúcou príslušné rozhranie. V jazyku C++ je to dosiahnuté dedičnosťou a polymorfizmom. Pluginy sú distribuované ako dynamicky linkovaná knižnica, v ktorej sa nachádza funkcia vytvárajúca inštanciu pluginu. Engine po načítaní knižnice lokalizuje túto funkciu a následne pomocou nej vytvorí inštanciu triedy príslušného pluginu.

4.5 Klient/server

V prípade verzie klient/server sa používajú namiesto manažérov zástupné (proxy) triedy, ktoré implementujú rovnaké rozhranie, avšak väčšinu volaní preposielajú po sieti serveru. Z pohľadu užívateľa sa riadne a zástupné triedy líšia len v inicializácií a čiastočne v synchronizácií.

4.5.1 Klient

V prípade sieťovej verzie nedochádza k vytváraniu manažéru dát, keďže renderovanie sa vykonáva na serveri. Dôjde však k pokusu o nadviazanie spojenia so serverom, ktoré musí byť úspešné. V prípade, že spojenie nemôže byť nadviazané, vytvorenie klientskej strany enginu zlyhá. Po úspešnom vytvorení je vzhľadom na identické rozhranie možné používať zástupné triedy rovnako, ako ich lokálne varianty. Vo všeobecnosti zástupné triedy zakódujú každú požiadavku do príslušného XML kódu a tento pošlú po sieti serveru. V prípade synchrónnych operácií sa čaká na odpoveď. Trieda implementujúca spojenie socketmi používa separátne načúvacie vlákno, ktoré čaká na správy od serveru. Tieto sú potom distribuované čakajúcim synchrónnym operáciám, alebo sú vykonané separátnym vláknom v prípade správ o ukončení asynchrónnej operácie. V prípade zlyhania spojenia dôjde k vzniku výnimky. Kvôli obmedzeniu množstva posielaných dát, scéna v zástupnej triede obsahuje lokálne kópie jej elementov. Tieto je možné upravovať bez nutnosti

posielania dát po sieti. Zmenené elementy sú zaslané až pri volaní, ktoré aplikuje zmeny elementov na interné štruktúry rendereru.

4.5.2 Server

Serverová aplikácia je tvorená objektom manažéru serverov, ktorý udržiava objekt serveru pre každé aktívne spojenie. Manažér serverov vlastní inštanciu hlavného manažéru, na ktorom sú vykonávané všetky požiadavky. Samotný manažér čaká na príchodzie spojenia, pre ktoré potom vytvára separátny objekt serveru. Tento obsahuje vytvorené spojenie a referenciu na globálny hlavný manažér. Server čaká na príchodzie požiadavky a tieto vykonáva. Požiadavky sú vykonávané simultánne viacerými vláknami. V prípade ukončenia spojenia dôjde k zrušenie príslušného serveru a k uvoľneniu jemu patriacich scén.

4.6 Analýza režijných nákladov

V súvislosti so zabezpečením odolnosti voči súčasnému prístupu viacerých vlákien bolo nutné využiť niektoré systémové synchronizačné objekty. Jednalo sa o vzájomné vylúčenia (mutual exclusion - mutex), podmienkové premenné (condition variable), semafory a zámok na čítanie/zapisovanie. Použitie týchto objektov prinieslo určité režijné náklady, ktoré sa pokúsime analyzovať. Analýza má význam iba pri časovo kritických operáciach.

- Každé volanie operujúce so scénou musí najprv podľa identifikátora scény vyhľadať v poli ukazateľ na inštanciu scény. Použité je binárne vyhľadávanie, pričom počas vyhľadávania je uzamknutý mutex chrániaci prístup k poľu. V prípade malého počtu scén (pri lokálnej verzii bežne 1) sa úkony v podstate redukujú na zamknutie a odomknutie tohoto mutexu.

- Pri editácií elementov scény je na scéne uzamknutý zámok na zápis. V prípade, že žiadne iné vlákno nevlastní zámok na scéne, volanie nespotrebuje žiaden ďalší čas čakaním. Po ukončení editácie elementov scény je nutné túto odomknúť.

- Pri synchrónnom renderovaní je vo všeobecnosti požiadavka predaná vláknu, ktoré vlastní buď užívateľský alebo interný kontext. Toto vlákno uzamkne scénu na čítanie a podľa požiadavky a schopnosti renderovacieho pluginu renderuje scénu do pamäte alebo do OpenGL kontextu. Ak treba, je obrázok extrahovaný z interného kontextu alebo

naopak vykreslený do užívateľského kontextu a volajúce vlákno ukončí volanie. Na zabezpečenie synchronizácie volajúceho a vykonávacieho vlákna sa využívajú semafory. V prípade, že užívateľ použije variantu renderovania do aktívneho kontextu, renderovanie sa uskutoční v rámci volajúceho vlákna.

- Asynchrónne renderovanie podobne ako v synchrónnom prípade odovzdá požiadavku internému vláknu a ukončí volanie. Po dokončení operácie je vytvorené separátne vlákno, z ktorého sa zavolá funkcia spätného volania.

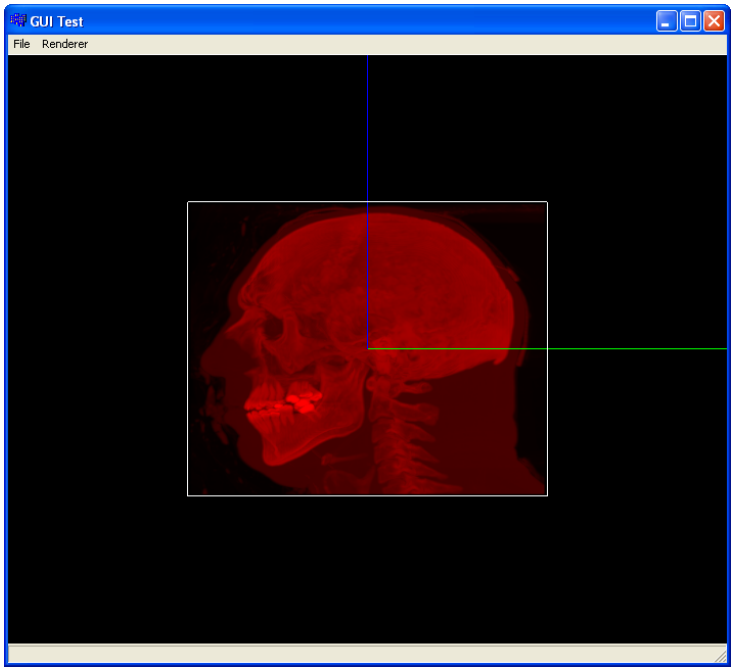
Ako vidieť, operácie vyžadujú konštantný počet synchronizačných úkonov a v asynchrónnom prípade aj vytvorenie nového vlákna. Dosiahnuteľné časy renderovania a spomalenie asynchrónnych operácií oproti synchrónnym je uvedené v tabuľkách v sekcii 5.

V prípade serverovej verzie je väčšina režijných nákladov spojená s posielaním správ a výsledkov po sieti. Volania sú vo všeobecnosti zakódované do XML kódu, ktorý je následne zaslaný serveru. Server správu dekóduje a vykoná. Výsledok opäť zakóduje a pošle späť klientovi. Vyrenderované obrázky sú posielané skomprimované algoritmom bzip2. Klient po prijatí správy dekóduje výsledok a ukončí volanie. Na synchronizáciu sa využívajú semafory. V prípade asynchrónnych volaní je serverom vrátené potvrdenie o prijatí operácie (ak nenastala chyba pri spúšťaní asynchrónnej operácie), po ktorého prijatí je užívateľské volanie ukončené. Po skončení operácie je zaslaný výsledok, ktorý je na klientskej strane spracovaný separátnym vláknom. Posielanie dát vyžaduje použitie istého počtu synchronizačných volaní a samotné posielanie správ ako aj ich príjem je spracovávaný sekvenčne.

4.7 Implementované aplikácie a pluginy

Okrem implementácie samotnej knižnice enginu sme implementovali aj niekoľko testovacích pluginov. Vytvorený bol loader plugin pre tzv. raw dáta, čiže dáta uložené v nekomprimovanej forme voxel po voxli. Ďalej sme implementovali dva builder pluginy, ktoré udržiavali dáta v hlavnej pamäti a v pamäti na grafickej karte ako 3D textúru. Na otestovanie priameho renderingu do pamäte sme vytvorili renderovací plugin implementujúci projekciu maximálnej intenzity. Renderovanie za pomoci OpenGL bolo testované pluginom implementujúcim rovnakú metódu za použitia 3D textúr.

Vytvorených bolo aj niekoľko klientských aplikácií, ktoré testovali použitie enginu z konzolovej aplikácie ako aj v jednoduchom grafickom užívateľskom rozhraní (obrázok 4.2). Pre výkonnostné testy spomínané v nasledujú-



Obrázok 4.2: Testovacie grafické užívateľské rozhranie

cej kapitole sme vytvorili špeciálnu klientskú aplikáciu a prázdny renderovací plugin. Všetky aplikácie boli použité aj pri testovaní vzdialenej vizualizácie.

Kapitola 5

Výsledky

Keďže engine sám nevykonáva žiadne renderovanie, za toto sú zodpovedné renderovacie pluginy, nemá význam merať časy renderovania nejakého objemu nejakým algoritmom. Réžia prítomná pri použití enginu, vzniká používaním vlákien a objektov slúžiacich na zachovanie integrity dát. Z tohto dôvodu má význam merať čas potrebný na vykonanie jedného volania renderovania, avšak bez toho, aby sa niečo zmysluplné renderovalo. Naše testy teda pozostávali z merania času potrebného na vykonanie dostupných štyroch druhov renderovania. Jedná sa o renderovanie do pamäte a renderovanie do OpenGL kontextu a to synchrónne aj asynchrónne. Keďže pri vzdialenej vizualizácií je výkon ovplyvnený hlavne rozmermi posielaného obrázku, vykonali sme skutočné renderovanie objemu a porovnali dosiahnuté časy s výsledkami staticky linkovanej verzie enginu.

5.1 Metodika merania

5.1.1 Použitý renderer

Vytvorili sme renderer, ktorý implementoval oba spôsoby renderovania. Pri renderovaní do pamäte bol alokovaný blok pamäte, kam by sa pri renderovaní ukladali výsledky a z tohto bloku bol vytvorený objekt, ktorý bol funkciou vrátený. Renderovanie do OpenGL kontextu najprv nastavilo aktuálnu farbu na náhodnú a následne vykreslilo jeden štvorec rozmerov okna. Týmto sa v oboch prípadoch simulovali úkony nutne potrebné na renderovanie, avšak samotné renderovanie sa neuskutočnilo.

Pre skutočné renderovanie sme použili OpenGL renderer vykonávajúci maximálnu intenzitovú projekciu. Plugin používa 3D textúry a ako proxy

geometriu objektovo orientované rezy. V prípade renderovania do pamäte dochádza k extrakcií vyrenderovaných dát z OpenGL kontextu.

5.1.2 Meranie času

Pri meraniach sme postupne vykonali oba druhy synchrónnych a potom oba druhy asynchrónnych operácií. Použitých bolo zakaždým 10000 operácií s výnimkou vzdialenej vizualizácie, kedy sme použili 100 operácií. Pri synchrónnych operáciach tieto boli vykonávané v cykle, pričom tesne pred a okamžite po každej operácií bol zistený časový údaj, ktorých rozdielom sme dostali čas potrebný na vykonanie operácie. Okrem toho sme merali aj úhrnný čas pre jednu sadu operácií, ktorý teda zahŕňal aj náklady spojené s behom cyklu a príležitostným vypisovaním priebehu na štandardný výstup. Pri asynchrónnych operáciach bolo meranie vykonávané z funkcie spätného volania. Po skončení asynchrónnej operácie bola zavolaná táto funkcia, v ktorej sa zaznamenal čas ukončenia operácie a vypočítal sa čas potrebný na jedno renderovanie. Následne došlo k zaznamenaniu času spustenia operácie a k iniciovaniu ďalšieho asynchrónneho volania. Asynchrónne volania sa teda v skutočnosti vykonávali synchrónne, takže časy neboli ovplyvnené súčasným behom viacerých operácií.

Na meranie času sme použili systémové funkcie `GetTickCount` pre Windows a `gettimeofday` pre Linux. Prvá vracia počet milisekúnd od naštartovania operačného systému, druhá vracia aktuálny čas v mikrosekundách. Skutočná presnosť oboch funkcií je samozrejme nižšia ako použitá jednotka, preto sme počítali priemerný čas na vykonanie operácie aj z úhrnného času. Časy sme zaznamenávali v milisekundách a po skončení renderovania sme vypočítali priemerný čas potrebný na vykonanie operácie ako aj štandardnú odchýlku. Následne kvôli vypovedacej hodnote sme získaný časový údaj prepočítali do obrázkov za sekundu (FPS). Nameraný úhrnný čas sa od sumy časov jednotlivých volaní líšil minimálne.

5.1.3 Vykonávanie funkcií

Pri renderovaní do pamäte sa v oboch prípadoch samotná operácia vykonávala z vlákna scény, ktoré vlastní interný kontext. V prípade, že by renderovací plugin neumožňoval renderovať do pamäte priamo, by totiž bolo treba použiť OpenGL kontext scény. Pri synchrónnom renderovaní sa čakalo na ukončenie operácie. Pri renderovaní do OpenGL sa v prípade synchrónneho variantu testovala verzia operácie vykonávaná na aktívnom kontexte, takže renderovanie prebiehalo z volajúceho vlákna. Kontext bol nastavený ako aktívny z

Procesor	AMD Athlon XP 2600+ (2088 MHz)
Pamäť	1 GB
Grafická karta	ATI Radeon X1650 PRO AGP8x
Grafická pamäť	256 MB/DDR3 (700MHz)
Operačný systém 1	Microsoft Windows XP SP2
Operačný systém 2	openSuSE 10.3
Verzia ovládačov grafickej karty	8.2 (Windows), 8.45.4 (Linux)

Tabuľka 5.1: Zostava 1 - ATI

Procesor	Intel Pentium 4 (2806 MHz)
Pamäť	1 GB
Grafická karta	NVidia GeForce 6800 AGP8x
Grafická pamäť	256 MB
Operačný systém 1	Microsoft Windows XP SP2
Verzia ovládačov grafickej karty	163.71

Tabuľka 5.2: Zostava 2 - NVidia

testovacej aplikácie pred začatím bloku renderovania. Tento spôsob poskytuje o niečo vyšší výkon, keďže netreba používať synchronizačné mechanizmy na zistenie ukončenia renderovania vláknom scény. Pri asynchrónnej verzii však renderovanie už muselo byt vykonávané z vlákna enginu.

5.1.4 Použité počítače

Merania sme vykonali na počítačoch s procesormi a grafickými kartami od všetkých relevantných výrobcov. Na zostave s procesorom AMD a grafickou kartou ATI/AMD sme porovnali výkon enginu aj medzi operačnými systémami Windows a Linux. Zvyšné merania prebehli pod operačným systémom Windows na počítači s procesorom Intel a grafickou kartou NVidia a na notebooku Intel s integrovaným grafickým riešením. Podrobnejšie informácie o použitých testovacích zostavách sa nachádzajú v tabuľkách 5.1, 5.2 a 5.3.

5.2 Analýza výsledkov

Tabuľka 5.4 uvádza namerané hodnoty obrázkov za sekundu pri statickom linkovaní enginu s testovacou aplikáciou. V tabuľke 5.5 sa nachádzajú namerané hodnoty pre sieťovú verziu enginu. Pre porovnanie sú uvedené časy

Procesor	Intel Pentium M (1.6 GHz)
Pamäť	512 MB
Grafická karta	Intel 915GM
Grafická pamäť	8 MB
Operačný systém 1	Microsoft Windows XP SP2
Verzia ovládačov grafickej karty	6.14.10.4020

Tabuľka 5.3: Zostava 3 - Intel

Meranie/zostava		ATI (W)	ATI (L)	NVidia	Intel
Pamäť	synch.	2078	13661	1941	2571
	asynch.	846	8149	926	1706
OpenGL	synch.	3107	2989	3558	815
	asynch.	989	986	1102	808

Tabuľka 5.4: Počet obrázkov za sekundu pri prázdnom renderovaní

renderovania pri statickom linkovaní. Časy sú rozdelené podľa rozmerov posielaného obrázku. Pri všetkých meraniach vidieť, že režijné náklady pri asynchrónnom renderovaní do pamäte sú približne dvojnásobné oproti synchrónnom variante. Vzhľadom na minimálny rozdiel vo vykonávaných operáciach sa dá vzniknutý rozdiel pripísať na vrub vytvárania separátneho vlákna pre spätné volanie z asynchrónneho renderovania. Rozdiely v počte dosiahnutých operácií za sekundu na úrovni rádov pri rozdielnych operačných systémoch sú vysvetliteľné efektívnejšou prácou s pamäťou v Linuxe alebo efektívnejšími operáciami na synchronizačných objektoch.

Meranie/rozmery obrázku		256x256			512x512		
konfigurácia		A	B	C	A	B	C
Pamäť	synch.	24	10	3	24	1.7	1.4
	asynch.	24	9	4	22	1.6	1.2
OpenGL	synch.	49	10	4	37	1.5	1.2
	asynch.	47	9	3	36	1.5	1.1

Tabuľka 5.5: Počet obrázkov za sekundu pri lokálnej a vzdialenej vizualizácií. Konfigurácie: statické linkovanie enginu (A), klient aj server na rovnakom počítači (B), klient a server na rozdielnych počítačoch spojených lokálnou sieťou (C)

Výsledky renderovania za použitia OpenGL sú v podstate rovnaké na oboch operačných systémoch. Integrované riešenie od firmy Intel zaostáva za výkonnejšími konkurenčnými grafickými kartami. Synchrónna varianta renderovania dosahuje oproti asynchrónnej približne trojnásobný výkon. Podobne ako v prípade renderovania do pamäte to bude spôsobené vytváraním vlákna ako aj menšou synchronizačnou réžiou, keďže synchrónne renderovanie pracuje v rámci volajúceho vlákna.

Merania pri sieťovej verzii vykazujú výrazné spomalenie. Toto spomalenie je spôsobené hlavne nutnosťou prenášať výsledný vyrenderovaný obrázok, čo dobre ilustruje pokles výkonu pri zvýšení objemu prenášaných dát. Práca [6] ukazuje, že kompresia spolu s podvzorkovaním dát pri manipulácií s objemom (napr. rotácia) umožňuje dosiahnuť uspokojivý výkon. Po skončení narábania s dátami sa tieto vyrenderujú v plnej kvalite.

Kapitola 6

Záver

V práci sme navrhli architektúru vizualizačného enginu, ktorý poskytuje vysokú rozšíriteľnosť o nové vizualizačné algoritmy alebo implementácie. Engine používa scénový model a poskytuje možnosť použitia viacerých scén. Engine umožňuje vysokú znovupoužiteľnosť kódu použitím pluginov a je vhodný ako vývojový alebo testovací nástroj. Engine sme implementovali aj ako server, umožňujúc tak vzdialenú vizualizáciu využívajúcu vzdialené hardvérové prostriedky.

Literatúra

[1] B. Bargen and P. Donnelly. *Inside DirectX: in-depth techniques for developing high-performance multimedia applications.* Microsoft Press, Redmond, WA, USA, 1998.

[2] J. Bloomenthal. Polygonization of implicit surfaces. *Comput. Aided Geom. Des.*, 5(4):341–355, 1988.

[3] H. N. Christiansen and T. W. Sederberg. Conversion of complex contour line definitions into polygonal element mosaics. *SIGGRAPH Comput. Graph.*, 12(3):187–192, 1978.

[4] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-quality volume rendering using texture mapping hardware. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 69–ff., New York, NY, USA, 1998. ACM.

[5] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, New York, NY, USA, 2001. ACM.

[6] K. Engel, O. Sommer, and T. Ertl. A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In *VisSym*, pages 167–177. Joint Eurographics and IEEE TCVG Symp. on Vis., 2000.

[7] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Commun. ACM*, 20(10):693–702, 1977.

[8] M. Hadwiger, J. M. Kniss, K. Engel, and C. Rezk Salama. High-Quality Volume Graphics on Consumer PC Hardware. In *Siggraph*, 2002.

[9] M. Hadwiger, I. Viola, T. Theul, and H. Hauser. Fast and flexible high-quality texture filtering with tiled high-resolution filters. 2002.

[10] E. Keppel. Approximating complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development*, 19(1), 1975.

[11] T. Klein, M. Weiler, and T. Ertl. A Volume Rendering Extension for the OpenSG Scene Graph API. In *Poster Compendium of IEEE Visualization '03*, pages 30–31. 2003.

[12] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA, 1994. ACM.

[13] M. Levoy. Display of Surfaces From Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[14] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[15] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design Galleries: a General Approach to Setting Parameters for Computer Graphics and Animation. *Computer Graphics*, 31(Annual Conference Series):389–400, 1997.

[16] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 100–107, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[17] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[18] B. K. Natarajan. On generating topologically correct isosurfaces from uniform samples. 1991.

[19] P. Ning and J. Bloomenthal. An evaluation of implicit surface tilers. *IEEE Comput. Graph. Appl.*, 13(6):33–41, 1993.

[20] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab. *Signals & systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[21] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 2.0)*, 2004.

[22] D. Weiskopf, M. Weiler, and T. Ertl. Maintaining constant frame rates in 3d texture-based volume rendering. In *CGI '04: Proceedings of the Computer Graphics International*, pages 604–607, Washington, DC, USA, 2004. IEEE Computer Society.

[23] O. Wilson, A. VanGelder, and J. Wilhelms. DIRECT VOLUME RENDERING VIA 3D TEXTURES. Technical Report UCSC-CRL-94-19, 1994.

[24] M. Šrámek. *Visualization of Volumetric Data by Ray Tracing.* PhD thesis, 1996.

Prílohy

Súčasťou práce je aj CD-ROM disk obsahujúci:

- Text diplomovej práce v elektronickej podobe (pdf)
- Zdrojové súbory enginu
- Súbory objemových dát
- Vzorové a testovacie pluginy a aplikácie