Architektúra enginu pre objemovú vizualizáciu

Michal Hučko

Školiteľ: RNDr. Michal Červeňanský

18. apríla 2008

Abstrakt

Cieľom práce je vytvoriť prostredie vhodné na rýchly vývoj vizualizačných algoritmov alebo ich implementácií s dôrazom na vysokú znovupoužiteľnosť kódu. Prostredie má byť založené na modeli grafu scény a má umožniť použitie s ľubovoľným uživateľským 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

Obsah

Kapitola 1

Úvod

Objemová vizualizácia je časť počítačovej grafiky zaoberajúca sa vizualizáciou objemových dát [13, 4] . Tieto dáta su 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 proramátorského, 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. 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. Použitie majú aj syntetické dáta, ktoré sú produktom nejakých matematických modelov. Na počítačoch bývajú dáta vzorkované v diskrétnych bodoch a kvantizované do konečného počtu úrovní.

1.2 Vizualizácia 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 [13]. 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).

1.2.1 Nepriama vizualizácia

Pri nepriamych metódach dochádza k vytváraniu povrchu z oblastí s rovnakými alebo podobnými vastnosťami. Získaný povrch je následne vykreslený 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) [7].

1.2.2 Priama vizualizácia

Priame metódy pracujú na úrovni voxelov objemu. Keďže namerané hodnory zväčša reprezentujú nejaké fyzikálne vlastnosti či veličiny, na skalárne dáta sú mapované optické vlastnosti akými sú napríklad farba, priehľadnosť, intenzita emitovaného svetla a iné [6]. Toto mapovanie sa najčastejšie vykonáva pomocou prenosových funkcií (transfer functions), čo sú vlastne tabuľky, ktoré fungujú podobne ako paleta. Používaný optický model potom určuje, ako sa tieto vlastnosti prejavia na výslednom obraze [4]. Modely najčastestejšie kombinujú absorbciu, emisiu, odrazy a/alebo rozptyl svetla [9].

Metódy sa dajú vo všeobecnosti rozdeliť do dvoch skupín na základe spôsobu prechádzania dát [13]. Objektovo orientované postupne prechádzajú jednotlivé voxle objemu a počítavajú prírastok daného voxlu k výslednému obrazu. Obrazovo orientované techniky zasa prechádzajú pixle výsledného obrazu a vypočítavajú jeho výslednú farbu. Medzi prvé patria napríklad metódy pracujúce s textúrami na grafických kartách, kde je objem postupne renderovaný po rezoch[12, 2]. Príkladom druhého prístupu je metóda sledovania lúča (ray tracing)[11, 13].

1.3 Vizualizačné riešenia

Aplikácie alebo knižnice implementujúce metódy objemovej vizualizácie vo všeobecnosti používajú dva odlišné architektonické prístupy. Model toku dát pozostáva z primitív - objektov implementujúcich nejakú operáciu na dátach. Tieto objekty maju definovanú vstupno-výstupnú charakteristiku a je možné ich spájať tzv. rúrami (pipes). Výsledná operácia pozostáva z orientovaného acyklického grafu čiastkových operácii. Keďže samotná objemová vizualizácia predstavuje v podstate atomický objekt, tento model má skôr význam pri predspracovaní, či úprave dát. Medzi prostredia používajúce model toku dát patrí napríklad SciRUN[1], MeVisLab[2].

Druhým prístupom je použitie tzv. grafu scény, čize hierarchickej organizácie virtuálnych objektov v priestore. Vizualizácia celej scény potom vzniká kombináciou vizualizácii podstromov scény. Keďže pojem grafu scény je pomerne abstraktný, môže sa jednať o prístup, ktorý iba nejakým spôsobom organizuje elementy scény, ale aj prístup, ktorý transformuje nejaký vizualizačný algoritmus do primitív nejakého prostredia. Príkladom môže byť rozšírenie prostredia OpenSceneGraph[3] o objemovú vizualizáciu pomocou textúr [5].

Bežne dostupné riešenia využívajúce nejakú reprezentáciu virtuálnej scény väčšinou predstavujú monolitické aplikácie. Pri modele toku dát sa dá funkcionalita rozšíriť priamočiaro pridaním nových modulov. V prípade scény nie je jednoznačne jasné, čo sa má dať rozšíriť a ako. Preto aplikácie implementujúce uživateľské rozhranie predstavujú jeden celok, ktorý nie je možné jednoducho rozšíriť, alebo použiť po častiach. Užívateľ síce najčastejšie môže špecifikovať vlastný postup zobrazovania pomocou shadera, má však obmedzené možnosti. Aplikácie najčastejšie implementujú nejakú množinu vizualizačných algoritmov, resp. ich implementácií na hardvéri a túto nie je možné meniť[4]. V prípade knižníc, ktoré neimplementujú uživateľské rozhranie je síce možné použiť vlastné, avšak obmedzenie renderovacích algoritmov ostáva.

[1] http://software.sci.utah.edu/scirun.html

[2] http://www.mevislab.de

[3] http://www.openscenegraph.org

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

1.4 Vzdialená vizualizácia

Sieťová konektivita je dnes, hlavne na lokálnych sieťach, dostatočná na to, aby sa renderovanie vykonávalo vzdialene. V práci [3] bolo ukázané, ako sa dá pomocou prostredia CORBA[5] prerobiť aplikácia lokálneho typu na vzdialenú. Obmedzila sa na aplikácie s uživateľským rozhraním založeným na OpenInventor[6] a Cosmo3D[7] prostrediach. Jednalo sa však naďalej o pôvodnú monolitickú aplikáciu, len renderovanie sa nevykonávalo lokálne. Opačným prístupom je použitie X serveru, keď aplikácia sa nachádza na serveri, ale zobrazovanie sa vykonáva na lokálnom stroji. Výhodou tohto prístupu je možnosť používať aplikácie nachádzajúce sa na vzdialenom počítači, z pohľadu výkonu však nemá veľké využitie.

[5] Common Object Request Broker Architecture, http://www.corba.org/
[6] http://oss.sgi.com/projects/inventor/
[7] Cosmo3D už nie je vyvýjaný

Kapitola 2

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 4.

2.1 Požiadavky

Jednou zo základných požiadaviek kladených na engine je umožniť jednoduchú rozšíriteľnosť o nové vizualizačné postupy a ďaľšou umožniť jeho použitie s ľubovoľným uživateľským rozhraním. Engine má byť postavený na scénovom prístupe, čiže vytvárať model virtuálneho priestoru obsahujúceho dáta a ďaľšie objekty.

Proces vizualizácie možno popísať nasledovnou postupnosťou krokov:

1. Výber renderovacieho algoritmu
2. Načítanie dát zo súboru
3. Nastavenie parametrov scény (kamera, prenosová funkcia, orezávacia geometria, ...)
4. Renderovanie dát

Formát, v ktorom budú dáta interne uložené je závislý od zvoleného renderovacieho algoritmu a reflektuje spôsob, akým algoritmus s dátami narába. V prípade interaktívnej aj dávkovej vizualizácie sa zväčša kroky 3 a 4 opakujú.

Požiadavka na rozšíriteľnosť o nové algoritmy implikuje vyňatie samotného algoritmu z implementácie enginu a vytvorenie samostatného modulu. Takéto moduly potom môžu vznikať nezávisle na engine. Aby mohlo dôjsť ku spolupráci modulu s enginom, musia oba používať vopred dohodnuté rozhranie, ktoré musí byť dostatočne flexibilné, aby funkčne nelimitovalo budúce renderovacie moduly.

V prípade práce s dátami treba vyriešiť dva problémy. Jedným je množstvo podporovaných dátových formátov a druhým podporované interné formáty dát. Interné formáty zodpovedajú použitým renderovacím algoritmom a potrebná implementácia sa može vyskytovať ako súčasť renderovacieho pluginu. V prípade oddelenia možno oba problémy vyriešiť zvolením fixnej množiny podporovaných formátov, čo však môže pôsobiť reštriktívne na renderovacie moduly. Z pohľadu rozšíriteľnosti sa javí výhodnejšie riešenie s použitím samostatných modulov, minimálne pre súborové formáty dát. V oboch prípadoch treba definovať rozhranie, cez ktoré budú moduly a engine navzájom komunikovať. To by malo umožniť prácu s čo najväčšou množinou dát s ohľadom na rozličné atribúty dát produkovaných rôznymi zariadeniami (bitová hĺbka, farebná hĺbka, číselný typ, ...).

2.2 Navrhnutá architektúra

Pri návrhu architektúry sme okrem zapracovania vyššie uvedených požiadaviek, sledovali aj zvýšenie používateľského komfortu a poskytnutie rozšírenej sady funkcií. Analýza režijných nákladov s tým spojených sa nachádza v kapitole 4. 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][10] 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.

2.2.1 Scéna

Kvôli možnosti použitia ľubovoľného vizualizačného algoritmu, resp. jeho implementácie je nutné scénu chápať iba ako kolekciu nastavení. Rôzne pluginy môžu vyžadovať rôzne formáty týchto parametrov vzhľadom na rozdielne spôsoby spracovania dát, a tak neexistuje univerzálna reprezentácia scény, ktorá by vyhovovala všetkým pluginom. Z tohto dôvodu sme zvolili prístup,

[1] http://www.opengl.org/
[2] http://msdn.microsoft.com/directX

kde jednotlivé elementy scény ako napríklad kamera, svetelné zdroje, či orezávacia geometria sú reprezentované samostatnými objektami. Aplikovanie nastavení reprezentovaných týmito objektami leží výhradne na renderovacom plugine, ktorý zodpovedá za svoju internú reprezentáciu týchto parametrov.

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

Na umožnenie špecifikovania ďalších hodnôt, ktoré môžu byť nevyhnutné pre prácu vizualizačného algoritmu, sme vytvorili 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 umožnujú ich správnu interpretáciu používateľom. Dodatočné parametre sú vytvárané renderovacím pluginom pri inicializácií. Keďže ich triedy sú vopred známe, uživateľské rozhranie má možnosť sprostredkovať uživateľovi tieto špecifické parametre algoritmu. Ďalšou možnosťou by bolo umožniť renderovacím pluginom vytvárať aj inštancie vlastných tried, pričom tieto by bolo možné použiť len s uživateľským rozhraním poznajúcim tieto triedy. Túto možnosť sme ponechali pre prípadné ďalšie verzie enginu.

2.2.2 Centralizácia

Používanie externých modulov (pluginov) vyžaduje určitú réžiu. V prípade, že funkcie alebo trieda reprezentujúca nejaký konkrétny plugin nie sú známe v dobe kompilácie, je nutné ich manuálne načítať z externého súboru. Tento súbor v závislosti na použitom programovacom prostredí obsahuje buď interpretovaný alebo preložený kód. Aby sme odľahčili uživateľa od nutnosti načítavať tento kód, rozhodli sme sa zahrnúť túto funkcionalitu do enginu. Po zvolení súboru je potom engine sám schopný načítať súbor a pracovať s kódom v ňom obsiahnutom.

Okrem manipulácie s modulmi, je nutné zabezpečiť aj vytvorenie scény,

jej elementov, priradenie objemových dát scéne a asociáciu s renderovacím pluginom. Na uľahčenie repetitívnych operácií súvisiacich s vytváraním potrebných objektov a ich previazaním sme sa rozhodli tieto úkony centralizovať v rámci enginu. Engine je tak reprezentovaný jednou hlavnou triedou, ktorá interne zabezpečuje všetky potrebné úkony spojené s inicializáciou objektov, pričom uživateľ s ňou komunikuje pomocu jej rozhrania. V engine sme nazvali túto triedu hlavným manažérom (main manager). Okrem hlavného manažéru existijú v engine ďaľšie dva manažéry a to manažér scén (scene manager) a manažér dát (data manager). Manažér scén je zodpovedný za správu scén a ich elementov, komunikáciu s vybraným renderovacím pluginom a umožňuje vykonávať renderovacie operácie. Manažér dát je zodpovený za správu načítaných dátových súborov. Inštancia hlavného manažéru predstavuje inštanciu enginu a umožňuje uživateľovi s ním pracovať. Podrobnejší popis manažérov sa nachádza v sekcii 2.2.5.

2.2.3 Serverový prístup

Bežne sa vo vizualizačných aplikáciach poskytujúcich grafické uživateľské rozhranie pracuje naraz iba s jednou scénou. Pri vývoji nových algoritmov však môže byť užitočné paralelne porovnávať výsledky vyvíjaného pluginu s nejakým existujúcim. Takisto môže mať význam zobrazovať objem zároveň z rôznych uhlov, či pracovať s rôznymi prenosovými funkciami zároveň (napríklad pri tzv. design gallery [8]). Keďže doteraz navrhnuté riešenie centralizuje správu scény, dát aj renderovacích pluginov v rámci enginu, možné sú dva spôsoby použitia - 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 manažéru podporujúceho viacero scén, sa užívateľ nemusí zaoberať zabezpečením komunikačného spojenia medzi viacerými inštanciami manažéru pri modeli podporujúcom iba jednu scénu.

Podpora viacerých scén ponúka možnosť 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 kapitole 4.

Pri vizualizácii 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 užívateľa vytvárať vlákna vykonávajúce zdĺhavé operácie, rozhodli sme sa do enginu zaradiť podporu asynchrónnych operácií. Takéto operácie 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 takejto operácie. Buď musí sám užívateľ kontrolovať, či operácia už skončila 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.

Operácie modifikujúce scénu (priradenie dátového súboru, aplikácia zmien elementov scény, 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.

2.2.4 Pluginy

V engine používame okrem renderovacích aj ďalšie dva druhy pluginov. Jednak sa jedná o pluginy zodpovedné za čítanie súborových formátov dát (v našom názvosloví loader) a potom 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 uživateľovi ako aj ďalším komponentom enginu a hlavne dekódovať príslušný súbor. 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 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.

2.2.5 Manažéry

Funkcionalita pokrývaná enginom sa dá rozdeliť do viacerých kategórií, ktoré v našom návrhu zastrešujú jednotlivé triedy manažérov. Konkrétne manažér dát zodpovedá za načítavanie dátových súborov, čiže obhospodaruje jednotlivé dostupné loader pluginy. Ďalej je zodpovedný za vytváranie dát v internom formáte a za ich uskladnenie. Manažér dát udržiava počítadlá referencií a na ich základe podľa potreby načítava dátové súbory z disku alebo načítané dáta uvoľnuje. Týmto spôsobom je zabezpečené, že konkrétne dáta v konkrétnom formáte sú načítané 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.

Engine je reprezentovaný triedou hlavného manažéra, ktorý integruje manažér dát aj scén. Umožňuje prístup k manažéru scén a poskutuje rozhranie pre časť funkcionality manažéru dát, ktorý je v engine privátny. Hlavný manažér slúži na zisťovanie dostupnosti renderovacích pluginov, ako aj na prechádzanie súborového systému, pričom súbory sú automaticky filtrované podľa dostupných loader pluginov.

2.2.6 Klient/Server

Navrhnutá architektúra poskytujúca možnosť použitia viacerých scén a prístupu z viecerých vlákien súčasne, umožňuje priamočiare rozšírenie na aplikáciu typu klient/server. Na strane serveru treba zabezpečiť načúvaniu príchodzím sieťovým spojeniam a následne vykonávať požiadavky zasielané po sieti a posielať späť výsledky. Na klienskej 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. Na klientskej strane sme vytvorili zástupné (proxy) triedy manažérov, ktoré implementujú rovnaké rozhranie, ako pôvodné manažéry, ale zabezpečujú iba transformáciu volaní metód na správy zasielané po sieti. Server obsahujúci inštanciu enginu tieto volania vykonáva a výsledky posiela späť.

Kapitola 3

Implementácia

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

3.1 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 pracujúce s .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. Narozdiel 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.

3.2 Použité knižnice

Pri implementácii enginu sme využili niektoré ďaľšie knižnice, ktoré tak musia byť dostupné pri kompilovaní. Na prácu s grafickými kartami sme použili knižnicu OpenGL, ktorá je na rozdiel od DirectX dostupná na väčšine platforiem. 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[1], vybrali sme ich pre použitie v engine. Na komunikáciu po sieti sme vybrali socket-y, 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 ako sú. Na prácu s XML kódom využívame knižnicu TinyXML[2], ktorá je voľne dostupná. Na narábanie so súbormi, reťazcami a dynamickými poľami využívame objekty štandardných C++ knižníc.

3.3 Architektúra

3.3.1 Manažéry

Engine je tvorený hlavnou triedou - hlavným manažérom. Implementované sú dve triedy implementujúce rozhranie hlavného manažéru (IMainManager), TMainManager a TMainManagerProxy. Prvá reprezentuje engine v rámci aplikácie, druhá slúži ako zástupná trieda pri vzdialenej vizualizácií. Rozhranie hlavného manažéru umožňuje prístup k objektu implementujúcemu rozhranie manažéru scén (ISceneManager). Pri lokálnom variante sa jedná o inštanciu triedy TSceneManager, pri vzdialenom o zástupnú triedu TSceneManagerProxy. Užívateľ má prostredníctvom hlavného manažéru prístup iba k manažéru scén, ostatné objekty sú privátne. Ďalej rozoberieme architektúru lokálnej verzie enginu, keďže až na výnimky zástupné triedy všetky požiadavky preposielajú na server.

[1] http://sourceware.org/pthreads-win32/
[2] http://www.grinninglizard.com/tinyxml/

Hlavný manažér poskytuje rozhranie umožňujúce užívateľovi enumerovať dostupné renderovacie pluginy. Keďže manažér dát je privátnym objektom hlavného manažéru, tento sprostredkuváva aj niektoré jeho funkcie. Konkrétne umožňuje enumerovať podadresáre zvoleného adresára a takisto enumerovať dátové súbory nachádzajúce sa v tomto adresári. Pracuje sa len so súbormi typov, rozpoznaných dostupnými loader pluginmi. O týchto súboroch je možné získať aj meta informácie.

Manažér scén obsahuje výlučne funkcie na pridávanie, uberanie a prácu so scénami. Po pridaní novej scény je vrátený jej číselný identifikátor, ktorý sa používa pri všetkých ostatných volaniach. Tieto zahŕňajú volania priraďujúce renderovacie pluginy k scéne, funckie asociujúce dátový súbor so scénou, volania priraďujúce scéne funkcie na spätné volanie ako aj metódy vykonávajúce renderovanie scény. Renderovanie môže byť vykonávané buť do pamäťového buffra, alebo do OpenGL kontextu. V prípade, že renderovací plugin implementuje len jednu z týchto možností, engine automaticky zabezpečí extrakciu dát z kontextu, či vykreslenie pamäťových dát do kontextu.

Okrem synchrónnych, sú niektoré operácie implementované aj ako asynchrónne. Medzi tieto patria renderovacie operácie, operácia priraďujúca scéne dátový súbor a operácia slúžiaca na úpravu interných štruktúr renderovacieho pluginu po zmene elementov scény. Naraz môže byť vykonávaná len jedna operácia, či už synchrónna alebo asynchrónna. Po skončení asynchrónnej operácie je v prípade, že bola nastavená, zavolaná funkcia spätného volania. Enkapsulovali sme ukazateľ na funkciu aj ukazateľ na metódu a tak funkcia spätného volania môže byť oboch typov.

Manažér dát zodpovedá za všetko narábanie s dátovými súbormi ako aj ich obrazmi v internom formáte. Pri vytvorení manažéru dôjde k lokalizácií všetkých dostupných loader pluginov, ktoré sú automaticky načítané. Tieto pluginy sú používané na enumerovanie dostupných súborov známych typov, čítanie meta informácií o súbore a dekódovanie súborov. Manažér dát obsahuje ďalej repozitáre (TDataRepository) pre každý interný formát dát. V prípade, že treba načítať dáta zo súboru do požadovaného interného formátu, dôjde k vytvoreniu príslušného repozitára a k vytvoreniu inštancie builder pluginu. Tá je zodpovedná za vytvorenie interných štruktúr uchovávajúcich načítané dáta. Po načítaní je inštancia zaradená do repozitára tak, aby pri ďaľšej požiadavke na rovnaké dáta v rovnakom formáte bolo možné využiť už načítané dáta.

3.3.2 OpenGL

Ako sme už spomenuli, engine umožňuje renderovať do pamäte alebo do OpenGL kontextu. 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. 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[3] 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 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.

3.3.3 Klient

Pri použití zástupnej triedy hlavného manažéru (TMainManagerProxy), táto v konštruktore iniciuje vytvorenie spojenia so serverom prostredníctvom socketu. V prípade zlyhania dôjde k vzniku výnimky a engine nie je možné použiť. 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 operáciam, 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 kôpie jej elementov. Tieto je možné upravovať bez nutnosti posielania dát po sieti. Zmenené elementy sú zaslané

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

až pri volaní, ktoré aplikuje zmeny elementov na interné štruktúry rendereru.

3.3.4 Server

Serverová aplikácia je tvorená objektom manažéru serverov, ktorý udržiava objekt serveru pre každé aktívne spojenie. Manažér serverov vlastní inštanciu hlavného manažéra, 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 simultálne viacerými vláknami. V prípade ukončenia spojenia dôjde k zrušenie príslušného serveru a k uvolneniu jemu patriacich scén.

3.4 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 uživateľskom rozhraní. Pre výkonnostné testy spomínané v nasledujúcej kapitole sme vytvorili špeciálnu klientskú aplikáciu a prázdny renderovací plugin. Všetky aplikácie boli použité aj pri testovaní vzdialenej vizualizácie.

Kapitola 4

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.

4.1 Metodika merania

4.1.1 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.

4.1.2 Meranie času

Pri meraniach sme postupne vykonali oba druhy synchrónnych a potom oba druhy asynchrónnych operácií. Použitých bolo zakaždým 10000 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 úhrný 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 vyrátal 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 súšasným 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 rátali priemerný čas na vykonanie operácie aj z úhrného času. Časy sme zaznamenávali v milisekundách a po skončení renderovania sme vyrátali 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 úhrný čas sa od sumy časov jednotlivých volaní líšil minimálne.

4.1.3 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 operácia vykonávala na aktívnom kontexte, takže renderovanie prebiehalo z volajúceho vlákna. Kontext bol nastavený ako aktívny z 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 tohto vlákna.

4.1.4 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

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 driverov grafickej karty	8.2 (Windows), 8.45.4 (Linux)

Tabuľka 4.1: 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 driverov grafickej karty	163.71

Tabuľka 4.2: Zostava 2 - NVidia

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 4.1, 4.2 a 4.3.

4.2 Analýza výsledkov

Tabuľka 4.4 uvádza namerané hodnoty obrázkov za sekundu. Identifikátor ATI-N (W) reprezentuje sieťovú verziu enginu, pričom server aj klient bežali na tom istom počítači. Pri všetkých meraniach vidieť, že režijné náklady pri asynchrónnom renderovaní do pamäte sú približne dvojnásobné oproti synchrónnom variante. Vzhľadom na minimálny rozdiel vo vykonávaných

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 driverov grafickej karty	6.14.10.4020

Tabuľka 4.3: Zostava 3 - Intel

Meranie/zostava		ATI (W)	ATI (L)	NVidia	Intel	ATI-N (W)
Pamäť	synch.	2078	13661	1941	2571	78
	asynch.	846	8149	926	1706	66
OpenGL	synch.	3107	2989	3558	815	93
	asynch.	989	986	1102	808	75

Tabuľka 4.4: Počet obrázkov za sekundu pri prázdnom renderovaní

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.

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.

Meranie sieťovej verzie sme uviedli len pre úplnosť. Časy sú silne ovplyvnené nutnosťou prenášať obrázky po sieti. Pri renderovaní sme použili rozmer prenášaného obrázku 512x512 pri troch bajtoch na pixel. Do budúcna počítame s použitím kompresie posielaných dát, čo by malo zvýšiť množstvo vyrenderovaných obrázkov za sekundu [3].

Kapitola 5

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] 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.

[3] 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.

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

[5] 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.

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

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

[8] 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.

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

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

[11] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195, 2005.

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

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