

Bc. Miroslav Beno

**Vizualizácia objemových dát pomocou
grafického procesora**

Diplomový projekt I

Vedúci diplomového projektu: Ing. Ondrej Hirjak

Máj 2011

ANOTÁCIA

Slovenská Technická Univerzita v Bratislave **FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLOGIÍ**

Študijný program: **INFORMAČNĚ SYSTĚMY**
Autor: **Bc. Miroslav Beno**
Diplomový projekt: **Vizualizácia objemových dát pomocou
grafického procesora**
Vedenie diplomového projektu: Ing. Ondrej Hirjak
Rok odovzdania: máj 2011

V súčasnosti umožňuje grafický hardvér podporu všeobecných výpočtov prostredníctvom plne programovateľných prúdových procesorov a príslušných programovacích rozhraní. Oproti tradičným procesorom ponúka násobne väčší výkon pri aplikovaní úloh, v rámci ktorých je možné využiť masívny paralelizmus najmä s ohľadom na nezávislé spracovanie dát. Takouto úlohou je aj vizualizácia objemových dát, kedy sú v skalárnej trojrozmernej mriežke vyjadrené parametre diskretných objemových elementov nejakého priestoru. Tieto dáta majú svoje významné použitie napríklad v medicínskej oblasti. Práca v časti analýzy zachytáva súčasný stav v sfére programovania grafických procesorov vrátane historických východísk, teoretických základov takéhoto programovania, a charakteristík konkrétnych programovacích rozhraní. Opísaný je tiež programovací model, ktorý objasňuje princípy organizácie a pristupovaniu k jednotlivým komponentom grafického hardvéru. Ďalej sa práca zaoberá analýzou objemových dát, ich zdrojov a ich formátu. Diskutovaný je aj optický model, na ktorého reprezentácii zobrazovanie objemových dát stavia. Kategorizované sú rôzne metódy vizualizácie objemových dát spolu s technikami, ktoré optimalizujú ich beh, pričom dôraz je kladený na pokročilejšie metódy, vhodné na implementáciu na súčasných programovacích rozhraniach grafického hardvéru.

ANNOTATION

Slovak University of Technology Bratislava **FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGY**

Degree Course:	INFORMATIC SYSTEMS
Author:	Miroslav Beno
Diploma project:	Volume data rendering on graphics hardware
Supervisor:	Ing. Ondrej Hirjak
Year of the submission:	May 2011

At present, graphics hardware supports general computations to be fully programmable through a stream processor with corresponding programming interface. Compared to traditional processors, it offers multiple times more power when applied to tasks in which massive parallelism can be exploited in particular for independent data processing. One such task is volume data rendering, where data is a three-dimensional scalar grid of parameters of some discrete volume elements in three-dimensional space. Major use for this data type is for example in the medical field. Part of work analyses the current situation in the realm of graphical programming processors, including historical background, theoretical foundations of such programming, and characteristics of specific programming interfaces. Programming model explaining the principles of organization and accessing the individual components of the graphical hardware is also described. The work deals with the analysis of volumetric data, their resources and their format and discussed is also the optical model, on which representation of volume data is built. Different volume rendering methods together with optimization techniques are categorized, while putting emphasis on more advanced methods, suitable for implementation on existing programming interfaces of graphics hardware.

Obsah

1	Úvod.....	1
2	Programovanie grafických procesorov	3
2.1	História a vývoj	3
2.2	Architektúra grafických kariet	6
2.3	Rozhrania pre programovanie grafických procesorov	6
2.3.1	OpenCL	6
2.3.2	CUDA.....	7
2.4	Programovací model rozhraní grafického hardvéru	7
2.4.1	Štruktúra kódu	7
2.4.2	Pamäťový model	8
2.4.3	Model vlákien.....	10
3	Vizualizácia objemových dát	13
3.1	Objemové dáta.....	13
3.1.1	Formát objemových dát.....	13
3.2	Optický model vizualizácie	14
3.3	Prehľad spôsobov vizualizácie	15
3.3.1	Všeobecné fázy vizualizácie objemových dát.....	15
3.3.2	Nepriame zobrazovanie.....	16
3.3.3	Priame zobrazovanie	17
3.3.4	Klasifikácia a prenosová funkcia	18
3.4	Ray casting	19
3.4.1	Optimalizačné techniky	20
4	Opis riešenia.....	22
4.1	Špecifikácia	22
5	Zoznam bibliografických odkazov	23

1 Úvod

Grafický procesor (GPU) na dnešných bežne dostupných grafických kartách sa postupne vyvinul vo veľmi výkonnú a flexibilnú výpočtovú jednotku. Posledné generácie grafických akceleratorov poskytujú veľké množstvá operačnej pamäte a výpočtového výkonu spolu s plne programovateľnými jadrami realizujúcimi všeobecné operácie. Z pohľadu architektúry grafický hardvér ponúka vykonávanie vysoko paralelizovaných prepočtov na veľkom počte výpočtových jednotiek. Pojmom General-purpose computing on graphics processing units (GPGPU) sa označuje postup, kedy sa na grafickej karte vykonávajú všeobecné výpočty mimo pôvodného dizajnu grafických akceleratorov, ktoré sú normálne určené na spracovanie klasickým procesorom (CPU). V súčasnosti existujú vysoko-úrovňové programovacie jazyky so štandardnou syntaxou a rozličné aplikačné rozhrania s vyššou úrovňou abstrakcie podporujúce GPGPU. Vďaka týmto rozhraniám nie je potrebná podrobná znalosť architektúry grafickej karty. Stále je však pre efektívne využitie výpočtového výkonu potrebné prostredníctvom GPGPU riešiť problémy realizovateľné paralelným spracovaním. Jedná sa o problémy, ktoré možno adaptovať pre využitie masívneho paralelizmu, čiže pozostávajú z viacerých dekomponovaných algoritmov na sebe viac alebo menej sekvenčne nezávislých. Vtedy možno sledovať výraznú akceleráciu výpočtu oproti vykonávaniu na tradičných univerzálnych procesoroch.

Jednou z vhodných aplikácií pre GPGPU spracovanie je vizualizácia objemových dát. Objemové dáta predstavujú model trojrozmerného objektu reprezentovaného veľkým množstvom diskretných bodov v priestore (voxelov), s rôznymi charakteristikami jednotlivých bodov alebo zoskupujúcich vrstiev. Zdrojom takýchto dát môže byť napríklad pokročilá röntgenová snímka z lekárskeho prostredia (tomografia) alebo sken rôznych archeologických objektov určený na analýzu. Pri vizualizácii objemových dát sa vytvára dvojrozmerná projekcia trojrozmerného objektu v takejto reprezentácii, pričom bolo navrhnutých niekoľko štandardných spôsobov a metód tejto vizualizácie. Okrem toho, že vizualizácia objemových dát je náročná na výpočtový výkon, býva ďalším problémom v závislosti od detailnosti modelu aj pamäťová náročnosť pri ich spracovávaní. Skúmajú sa teda aj rôzne optimalizačné techniky umožňujúce redukovať alokáciu potrebných pamäťových zdrojov.

V rámci teoretickej časti sa táto diplomová práca zaoberá v prvých dvoch kapitolách analýzou existujúcich aplikačných rozhraní na prístup ku grafickej karte podporujúcich všeobecné inštrukcie s použitím vyšších programovacích jazykov a zhodnotením vývoja oproti predchádzajúcim spôsobom realizovania GPGPU výpočtov. Z hľadiska konkrétneho využitia výpočtovej sily grafického hardvéru budú predmetom skúmania aj jednotlivé metódy vykresľovania objemových dát, spolu s porovnaním ich zložitosti, efektívnosti a výslednej použiteľnosti. S tým súvisí aj rozbor v doméne optimalizačných techník pre zobrazovanie konkrétnymi metódami.

Praktickú časť diplomovej práce opísanú v ďalších kapitolách bude tvoriť implementácia programu na vizualizáciu objemových dát pomocou zvoleného rozhrania a príslušného hardvéru. Návrh tejto implementácie bude založený na výstupe analýzy a bude zohľadňovať výber takých metód, aby ich bolo možné vykonávať v reálnom čase. Cieľom je dosiahnuť zakomponovanie interaktívneho prehliadania modelu, respektíve animovať výsledné zobrazenie. Pomocou vhodného používateľského rozhrania bude zabezpečená aj parametrizácia vizualizácie prostredníctvom výberu rôznych metód a možnosť ich vzájomného porovnávania. Aplikácia bude testovaná na súbore voľne dostupných objemových dát, s dôrazom na variabilitu ich komplexnosti a veľkosti pre overenie použitých metód a techník. Výsledkom testovania bude súhrn o súčasnej úrovni možného využitia prístupných grafických kariet pre vizualizáciu objemových dát, spolu s najefektívnejšími metódami a dosiahnutými charakteristikami kvality zobrazenia modelu. Keďže implementovaný kód bude po miernej modifikácii spustiteľný aj na tradičných výpočtových procesoroch, bude možné zahrnúť do výstupu praktickej časti aj porovnanie efektivity implementácie algoritmov medzi CPU a GPU.

2 Programovanie grafických procesorov

2.1 História a vývoj

Za posledné desaťročia prekonali klasické výpočtové mikroprocesory radikálny vývoj prostredníctvom kontinuálneho zvyšovania výkonu a zároveň znižovania ceny na jednotku výkonu. Výkon procesorov z kategórie bežných a najrozšírenejších spotrebných typov prítomných v notebookoch a pracovných staniciach dosiahol rádovo jednotky gigaflopov. To umožnilo rozmach náročnejších softvérových aplikácií spoliehajúcich sa na tento výkon, s väčším množstvom funkcionality a zároveň vyššou mierou užitočnosti pre koncových používateľov informačných technológií. Tento vývoj bol však okrem iného z technického hľadiska založený aj na zvyšovaní frekvencie jadra procesorov, pričom takýto spôsob okolo roku 2003 narazil na hranice výhodnosti v súvislosti so spotrebou elektrickej energie a zahrievaním. Riešením bol radikálnejší zásah do architektúry mikroprocesorov v podobe využitia viacerých výpočtových jadier na jednom čipe procesora so zámerom zvýšiť výkon.

Takéto zvýšenie nie je automaticky úmerné počtu použitých jadier, keďže musí byť zakomponované a zohľadnené aj pri vývoji softvérových aplikácií, ktoré sú tradične reprezentované sekvenčnou postupnosťou inštrukcií. Transformácia programov tak, aby bolo možné vykonávať viacero ich častí súčasne spadá pod metódy takzvaného paralelného programovania. S postupným vývojom výpočtových mikroprocesorov, ktoré počíta so zvyšovaním škály výkonu naďalej prostredníctvom zvyšovania počtu jadier (momentálne sú komerčne dostupné osemjadrové procesory) je dôležitosť paralelného programovania o to viac evidentná. Spolu s ním sa pozornosť upriamila však aj na hardvér, ktorý multijadrovú architektúru využíva už z podstaty dát s ktorými narába. Je ním hardvér určený na spracovávanie a akcelerovanie grafických dát, alebo skratene grafické karty. Na porovnanie ich hrubý výkon pri využití paralelného spracovania dosahuje v súčasnosti stovky gigaflopov. Tu vyplynula motivácia tvorcov hardvéru a zároveň vývojárov softvéru presunúť výpočtovo náročné časti softvéru, pokiaľ je to vhodné, na grafický procesor. Samozrejým východiskom rýchleho vývoja tejto idey bol dostatočne široký trh aplikácie, čo pri masovej rozšírenosti grafických kartách nachádzajúcich sa na miliónoch spotrebných počítačoch nebol problém.

Hardvér podporujúci spracovanie grafických informácií do podoby v akej ho poznáme dnes prešiel, podobne ako tradičné mikroprocesory, dlhším vývojom. Je evolúciou veľkých a drahých systémov z 80-tych rokov, prechádzajúcou cez menšie dedikované pracovné stanice, až ku grafickým akceleratorom osobných počítačov v podobe kariet a čipov. Počas tohto obdobia sa násobne znižovala cena a zvyšoval výkon týchto zariadení, pričom dôvodom nebolo len zlepšovanie fyzických charakteristík komponentov ale aj v inovovaní grafických algoritmov a hardvérového dizajnu. Tento vývoj bol poháňaný trhom požadujúcim vysoko kvalitné grafické zobrazenie v reálnom čase, neskoršie hlavne dravým videoherným priemyslom. V rámci tohto vývoja sa grafický hardvér pretransformoval z jednoduchého nástroja na zobrazovanie diagramov na komplexnú, vysoko paralelnú architektúru schopnú

vykresľovať zložité interaktívne trojdimenzionálne modely. Zároveň sa funkčnosť ponúkaná grafickými procesormi stala oveľa sofistikovanejšou a modifikovateľnejšou.

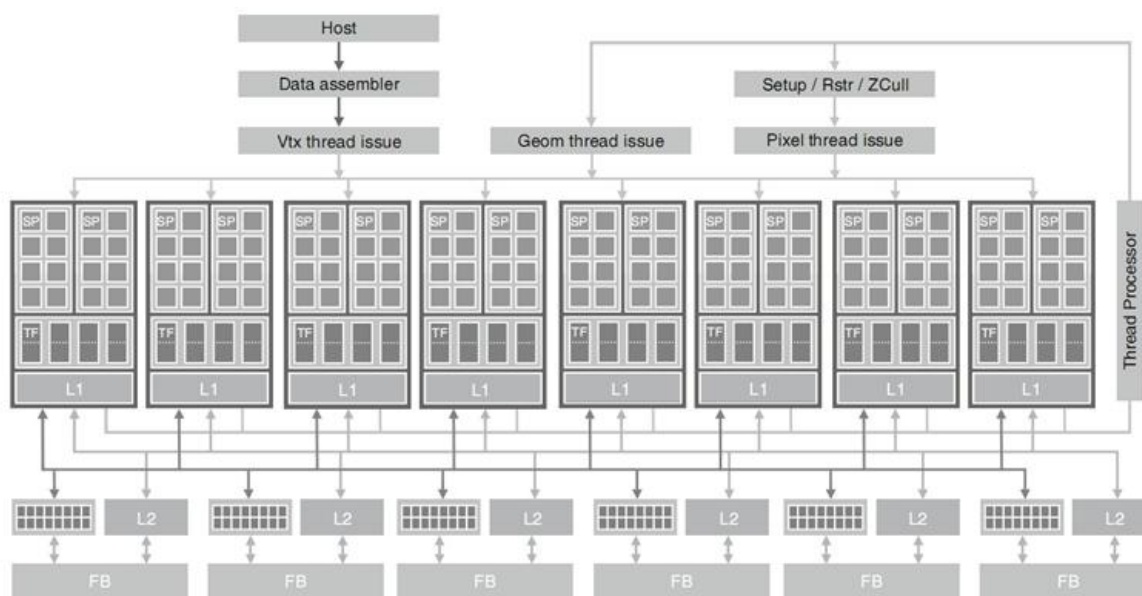
V prvej ére spotrebných grafických akcelerátorov, počas 90-tych rokov, pracoval grafický hardvér na základe pevnej množiny funkcií spracovania a vykresľovania grafických dát, ktoré mali rôzne konfigurovateľné parametre, ale nebolo možné programovať nové funkcie. Išlo o takzvanú *fixed-function graphic pipeline*, v rámci ktorej bola presne definovaná postupnosť spracovania dát spolu s príslušnými metódami v jednotlivých fázach. Medzi zástupcov týchto spotrebných grafických akcelerátorov patria prvé grafické karty od spoločností *nVidia* a *3Dfx*. Na prístup k funkciám týchto kariet slúžili štandardizované rozhrania umožňujúce komunikovať s grafickým hardvérom a využívať presne vymedzenú funkcionálnu. Medzi tieto rozhrania, ktoré sa s aktualizovanými verziami používajú dodnes, patrili komerčne najrozšírenejší *DirectX* pevne naviazaný na prostredie *Microsoft Windows*, a otvorený štandard *OpenGL*, podporovaný viacerými výrobcami, a sprístupnený napríklad aj na operačnom systéme *Linux*.

Evolúciou, ktorá nastala v roku 2001 s treťou generáciou kariet *nVidia GeForce*, bolo umožnenie základnej modifikácie a programovania najvýznamnejších častí spracovania grafických dát. Tento spôsob sa označuje ako *programmable graphic pipeline* a sprisťušoval inštrukčnú sadu spracovania *vertex* dát a procesov vo fáze *shader* spracovania (tiež označované ako pixel alebo fragment shader). *Vertexami* sa označujú vrcholy polygónov, respektíve trojuholníkov, v ktorých grafický hardvér reprezentuje objekty, pričom sa aplikujú rôzne transformačné metódy na tieto vrcholy. V rámci *pixel shader* fázy sa zase určuje výsledná farba každého bodu obrazu (*pixelu*), a to rôznymi spôsobmi napríklad mapovaním textúry na objekty, rozličnými svetelnými a inými efektmi, alebo ich kombináciou. Pre hardvérové jednotky akcelerujúce tieto fázy spracovania obrazu bolo umožnené písať programy s danou inštrukčnou sadou sprístupnenou opäť prostredníctvom rozhraní a príslušných programovacích jazykov (takzvané *Shading languages*). Tieto ponúkajú zväčša vyššiu úroveň abstrakcie a vychádzajú zo všeobecných programovacích jazykov. Existuje niekoľko variantov s rôznymi špecifikami:

- ARB (*ARB low-level assembly language*) - jazyk používajúci nízkoúrovňové inštrukcie, úrovňou podobajúci sa na assembler.
- GLSL (*OpenGL shading language*) - viaže sa na rozhranie *OpenGL*, postavený na ARB, so syntaxou vychádzajúcou z jazyka C. Umožňuje používanie cyklov, a vetvení programu.
- HLSL (*High Level Shader Language*) - jazyk úrovňou podobný GLSL, pre rozhranie *DirectX* od spoločnosti *Microsoft*.
- Cg - jazyk vyvíjaný spoločnosťou *nVidia*, úrovňou opäť podobný predchádzajúcim dvom jazykom, snažiaci sa o nezávislosť od rozhrania. Súvisí s ním aj množstvom nástrojov podporujúcich efektivitu vývoja.

Tieto jazyky však stále neboli priamo určené na implementovanie všeobecných výpočtov, jednalo sa o umožnenie väčšej flexibilitnosti pri operáciách s grafickými dátami. Aj keď všeobecné výpočty v určitej miere sprístupňovali, pri ich realizácii existovali obmedzenia a bola potrebná podrobná znalosť domény grafických operácií na grafickej karte. Napríklad bolo nutné pre prístup k dátam používať textúry alebo obchádzať rôzne nežiaduce transformácie a zásahy do dát neprogramovateľnými časťami procesu spracovania, pričom výsledok generovaný pomocou *pixel shadera* bol reprezentovaný v podobe súboru *pixelov*, z ktorého sa následne extrahovali potrebné informácie.

V roku 2006 pri vydaní ôsmej generácie kariet *nVidia* bolo predstavených niekoľko zlomových princípov. Pribudol napríklad nový *geometry shader* umožňujúci pokročilé spracovanie súborov *vertex* údajov. Dôležitejšie však bolo že z hľadiska hardvérovej architektúry, ale aj softvérovej reprezentácie, sa stali výpočtové jednotky (procesorové jadrá) unifikovanými [1]. To znamená, že fyzická jednotka realizujúca operácie výpočtov je svojou funkčnosťou závislá iba na softvérovom naprogramovaní, a teda vhodná na akýkoľvek typ *shaderu* (programu). V praxi sa pri *graphic pipeline* s unifikovanými procesormi stretávame s tromi cyklami, kedy pri každom cykle je využitie procesorov dynamicky alokované pre konkrétny typ *shaderu* (*vertex*, *geometry*, *pixel*), s príslušnou operačnou a dátovou logikou uskutočňovanou medzi týmito cyklami (Obr. 1).



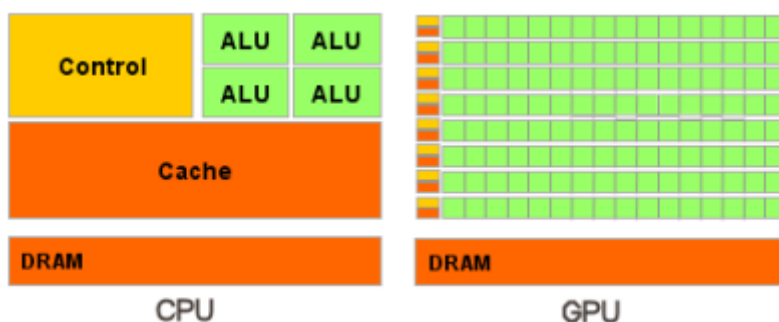
Obr. 1 Ukážka *graphic pipeline* s polom unifikovaných procesorov [1].

S unifikovaným procesormi prišlo aj mierne zníženie výkonu oproti predošlým procesorom s menšími presne definovanými inštrukčnými sadami, čo však bolo vyvážené ďalšími vylepšeniami s pohľadu celkovej architektúry karty, vyšších taktov procesorov a plnom využití všetkých jednotiek pri každej z fáz *pipeline*. Mimo domény spracovania grafických dát tým zároveň došlo k výraznému zjednodušeniu a umožneniu oveľa širšieho záberu v rámci všeobecných výpočtov na takýchto procesoroch, keďže sa svojou programovateľnosťou priblížili tradičným počítačovým procesorom.

Zároveň boli špecifikované nové rozhrania a súbory funkčných knižníc, ktoré ponúkali generickejší prístup ku grafickému hardvéru a podporovali ideu realizácie všeobecných výpočtov na grafických kartách.

2.2 Architektúra grafických kariet

Pri programovaní pre grafické procesory je potrebné oboznámiť sa aj s fyzickou hardvérovou architektúrou takýchto zariadení, ktorá ovplyvňuje a ozrejmuje spôsob logického spracovania údajov. Zatiaľ čo tradičné výpočtové mikroprocesory sa snažia pri použití viacerých jadier (*multi-core*) stále zameriavať na udržanie výkonu primárne pre sekvenčné programy, grafický hardvér s mnohými jadrami (*many-core*) sa sústreďuje výhradne na výkonovú priepustnosť paralelného spracovania [1]. Od toho sa odvíja aj ich odlišne poňatá architektúra, skladajúca sa v základe z riadiacich jednotiek, výpočtových jednotiek (jadrá) a niekoľko druhov pamätí. Pri tradičnom procesore je dôležitou časťou komplexná riadiaca jednotka, ktorá prerozdeľuje inštrukcie výpočtovým jadrám, napríklad aj v rámci jedného sekvenčného programu, a dôraz je kladený aj na cache pamäť pre urýchlenie prístupu k dátam a inštrukciám. Pri grafických procesoroch existuje niekoľko jednoduchších riadiacich jednotiek (Obr. 2), ktoré spolu s menšími cache pamäťami zdieľajú väčšie počty jadier usporiadaných v blokoch.



Obr.2 Porovnanie architektúry tradičného procesora a grafického hardvéru [2].

2.3 Rozhrania pre programovanie grafických procesorov

2.3.1 OpenCL

Open Computing Language alebo skratene OpenCL predstavuje otvorený štandard pre programovanie aplikácií, ktorých kód bude spustiteľný na rôznych hardvérových platformách naprieč tradičnými procesormi, procesormi grafických kariet a ďalšími špecializovanými procesormi. Primárne je však používaný najmä na sprístupnenie všeobecných výpočtov na grafických procesoroch. Vyvinutý bol spočiatku spoločnosťou Apple, neskôr prešla správa vývoja na konzorcium Khronos Group, ktoré združuje veľké množstvo významných IT spoločností ako ATI, nVidia, IBM, Intel alebo Nokia. Oficiálna špecifikácia prvej verzie bola vydaná na konci roku 2008.

OpenCL pozostáva z programovacieho jazyka, ktorý vychádza z jazyka C s niekoľkými obmedzeniami a rozšíreniami a knižnice aplikačných rozhraní pre prístup ku takzvanej heterogénnej výpočtovej platforme. To znamená, že program využívajúci toto rozhranie sa bude dať spustiť na grafických kartách rôznych výrobcov, ktorí podporujú toto rozhranie v ovládačoch ku svojmu hardvéru, rovnako ako aj na tradičnom procesore. OpenCL sa snaží prístup k týmto výpočtovým zdrojom zjednotiť a výrazne tak zvýšiť portabilitu výpočtovo náročných aplikácií.

Definované sú paralelné časti kódu alebo funkcie – kernely, spúšťané vláknami v na sebe nezávislých skupinách pričom je možná synchronizácia vlákien v rámci skupiny. Platformový model OpenCL zahŕňa delenie na domovské prostredie a jedno alebo viacero výpočtových zariadení. V rámci narábaní s dátami sa rieši explicitná manipulácia v rôznych typoch pamätí definovaných pamäťovým modelom. Pri programovacom jazyku medzi obmedzenia patrí napríklad nepoužívanie ukazovateľov na funkcie, rekurzie, premenlivej dĺžky polí. Na druhej strane knižnica rozhraní obsahuje rôzne nadštandardné metódy na manipuláciu obrazu alebo rôzne matematické operácie.

2.3.2 CUDA

Compute Unified Device Architecture predstavuje architektúru pre sprístupnenie všeobecných paralelných výpočtov na grafických kartách spoločnosti nVidia, čiže je závislá na konkrétnej hardvérovej platforme. Pozostáva z jazyka pomenovaného *C for CUDA*, ktorý predstavuje jazyk C s rôznymi rozšíreniami a obmedzeniami, a knižnice aplikačných rozhraní pre prístup k funkciám hardvéru. Predstavená bola na začiatku roku 2007 práve spoločnosťou nVidia a pre jej karty predstavuje teoreticky rovnocennú alternatívu OpenCL štandardu pre všeobecné výpočty.

2.4 Programovací model rozhraní grafického hardvéru

V tejto kapitole sú opísané základné princípy a postupy v rámci programovacieho modelu pre grafické karty, pričom sú konkrétne demonštrované na architektúre CUDA [1, 2].

2.4.1 Štruktúra kódu

CUDA programy pozostávajú zo zjednoteného zdrojového kódu kódu, ktorý obsahuje časti vykonávané sekvenčne na tradičnom CPU (*host*) aj paralelizovateľné časti implementované pre grafický hardvér (*device*). Z hľadiska programovacieho jazyka sa používa štandardná syntax ANSI C, s rozšíreniami v podobe kľúčových slov a dátových štruktúr pre *device*-časť kódu. Tieto paralelizovateľné časti kódu sa nazývajú *kernel* funkcie a spúšťajú sa spravidla vo veľkom počte vlákien, podľa množstva údajov, na ktorých budú vykonané. Po skončení vykonávania vlákien program môže sekvenčne pokračovať ďalej a napríklad spúšťať ďalšie *kernel* funkcie s inými dátami. Preto môžeme konštrukciu CUDA programu chápať ako

klasický sekvenčný kód s vhodne identifikovanými *kernel* časťami spracovania, ktoré sú „outsourcované“ na grafický hardvér.

Na označenie *kernel* funkcií vyvolávajúcich vykonávanie vlákien na karte sa používajú kľúčové slová, ktoré sa uvádzajú v rámci deklarácie funkcie.

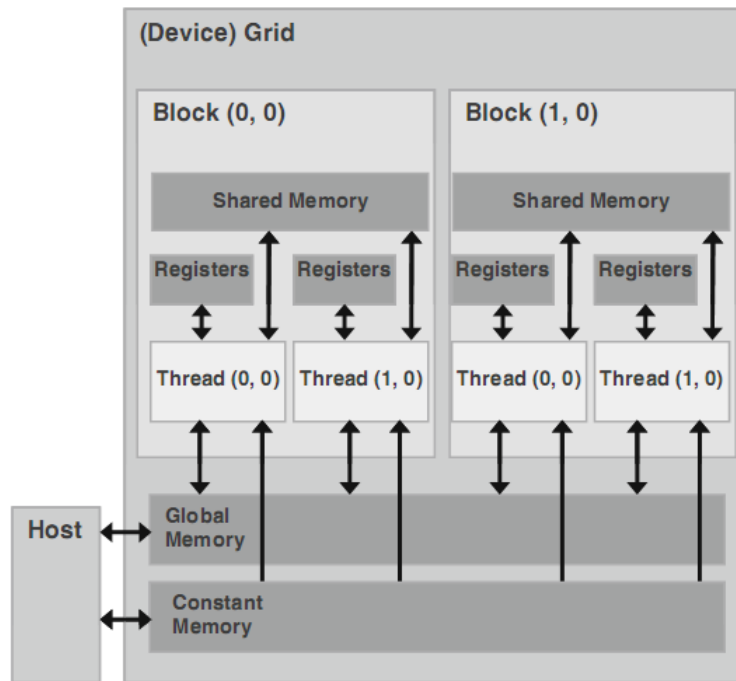
- `__global__` – označuje *kernel* funkciu, ktorá môže byť zavolaná z *host* časti programu, a jej telo sa vykonáva na device v podobe vytvorenia paralelných vlákien.
- `__device__` – označuje funkciu, ktorá môže byť volaná iba z inej *kernel* alebo `__device__` funkcie, vykonávaná je na device.
- `__host__` – označuje tradičnú C funkciu – môže byť volaná iba z inej *host* funkcie, vykonávaná na *host*. Funkcie bez uvedeného kľúčového slova sú implicitne `__host__` funkciami.

Funkcie vykonávané na *device* nemôžu obsahovať rekurzívne volania, alebo nepriame volania cez ukazovatele. Je možné použiť aj kombináciu `__host__` a `__device__` príznakov, pričom v takomto prípade sa pri kompilovaní vygenerujú dve verzie funkcie s príslušnými vlastnosťami volania.

2.4.2 Pamäťový model

Pri práci s dátami CUDA programy pracujú s dvoma oddelenými pamäťovými priestormi. Pamäť *host* prostredia predstavuje obyčajne klasická DRAM, prístupná pre CPU a všetky bežiacie programy a operačný systém. V rámci vykonávania na *device* táto nie je priamo prístupná, pracuje sa tu s vlastnou pamäťou grafického hardvéru. Pri spracovávaní dát v *kernel* funkciách je potrebné najskôr miesto v tejto pamäti alokovať a preniesť do nej dáta z *host* prostredia, a následne po výpočte uskutočniť opačný proces a pamäť uvoľniť. Pamäťový model grafického hardvéru podporujúceho CUDA je možno vidieť na Obr. 3, a je rozdelený na niekoľko rôznych typov pamätí podľa veľkosti, latencie a prístupu:

- Globálna pamäť je najväčšia hlavná pamäť grafickej karty. Jej charakteristikami sú veľká kapacita (v súčasnosti až v jednotkách gigabajtov), vysoká priepustnosť a vysoká miera latencie prístupu. sa vyznačuje väčšou prístupovou latenciou. Používanie len globálnej pamäte preto výrazne limituje výkonový potenciál grafických procesorov, keďže kvôli čakaniu na zdroje dát sú vlákna nečinné. Prostredníctvom všeobecných pamäťových inštrukcií k nej môže pristupovať *host* a každý z procesorov vykonávajúcich vlákna. Efektivitu kódu *kernelu* v súvislosti s pristupovaním ku globálnej pamäti možno vyjadriť pomerom výpočtových inštrukcií ku inštrukciám na prístup do globálnej pamäte. Jedná sa o výkonovo dôležitý parameter keďže hrubý výpočtový výkon je výrazne závislý od rýchlosti, akou sa dá pristupovať k dátam na ktorých sú výpočty vykonávané – preto existuje snaha časté prístupy ku globálnej pamäti minimalizovať.



Obr. 3 Pamäťový model CUDA zariadenia [1].

- Registre a zdieľané pamäte sú takzvané „*on-chip*“ pamäte nachádzajúce sa priamo pri výpočtovom multiprocesore a v porovnaní s globálnou pamäťou poskytujú vysokú rýchlosť aj pri paralelnom prístupe. Tieto pamäte však ponúkajú obmedzenú kapacitu a efektívne narábanie s nimi je často špecifické pre konkrétny algoritmus. Registre sú prístupov logicky naviazané na konkrétne jedno vlákno a používajú sa na uchovávanie často dotazovaných premenných privátnych pre každé vlákno. Zdieľaná pamäť predstavuje efektívny spôsob kooperácie a zdieľania dát medzi vláknami v rámci bloku.
- Pamäť konštánt je menej používaná pamäť pre konštanty, s malou kapacitou, prístupná pre *host* do na operácie zápisu aj čítania, device k nej prístupuje len na čítanie, pričom poskytuje vo väčšine prípadov malú latenciu.

Typ pamäte kde bude hodnota premennej uložená špecifikujú kvalifikátory pri deklaráciách premenných. Tieto zároveň špecifikujú viditeľnosť (obor), životnosť a rýchlosť prístupu k premennej. Automaticky generované premenné vlákien sú ukladané v registroch vlákna, poprípade v globálnej pamäti, ak sa jedná o polia, a existuje vlastná kópia pre každé vlákno. Kľúčové slovo `__shared__` označuje zdieľanú premennú prístupnú pre všetky vlákna jedného bloku a je uložená v zdieľanej pamäti, čiže prístup k nej je pomerne efektívny. Často sa používa na udržiavanie intenzívne využívannej časti dát z globálnej pamäte. Kľúčové slovo `__constant__` označuje konštanty, existujúce počas celého behu programu a viditeľné zo všetkých spúšťaných vlákien, uložené sú v pamäti konštánt, rýchly prístup je umožnený vďaka *cacheovaniu*. Kľúčové slovo `__device__` označuje globálnu premennú, uloženú v globálnej pamäti. Používa sa na uchovávanie vstupných a výstupných dát pred a po behu

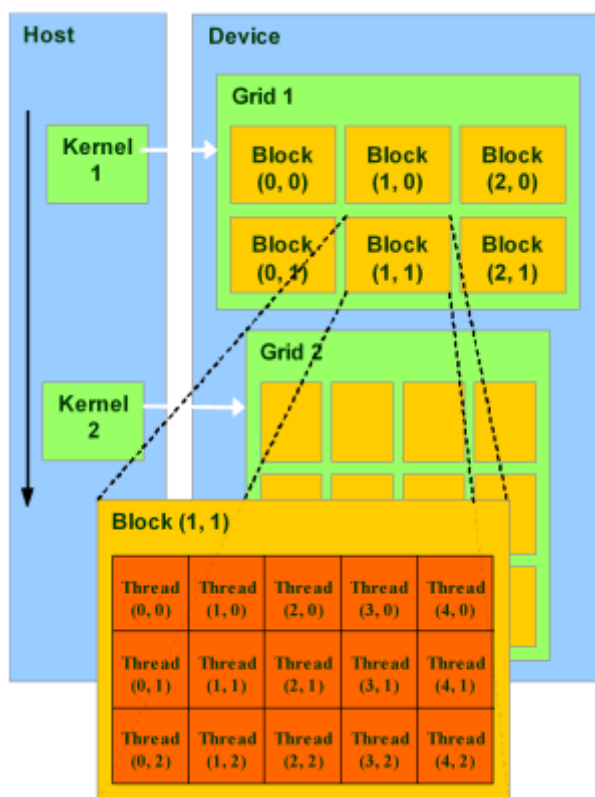
kernel funkcie, nakoľko prístup k nej z blokov bežiacich vlákien je zo spomínaných typov najpomalší a nie je synchronizovaný.

Operácie s pamäťou sú realizované prostredníctvom funkcií aplikačného rozhrania, pričom predobrazom sú klasické funkcie C na prácu s pamäťou.

- *cudaMalloc* – umožňuje alokovať priestor v globálnej pamäti, podobne ako štandardná *malloc* funkcia. Pracuje s generickými adresnými ukazovateľmi a veľkosťami v bajtoch.
- *cudaMemcpy* – umožňuje presunúť dáta medzi *host* pamäťou a globálnou *device* pamäťou. Udáva sa cieľová adresa, zdrojová adresa, veľkosť dát, a smer prenosu.
- *cudaFree* – uvoľňuje predtým alokovaný priestor v globálnej pamäti na základe ukazovateľa.

2.4.3 Model vlákien

Keď je *kernel* funkcia vyvolaná, je spustená na mriežke paralelných vlákien. Pri efektívnom využití potenciálu fyzických procesorov grafickej karty pozostáva logická mriežka vlákien typicky z tisícok až miliónov vlákien, čoho východiskom je samozrejme masívne použitie dát podliehajúcich dátovému paralelizmu. Vlákna sú organizované v dvojúrovňovej hierarchii multidimenzionálnych štruktúr.



Obr. 4 Logická organizácia vlákien vykonávajúcich *kernel* funkcie [1].

Na prvej úrovni mriežka pozostáva z jedného alebo viacerých blokov s rovnakým vnútorným usporiadaním a počtom vlákien na jeden blok. V rámci mriežky sú tieto bloky usporiadané dvojrozmerné a identifikované indexmi *blockIdx.x* a *blockIdx.y*. Rozmer jednej dimenzie blokov môže dosahovať hodnoty od 1 do 65 535. V rámci jedného bloku sú vlákna, ktorých počet môže byť maximálne 512, usporiadané trojrozmerné a identifikované indexmi *threadIdx.x*, *threadIdx.y* a *threadIdx.z*. Premenné indexov alebo koordinátov sú predefinované premenné, ku ktorým je možné pristupovať v rámci každého vlákna. Ich úlohou je jednoznačne identifikovať každú inštanciu vlákna, a hlavne identifikovať úsek dát, ktorý bude dané vlákno spracovávať.

Kvôli obmedzeniu na počet vlákien pre jeden blok je v praktických aplikáciách potrebné mapovať dáta na viacero blokov s použitím premenných indexov. Napríklad, ak sa spracováva jednorozmerné pole údajov, a každé vlákno má za úlohu pracovať s jedným prvkom určeným jedným indexom poľa, tak v *kernel* funkcii určíme identifikátor vlákna predstavujúci zároveň jemu pridelený index ako $threadID = blockIdx.x * blockDim.x + threadIdx.x$;. Mriežka aj bloky sú v tomto prípade jednorozmerné, ďalšie dimenzie sa ignorujú, respektíve sú nastavené na 1.

Inicializácia organizácie vlákien a blokov vlákien pred spustením *kernelu* je deklarovaná pri *kernel* funkciách prostredníctvom parametrov označených značkami <<< a >>>. V týchto parametroch sa udávajú rozmery mriežky blokov, a rozmery (počty vlákien) dimenzií samotných blokov. Prvý parameter určuje rozmery mriežky v zmysle počtu blokov, a druhý rozmery bloku v zmysle počtu vlákien. Parametrami sú dim3 dátové typy, ktoré združujú tri celé čísla x,y,z reprezentujúce veľkosti dimenzií. Po spustení *kernelu* sa už organizácia mriežky nemení až do skončenia jeho vykonávania.

Pre synchronizáciu vlákien sa používa funkcia *_syncthreads()*, ktorá zastaví vykonávanie vlákna, pokým nedosiahnu rovnakú pozíciu v *kernel* funkcii aj ostatné vlákna v rámci bloku. Tým je umožnený barierový spôsob synchronizácie, ktorý umožňuje rozdeliť paralelný proces do niekoľkých fáz. Vlákna sa synchronizujú vždy v jednom synchronizačnom bode, ktorý predstavuje volanie *_syncthreads()* na konkrétnom rovnakom mieste v kóde, pričom takýchto bodov môže byť vo funkcii viacero. Synchronizácia je podporená aj tým, že prístup k žiadaným pamäťovým prostriedkom sa prideliť celému bloku vlákien naraz. Medzi blokmi sa vlákna nemôžu synchronizovať, čo dovoľuje spúšťať bloky vlákien nezávisle na sebe v rôznom poradí a súbežnosti. Tým je zabezpečená transparentná škálovateľnosť rýchlosti kódu, kedy pri väčšom počte fyzických procesorov na hardvéri sa vykonáva kód rýchlejšie vďaka paralelnému behu viacerých blokov zároveň. V súčasnosti sa dá pri spotrebných grafických kartách počítať s rádovo desiatkami výpočtových procesorov na jednej karte, pričom v závislosti jednému môže byť pridelené na vykonávanie súčasne najviac 8 blokov alebo maximálne 1024 vlákien. Z hľadiska časovania a prideľovania spúšťania vlákien sú tieto združené do takzvaných *warp* množín, ktoré sú následne prioritizované na základe dostupnosti zdrojov (pamäte), pričom cieľom je eliminovať väčšiu latenciu napríklad globálnej pamäte grafického hardvéru.

Ukážku elementárneho CUDA kódu demonštrujúcu použitie prvkov programovacieho model je možno vidieť na Obr. 5.

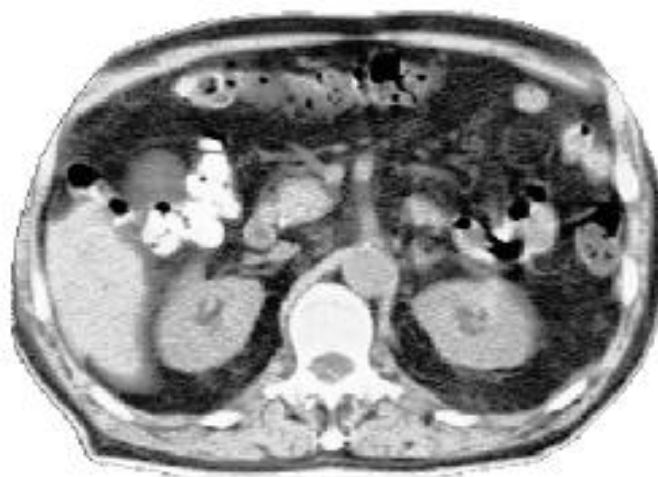
<pre> __global__ void matrixMul(float* C, float* A, float* B, int wA, int wB) { int tx = threadIdx.x; int ty = threadIdx.y; float value = 0; for (int i = 0; i < wA; ++i) { float elementA = A[ty * wA + i]; float elementB = B[i * wB + tx]; value += elementA * elementB; } C[ty * wA + tx] = value; } int main(int argc, char** argv) { unsigned int size_A = 3 * 3; unsigned int mem_size_A = sizeof(float) * size_A; float* h_A = (float*) malloc(mem_size_A); unsigned int size_B = 3 * 3; unsigned int mem_size_B = sizeof(float) * size_B; float* h_B = (float*) malloc(mem_size_B); unsigned int size_C = 3 * 3; unsigned int mem_size_C = sizeof(float) * size_C; float* h_C = (float*) malloc(mem_size_C); float* d_C; cudaMalloc((void**) &d_C, mem_size_C); readMatrixData(h_A, size_A); readMatrixData(h_B, size_B); </pre>	<pre> float* d_A; float* d_B; cudaMalloc((void**) &d_A, mem_size_A); cudaMalloc((void**) &d_B, mem_size_B); cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice); cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice); dim3 threads(3, 3); dim3 grid(WC / threads.x, HC / threads.y); matrixMul<<< grid, threads >>>(d_C, d_A, d_B, 3, 3); cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost); free(h_A); free(h_B); free(h_C); cudaFree(d_A); cudaFree(d_B); cudaFree(d_C); } </pre>
--	---

Obr. 5 Ukážka elementárneho CUDA programu – násobenie dvoch matic.

3 Vizualizácia objemových dát

3.1 Objemové dáta

Objemové dáta majú uplatnenie v rôznych oblastiach, primárne v rámci medicínskej aplikácie, ale napríklad aj v odboroch antropológie, bioinformatiky alebo geológie. Pri medicínskej aplikácii sú objemové dáta využívané predovšetkým v súvislosti s metódami pre získavanie dát o telesách z tomografie (*Computed Tomography* - CT) a magnetickej rezonancie [3]. Tieto produkujú trojrozmerné dáta vo forme súboru rovinných rezov telesom (Obr. 6), ktoré sa dajú naskladaním jednoducho previesť na množinu objemových elementov. CT pre získavanie objemových dát používa röntgenové lúče, ich snímачe a počítač pre vytvorenie obrazu pričného rezu telom pacienta. Pri CT sa rozlíšenie dát pre diagnostické účely pohybuje okolo 1/3 mm až 5 mm a hodnota dát predstavuje hustotu absorbovaných röntgenových lúčov na snímači. Pri magnetickej rezonancii sa zase dáta dvojrozmerného tomografického obrazu získavajú pomocou zaznamenania zmien gradientu magnetického poľa a vyjadrený je nimi počet protónových jadier v nasnímanom objeme. Vo všeobecnosti je interpretácia takýchto hrubých dát pomerne náročnou úlohou kvôli ich rozsiahlosti a vnútornej komplexnosti. Keď už sú takto zozbierané objemové dáta rôznymi spôsobmi vizualizované, použité sú zväčša na diagnostiku ochorení alebo anomálií. Okrem štandardných zobrazení je zaujímavou aj aplikácia takzvaného virtuálneho zobrazenia z vnútra zosnímaného objektu používaná na virtuálny pohyb v tele pacienta.



Obr. 6 Snímok prierezu brušnej dutiny získaný pomocou metódy CT [3].

3.1.1 Formát objemových dát

Pri objemových dátach je časť trojrozmerného priestoru je rozdelená na veľké množstvo pravidelných elementárnych objemových jednotiek, ktoré sa nazývajú *voxely*. Voxely si môžeme predstaviť ako kocky, ktoré diskretizujú priestor a vyjadrujú jeho objemovú

informáciu prostredníctvom množiny usporiadaných mikroobjemov. Tvorí zvyčajne pravidelnú rastrovú trojrozmernú mriežku v tvare kvádra, pričom jeho rozmery dosahujú typicky hodnoty stoviek až tisícok voxelov v jednej dimenzii. Hodnota voxelu môže byť reprezentovaná v elementárnych prípadoch jednobitovým číslom, ktoré hovorí, či sa objekt v danej časti priestoru nachádza, alebo nie. Pre reálne použitie opodstatňujúce objemový typ dát má však reprezentácia voxela podobu čísla s veľkosťou typicky 8 až 12 bitov. Potom hodnotu každého voxela môžeme chápať ako hodnotu opisujúcu hustotu, intenzitu alebo farbu daného objemového elementu. Veľkosť reálnych objemových dát, ktoré vznikajú v rámci rôznych meraní alebo simulácií sa vo výsledku pohybuje rádovo v desiatkach megabajtov, niekedy až gigabajtov.

Medzi používané súborové formáty, určené na uskladnenie objemových dát patria napríklad formáty RAW (čisté dáta), PVM, F3D (*Format 3 Dimensional*), DF3 (*POVRay Density*), NRRD (*Nearly Raw Raster Data*), pričom prvý menovaný je najuniverzálnejším spôsobom ukladania. Dáta v týchto súboroch sú organizované rozličnými spôsobmi, ale všeobecným princípom je lineárne uloženie voxelov (pri príslušnom bitovom rozsahu) v podobe veľkého pola s určenými virtuálnymi dimenziami. Potom ku konkrétnemu voxelu na pozícii (x, y, z) pri veľkosti objemu napríklad 512*256*200 môžeme pristúpiť pomocou indexu $pole[z * 256 * 512 + y * 512 + x]$ s využitím veľkosti jednotlivých dimenzií.

Takto charakterizované objemové dáta predstavujú typ dát, ktorý sa v rámci grafického hardvéru obtiažnejšie vizualizuje klasickými spôsobmi. Geometria telesa tu nie je opisovaná jeho povrchom, ako je to typické pre väčšinu modelov, ktoré grafické karty natívne vizualizujú. Vďaka takejto reprezentácii je možné modelovať aj inak náročnejšie objekty napríklad tekutín, plynov alebo prírodných živlov ako ohňa. Takto definované objekty sa ale ťažšie prevádzajú na polygónovú reprezentáciu, a navyše pri tomto prevode dochádza napríklad ku strate vnútornej štruktúry objektu. V porovnaní s povrchovou reprezentáciou objektov, majú objemovo reprezentované objekty výrazne vyššie nároky na pamäť, čo súvisí s veľkým množstvom dát nutných pre zobrazenie scény. Pred príchodom programovateľných grafických kariet, ktoré sú dnes bežne dostupné, bolo kvalitné objemové zobrazovanie doménou špecializovaného hardvéru alebo komplexných pracovných staníc. Problémom bola však ich minimálna rozšírenosť a vysoká cena.

3.2 Optický model vizualizácie

Objemové dáta ponúkajú fyzikálne vlastnosti objektu, ktorý reprezentujú. Pre syntézu obrazu počas vizualizácie sú tieto fyzikálne vlastnosti použité na výpočty prenosu a interakcie svetla s prostredím [4]. Rozlišujeme niekoľko takýchto interakcií: pohlcovanie (absorpcia), vyžarovanie (emisía) a rozptyl svetla. Pri objemových dátach je najčastejšie uvažovaný absorpčno-emitálny model interakcie svetla, ktorý berie do úvahy pohlcovanie a vyžarovanie. Opis šírenia svetla pre jeden lúč sa dá vyjadriť pri tomto modeli rovnicou objemového zobrazovania:

$$\frac{dI(s)}{ds} = -\kappa(s)I(s) + q(s) ,$$

kde je veličina I , predstavujúca vyžarovanie svetelnej energie, vyjadrená pomocou funkcií koeficientov absorpcie κ a emisie q . Vyžarovanie na konkrétnej pozícii lúča je určené v závislosti na vzdialenosti od svetelného zdroja, ktorú reprezentuje parameter s . Integrovaním uvedenej rovnice pozdĺž celého lúča, od jeho zdroja v bode $s = s_0$ po konečný bod $s = D$, získame integrál objemového zobrazenia:

$$I(D) = I_0 e^{-\int_{s_0}^D \kappa(t) dt} + \int_{s_0}^D q(s) e^{-\int_{s_0}^s \kappa(t) dt} ds .$$

Hodnota I_0 tu predstavuje intenzitu svetla vstupujúceho do objemového priestoru v bode s_0 .

Algoritmy objemovej vizualizácie sa typicky implementujú optický model pomocou diskretnej numerickej aproximácie tohto objemového integrálu. V rámci tejto aproximácie je vyjadrený iteratívny výpočet integrálu na základe kompozície optických vlastností. Takáto iteratívna kompozícia (alebo skladanie) je zadefinované v podobe predpisu:

$$C_{ciel} \leftarrow C_{ciel} + (1 - \alpha_{ciel}) C_{zdroj}$$

$$\alpha_{ciel} \leftarrow \alpha_{ciel} + (1 - \alpha_{ciel}) \alpha_{zdroj} ,$$

s veličinou C reprezentujúcou farbu a veličinou α vyjadrujúcou priehľadnosť. Tento predpis sa uplatňuje v prípade, že kompozícia prebieha spredu dozadu, teda so začiatkom lúča v bode pozorovateľa smerom do objemového priestoru.

3.3 Prehľad spôsobov vizualizácie

Vizualizáciu objemových dát môžeme stručne charakterizovať ako zobrazovanie trojrozmerných skalárnych mriežok voxelov do dvojrozmernej roviny. Ambíciou je dosiahnuť vysokú kvalitu vizualizovaného modelu objemových dát na interaktívnej úrovni rýchlosti zobrazovania. Interaktivita pri prehliadaní zvyšuje priestorové vnímanie vizualizovaného objektu a spolu s vysokým rozlíšením dát a modelu zabezpečuje rýchlejšie inšpekciu objektu. Optimálnou možnosťou, na ktorú sú zamerané aj ďalej opísané metódy je využitie bežného, finančne nenáročného, spotrebného grafického hardvéru.

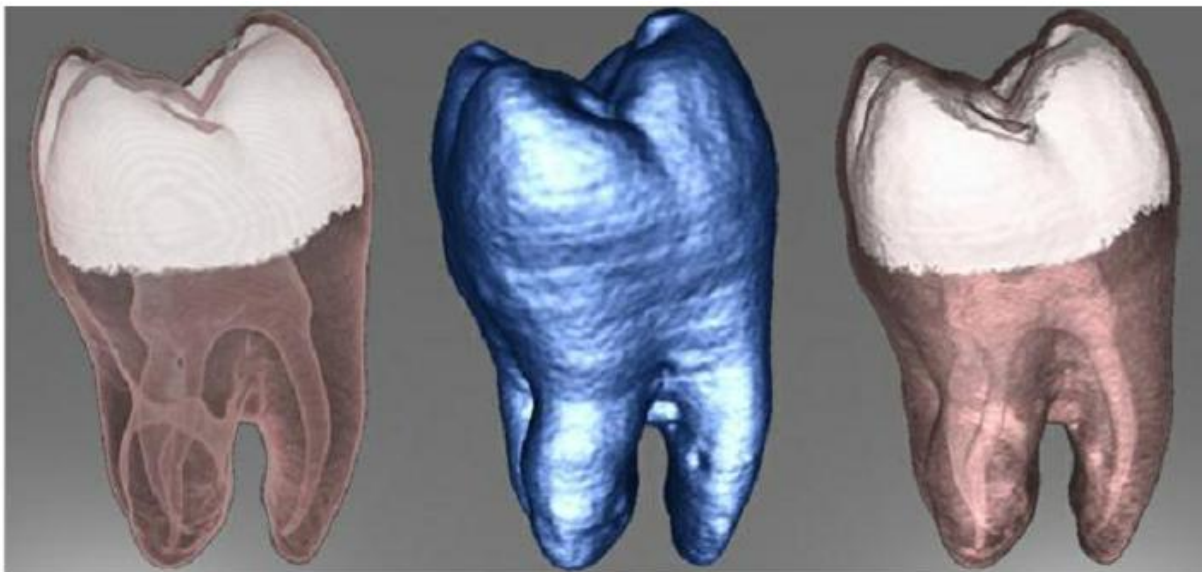
3.3.1 Všeobecné fázy vizualizácie objemových dát

Nezávisle od techniky vykresľovania sa pri vizualizácii objemových dát uplatňujú všeobecné fázy spracovania týchto dát. Tieto spolu tvoria postupnosť procesu zobrazovania objemových dát, takzvanú *volume-rendering pipeline* [4].

- **Prechádzanie dátami** – na začiatku sú určené pozície vzorkovania naprieč objemom. Tieto slúžia ako základ pre diskretizáciu integrálu objemového zobrazovania.

- **Interpolácia** – určené pozície vzorkovania dát tvoria pravidelnú mriežku a musia byť pri vizualizácii zrekonštruované do spojitých oblastí. Najčastejšie sa na tento proces používa trilineárna interpolácia.
- **Výpočet gradientov** – gradient skalárnej mriežky je využívaný na výpočet lokálneho osvetlenia.
- **Klasifikácia** – klasifikácia mapuje hodnoty objemových dát na optické vlastnosti voxelov. Obyčajne je implementovaná pomocou prenosových funkcií.
- **Tieňovanie a osvetlenie** – tieňovanie objemového modelu sa dosahuje pridaním koeficientu do výrazu vyžarovania svetla v rámci integrálu objemového zobrazovania. Efekty osvetlenia sú simulované za pomoci gradientovej mapy.
- **Kompozícia** – kompozícia je základom pre postupný výpočet diskretizovaného integrálu objemového zobrazovania. Kompozičná rovnica závisí od poradia prechádzania objemového priestoru, pričom môže mať dva tvary, buď pri prechádzaní smerom od pozorovateľa alebo naopak.

Fázy interpolácie, výpočtu gradientov, tieňovania a klasifikácie sú uplatňované pri spracovávaní konkrétneho bodu vzorkovania, čiže ich možno aplikovať zvyčajne nezávisle od celkovej techniky vykresľovania.



Obr. 7 Výsledok vizualizácie objemových dát rôznymi spôsobmi – zľava doprava priame zobrazovanie, nepriame povrchové zobrazovanie, a priame zobrazovanie s aplikovaným tieňovaním [3].

3.3.2 Nepriame zobrazovanie

Techniky nepriameho zobrazovania pracujú na báze extrakcie povrchu (Obr. 7). Algoritmy vizualizujúce povrchy vytvárajú zo vstupných dát pomocnú geometrickú štruktúru, ktorou je tento povrch objektu vyjadrený. Za povrch sa pri objemových dátach považuje plocha spájajúca body s určitými rovnakými vlastnosťami respektíve hodnotami, nazývaná aj izopovrch. Na vyjadrenie tohto povrchu slúžia jednoduchšie geometrické primitívy,

najčastejšie trojuholníky. Pre prevod dát z formy skalárnej mriežky do povrchovej reprezentácie s väčšou alebo menšou presnosťou sa používajú algoritmy ako kontúrová detekcia alebo „Marching Cubes“ [3]. S takouto reprezentáciou je možné následne narábať tradičnými rozšírenými metódami vizualizácie, natívnymi pre hardvérové grafické akcelerátory. Podstatnou nevýhodou techník povrchového zobrazenia je strata významnej časti informácií z pôvodných objemových dát a nemožnosť korektne vizualizovať objekty s nejasnými hranicami ako objekty simulujúce hmlu alebo napríklad oblaky.

3.3.3 Priame zobrazovanie

Techniky priameho zobrazovania vizualizujú všetky relevantné objemové dáta naprieč objemovým priestorom. Umožnené je tak zobrazovanie napríklad aj objektov s rôznou mierou priehľadnosti a vnútornej štruktúry (Obr. 7), alebo tiež filtrácia rozsahu priehľadnosti a intenzity v objemových dátach.

Techniky priameho vykresľovania možno kategorizovať do dvoch hlavných smerov [4]. Obrazovo orientované techniky vychádzajú z dvojrozsmernej plochy, predstavujúcej zobrazovaciu oblasť, kde sú umiestnené štartovacie body pre prechádzanie priestoru objemového modelu. Objektovo orientované techniky zase používajú určitú schému na preskenovanie priestoru objemového modelu. Následne sú spracované časti zobrazené do vykresľovaného priestoru.

Objektovo orientované techniky sú založené na priestorových rezoch objemového priestoru. Trojdimenzionálny priestor sa tak delí na dvojrozmerné vrstvy, na ktoré sa následne vzorkujú voxelové dáta. Vrstvy sú finálne zobrazované zmiešavaním podľa adekvátnej orientácie späť do jednoduchého zobrazovacieho priestoru podľa kompozičnej schémy. Tento spôsob je podporovaný aj tradičnými vykresľovacími metódami grafických akceleratorov, keďže vrstvy objemových dát môžu byť reprezentované textúrami. Vtedy sa ťaží hlavne z podpory natívnej interpolácie textúr, nevýhodou je ale pri previazanosti na funkcie grafického hardvéru menšia flexibilita algoritmov. V rámci takéhoto použitia existujú dva prístupy technického charakteru a to použitie:

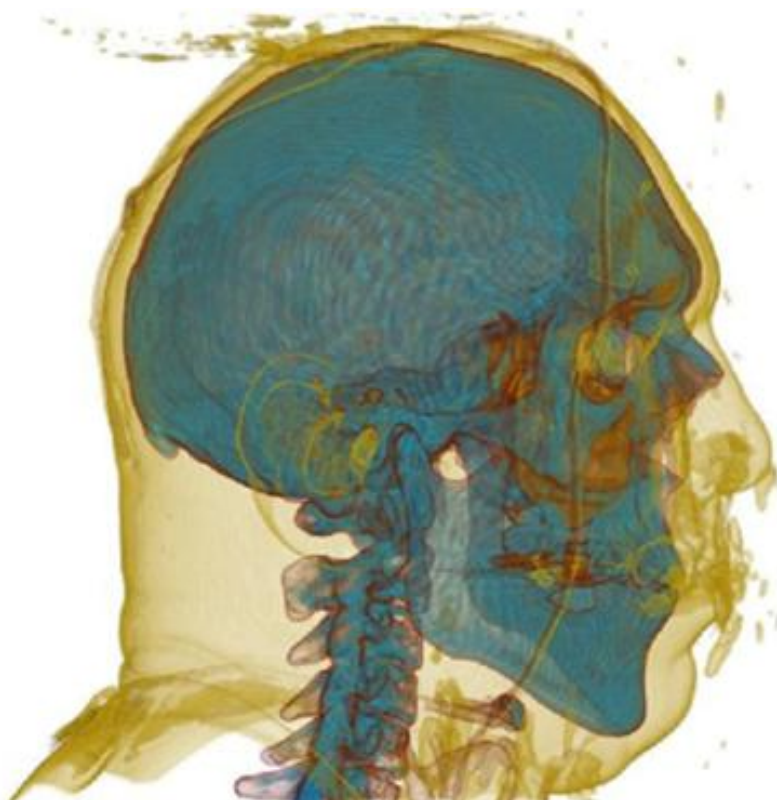
- 2D textúr – rezy sú orientované podľa objektu. Prístup je menej náročný na výkon, ale je použitá menej kvalitná interpolácia a nastávajú artefakty pri zobrazovaní v dôsledku rôznych vzorkovacích vzdialeností v závislosti od uhla pozorovania.
- 3D textúry – rezy sú pohľadovo orientované. Vylepšenie oproti predchádzajúcemu prístupu v rámci interpolácie a odstránenia artefaktov, nevýhodou je o niečo vyššia náročnosť na výkon a menšia dostupnosť z hľadiska podpory akcelerácie.

Obrazovo orientované techniky spočívajú v prístupe, pri ktorom sú východiskom vizualizácie objemových dát jednotlivé zobrazované body výsledného obrazu. Prechádzanie objemových dát v takomto prípade prebieha smerom od pozorovateľa scény naprieč objemovým modelom, alebo naopak. Tento prístup nepočíta priamo s využitím tradičnej funkcionality grafického hardvéru a je preto flexibilnejší v zmysle ľahšej modifikovateľnosti a napríklad

aplikovateľnosti optimalizačných techník. Typickým zástupcom tejto kategórie je algoritmus vysielania lúčov (ray casting), detailnejšie opísaný v nasledujúcej podkapitole.

3.3.4 Klasifikácia a prenosová funkcia

Okrem explicitnej vizualizácie hustoty objemových dát v priestore, môžeme vizualizovať aj kategorizované časti dát za pomoci farebného odlišenia a priehľadnosti (Obr. 8). Klasifikácia týchto častí býva pri objemových dátach definovaná prenosovou funkciou (*transfer function*). Prenosová funkcia transformuje skalárne dáta voxelu na jeho optické parametre. Výsledkom je pre každý objemový element namapovaná štvorzložková farba (vrátane alfa kanálu priehľadnosti) a absorpčný a emitačný koeficient. Pri vizualizácii s použitím prenosovej funkcie je potom voxel s konkrétnou hodnotou vykreslený namapovanou farbou a priehľadnosťou, pričom farebné prechody medzi jednotlivými voxelmi sú interpolované pre plynulejšie a prirodzenejšie zobrazenie. Táto interpolácia môže byť vykonaná pred alebo po klasifikácii, čo má dopad na kvalitu zobrazenia klasifikovaných oblastí. Prenosová funkcia pre špecifické dáta môže byť vytvorená manuálne empirickým zisťovaním, kedy sa pre každú hodnotu voxelu staticky špecifikuje RGBA hodnota, alebo vizuálnym dizajnom s pomocou nástrojov, kedy sa upravujú parametre funkcie na základe zobrazeného výsledku v reálnom čase. Konkrétnym príkladom klasifikácie je odlišenie materiálu kosti a kože na tomografickej snímke – zodpovedajúca prenosová funkcia je na Obr. 9.



Obr.8 Aplikácia klasifikácie na rôzne materiály zachytené objemovými dátami [4].



Obr. 9 Príklad prenosovej funkcie implementovanej v jednorozmernej textúre, úseky reprezentujú materiály kože a kostí.

3.4 Ray casting

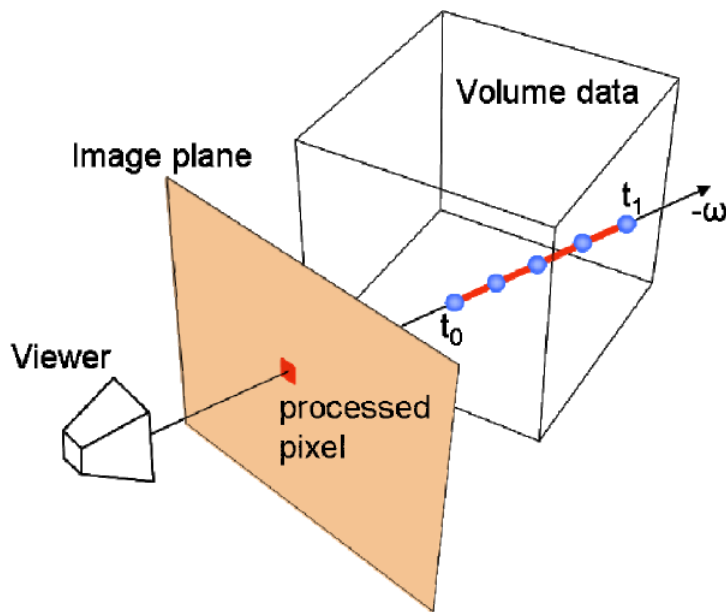
Všeobecná metóda vizualizácie vrhaním lúčov (*ray casting* alebo *ray marching*) spočíva v princípe pozorovateľa, plochy predstavujúcej displej a sledovanej scény, ktorá sa celá nachádza za touto plochou [5]. Z bodu pozorovateľa sa cez plochu vysielajú lúče, a sleduje sa, ako tieto lúče interagujú s objektmi na scéne. Takto sa premieta obsah scény na každý bod zobrazovacej plochy prostredníctvom jedného lúča s využitím kompozície.

Pri objemovom zobrazovaní je tento princíp zachovaný, pričom lúč postupuje cez objemové dáta a interaguje s navzorkovanými voxelmi (Obr. 10). Konkrétnejšie možno postupnosť vykonávania pri metóde *ray castingu* charakterizovať nasledujúcou schémou:

1. Vygenerovanie parametrov lúčov pre body zobrazenia.
2. Pre každý vyslaný lúč vzorkovací cyklus:
 - a. Pokiaľ sa nachádzame vnútri priestoru objemových dát:
 - i. Načítanie objemových dát.
 - ii. Namapovanie farby a priehľadnosti na načítané dáta.
 - iii. Akumulovanie farby.
 - b. Zapísanie farby pre daný lúč do výsledného buffera zobrazenia.

Pri generovaní môžeme každý vyslaný lúč charakterizovať ako skalárne súradnice východzieho bodu a prislúchajúci vektor smerovania. Pre vizualizáciu potrebujeme také parametre lúčov, ktoré majú počiatok v bode pozorovateľa (alebo kamery) a pretínajú oblasť objemových dát. Oblasť objemových dát predstavuje spravidla priestor v tvare kvádra, pri ktorom sa dajú vypočítať body, kde ho lúče pretínajú, a tým určiť ich smerový vektor.

Objemové dáta sú následne iteratívne vzorkované v rámci určeného počtu krokov, v bodoch oddelených konštantnou dĺžkou pozdĺž konkrétneho lúča, pričom prostredníctvom kompozície dostávame výslednú farbu obrazového bodu vizualizácie. V rámci kompozície sa akumulujú hodnoty farby a priehľadnosti tak, že nová komponovaná farba predstavuje farbu na aktuálnej pozícii váhovanú koeficientom priehľadnosti, zmiešanú s doteraz naakumulovanou farbou. Lúč môže prechádzať objemovými dátami dvoma smermi – spredu dozadu a naopak, kedy sa aplikuje upravená kompozícia.



Obr. 10 Princíp ray castingu [5].

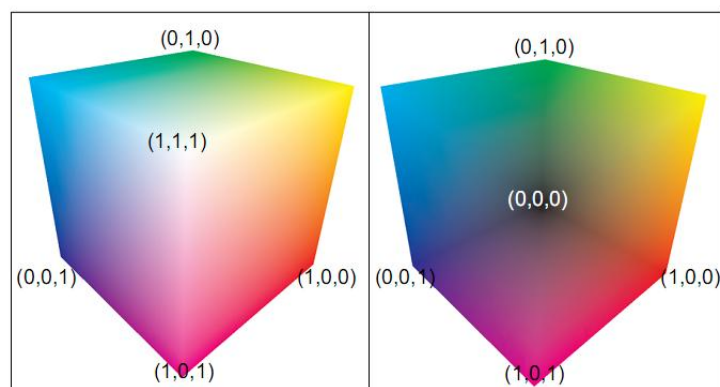
3.4.1 Optimalizačné techniky

Rýchlosť vykonávania algoritmu *ray castingu* možno akcelerovať niekoľkými vyvinutými technikami [5,6]:

- **Adaptívne vzorkovanie obrazového priestoru** – Pri generovaní lúčov nie je potrebné vytvoriť lúč pre každý zobrazovaný bod. Namiesto toho sa priestor obrazových bodov rozdelí na regióny o konštantnej veľkosti a pre každý sa vyšle jeden skúšobný lúč, ktorý určuje na základe akumulovaných objemových dát dôležitosť daného regiónu podľa určenej klasifikácie. Následne môže byť región zobrazovaný lúčmi s nižším rozlíšením vzorkovania.
- **Preskakovanie prázdneho priestoru alebo nerelevantných dát** – V praxi pri vizualizácii iba 5 až 10 percent objemových dát prispieva k výslednému obrazu modelu. Pre optimalizáciu akcie prechádzania objemových dát sa teda nemusia spracovávať všetky, pričom rozhodujúcim faktorom je prenosová funkcia. Pri vynechávaní nepodstatných objemových dát je možné do paralelnej štruktúry ukladať koordináty prázdneho priestoru okolo relevantných dát, ktorý sa neskôr implicitne preskakuje. Používané sú pokročilejšie stromové štruktúry, ktoré rozdeľujú priestor na základe maximálnych a minimálnych hodnôt dát podľa oblastí, napríklad blokov. Rovnako sa dajú ďalším prístupom ukladať pozície vstupných bodov lúčov do relevantnej časti objemových dát.
- **Pamäťové usporiadanie a manažment dát** – Optimalizovať je možné aj rýchlosť prístupu k objemovým dátam. Objemové dáta všeobecne vyžadujú vysoké nároky na pamäťové parametre, a pri spracovávaní generujú nesequenčný, roztrúsený prístup k pamäti. To má za následok spomalenie napríklad aj v dôsledku eliminácie cacheovania blokov dát. Preto sa používa spôsob načítavania objemových dát do pamäte

odlišný od lineárneho. Je nepravdepodobné že lúč bude pretínať voxely v dátach pri takomto načítaní susediace. Preusporiadaním dát do blokového rozloženia sa dosiahne to, že pamäť bude cache-ovať dáta, ktoré sa s väčšou pravdepodobnosťou nachádzajú pozdĺž lúča, čo povedie k zvýšenému výkonu z hľadiska pristupovania k pamäti.

- **Obmedzenie tieňovania** – Pri mapovaní farby, je tiež niekedy vhodné preskočiť fázu tieňovania, ktorá môže znamenať značné spomalenie. Táto praktika je aplikovateľná aj selektívne kedy sa preskakuje tieňovanie vzoriek s vysokou priehľadnosťou, ktoré nemajú veľký dopad na výsledne zobrazenie.
- **Skoré ukončovanie lúčov** – Veľmi efektívnou technikou je zastavenie akumulovania dát pri kompozícii, ak je hodnota farby pre lúč dostatočne satureovaná. Ďalšou podmienkou môže byť dostatočne nízka hodnota momentálnej priehľadnosti.
- **Zníženie rozlíšenia vizualizácie** – pri dátach, ktoré majú nižšiu frekvenciu, napríklad objekt simulujúci hmlu, môžeme vykreslovať model do *buffera* s polovičnou veľkosťou, a na konci procesu pri zobrazení priradiť tomuto *bufferu* dvojnásobnú škálu, pričom dopad na kvalitu v závislosti od dát nie je významný.
- **Efektívna metóda na určenie počiatočných bodov a smerových vektorov** – Namiesto vypočítavania priesečníkov s kvádom objemového priestoru sa využíva funkcionálna grafického hardvéru na spracovanie textúr. Do dvojrozmernej textúry sa vykreslí kváder ohraničujúci objemové dáta s tým, že na predné steny je aplikované mapovanie, kde konkrétne farebné body reprezentujú koordináty bodu steny kvádra v priestore (Obr. 11). Koordináty sú uložené v textúre prostredníctvom jednotlivých farebných zložiek daného bodu. Podobným spôsobom sa vygeneruje textúra so zadnými stenami kvádra. Následne sa odpočítaním textúry zadnej časti kocky od prednej pri danom uhle získa normalizovaný smerový vektor pre každý zobrazovaný bod predstavujúci smer lúča. Začiatkový bod lúča v objemovom priestore vyjadrujú body na prvej textúre kvádra.



Obr. 11 Textúry predných a zadných stien slúžiace na efektívne určenie parametrov lúčov [6].

4 Opis riešenia

4.1 Špecifikácia

V rámci praktickej aplikácie bude implementované riešenie v podobe programu na vizualizáciu objemových dát na grafickom procesore. Použitím programovacím rozhraním bude CUDA, pričom grafický výstup programu bude zabezpečený cez rozhranie OpenGL. Implementovaná bude technika zobrazovania *ray-casting*, s aplikovanými niekoľkými optimalizačnými technikami. Pri vizualizovaní objemových dát bude zabezpečená interaktivita prehliadania modelu, respektíve animovanie výsledného zobrazenia. Pomocou vhodného používateľského rozhrania bude zabezpečená aj parametrizácia vizualizácie prostredníctvom výberu rôznych optimalizačných techník. Aplikácia bude testovaná na súbore voľne dostupných objemových dát. Pre účely porovnania efektivity implementácie algoritmov medzi implementáciou na grafickej karte a tradičným procesorom bude umožnené spustiť beh kódu prostredníctvom dostupnej emulácie CUDA, ktorá využíva tradičné procesory. Medzi rozširujúce požiadavky špecifikácie patrí implementovanie použitia prenosovej funkcie a interaktívna manipulácia s ňou.

5 Zoznam bibliografických odkazov

1. David B. Kirk, Wen-mei W. Hwu. 2010. *Programming Massively Parallel Processors: A Hands-On Approach* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
2. *NVIDIA CUDA C Programming Guide : Version 3.2* [online]. 9.11.2010 [cit. 20.4.2011]. Santa Clara, CA, 2010. Dostupné z: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf.
3. Michal Červeňanský. *Volume Rendering* [online]. 31.3.2010 [cit. 1.5.2011]. Dostupné z: http://www.sccg.sk/~cervenansky/rtr/pr6_vr.pdf
4. Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, et al.. 2006. *Real-Time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA.
5. Marsalek, L., Hauber, A., Slusallek, P. 2008. High-speed volume ray casting with CUDA. In *IEEE Symposium on Interactive Ray Tracing*. Saarland Univ., Saarbrücken, p 185 – 185.
6. J. Kruger and R. Westermann. 2003. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (VIS '03). IEEE Computer Society, Washington, DC, USA, 38-.