Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Informačné systémy

Bc. Miroslav Beno

Vizualizácia objemových dát pomocou grafického procesora

Diplomový projekt II

Vedúci diplomového projektu: Ing. Ondrej Hirjak

December 2011

ANOTÁCIA

Slovenská Technická Univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Študijný program:	**INFORMAČNÉ SYSTÉMY**
Autor:	**Bc. Miroslav Beno**
Diplomový projekt:	**Vizualizácia objemových dát pomocou grafického procesora**
Vedenie diplomového projektu:	Ing. Ondrej Hirjak
Rok odovzdania:	december 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 diskrétnych 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:	December 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

Grafický procesor (GPU) na dnešných bežne dostupných grafických kartách sa postupne vyvinul vo veľmi výkonnú a flexibilnú výpočtovú jednotku. Posledné generácie grafických akcelerátorov 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 akcelerátorov, 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 rozhraniam 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 diskrétnych 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 eminentná. 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 skrátene 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. Samozrejmým 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 akcelerátorom 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.

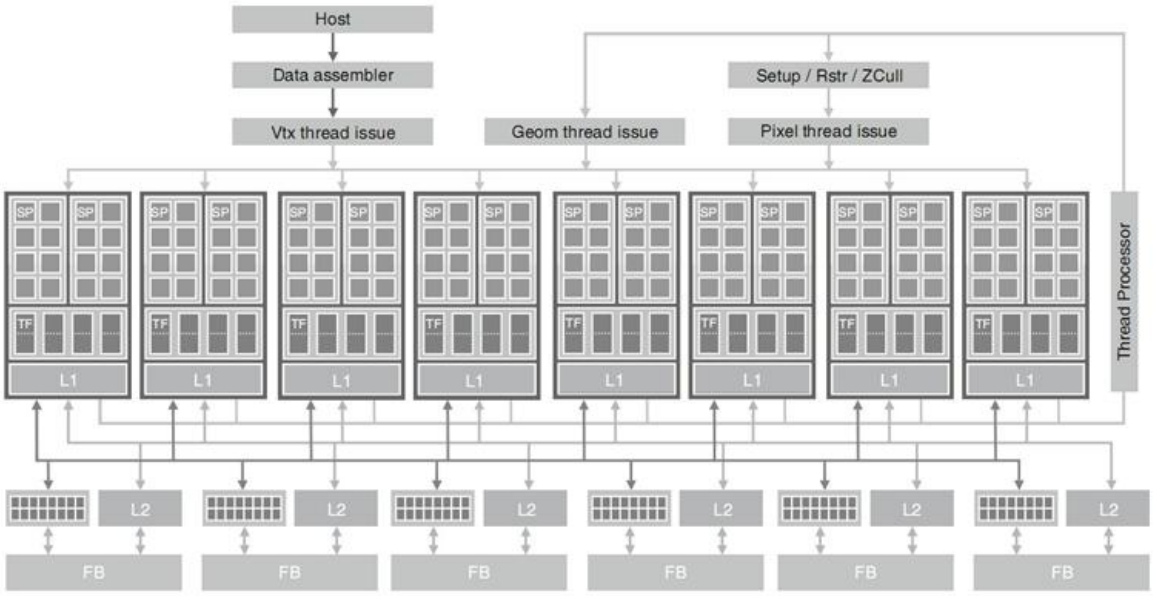
V prvej ére spotrebných grafických akcelerátorov, počas 90-tych rokov, pracoval grafický hardvér na základe pevnej množiny funkcií spracovania a vykresľovania grafických dát, ktoré mali rôzne konfigurovateľné parametre, ale nebolo možné programovať nové funkcie. Išlo o takzvanú *fixed-function graphic 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 funkcionalitu. 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 sprístuňoval 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 množstvo nástrojov podporujúcich efektivitu vývoja.

Tieto jazyky však stále neboli priamo určené na implementovanie všeobecných výpočtov, jednalo sa o umožnenie väčšej flexibility 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).



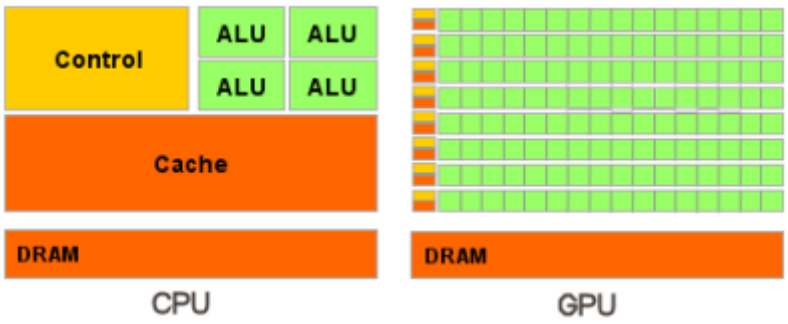
Obr. 1 Ukážka *graphic pipeline* s polom unifikovaných procesorov [1].

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 exituje 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 skrátene 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 grafický 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 pamäťovom modeli. Pri programovacom jazyku medzi obmedzenia patrí napríklad nepoužívanie ukazovateľov na funkcie, rekurzie alebo 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, 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.

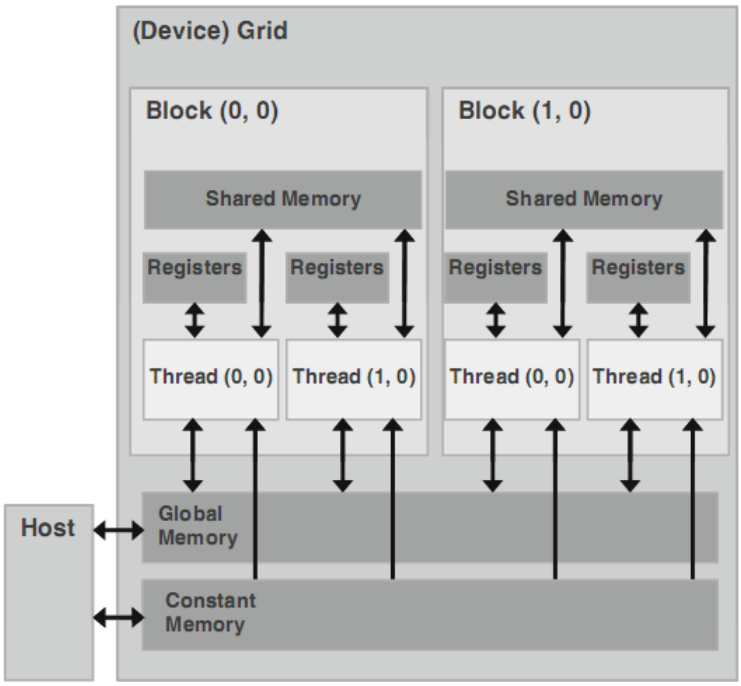
- *__global__* – označuje *kernel* funkciu, ktorá môže byť zavolaná z *host* časti programu, a jej telo sa vykonáva na *device* v podobe vytvorenia paralelných vlákien.
- *__device__* – označuje funkciu, ktorá môže byť volaná iba z inej *kernel* alebo *__device__* funkcie, vykonávaná je na *device*.
- *__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 bežiace 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. 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 prístupovo logicky naviazané na konkrétne jedno vlákno a používajú sa na uchovávanie často dopytovaných 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* na operácie zápisu aj čítania, device k nej pristupuje 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 využívanej 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 *cache-ovaniu*. Kľúčové slovo *__device__* označuje globálnu premennú, uloženú v globálnej pamäti. Používa sa na uchovávanie vstupných a výstupných dát pred a po behu

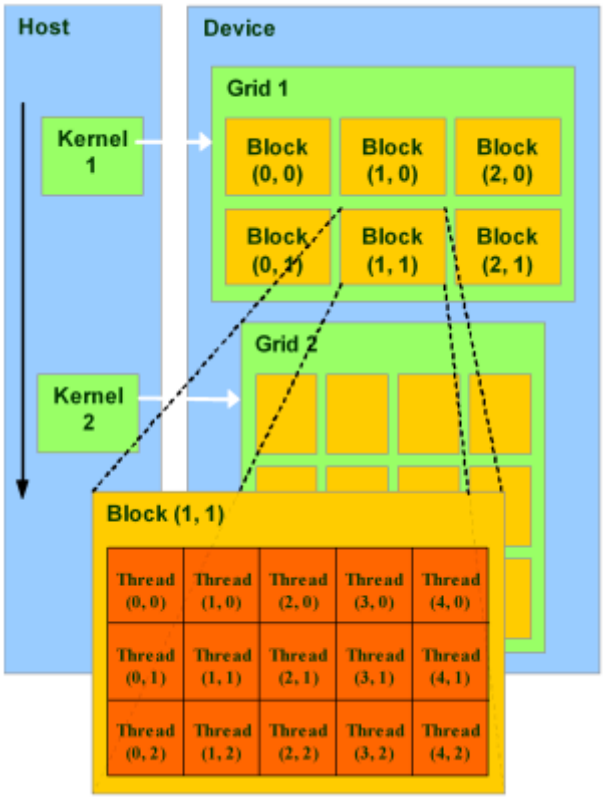
kernel funkcie, nakoľko prístup k nej z blokov bežiacich vlákien je zo spomínaných typov najpomalší a nie je synchronizovaný.

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 multi-dimenzionálnych 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 dvojrozmerne 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 trojrozmerne 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 bariérový 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 prideľuje 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 modelu je možné vidieť na Obr. 5.

```
__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;
}
```

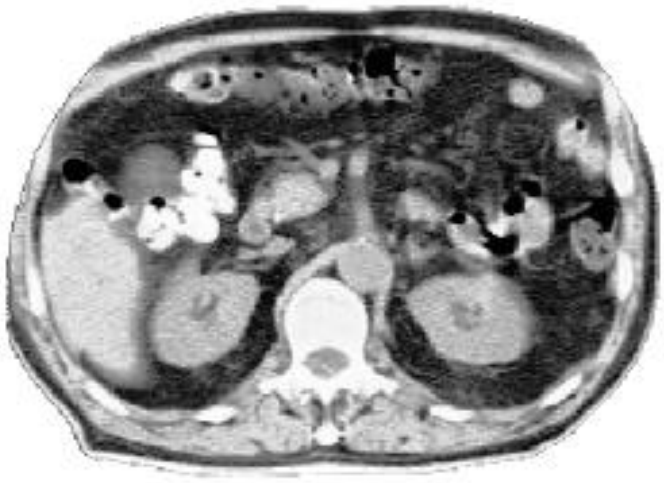
```
int main(int argc, char** argv)
{
  unsigned int size_A = 3 * 3;
  unsigned int mem_size_A = sizeof(float) * size_A;
  float* h_A = (float*) malloc(mem_size_A);
  unsigned int size_B = 3 * 3;
  unsigned int mem_size_B = sizeof(float) * size_B;
  float* h_B = (float*) malloc(mem_size_B);
  unsigned 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);
  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);
}
```

Obr. 5 Ukážka elementárneho CUDA programu – násobenie dvoch matíc (*kernel* funkcia vľavo).

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 snímače a počítač pre vytvorenie obrazu priečneho 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. Tvoria 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, intezintu alebo farby 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 gigabajtoch.

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 zložitejšie 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 (emisia) a rozptyl svetla. Pri objemových dátach je najčastejšie uvažovaný absorpčno-emitačný 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:

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

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

Algoritmy objemovej vizualizácie sa typicky implementujú optický model pomocou diskrétnej 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:

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

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

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.

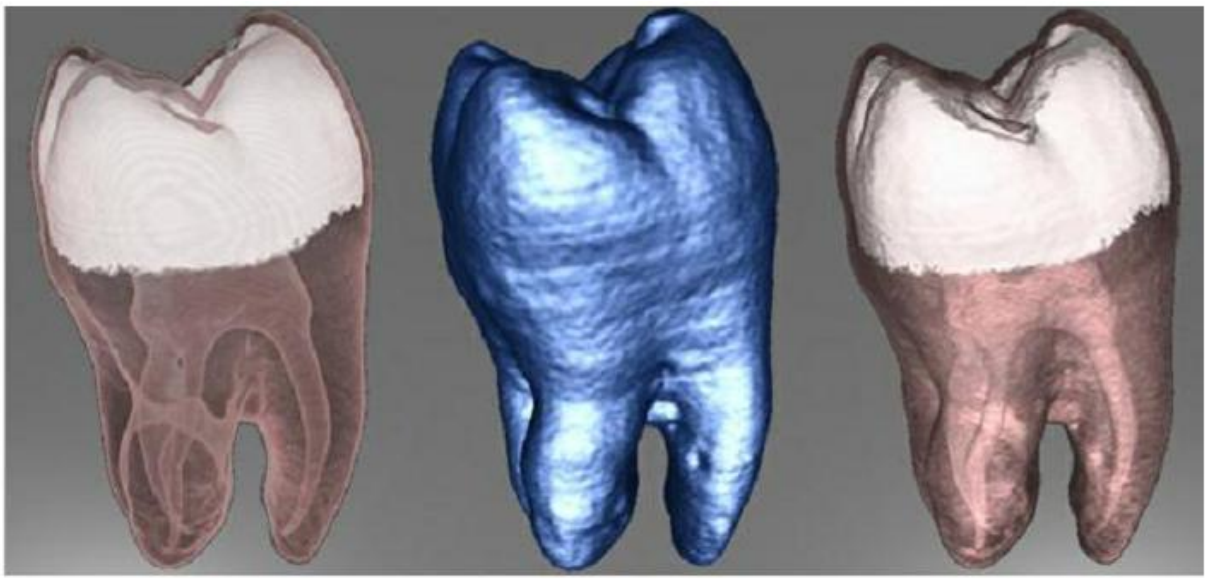
3.3.1 Všeobecné fázy vizualizácie objemových dát

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 zobrazenie, nepriame povrchové zobrazenie, a priame zobrazenie 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 dvojrozmernej 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 spätne do jednoduchého zobrazovacieho priestoru podľa kompozičnej schémy. Tento spôsob je podporovaný aj tradičnými vykresľovacími metódami grafických akcelerátorov, 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 flexibilnosť algoritmov. V rámci takéhoto použitia existujú dva prístupy technického charakteru a to použitie:

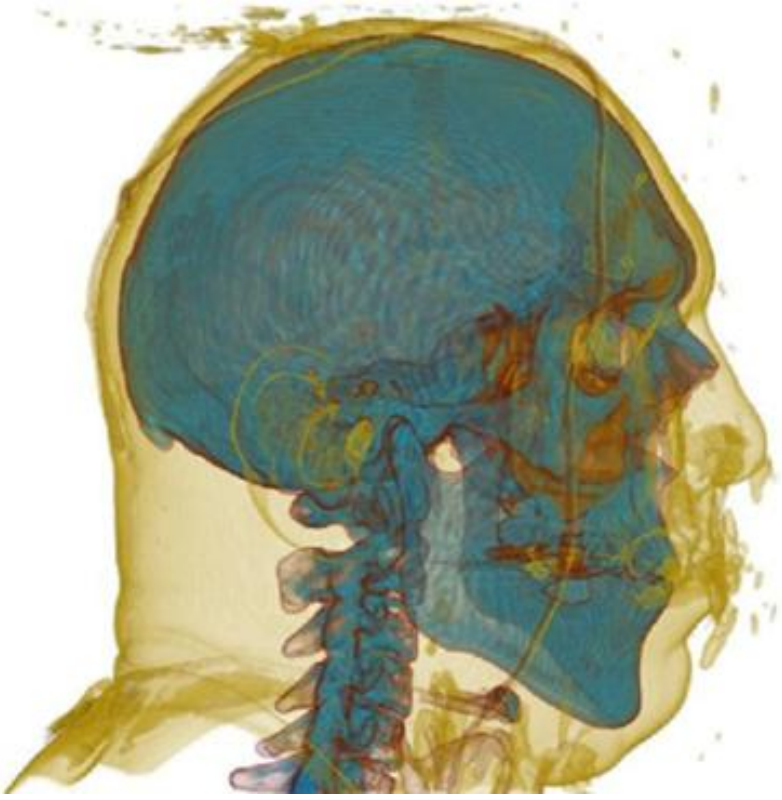
- 2D textúr – rezy sú orientované podľa objektu. Prístup je menej náročný na výkon, ale je použitá menej kvalitná interpolácia a nastávajú artefakty pri zobrazovaní v dôsledku rôznych vzorkovacích vzdialeností v závislosti od uhla pozorovania.
- 3D textúry – rezy sú pohľadovo orientované. Vylepšenie 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 podkapitole 3.4.

3.3.4 Klasifikácia a prenosová funkcia

Okrem explicitnej vizualizácie hustoty objemových dát v priestore, môžeme vizualizovať aj kategorizované časti dát za pomoci farebného odlíšenia 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 hodnotu 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 empirické 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 odlíšenie 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

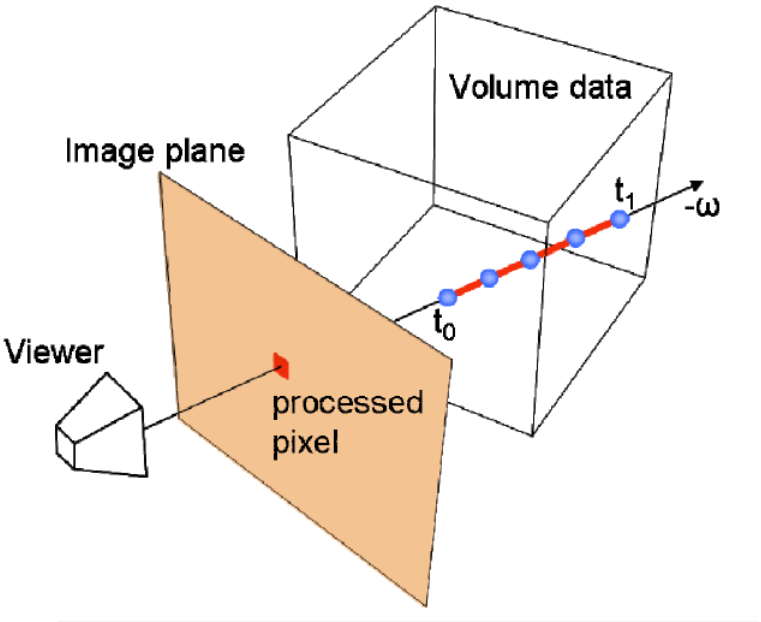
Všeobecná metóda vizualizácie vrhaním lúčov (*ray casting* alebo *ray marching*) spočíva v princípe pozorovateľa, plochy predstavujúcej displej a sledovanej scény, ktorá sa celá nachádza za touto plochou [5]. Z bodu pozorovateľa sa cez plochu vysielajú lúče, a sleduje sa, ako tieto lúče interagujú s 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 (bližšie opísané v kapitole 3.2). Lúč môže prechádzať objemovými dátami dvoma smermi – spredu dozadu a naopak, kedy sa aplikuje upravená kompozícia.



Obr. 10 Princíp *ray castingu* [5].

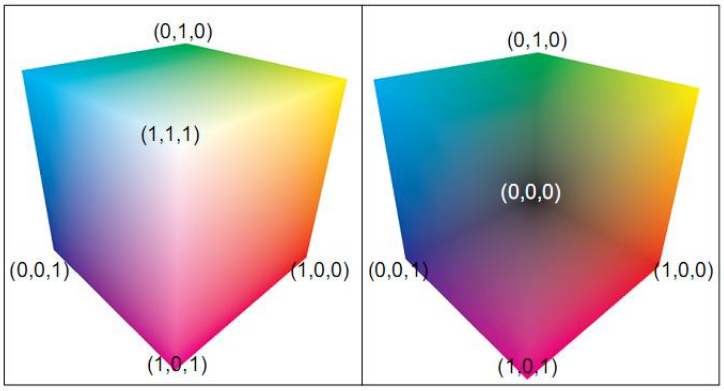
3.4.1 Optimalizačné techniky

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 nesekvenčný, roztrúsený prístup k pamäti. To má za následok spomalenie napríklad aj v dôsledku eliminácie cache-ovania 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 saturovaná. Ďalšou podmienkou môže byť dostatočne nízka hodnota momentálnej priehľadnosti.
- **Zníženie rozlíšenia vizualizácie** – pri dátach, ktoré majú nižšiu frekvenciu, napríklad objekt simulujúci hmlu, môžeme vykresľovať 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 kvádrom objemového priestoru sa využíva funkcionalita 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. Začiatočný 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 Návrh riešenia

Cieľom riešenia bude preveriť súčasné možnosti použitia grafického procesora na iné ako tradičné grafické zobrazenie, a to konkrétne na algoritme priamej vizualizácie objemových dát. Výsledkom by mal byť prehľad o aktuálnych charakteristikách technológií grafického procesora a jeho aplikačného rozhrania pričom budú preverené výkonnostné parametre vo vzťahu k použitým algoritmom vizualizácie. Predmetom skúmania bude interaktívne zobrazenie objemových dát spolu s hraničnými vlastnosťami zobrazovania, s ktorými je možné interaktivitu dosiahnuť a taktiež to, ktoré technológie grafického hardvéru majú najväčší dopad na efektivitu takéhoto zobrazovania. Vizualizačný algoritmus by mal pre relevantnosť výsledkov pracovať s najrozšírenejšími optimalizačnými technikami v doméne objemového zobrazovania. Meranými veličinami pre overenie riešenia budú predovšetkým čas a kvalita vizualizácie, prípadne vzťah medzi nimi pri použití rôznych parametrov.

Cieľ riešenia bude realizovaný implementáciou aplikácie spĺňajúcej požiadavky špecifikované v nasledujúcej podkapitole. Spolu s touto špecifikáciou bolo potrebné zvoliť aj konkrétne programovacie rozhranie grafických procesorov a kvôli zachovaniu rozumného rozsahu riešenia aj typ vizualizačného algoritmu. Napríklad architektúru aplikácie je totiž vhodné prispôsobiť týmto voľbám, pre zvýšenie efektivity a použiteľnosti výsledku. Ako vyplynulo počas analýzy problematiky vhodným výberom rozhrania je technológia **CUDA**. Umožňuje jednoduchú inicializáciu kernelov, niektoré výhodné pamäťové mechanizmy špecifické pre grafické karty a disponuje aj pokročilejšou výbavou na podporu vývoja. Nevýhodou je len viazanie sa na konkrétnu hardvérovú platformu, aj tú však možno hodnotiť relatívne, keďže na druhej strane robustné OpenCL nemôže v dôsledku podpory širokého spektra hardvéru vo väčšej miere optimalizovať výkonnosť paralelného behu programov. Z hľadiska vizualizačného algoritmu bol zvolený **Ray casting** z dôvodu dobrých vlastností na paralelizáciu, potrebných pre implementáciu na grafickom procesore.

4.1 Špecifikácia požiadaviek

V rámci praktickej aplikácie bude implementované riešenie v podobe softvéru na vizualizáciu objemových dát na grafickom procesore. Jeho základné charakteristiky v podobe funkcionálnych a nefunkcionálnych požiadaviek sú obsiahnuté v niekoľkých bodoch alebo kategóriách špecifikácie. Tieto sú označené identifikátorom pre neskoršie referencovanie, hlavne v kapitole návrhu riešenia.

P1: Vizualizácia objemových dát – Základnou požiadavkou je elementárne zobrazenie objemových dát projekciou trojrozmerného priestoru a modelu v ňom umiestneného z rôznych strán a úrovní priblíženia. Vizualizačný algoritmus, ktorý toto zobrazenie realizuje, by mal byť dostatočne netriviálny, tak aby bolo možné skúmať viaceré jeho varianty a ich vplyv na kvalitu zobrazenia a zároveň aby vizualizácia predstavovala pomerne hodnovernú reprezentáciu zobrazovaného objektu. Pri algoritme sa nepredpokladá použitie pokročilejších

techník skvalitňujúcich alebo modifikujúcich vizualizáciu ako náročnejšie tieňovanie alebo odstraňovanie minoritných vizualizačných artefaktov.

P2: Interaktivita – Kľúčom pri získavaní informácií a celkového pohľadu na zobrazovaný trojrozmerný objemový objekt je interaktivita vizualizácie. Napríklad spomínaná možnosť pozorovať objekt z viacerých uhlov dodáva vizualizácii objektu informáciu o jeho skutočnom trojrozmernom tvare avšak táto sa stáva pri neinteraktívnom zobrazení nekonzistentnou a ťažšie interpretovateľnou. Je preto potrebné, aby používateľ mohol interagovať s vizualizovaným modelom prostredníctvom vstupných rozhraní ako myš a klávesnica a aby bola zabezpečená možnosť rotácie, translácie a priblíženia modelu vo virtuálnom priestore. S interaktivitou zobrazenia súvisí tiež požiadavka na použitie takých technológií a algoritmov, ktoré zabezpečia minimálne pri vhodných parametroch dostatočne krátky čas vizualizácie na to, aby relatívne plynule a s malou odozvou reagoval na interakciu používateľa.

P3: Viaceré technológie a algoritmy výpočtu – Jedným z hlavných prípadov použitia špecifikovanej aplikácie je demonštrácia rozdielov výkonu vizualizácie na rôznych hardvérových platformách – tradičný procesor a grafický multi-procesor, a s použitím viacerých technológií v rámci týchto platforiem. Je preto potrebné v rámci aplikácie implementovať spomínaný vizualizačný algoritmus niekoľkokrát s použitím daných technológií, pričom je nutné zaručiť čo najmenšie odchýlky v samotnom vykonávaní algoritmu pre možnosť komplexného porovnania. Používateľ by mal mať tiež možnosť prepínať medzi jednotlivými implementáciami a pozorovať a konfrontovať vlastnosti vizualizácie najlepšie počas jedného behu aplikácie.

P4: Parametrizácia – Dôležitou vlastnosťou aplikácie je jej parametrizácia, a teda možnosť prispôsobenia jej správania. Konkrétne by malo byť možné zadávať rôzne parametre, ktoré ovplyvňujú samotný vizualizačný algoritmus, parametre rozhodujúce pre rôzne typy implementácií tohto algoritmu, mód spustenia aplikácie, alebo informácie o zdroji objemových dát. Pre tento účel je nutné navrhnúť vhodné rozhranie s používateľom zadávajúcim tieto parametre, pričom treba dbať na jeho zrozumiteľný opis a praktické ovládanie (samozrejme s predpokladom o znalosti všeobecných princípoch domény objemovej vizualizácie). Rozhranie by malo tiež komunikovať aktuálny stav a obmedzenia daných parametrov a funkcií.

P5: Bezpečnosť – V rámci aplikovania parametrov treba dbať aj na ošetrenie spracovania neplatných hodnôt a spôsobených výnimočných stavov behu programu tak, aby nevyústili v neúspešné ukončenie ale len chybové hlásenie. Predpokladá sa aj použitie bezpečnostných mechanizmov použitých technológií, pretože napríklad nekorektné narábanie s pamäťovými prostriedkami môže mať vo výpočtovo a dátovo náročnej aplikácii závažné následky ako pád celého systému a podobne.

P6: Merateľnosť – Keďže sa jedná o aplikáciu, ktorej zameraním je prezentovanie a porovnávanie efektívnosti algoritmov a výpočtovej sily hardvérových komponentov, je žiaduce tieto aspekty aplikácie kvantifikovať a merať. Používateľ musí mať možnosť zobraziť

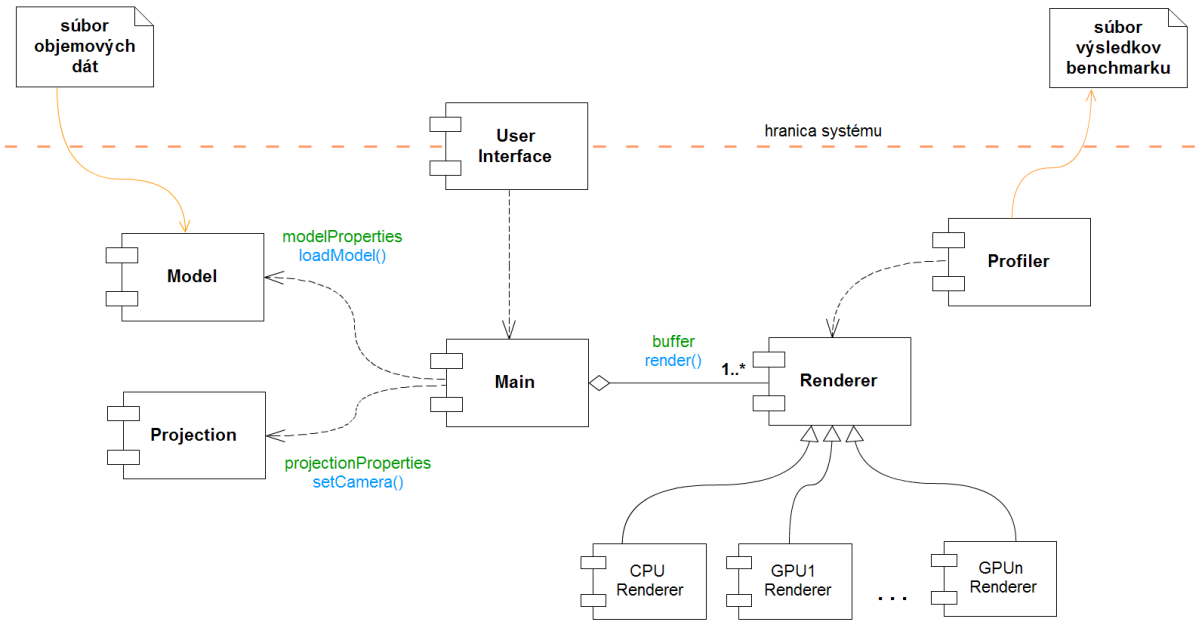
informáciu o momentálnom alebo priemernom výkone vizualizácie so zvolenými parametrami. Tiež by malo byť možné zaznamenať a uchovať tieto informácie normalizovaným spôsobom pre účel agregovania a následného vyhodnotenia, napríklad v podobe výstupného textového súboru s výsledkami. S tým súvisí aj umožnenie behu aplikácie v móde, ktorý v krátkom čase overí jej výkon a vyprodukuje sumárne vyhodnotenie, čím sa uľahčí napríklad porovnávanie výkonu aplikácie naprieč rôznym hardvérom.

P7: Podpora formátov dát – Z hľadiska vstupov do aplikácie by mal byť podporovaný dostatočný rozsah formátov súborov objemových dát s dôrazom kladeným na najrozšírenejšie formáty, v ktorých sú uložené na internete voľne dostupné objemové dáta. Vzhľadom na to, že niektoré formáty neposkytujú metadáta opisujúce špecifiká reprezentácie objemových dát v súbore, je potrebné umožniť pri takýchto dátach zvoliť manuálne zadanie týchto parametrov pred načítaním samotného súboru.

P8: Multiplatformovosť – Pre umožnenie spustenia aplikácie na čo najširšom množstve prostredí je nefunkcionálnou požiadavkou multiplatformovosť riešenia. Aplikácia by mala byť spustiteľná na všetkých operačných systémoch, pre ktoré existujú ovládače sprístupňujúce grafický hardvér a inak vylúčiť použitie platformovo závislých komponentov.

4.2 Architektúra

Návrh architektúry aplikácie vychádza zo špecifikovaných požiadaviek, pričom na jeho charakteristické črty vplýva najmä koncepcia možnosti porovnávania viacerých implementácií algoritmu vizualizácie opísaná v požiadavke *P3*. Pre realizáciu tohto princípu je potrebné identifikovať a oddeliť časti funkcionality, ktoré sú spoločné pre viaceré implementácie vizualizačného algoritmu a zachovávajú konzistentné správanie a stav aplikácie pre každú z týchto implementácií. Jednotlivé implementácie vizualizácie potom vystupujú v aplikácii ako vymeniteľné moduly s rovnakým rozhraním, ktoré možno zapojiť do celkového procesu vizualizácie na pokyn používateľa. Výsledkom tejto modulárnosti by mala byť tiež jednoduchá rozšíriteľnosť pri pridávaní ďalších implementácií algoritmov a minimalizácia duplicitného kódu potrebného pre integráciu každej ďalšej implementácie. Ostatná všeobecná funkcionalita je potom dekomponovaná do logických celkov s vhodnou mierou súdržnosti a granularity. Celkový pohľad na architektúru v podobe diagramu komponentov možno vidieť na Obr. 12. Na diagrame sa okrem navrhovaných komponentov aplikácie a ich vzťahov nachádzajú aj jej vstupy a výstupy, znázornené za hranicou systému oddeľujúcou vnútorné prostredie aplikácie a externé objekty, s ktorými aplikácia prichádza do styku. Zároveň sú tu pre lepšiu názornosť naznačené možné správy a atribúty, predstavujúce rozhranie dôležitejších komponentov.



Obr. 12 Architektúra aplikácie vyjadrená prostredníctvom diagramu komponentov.

Základným komponentom, v ktorom bude sústredená aplikačná logika je komponent *Main*. Okrem úloh ako spracovanie vstupných parametrov aplikácie a korektné ukončenie aplikácie bude spracovávať udalosti z komponentu používateľského rozhrania a delegovať príslušné akcie ďalším komponentom.

Komponenty *Model* a *Projection* budú uchovávať stav vizualizácie nezávisle od samotných vizualizačných komponentov a vykonávať akcie na jeho modifikáciu. Konkrétne komponent *Projection* bude riešiť manipuláciu virtuálnej kamery podľa požiadavky *P2* spolu s príslušnými výpočtami parametrov pre dvojrozmernú projekciu. Komponent *Model* bude obsahovať reprezentáciu objemových dát uložených v pamäti spolu s funkciami pre prístup k tejto reprezentácii a bude zodpovedný za jej načítanie z externého súboru. Jeho súčasťou teda budú aj viaceré funkcie na parsovanie vstupných súborov podľa požiadavky *P7*. Tieto dva komponenty budú komunikovať s hlavným komponentom prostredníctvom rozhraní, ktoré budú sprístupňovať ich funkcionalitu, ale tiež parametre, ktoré uchovávajú. Na Obr. 12 sú načrtnuté príklady ako operácie načítania súboru alebo zmena polohy kamery. Pre získanie parametrov vizualizácie budú komponenty poskytovať atribúty ako napríklad rozmery objektu vo voxeloch a veľkosť načítaných objemových dát pri komponente *Model* alebo parametre o umiestnení projekčnej plochy vo virtuálnom priestore pri komponente *Projection*.

Funkcie používateľského rozhrania bude zoskupovať komponent *User Interface*. Medzi ne patrí vytvorenie samotného okna vizualizácie spolu s definovanými interakciami pre používateľa pri použití myši a klávesnice podľa požiadavky *P2*. Súčasťou tohto komponentu bude tiež umožnenie parametrizácie (požiadavka *P4*). Používateľovi budú zobrazené možné parametre vizualizačného algoritmu, pričom zadávanie hodnôt a stavov bude realizované cez klávesové skratky a/alebo grafické ovládacie prvky, ktoré budú zadefinované v tomto komponente. S tým súvisí aj zobrazovanie povoleného rozsahu parametrov a ich prípadná

verifikácia po zadaní. Okrem toho bude tiež sprístupňovať prepínanie implementácií vizualizačného algoritmu podľa požiadavky *P3*, podobným spôsobom ako pri modifikácii parametrov. Úlohou komponentu bude aj zobrazenie chybových hlásení používateľovi.

Komponent *Renderer* je zovšeobecnením komponentov realizujúcich vizualizačný algoritmus. Definuje potrebné rozhranie na komunikáciu s komponentom *Main*, pričom toto rozhranie budú realizovať konkrétne implementácie algoritmov podľa požiadavky *P1*, v diagrame označené ako *CPU Renderer*, *GPU Renderer* a podobne. Rozhranie bude koncipované tak, že vstupmi pre vizualizačný algoritmus budú parametre vizualizácie z komponentov *Model* a *Projection* a výstupom bude vyrenderovaná vizualizácia uložená v nejakom pamäťovom bufferi určená na zobrazenie. Táto úroveň abstrakcie umožňuje modulárne použiť vizualizačné algoritmy s rôznorodou vnútornou organizáciou, podľa princípu popísaného v úvode podkapitoly (požiadavka *P3*). Zároveň budú v komponente *Renderer* obsiahnuté inicializačné a finalizačné akcie spoločné pre všetky konkrétne implementácie.

Komponent *Profiler* bude slúžiť na meranie výkonu vizualizačných algoritmov a ich zaznamenávanie podľa požiadavky *P6*. Jeho úlohou bude tiež celkové ohodnotenie použitých algoritmov pri špeciálnom takzvanom *benchmarkingovom* móde spustenia aplikácie a zapísanie výsledkov do externého súboru.

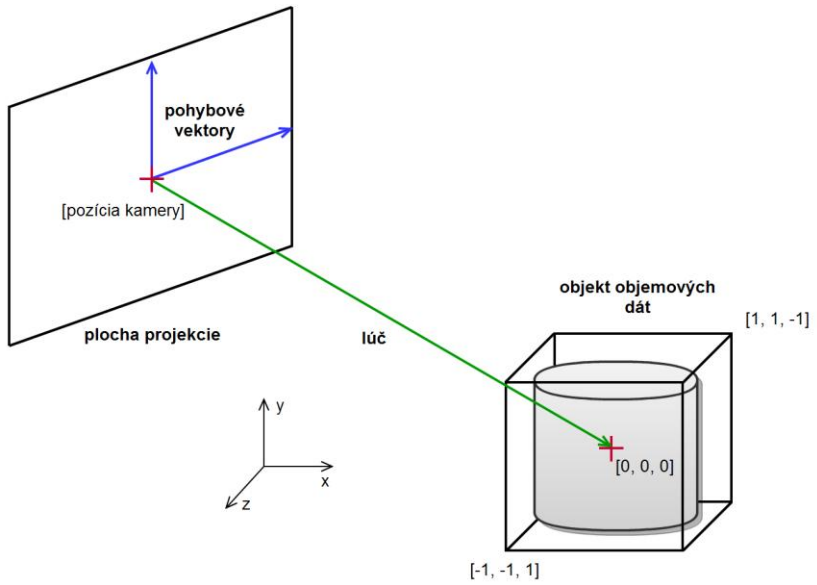
4.3 Vizualizačný algoritmus

Vizualizačný algoritmus objemových dát *ray casting* všeobecne opísaný v kapitole 3.4 môžeme pre potreby návrhu zhrnúť do troch fáz. Prvou je nájdenie zodpovedajúcich lúčov pre každý bod zobrazenia (pixel) v závislosti na projekcii. Ďalej sa vypočítajú priesečníky s kvádrom ohraničujúcim objemové dáta vo virtuálnom priestore. Finálne sa iteratívne vzorkujú a akumulujú hodnoty dát pozdĺž lúča.

4.3.1 Projekcia a parametre lúčov

Objekt objemových dát môžeme ohraničniť kvádrom. Tento kváder umiestnime vo virtuálnom priestore, ktorého projekciu budeme realizovať tak, že jeho stred bude zároveň stredom, respektíve bodom s koordinátmi [0, 0, 0], pravotočivej trojrozmernej sústavy s osami x, y, z (Obr. 13). Pre názornosť predpokladajme situáciu ortogonálnej projekcie, kedy je poloha virtuálnej kamery stredom plochy projekcie (táto vlastne predstavuje to, čo bude zobrazené používateľovi na obrazovke). Spojnicu pozície kamery a stredu objemového objektu predstavuje smer alebo vektor lúča, ktorý použijeme pre vzorkovanie objemových dát. Tiež môžeme predpokladať, že plocha projekcie je vždy kolmá na spojnicu pozície kamery a stredu objemového objektu, to znamená, že používateľ sa pozerá vždy kolmo na objemový objekt. Takto potom môžeme získať pomocou vektorového súčinu pohybové vektory, ktoré nám definujú projekčnú plochu. Pre prvý pohybový vektor môžeme vynásobiť vektor lúča a y-psilonový vektor [0,1,0], čím dostaneme na tieto dva vektory kolmý pohybový

vektor, pre horizontálny pohyb po projekčnej ploche. Ďalšou aplikáciou vektorového násobnie na práve získaný pohybový vektor, a vektor lúča dostávame druhý, vertikálny pohybový vektor. Pomocou danej veľkosti projekčnej plochy potom násobkami pohybových vektorov vieme vypočítať pozíciu pixela projekčnej plochy vo virtuálnom priestore, odkiaľ bude smerovať vzorkovací lúč, pričom vektor smerovania je pri ortogonálnej projekcii rovnaký ako pôvodný lúč medzi stredom projekčnej plochy a stredom objemového objektu. Pri perspektívnej projekcii treba body projekčnej plochy spájať s pozíciou kamery, ktorá je posunutá za projekčnú plochu.



Obr.13 Umiestnenie plochy projekcie a objemového objketu vo virtuálnom priestore.

4.3.2 Nájdenie priesečníkov

Pre návrh spôsobu nájdenia priesečníkov predpokladáme rovnaké umiestenie kvádra objemových dát so stredom v strede sústavy [0,0,0] a maximálnou dĺžkou hrany 2. Najmenší a najväčší hraničný bod vo všetkých dimenziách označíme ako N a M, v prípade rovnakých hrán a teda kocky ohraničujúcej objemový objekt:

$$N = [-1,-1,-1], \qquad M = [1,1,1].$$

Priesečníky získame porovnaním parametrov smerového vektoru pretínajúceho lúča s hodnotami hraničný bodov kvádra. Body lúča L (priamky v priestore) sú definované počiatočným bodom O, a smerovým vektorom D, pričom k je vzdialenosť bodu lúča od bodu O:

$$L(k) = O + kD.$$

Cieľom je získať hodnoty parametrov k_1 a k_2, označujúce body lúča, ktoré patria hranám kvádra. Tieto získame pre každú zložku dimenzie výpočtom:

$$k_1 = N - \frac{O}{D};\ k_2 = M - \frac{O}{D}.$$

Uplatnením prieniku intervalov pre každú dimenziu získavame parametre priesečníkov lúča a kvádru, alebo prázdnu množinu, ktorú interpretujeme tak, že lúč kváder nepretína. Ošetriť treba ešte situáciu, kedy počítame prienik lúčov rovnobežných s niektorou z osí priestoru, kedy je zložka smerového vektoru D rovná nule. Vtedy zistíme, či leží bod O v rozmedzí bodov N a M pre danú dimenziu, a uplatníme buď nulový interval ak nie, a nekonečný interval ak áno.

4.3.3 Akumulácia farby a vzorkovanie

Akumulácia farby a vzorkovanie pozdĺž lúča bude prebiehať podľa algoritmu opísaného v kapitolách 3.2 a 3.4. Pre účely aplikácie môžeme použiť jednoduchý optický model, kde na každý voxel dopadá konštatné svetlo, bez tieňovania, a hodnoty akumulovať výrazmi:

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

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

Pri ukončení akumulovania lúča ešte so zbytkovou hodnotou α prirátame farbu pozadia vizualizácie. Interpoláciou vzorkovania dát sa nebudeme v návrhu zaoberať, a v aplikácii bude použitá len tam, kde jej realizáciu umožňujú procesy v rámci grafickej karty.

5 Implementácia prototypu

V rámci druhého semestra diplomového projektu bol implementovaný prototyp, ktorého účelom bolo overiť možnosti realizácie aplikácie špecifikovanej a navrhnutej v predchádzajúcej kapitole. Fáze implementácie predchádzal výber vhodných dodatočných technológií, programovacieho jazyka a externých komponentov a tiež bolo potrebné pripraviť funkčné prostredie na vývoj. Podarilo sa implementovať fundamentálne body zo špecifikácie s vynechaním niektorých špecifík, hlavne čo sa týka rozhrania a bezpečnosti aplikácie. Prostredníctvom tohto výsledku bola tiež vykonaná množina experimentov skúmajúcich parametre vizualizácie.

5.1 Výber technológií

Ako bolo spomenuté v návrhu riešenia ako technológia realizujúca sprístupnenie behu algoritmu na grafických procesoroch bola zvolená **CUDA**. Tým bol determinovaný aj implementačný jazyk, keďže CUDA priamo podporuje použitie jazyka **C** respektíve v obmedzenej miere aj **C++.** Existujú síce projekty sprístupňujúce rozhranie CUDA v rámci iných jazykov (pomocou takzvaných wrapper funkcií), avšak na ich použitie neboli z hľadiska špecifikácie výraznejšie dôvody.

Okrem tohto rozhrania bolo na grafický výstup aplikácie zvolené štandardizované a otvorené rozhranie **OpenGL** pre jeho multiplatformovosť a relatívnu jednoduchosť použitia. Tá bola sprostredkovaná rozšírenými knižnicami **Glew** a **Glut**, ktoré odbremeňujú proces volania funkcií OpenGL (hlavne riešenie hardvérových a systémových závislostí) a poskytujú kompaktný framework pre zobrazenie jednoduchého okna grafického výstupu spolu s odchytávaním udalostí interakcie používateľa. Aj tieto knižnice sú zamerané na podporu viacerých platforiem, takže výslednými požiadavkami aplikácie by mal byť grafický hardvér podporujúci CUDA (bežné karty značky Nvidia) a operačný systém pre ktorý existujú ovládače pre tento hardvér.

Vývoj prototypu prebiehal v prostredí Windows, takže logickou voľbou integrovaného vývojového prostredia bolo overené **MS Visual Studio.** Pri vývoji bolo okrem runtime prostredia CUDA poskytujúceho kompilátor a ďalšie nástroje použitý aj **Nvidia GPU Computing SDK** – balík pre podporu vývoja s užitočnými príkladmi implementácií a napríklad aj podporou pre Visual Studio v podobe predpripravených projektov.

Počas vývoja a testovania prototypu bol zatiaľ použitý jeden grafický hardvérový komponent a to grafická karta **GeForce GT 240** s 96 CUDA jadrami taktovanými na 1340 MHz a 1GB globálnej pamäte na karte a tradičnom procesore **Intel Pentium E5200** s taktom 2500 Mhz**.**

5.2 Implementovaná funkcionalita

V rámci prototypu bola implementovaná základná architektúra a komponenty aplikácie podľa návrhu poskytujúce nasledovnú funkcionalita zhrnutá v nasledovných bodoch:

- Načítanie objemových dát z externého súboru (formát *raw*)
- Vizualizačný algoritmus ray castingu pre tradičný procesor aj pre CUDA hardvér (bez transferovej funkcie)
- Niekoľko variant algoritmu pre CUDA hardvér s využitím rôznych technológií s možnosťou prepínania počas behu programu
- Parametrizácia algoritmu (škála vzorkovania, posunutie transferovej funkcie, hranica nasýtenosti lúča)
- Ortogonálna projekcia a pohyb kamery pre rotáciu a priblíženie
- Interakcia s používateľom prostredníctvom klávesnice na manipuláciu kamery a parametrov vizualizácie
- Zobrazovanie času vizualizácie

Implentovaný algoritmus Ray casting nepoužíva akékoľvek tieňovanie a aplikovanou optimalizáciu je skoré ukončovanie lúčov. Podarilo sa dosiahnuť jeho implementáciu tak, aby jednotlivé fázy v jadre algoritmu boli konzistentné medzi spracovaním na tradičnom procesore a grafickej karte, a to najmä vďaka možnosti CUDA rozhrania, ktorá dovoľuje písať funkcie, ktoré budú spustiteľné na oboch platformách a líši sa len spôsob spúšťania algoritmu (spustenie kernelu na grafickej karte). Tým sa tiež z väčšej časti zamedzilo duplicitným častiam kódu.

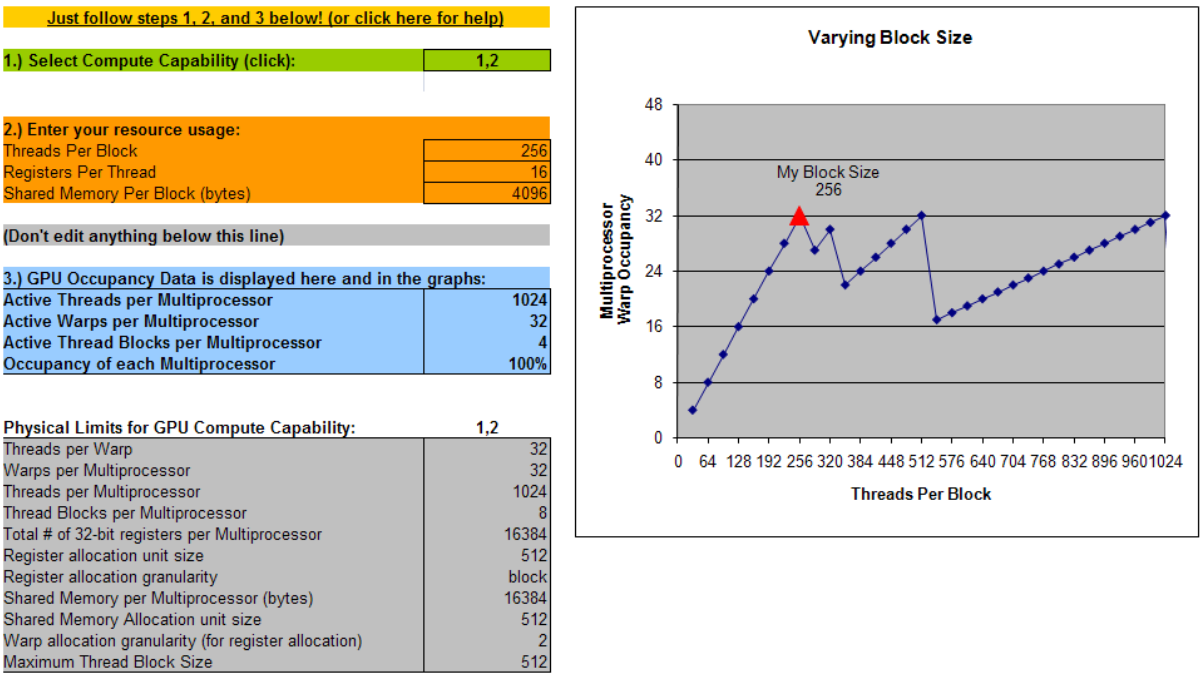
Použitými technológiami CUDA pre rôzne varianty implementácie algoritmu, ktoré sa líšia hlavne spôsobmi prístupu a uloženia objemových dát do pamäti grafickej sú:

- Predávanie parametrov vizualizácie ako vstuné parametre kernelu a uloženie objemových dát v globálnej pamäti grafickej karty
- Uloženie parametrov vizualizácie v štruktúrach v konštantnej pamäti pre lepšie cacheovanie a zrýchlenie prístupu jednotlivými threadmi
- Uloženie objemových dát v špecializovanej dátovej štruktúre *CUDA Array*, ktorá následne umožňuje pristupovať k nim ako k trojrozmernej textúre s výhodami cacheovania, lepšieho usporiadania dát a implicitnej interpolácie dát
- Zapisovanie priamo do *pixel buffer* objektu na zobrazenie pomocou OpenGL bez toho, aby bolo potrebné kopírovanie tohto bufferu mimo pamäte grafickej karty

Obmedzujúcim faktorom implementácii pre grafickú kartu bolo zdieľanie zdrojov na pamäti grafickej karty. V rámci host prostredia, ktoré spája jednotlivé implementácie je možné štandardne predávať ukazovatele na miesto v pamäti grafickej karty, ktoré sú následne dereferencované v kerneloch a týmto spôsobom je napríklad riešené zdieľanie buffera, do ktorého sa zapisuje výsledok vizualizácie. Problém však nastáva pri zdieľaní referencií do

konštantnej pamäte, keďže ich premenné treba deklarovať a pristupovať k nim v rovnakom zdrojovom súbore, ktorý obsahuje aj kernel. Preto má momentálne každý modul GPU implementácie v aplikácii alokované vlastné konštantné premenné. Podobne je to s pamäťou textúr, kedy sa samotný pamäťový priestor zdieľa (referencia na CUDA Array), ale referencia na objekt textúry, ktorá je nutná pri prístupe k takto uloženým dátam je pre každý komponent zduplikovaná. Tento prístup k dátam totiž nie je volaním funkcie CUDA rozhrania, ale je v čase kompilácie nahradený špeciálnou inštrukciou pre spracovanie textúr, kedy je označenie textúry dané staticky.

Pri spúšťaní bolo tiež potrebné stanoviť pre pixely výslednej vizualizácie vykonávajúce thready a ich usporiadanie do blokov. Na zistenie najefektívnejších parametrov poslúžil nástroj dodávaný v CUDA SDK nazvaný *CUDA Occupancy Calculator*. Ten obsahuje databázu technických špecifikácií grafických kariet Nvidia, a dokáže vypočítať ideálne rozvrhnutie threadov do blokov pre dosiahnutie čo najvyššieho využitia potenciálu hardvéru grafickej karty. Pri karte, s pomocou ktorej bola aplikácia vyvíjaná, bol zvolený počet threadov na blok 16x16, čo podľa nástroja by malo zaručiť optimálne rozloženie záťaže, ako je vidieť na grafe na Obr. 14.



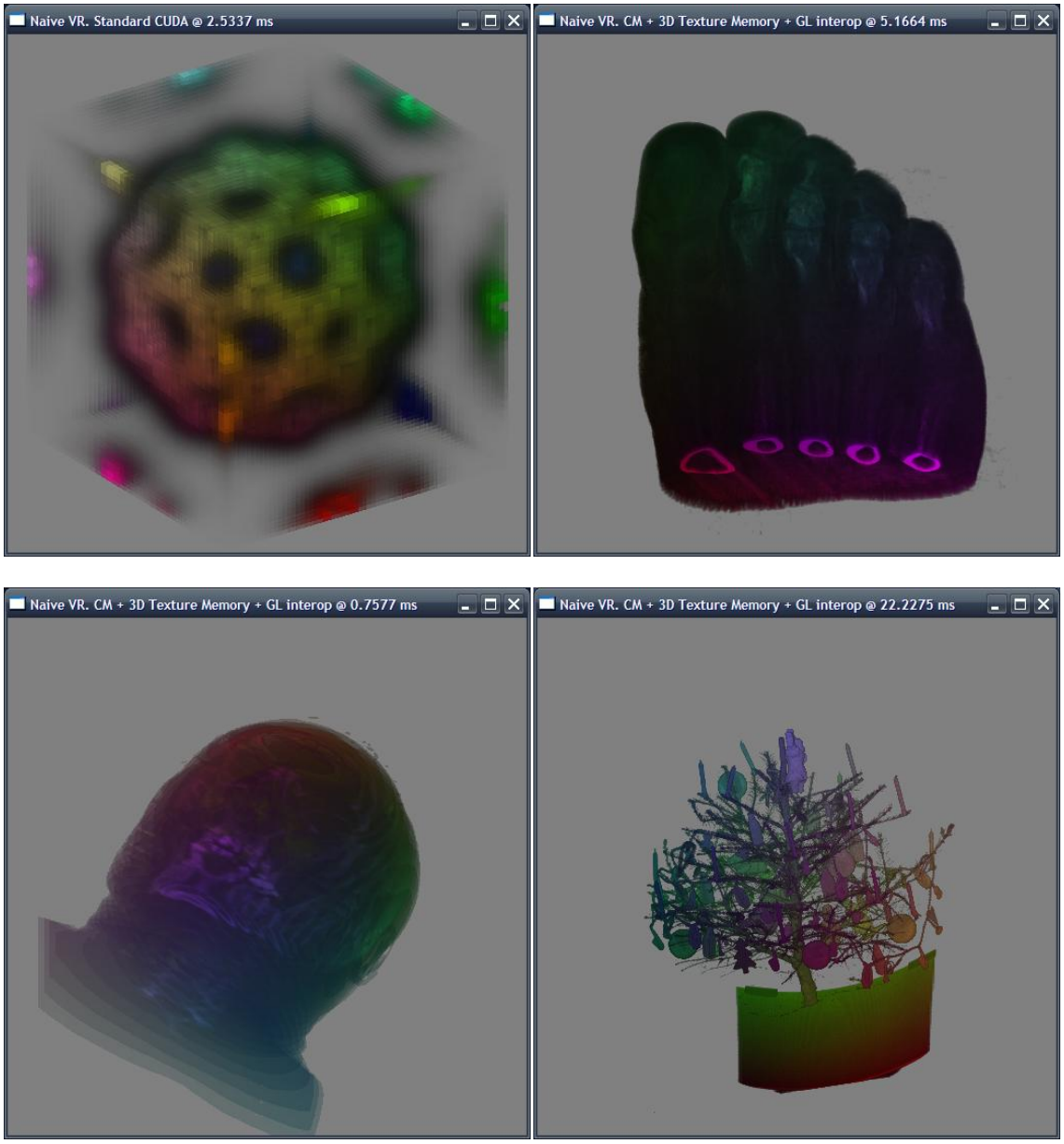
Just follow steps 1, 2, and 3 below! (or click here for help)	
1.) Select Compute Capability (click):	1,2
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	16
Shared Memory Per Block (bytes)	4096
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	100%

Physical Limits for GPU Compute Capability:	1,2
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2
Maximum Thread Block Size	512

Obr. 14 Ukážka rozhrania nástroja CUDA Occupancy Calculator.

Rozhranie aplikácie prototypu pozostáva z jednoduchého grafického okna, ktoré slúži na zobrazenie výsledku vizualizácie. Parametre vizualizácie komunikuje rozhranie na titulke okna, a v rámci konzolového okna, ktoré je takisto zobrazené pri spustení aplikácie. Interaktivita je dosiahnutá pomocou klávesnice, kedy je jednotlivým klávesom priradená funkčnosť na manipuláciu kamery, zmenu parametrov a ukončenie aplikácie. Ukážky

rozhrania spolu s príkladmi dosiahnutých vizualizácií je možné vidieť na Obr. 15. V prvej vizualizácii, ktorá neimplementuje interpoláciu, je vidno jasné prechody medzi voxelmi objemových dát kvôli ich malému počtu. Na druhej vizualizácii je časť ľudskej nohy, kde je vidieť kosti vďaka vyšším parametrom priesvitnosti materiálu kože a svalstva. Pri tretej vizualizácii možno pozorovať skreslenie spôsobené nastavením nízkeho rozlíšenia vzorkovania dát pozdĺž lúča (zobrazovaná je ľudská hlava). Štvrtou vizualizáciou je vianočný stromček – najväčšie testované dáta o veľkosti 127 MB. Farby vo vizualizácii nenesú relevantnú informačnú hodnotu – jedná sa o pseudozafarbenie vyjadrujúce polohu bodu v rámci kvádra ohraničujúceho objemové dáta vo virtuálnom priestore.



Obr. 15 Ukážky prototypom vizualizovaných objemových dát.

5.3 Experimenty

Počas vývoja prototypu sa uskutočnilo aj niekoľko experimentov s rôznymi objemovými dátami, za účelom prvotného pohľadu na výkonnosť aplikácie a vzťahy medzi jednotlivými implementáciami. Keďže nebol vykonávaný veľký počet testov, alebo použitá nejaká štatistická metóda na vyhodnotenie, výsledky prototypu sú zatiaľ vo väčšej miere informačné, a nemusia komplexne reprezentovať vlastnosti technológií a algoritmov. Výsledky sú zhrnuté v niekoľkých grafoch, pričom vertikálna os vyjadruje čas vykonávania vizualizačného komponentu v milisekundách, a horizontálna použitú implementáciu respektíve typ objemových dát. Charakteristika tých, ktoré boli použité ako reprezentačná vzorka pre testovanie je nasledujúca (ich vizualizácia je ukázaná aj na Obr. 15):

- Bucky – 32 x 32 x 32 voxelov, 32 KB raw
- Male – 128 x 256 x 256 voxelov, 8,1 MB raw
- Foot – 256 x 256 x 256 voxelov, 16,3 MB raw
- Tree – 512 x 499 x 512 voxelov, 127,7 MB raw

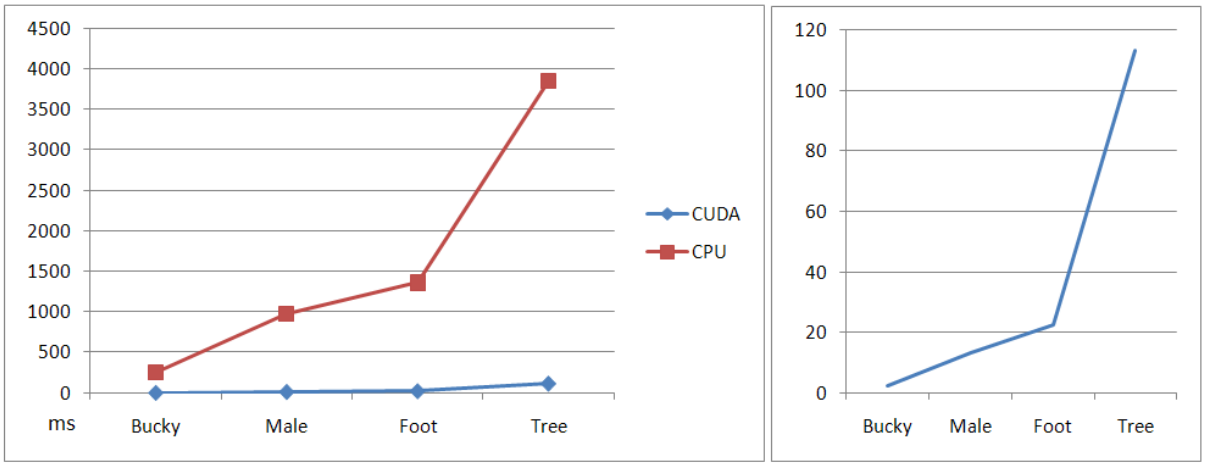
Implementácie vizualizačného algoritmu sú označené podľa použitých technológií (opísané aj v kapitole 5.2), respektíve platforiem:

- CPU – tradičný procesor
- CUDA, Standard – priamočiara implementácia algoritmu pre grafický multiprocesor
- CM – používanie konštantnej pamäte na uloženie parametrov vizualizácie
- CM+TM – používanie pamäti textúr na uloženie objemových dát (3D textúry)
- CM+TM+GLIO – použitie priameho zápisu do OpenGL *Pixel Buffer Objectu* (označuje sa tiež ako *GL Interoperation*), eliminácia kopírovania bufferu späť na *host* prostredie

Vizualizácie prebiehali pri rozlíšení 1024*1024 pixelov (okrem prvého testu), a ak nie je uvedené inak, počet vzoriek bol automaticky nastavený tak, aby sa vzorkoval približne každý voxel pozdĺž lúča zo zdrojových dát, a hranica skorého ukončovania lúčov bola nastavená na 95% hodnotu.

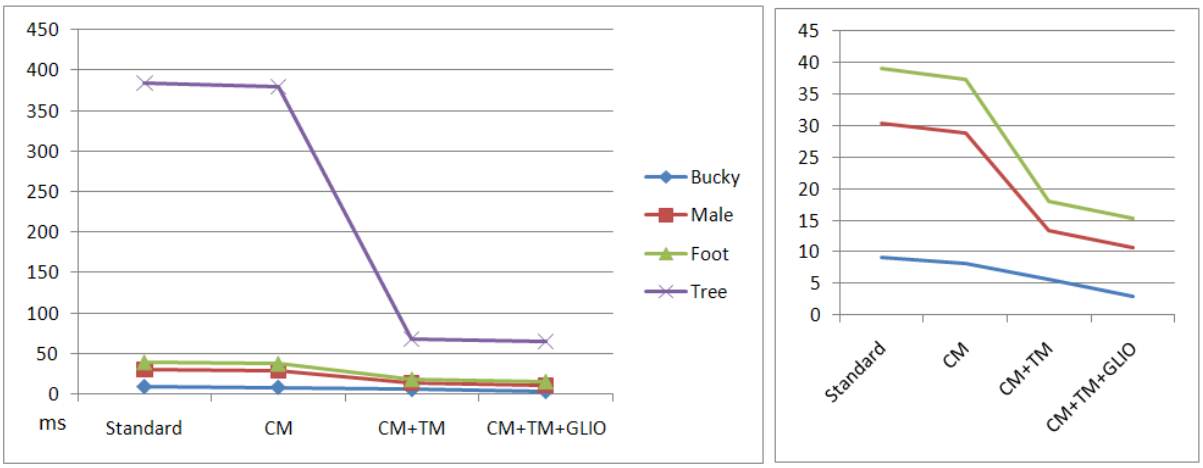
Pri prvom teste bolo zámerom obligátne overenie zrýchlenie času vizualizácie medzi tradičným a grafickým procesorom (Obr. 16). Použité boli 4 typy objemových dát, a renderovala sa vizualizácia o veľkosti 512*512 pixelov. Podľa nameraných časov behov prinieslo priamočiare prepísanie algoritmu do CUDA priemerne 67-násobné zníženie času vizualizácie. Toto enormné číslo poukazuje na výkonový potenciál grafickej karty a tiež vhodnosť ray castingu pre paralelné spracovanie, ktoré dokáže značne vyťažiť behom na grafickej karte. Každopádne tento pomer možno pripísať aj nie veľkej optimalizácii algoritmu z hľadiska kódu, kde CPU verzia stráca na kopírovaní hodnôt premenných pri niektorých funkciách, pričom pri kompilácii CUDA sa všetky funkcie implicitne inline-ujú. Vo pokročilejšej fáze aplikácie sa očakáva o niečo menej dramatický rozdiel. V menšom grafe je

pre prehľadnosť uvedená iba CUDA implementácia, pre porovnanie závislosti veľkosti vstupných dát (resp. počtu voxelov) a času vizualizácie.



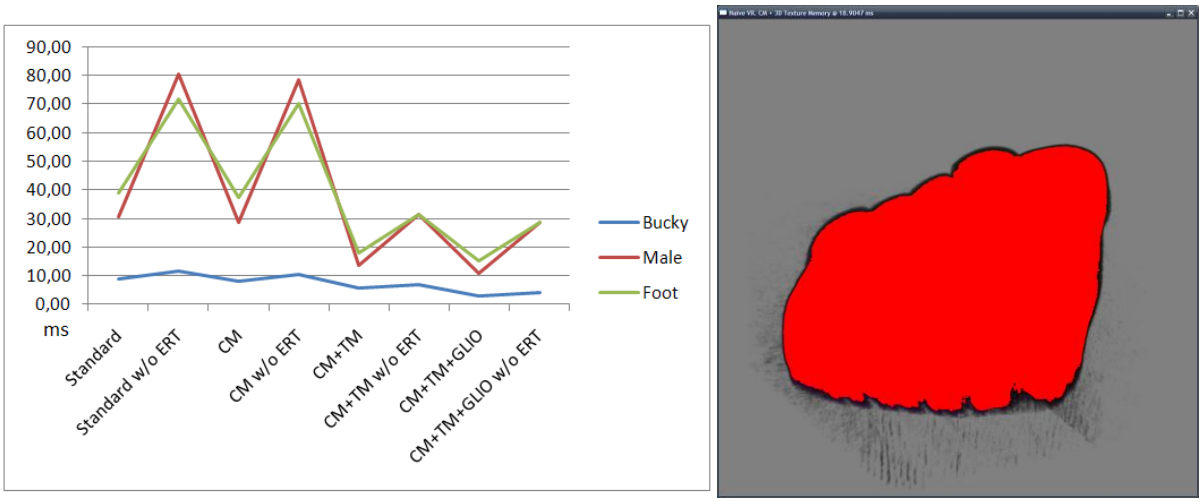
Obr. 16 Porovnanie výkonnosti CPU a CUDA implementácií.

Druhý test sa zameral na jednotlivé implementácie pre grafickú kartu (Obr. 17). Na výsledkoch možno konštatovať, že konštantná pamäť prináša malé, ale pozorovateľné zlepšenie. Parametre sú uložené iba na čítanie a cache-ované, čo poskytuje menšiu latenciu prístupu ako pri zdieľanej pamäti, kde sú parametre uložené, keď sú predávané ako argumemty kernelu. Dramatické zlepšenie hlavne pri veľkom množstve objemových dát poskytuje mechanizmus pamäti textúr, opäť to možno pripísať cache-ovaniu, optimalizovanému usporiadaniu dát v pamäti, a využitiu špecifických hardvérových komponentov pri prístupe k textúre, ktoré sa pri všeobecnom používaní globálnej pamäti v CUDA nevyužívajú. Zapisovanie priamo do grafického bufferu pre vykreslenie vizualizácie šetrí čas kopírovania dát bufferu medzi kartou a *host* prostredím nezávisle na objemových dátach. V prípade 1024*1024 veľkosti bufferu (presne 4MB dát) bolo toto zrýchlenie konštantne okolo 2,5 milisekúnd. V menšom obrázku priblíženie hodnôt bez najväčších objemových dát.



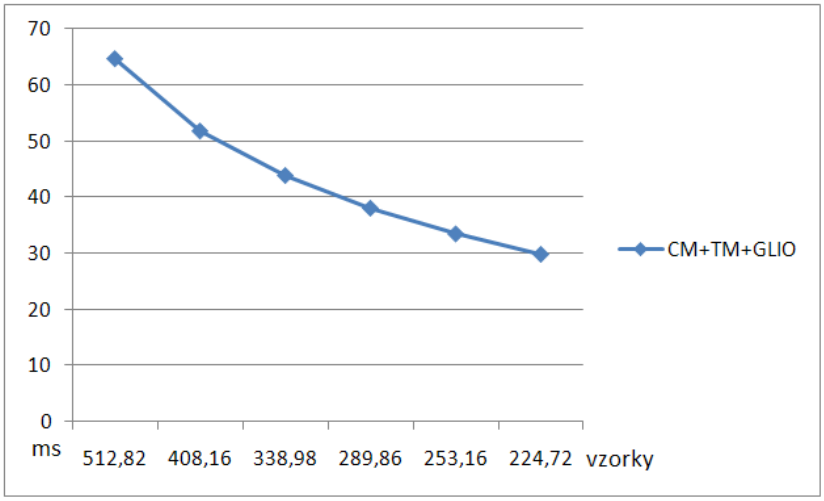
Obr. 17 Porovnanie viacerých implementácií s rôznymi CUDA technológiami.

Tretí test skúmal dopad optimalizácie skorého ukončovania lúčov (kapitola 3.4.1) – varianty označené *w/o ERT* nevyužívali túto optimalizáciu (Obr. 18). Ukazuje sa, že popri jej veľmi jednoduchej implementácii je jej dopad pomerne veľký, často ide až o dvojnásobný čas vykonania. Zaujímavosťou na grafe, je pri vypnutej optimalizácii rýchlejšie vykreslenie objemových dát Foot napriek väčšej veľkosti oproti dátam Head. Je to spôsobené tým, že pri dátach Head zaberá objekt veľkú časť priestoru, a tým pádom prichádza rýchlejšie k nasýteniu lúča. Na obrázku je tiež znázornený pokus určenia bodov, pri ktorých dochádza k skorému ukončeniu lúča. Pri hodnote 95% hranice naakumulovanej farby sú to pri modeli Foot v podstate všetky lúče, ktoré pretínajú samotný objekt (znázornené červenou farbou).



Obr. 18 Porovnanie výkonnosti pri použití skorého ukončovania lúčov.

Posledný test vyjadruje závislosť rozlíšenia vzorkovania lúča a času vizualizácie na dátach Tree a s použitím najvýkonnejšej implementácie (Obr. 19). Horizontálna os grafu vyjadruje počet vzoriek pri lúči kolmom na kváder objemového objektu. Cieľom bolo overiť ako sa dá stlačiť čas vykonania pri znižovaní kvality – pri polovičnom rozlíšení vzorkovania boli už na objekte viditeľné artefakty, čas bol podľa očakávania takmer priamo úmerný znižovaniu rozlíšenia.



Obr. 19 Vzťah medzi časom a kvalitou vizualizácie (počet vzoriek pri lúčoch).

6 Zhodnotenie

Počas prvého semestra prác na diplomovom projekte som sa venoval analýze problematiky, ktorá je zhrnutá v kapitolách 2 a 3 dokumentu. Podarilo sa mi zorientovať sa v doméne využitia grafických procesorov ako univerzálnych prúdových multiprocesorov. Okrem oboznámenia sa so súčasným stavom problematiky a histórie z ktorej tento stav vychádza som sa podrobnejšie venoval praktickým možnostiam použitia dvoch konkrétnych technológií respektíve rozhraní sprístupňujúcich grafický hardvér. Vďaka lepšej dokumentácii, väčšej rozšírenosti a dostupnosti literatúry som sa primárne venoval štúdiu technológie CUDA, ktorá je však čo sa týka základných princípov zhodná s druhou spomínanou technológiou OpenCL. To umožnilo pri vytváraní textu prvej kapitoly obsiahnuť okrem špecifických jazykových konštrukcií CUDA aj všeobecnejšie prístupy pri programovaní masívnych multiprocesorov. V druhej fáze analýzy bola predmetom skúmania doména vizualizácie objemových dát, ktorá je niektorými svojimi algoritmami mimoriadne vhodná na realizáciu paralelným spracovaním ponúkaným grafickými procesormi. Ukázalo sa, že existuje väčšie množstvo prístupov a metód, ktoré vznikli počas dlhšej doby kedy sa možnosti vizualizácie objemových dát skúmali. Avšak až nedávno sa podarilo dosiahnuť dostatočne rýchle a zároveň kvalitné zobrazenie pre poskytnutie interaktivity v reálnom čase s použitím dostupného hardvéru, ktorým boli práve grafické karty. Z tejto časti analýzy vyplynulo to, že vhodné bude sústrediť sa v rámci diplomového projektu na overenie výhod použitia technológií grafických kariet a vizualizačné algoritmy objemových dát využiť na ich demonštráciu. Nadobudnuté poznatky boli zhrnuté v tretej kapitole. Závery boli zohľadnené v špecifikácii aplikácie, účelom ktorej je overiť spomínané vlastnosti analyzovaných technológií a algoritmov.

V druhom semestri som sa sústredil hlavne na praktickú skúsenosť s rozhraním CUDA a vývojom pre neho. Počiatočné pokusy priniesli širší pohľad na možnosti platformy a s jeho pomocou bolo možné podrobnejšie navrhnúť koncepciu vyvíjanej aplikácie, opísanú v kapitole 4. V návrhu bol kladený dôraz na modularitu a merateľnosť aplikácie, keďže tieto prístupy sa ukázali ako fundamentálne pre naplnenie požadovaných cieľov práce v podobe získania komplexných výsledkov pri hodnotení a overovaní riešenia. Okrem toho bolo potrebné zvoliť a podrobnejšie preštudovať konkrétne algoritmy vizualizácie a projekcie a navrhnúť spôsob ich implementácie. Výsledkom prác v druhom semestri je funkčný prototyp aplikácie opísaný v kapitole 5. Ten zahŕňa implementáciu ray castingu pre tradičný procesor aj pre grafický hardvér, spolu s použitím viacerých technológií. Aj keď ešte nespĺňa mnohé požiadavky opísane v špecifikácii, hlavne čo sa týka reálnej použiteľnosti, implementovaná funkcionalita bola dostatočná pre vykonanie počiatočných experimentov. Z nich vyplynul enormný prínos použitia paralelného algoritmu na grafickej karte v podobe dramaticky kratšieho času vykresľovania a tiež dopady použitia optimalizácií a rôznych typov pamätí na grafickej karte.

6.1 Plán práce do ďalšieho semestra

Náplň ďalšieho semestra by mala stáť hlavne na troch pilieroch:

1) Intenzívnejšia implementácia funkcií prototypu podľa špecifikácie
2) Rozšírenie použitých metód vizualizačného algoritmu
3) Komplexné experimenty, testovanie a vyhodnotenie výsledkov

Z hľadiska použiteľnosti aplikácie na rozsiahlejšie vyhodnotenie technológií je potrebné dopracovať hlavne komponent *Profiler*, ktorý bude zaznamenávať výkon aplikácie, a poskytovať výstup v podobe formátovaného textového súboru. Žiaduce je aj pridanie mnohých funkcií rozhrania (napr. ovládanie kamery myšou) a s tým súvisiacu podporu väčšej parametrizácie vizualizácie. Pri vizualizačnom algoritme je v prototype podstatným opomenutím nepoužitie transferovej funkcie, pridanie ktorej poskytne reálnejšie výsledky v zmysle priblíženia sa ku v praxi používaným procesov zobrazenia objemových dát. Plánované je aj pridanie podpory načítavania *pvm* súborov objemových dát.

Čo sa týka rozširujúcej funkcionality, zaujímavou pridanou hodnotou by bol jednoduchý editor transferovej funkcie zabudovaný do aplikácie. Taktiež by bolo vhodné zakomponovať aj perspektívnu projekciu pre hodnovernejšiu vizualizáciu.

Po dokončení aplikácie bude nasledovať fáza experimentov, z ktorých zaujímavé výsledky a podstatné súvislosti budú zhrnuté v samostatnej kapitole overenia riešenia v tejto diplomovej práci. Predpokladá sa použitie rozmanitejšieho množstva objemových dát, a aj porovnanie na viacerých kusoch grafického hardvéru, prípadne celkových systémových platforiem.

7 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., Saarbrucken, 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-.

Príloha A – Návod na inštaláciu prototypu a obsah elektronického média

Na priloženom DVD sa nachádza implementácia prototypu. Pre jej spustenie sú stanovené nasledovné systémové požiadavky

- Grafická karta Nvidia, so schopnosťou podpory CUDA (od generácie 8000)
- Najnovšie ovládače CUDA (súčasť štandardných ovládačov grafickej karty)
- Nvidia CUDA Toolkit, odporúčaná verzia 3.2, pre kompiláciu zdrojových súborov

V adresári bin sa nachádza predkompilovaná verzia pre 32-bitové systémy Windows, ktorú možno priamo spustiť. Zdrojové súbory aplikácie sú v zložke VolumeRendering, spolu s pripraveným projektom pre MS Visual Studio 9. Potrebné knižnice pre kompilovanie pod 32-bitovými systémami Windows sú v zložke libs. Pre ostatné systémy sa dajú ich najnovšie verzie stiahnuť z domovských stránok projektov, a projekt Visual Studia konvertovať na kompatibilný kompilačný súbor napríklad nástrojom *CMake*.

Pri spustení aplikácie sa v konzolovom okne vypíše manuál na ovládanie, so všetkými dostupnými klávesmi sprostredkúvajúcimi interakciu. Súčasťou DVD sú aj vzorové objemové dáta, ktoré boli použité pri experimentoch – súbory s príponou *raw*.

V adresári doc sa nachádza tento dokument diplomovej práce v elektronickej forme.