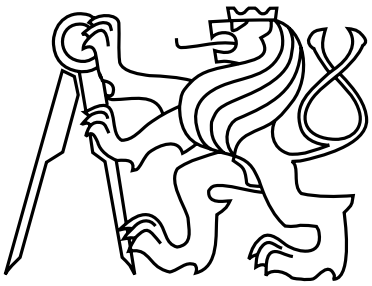
České vysoké učení technické v Praze
Fakulta elektrotechnická



Diplomová práce

Metoda sledování paprsku na grafických akcelerátorech

Martin Zlatuška

Vedoucí práce: Ing. Vlastimil Havran, Ph.D.

Studijní program: Elektrotechnika a informatika, dobíhající

Obor: Výpočetní technika

leden 2009

Poděkování

Na tomto místě bych rád poděkoval všem, kteří mě podporovali při psaní této diplomové práce. Především bych chtěl poděkovat vedoucímu mé diplomové práce Ing. Vlastimilu Havranovi, Ph.D. za jeho vstřícnost, věcné připomínky a konzultace. Také děkuji svým rodičům, kteří mi byli oporou nejen v době psaní této práce, ale při celém studiu.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne ....

Abstract

The ray tracing method is often cited as a future standard in real-time graphics and therefore is currently experiencing a surge of active research interest. This thesis goal is to demonstrate the viability of implementing ray tracing using programmable graphics processing units and compare the results of well known acceleration methods. For this purpose a ray tracing algorithm is implemented using one of the following acceleration structures: Uniform grid, kd-tree and Bounding volume hierarchy (BVH).

Abstrakt

Metoda sledování paprsku je často označována za budoucí standard v oblasti real-time grafiky a v této oblasti probíhá v současnosti aktivní výzkum. Cílem práce je ověřit správnost implementace metody sledování paprsku na grafickém akcelerátoru a porovnat výsledky dobře známých akceleračních metod. Pro tyto účely je implementován algoritmus sledování paprsku za použití následujících akceleračních struktur: uniformní mřížky, kd-stromu a hierarchie obálek BVH.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Je zcela nezpochybnitelnou skutečností, že počítačová grafika zaznamenala v posledních dvou desetiletích nebývalý rozmach a dnes již snad ani neexistuje sofistikovaná oblast lidského života, v které bychom se s ní nesetkali v nejrůznějších podobách. Své místo si postupně našla ve vojenství, kosmickém výzkumu, průmyslu, vědě, medicíně a v neposlední řadě i v zábavním průmyslu. Během krátké doby se ukázalo, že způsob tvorby obrazu se v jednotlivých oblastech výrazně odlišuje. V současnosti jsou profesionální produkty využívány pro složité a dlouhodobě prováděné výpočty a například při tvorbě filmů dosahují vysoké úrovně přiblížení se realitě, zatímco v oblasti tzv. „domácí“ a profesionální real-time grafiky převažují méně náročné metody a postupy. S postupným zdokonalováním výpočetní techniky roste i ze strany uživatelské veřejnosti stále větší poptávka po metodách a způsobech realističtějšího grafického zobrazování. Za představitele tohoto procesu (i když ne jediného) lze považovat právě metodu sledování paprsku.

Přestože je metoda sledování paprsku teoreticky známa delší dobu, značné nároky na výpočetní výkon doposud bránily v jejím výraznějším použití pro potřeby interaktivní nebo real-time grafiky. Se vzrůstajícím výpočetním výkonem procesorů se přiblížila doba, ve které lze dosažení tohoto cíle reálně předpokládat. Tomuto přesvědčení napomáhá jak vývoj nových rychlejších algoritmů sledování paprsku, tak i snaha využít jiné procesorové architektury než té, která reprezentuje doposud používané procesory počítačů. Jednu z mnoha možných architektur představuje grafická karta.

Byl to prudký růst obliby počítačových her, který ve svém důsledku vyvinul značný tlak na neustálé zlepšování kvality zobrazení a inicioval výrazný rozvoj grafických akcelerátorů, převážně zaměřených na urychlení herního zobrazení. A byla to i potřeba tvorby specifických programů, která umožnila samostatný běh vlastních programů na grafické kartě. Současná generace grafických akcelerátorů v podstatě představuje obecný programovatelný procesor, jehož lze využít i na obecné výpočty metody sledování paprsku. Provedení implementace metody sledování paprsku na programovatelné grafické kartě je hlavním cílem této diplomové práce.

Teoretickému shrnutí dosavadního vývoje grafických karet je věnována druhá kapitola. Pozornost je upřena k přeměně jednoduchého zařízení v plně programovatelný procesor s možností jeho využití i na jiné než grafické úlohy. V převážné části kapitoly je proveden rozbor tzv. platformy CUDA a její přínos pro obecné výpočty na grafické kartě. Dále je představena architektura a vlastnosti procesoru použitého na podporovaných kartách, uvedeny jsou způsoby programování a základní pravidla pro dosažení vysokého výkonu. Závěrem jsou zmiňovány některé ze softwarových a hardwarových alternativ této platformy.

Ve třetí kapitole je představena metoda sledování paprsku, popsáno je srovnání se široce používanou rasterizací. Jsou zde zmíněny některé další možnosti zkvalitnění a způsoby urychlení metody sledování paprsku. Kapitola obsahuje i přehled akceleračních struktur včetně základních metod jejich stavby a procházení. V závěru jsou shrnuty dosavadní pokusy o implementaci sledování paprsku na grafické kartě a zmiňují se úskalí tohoto postupu.

Čtvrtá kapitola je věnována souhrnnému popisu samotné problematiky implementace. Obsahuje nejen informace o implementaci nahrávání modelů a stavbě akcelerační struktury, ale i o sledování paprsku na grafické kartě. V této části je poukázáno i na možné komplikace spojené s architekturou odlišnou od klasického procesoru.

Závěrečná kapitola obsahuje porovnání výkonu jednotlivých akceleračních struktur jak při jejich stavbě, tak i při jejich procházení. Jsou zde popsány změny výkonu v závislosti na změně rozlišení, druhu grafické karty, na scéně, kvalitě zobrazení ad. V samotném závěru jsou shrnuty přednosti a nedostatky implementace sledování paprsku na grafickém akcelerátoru.

2 Obecné výpočty na grafickém akcelerátoru

2.1 Vývoj grafických akcelerátorů

Pojem počítačová grafika se poprvé objevil v 60. letech minulého století zásluhou Williama Fettera, designéra firmy Boeing. První aplikace počítačové grafiky pomáhaly především při návrzích rozsáhlých strojírenských projektů. Za přelomový okamžik lze označit použití počítačové grafiky pro účely řízení protiletadlové obrany. Rozsah aplikací se postupem doby rozšiřoval a našel své uplatnění zvláště v oblasti zábavního průmyslu. V této době byla převážná část počítačové grafiky vytvořena pomocí běžného procesoru počítače. V oblasti průmyslového nasazení existovaly speciální akcelerátory, zaměřené výhradně na ulehčení činnosti procesoru od grafických úloh. Tyto akcelerátory však byly velmi drahé a to rozhodlo o jejich nasazení v poměrně malých seriích.

V té době představovaly grafické karty v domácích počítačích velmi jednoduchá zařízení. Často byly tvořeny pouze pamětí pro uložení obrazu a obvody pro převod do analogové podoby za účelem zobrazení na monitoru. Teprve počátkem devadesátých let minulého století dosáhly procesory osobních počítačů dostatečný výkon, potřebný pro zobrazování základní prostorové grafiky. Program dokázal vykreslit kompletní scénu pomocí vlastních algoritmů, přičemž grafická karta sloužila pouze k zobrazení již hotového obrazu.

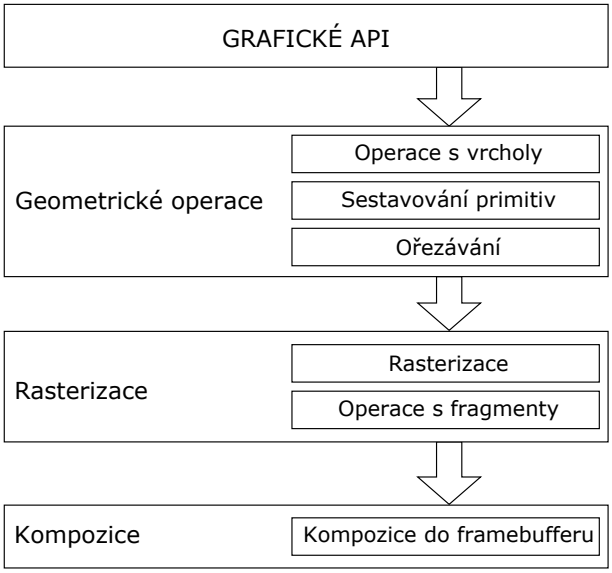
2.1.1 Akcelerátory s fixní funkcionalitou

Postupem času se začaly požadavky na kvalitu 3D zobrazení zvyšovat. K naplnění těchto požadavků měl napomoci i stále se zvyšující výkon CPU, ale ani ten uživatelská očekávání nesplnil. 3D grafika ve vysokém rozlišení, s vysokým počtem barev a s filtrovanými texturami byla nad výpočetní možnosti i nejrychlejších procesorů. Nové herní konzole obsahující specializované obvody pro prostorovou grafiku ohrožovaly pozici PC na herním trhu. A právě pro vyřešení tohoto problému byl navržen specializovaný grafický akcelerátor. Akcelerátor měl především umožnit odlehčení procesoru od náročných operací opakujících se pro každý pixel, přičemž jeho hlavním posláním mělo být docílení rasterizace trojúhelníků a nanášení filtrované textury. Určitou nevýhodou v jeho použití, oproti čistě softwarovému řešení, představovalo omezení možností „svobodné volby“ u programátora, zapříčiněné předem pevně danými funkcemi grafického API.

S rostoucím výkonem akcelerátoru narůstaly i jeho možnosti, tzn. rozšiřovala se nabídka jeho funkcí a efektů. Akcelerátor převzal další část vykreslovacího řetězce, tj. transformaci a osvětlení, ale právě výše uvedené omezení způsobené fixní funkcionalitou stále více limitovalo možnosti samotných uživatelů. Nutnost přidávat nové efekty byla čím dál tím zřejmější. Z tohoto důvodu se zlepšování schopností akcelerátoru pomocí přidávání dalších fixních funkcí grafického čipu jevilo jako nedostatečné, kvalitativně nic neřešící. Možné řešení tohoto problému nabízel programovatelný grafický procesor (GPU).

2.1.2 Programovatelné akcelerátory

S rostoucími možnostmi výroby grafických čipů se výpočet celého zobrazovacího řetězce (obr. 2.1) přesunul na grafickou kartu. Stále se však jednalo o bloky s fixní funkcí a s omezenou možností ovlivnit jejich výsledek. K dosažení požadovaných schopností grafické karty byly dva bloky zobrazovacího řetězce s fixními funkcemi nahrazeny programovatelnými procesory, tzv. *shadery*. Provedená změna umožnila nahradit blok „Operace s vrcholy“ programovatelným Vertex shaderem a operace s fragmenty převzal rovněž programovatelný Fragment (Pixel) shader. Schopnosti shaderu se většinou porovnávají (podle terminologie zavedené rozhraním DirectX) tzv. *Shader modelem.*



Obrázek 2.1: Klasický zobrazovací řetezec.

Shader model 1 Možnosti tohoto modelu byly značně limitované, např. maximální počet instrukcí a použitelných registrů byl výrazně omezen. Fragment shader navíc pracoval pouze s celými čísly a výstup mohl být uložen pouze do framebufferu. Z těchto důvodů byly naprogramovatelné efekty jednoduché, i když poměrně působivé, ale využití grafické karty pro obecné výpočty bylo takřka nemožné.

Shader model 2 Tento model představoval skutečný praktický rozmach v používání shaderů, vždyť od tohoto okamžiku všechny shadery podporovaly čísla s plovoucí řádovou čárkou. Podstatně vzrostl počet instrukcí a registrů a za další neméně významný kvalitativní krok lze považovat i podporu ukládání výsledků do textur. Grafický akcelerátor šlo již použít pro obecné výpočty, i když pouze s využitím grafického rozhraní.

Shader model 3 Tento model lze označit za vylepšenou verzi modelu 2, přičemž za nejvýraznější novinku lze považovat možnost větvení kódu a cyklů.

Shader model 4 Model 4 přinesl revoluční změnu srovnatelnou s přechodem z verze 1 na verzi 2. Počet instrukcí se stal prakticky neomezeným. Počet použitelných registrů vzrostl na několik tisíc, navíc přibyla plnohodnotná podpora celočíselných typů. Nahrazení fixního bloku sestavování primitiv programovatelným *Geometry shaderem* vůbec poprvé umožňovalo vytváření trojúhelníků přímo na grafické kartě.

Hardwarová implementace shader modelu 4 přináší významnou změnu i do samotné architektury grafického procesoru. V předchozích verzích programy Vertex a Fragment shaderů vykonávaly dva odlišné typy vzájemně nekompatibilních výpočetních jader. Toto řešení představuje omezení jednak funkční (např. problematický přístup k texturám ve vertex shaderu), tak i výkonnostní. Při každém vykreslování tělesa mohou být kladeny rozdílné požadavky na výkon jednoho ze shaderů, tzn. může dojít k přetížení jednoho shaderu a druhý - nevyužitý - naopak čeká na jeho výsledky. Nově zavedený Geometry shader navíc vyžaduje implementaci zcela nového druhu výpočetního jádra. Komplikovanou situaci vyřešilo zavedení jednotného výpočetního jádra pro všechny funkce, tzv. *Uniformního shaderu.*

Procesor s uniformními shadery je tvořen identickými obecnými výpočetními jednotkami, kterým se přiřazují funkce podle potřeby. Veškerá výpočetní síla je tvořena jedním typem jader se

stejnou instrukční sadou. Tato uniformita umožňuje vytvoření jednoduchého způsobu programování grafického procesoru a využití jeho plného výkonu s pomocí obecného programovacího jazyka, nezávislého na grafickém API.

2.1.3 Obecné výpočty pomocí grafického rozhraní

Se stále rostoucí mírou programovatelnosti a i rostoucím výkonem procesoru se přímo nabízela myšlenka na využití grafického akcelerátoru i pro jiné účely než jen pro 3D grafiku. Jediný způsob využití akcelerátoru ale stále představovalo programování prostřednictvím jednoho z grafických API. Pro obecné výpočty je grafický akcelerátor používán jako tzv. *Proudový procesor*. Toto paradigma předpokládá proud vstupních dat, nad kterými provádí výpočet tzv. kernel, přičemž výsledky jsou následně ukládány do výstupního proudu. Při implementaci za použití grafického akcelerátoru jsou vstupní a výstupní proudy představovány texturami. Kernel je program pro jeden z nabízených shaderů (většinou fragment shaderu). Výpočet kernelu se spouští vykreslením grafického primitiva.

Programování je možné provádět přímo v jazyce pro Shadery (GLSL, HLSL, Cg) a data do/z textur ukládat manuálně. V této souvislosti byly pro potřeby programování vytvořeny speciální jazyky, pokoušející se o obecnou paralelizaci výpočtů na základě C/C++. Zástupci tohoto trendu jsou např. jazyky Brook a RapidMind [8]. Tyto jazyky nejsou zaměřeny čistě na grafické karty, ale i na vícejádrové procesory a jiné architektury. Při jejich použití pro programování grafických karet je pro uskutečnění výpočtu opět použito standardní grafické API.

2.2 Compute Unified Device Architecture

Programování obecných výpočtů pomocí grafického API má však řadu nedostatků. Pokaždé je přítomna dodatečná režie rozhraní, zaměřeného na grafické výpočty. Psaní programu vyžaduje značné znalosti a nelze se ho jen tak snadno naučit. Rovněž neexistuje jednotný přístup pro ladění programů. Zásadním problémem je také skutečnost, že jednotlivé paralelně spuštěné části programu nemohou mezi s sebou spolupracovat, např. data je možno vyměňovat pouze tím způsobem, že se zapíší do textury a opětovně se spustí výpočet. Výsledkem je zbytečné vyžadování dodatečné propustnosti paměti.

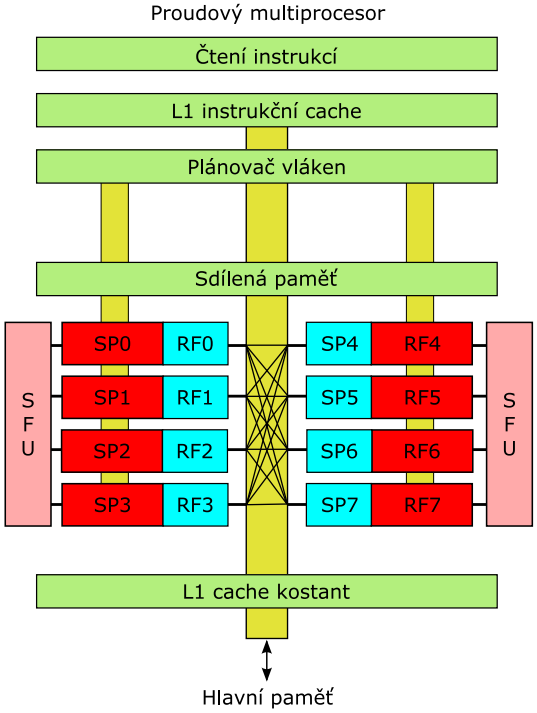
Pro rozšíření provádění obecných výpočtů prostřednictvím grafických karet se jeví jako nutné nalezení zcela nového přístupu. Za takovýto nový přístup lze označit platformu CUDA (Compute Unified Device Architecture) firmy Nvidia (Santa Clara, USA).

CUDA totiž představuje ucelenou platformu použitelnou pro obecné výpočty na grafické kartě, která se skládá z následujících komponentů:

- Architektura čipu G80.
- Ovladače s podporou CUDA.
- Rozšíření programovacího jazyka C.
- Instrukční sada PTX.
- Překladač jazyka C do instrukční sady PTX.

2.2.1 Architektura procesoru

Nejzákladnějším blokem procesoru je *Proudový procesor*. V porovnání se standardním procesorem je jeho funkčnost omezená, protože neobsahuje obvody pro dekódování instrukcí a není tak schopen pracovat samostatně. Instrukce obdrží již předem zpracované a to z vyšší části hierarchie. ALU dokáže v procesoru zpracovávat základní matematické operace. Je zajímavé, že (na rozdíl od starších architektur) jednotlivé procesory jsou *čistě skalární* a neobsahují žádnou specializovanou jednotku pro operace nad vektory. Instrukční sada navíc neobsahuje instrukce



Obrázek 2.2: Schéma multiprocesoru.

pracující nad vektory. Toto řešení umožňuje lepší využití výkonu procesoru při obecných výpočtech, které často nejsou vektorové. Vysokého výkonu je dosaženo pipelinovaným designem ALU, s možností spuštění instrukce v každém taktu. Rychlé přepínání vláken umožňuje zásobovat ALU daty v každém taktu.

Jednotlivé procesory jsou sdruženy v *Proudový Multiprocesor*, obsahující osm proudových procesorů, v kterých se provádí dekódování instrukcí a plánování přepínání mezi vlákny a to společně pro všechny obsažené procesory. Multiprocesor obsahuje dvě jednotky pro speciální, méně časté výpočty (trigonometrické funkce, převrácená hodnota, odmocnina). Multiprocesor však stále ještě nemůže pracovat samostatně, protože neobsahuje obvody pro přístup do paměti. Schéma multiprocesoru je zobrazeno na obrázku 2.2.

Multiprocesory jsou následně sdruženy do *Texture Processor Clusteru*, který rozšiřuje funkčnost o texturovací jednotku a rozhraní pro přístup do paměti. Počet sdružených multiprocesorů se liší v závislosti na konkrétní generaci čipu. V současnosti existují verze se dvěma či s třema multiprocesory. Cluster je již kompletní samostatnou výpočetní jednotkou schopnou výpočtu. Umístění různého počtu clusterů do čipu nebo jejich vypínání umožňuje vytváření různých modelů grafických karet s odlišným výkonem.

Architektura procesoru se postupně vyvíjí a různě vylepšuje. Nové generace čipů vycházejí z generací předchozích s tím, že mírně vylepšují některé jejich vlastnosti. Pro jednoznačné určení vlastností procesoru se používá pojem *Výpočetní schopnost* (Compute Capability).

2.2.1.1 Paměťový model

Grafický akcelerátor obsahuje několik typů paměti, které se od sebe navzájem liší svými schopnostmi (zápis/pouze čtení), rychlostí (0-600 cyklů latence přístupu) a velikostí (16 KB až cca 1 GB). Pro plné využití akcelerátoru je důležitá znalost rozdílů mezi těmito paměťmi. Hierarchie paměti je zobrazena na obrázku 2.3.

Globální paměť Jedná se o paměť grafické karty disponující obdobnou funkcí jako má hlavní paměť počítače, tj. slouží jako hlavní úložiště dat, přičemž jednotlivá jádra do ní můžou zapisovat, ale i z ní číst. Přístup do paměti *není* cachován a trvá několik stovek cyklů. K dosažení plné rychlosti přístupu je nutný speciální přístupový vzorec. Alokace této paměti je možná pouze prostřednictvím programu hostitele. Hostitelský program může kopírovat data mezi touto pamětí a pamětí hostitele pomocí knihovních funkcí CUDA. Velikost paměti závisí od modelu grafické karty a může dosahovat až 1 GB.

Registry Registry představují primární úložiště proměnných kernelu. Cena přístupu je již zahrnuta v množství cyklů potřebných na vykonání jedné instrukce a ve většině případů ji není potřeba brát v potaz. V jistých případech vznikají delší prodlevy (např. při RAW hazardu), které ale programátor nemůže nijak ovlivnit a v takovém případě se musí plně spolehnout na optimalizátor překladače, který by se měl snažit takovýmto situacím vyhnout. Počet dostupných registrů je určen výpočetní schopností procesoru, přičemž jejich současná hodnota je 8192 (Verze 1.0, 1.1) nebo 16384 (Verze 1.2, 1.3).

Lokální paměť Pokud kernel používá většího množství proměnných, může se vyskytnout situace, že není k dispozici dostatek volných registrů, a v takovém případě se může část proměnných umístit do lokální paměti. Takovéto řešení má většinou negativní vliv na rychlost programu, protože rychlost lokální paměti je stejná jako v případě globální paměti. Umístění proměnných do lokální paměti nemůže programátor nikterak ovlivnit, je totiž zcela v režii překladače.

Sdílená paměť Multiprocesor obsahuje malé množství (16 KB) paměti sdílené všemi proudovými procesory. Hlavním účelem této paměti je umožnit komunikaci mezi vlákny. Samotná paměť je rozdělena do 16 nezávislých bank. Přístup do paměti je stejně rychlý jako do registrů za předpokladu, že každé vlákno přistupuje do jiné banky.

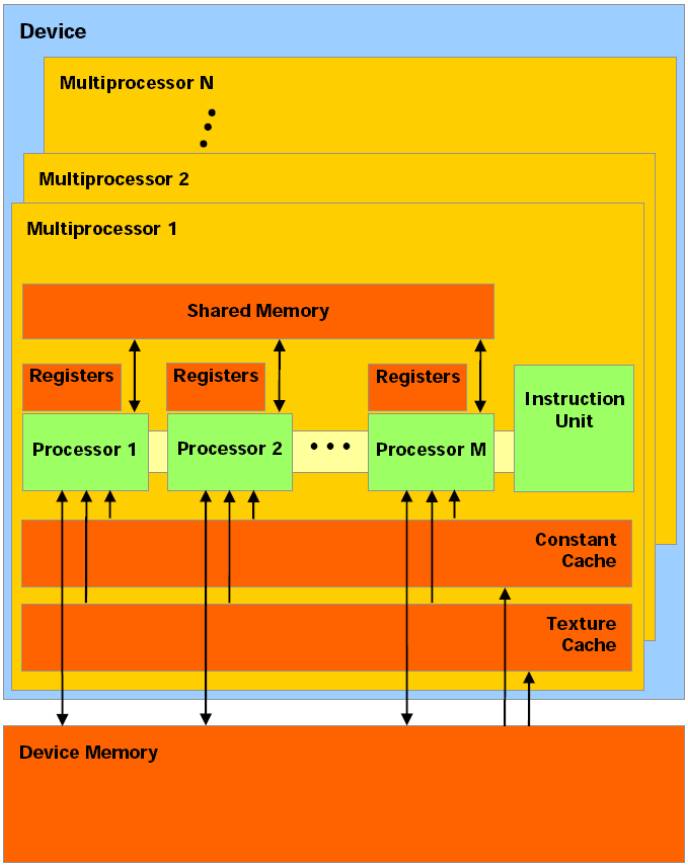
Paměť konstant Pro data, jejichž hodnoty jsou známy ještě před spuštěním jádra, lze použít paměť konstant. Tato paměť je cachovaná, tzn. že pokud jsou požadovaná data v cache, je stejně rychlá jako registry. Paměť je najednou (v jednom kroku) schopna obsloužit pouze jeden požadavek. Cena přístupu tedy roste lineárně s množstvím různých požadovaných adres v jednom půl-warpu.

Cache textur Cache slouží k zrychlení přístupu k texturám uloženým v hlavní paměti. Je možno ji použít pro data, pro která nejsme předem schopni odvodit přístupový vzorec, víme však, že má jistou souvislost a spojitost.

2.2.2 Programovací model

Program pro grafický akcelerátor je napsán v jazyce C a obsahuje několik specifických rozšíření, spočívajících v kvalifikátorech funkcí a proměnných. Kupříkladu umožňují specifikovat, kdo konkrétně může volat funkce. Jejich prostřednictvím lze také určit, ve kterém paměťovém prostoru se budou nacházet proměnné. Další rozšíření umožňuje spouštět výpočetní kernel s danou konfigurací a navíc jsou zde podporovány některé konstrukce jazyka C++. Je také dovoleno přetížit operátory a použít omezenou implementaci šablonového metaprogramování.

Při srovnání se standardním jazykem C vykazuje verze pro grafický akcelerátor i některá omezení. Volané funkce jsou standardně při překladu „inlinovány“, tzn. že funkce není možno volat rekurzivně. Rovněž zde neexistuje obdoba ukazatele na funkci. Za uživatelsky nepříjemné je možno považovat i výrazné omezení použití polí jako proměnných kernelu, tzn. že proměnné jsou uloženy v registrech, které nelze indexovat. Indexace pole vyústí v umístění dat do lokální



Obrázek 2.3: Organizace paměti v procesoru. Převzato z [2].

paměti, kterou lze indexovat, ale na druhou stranu je tato paměť výrazně pomalejší než samotné registry. Tato problematika je dále řešena v části o implementaci.

Při překladu je kód programu rozdělen na dvě části. Jedna část, kód pro hostitele sloužící k spuštění výpočtu na grafické kartě, je dále překládán standardním překladačem (např. v linuxu pomocí GCC). Druhá část, kód určený pro zařízení, je přeložena do instrukční sady virtuálního paralelního počítače PTX, která je po spuštění hostitelského programu za běhu přeložena ovladačem grafické karty do instrukční sady skutečné grafické karty.

2.2.2.1 Aritmetické operace

Proudový procesor podporuje jak čísla s plovoucí řádovou čárkou podle standardu IEEE-754 s jednoduchou přesností, tak i celočíselné datové typy. Implementace nezapře zaměření na počítačovou grafiku a v ní nejčastěji používané funkce. V některých případech implementace čísel s plovoucí řádovou čárkou nedodržuje přesně normu IEEE. Výsledky instrukcí násobení a sčítání požadavky normy splňují, operace jsou však často slučovány do jedné instrukce (FMAD), která zaokrouhluje mezivýsledek. Dělení je prováděno nestandardně pomocí násobení obrácenou hodnotou. Trigonometrické funkce jsou dostupné v rychlé verzi s omezenou přesností a ve verzi s iterativním zpřesňováním, která je několikanásobně pomalejší. Od výpočetní schopnosti 1.3 dokáží procesory provádět výpočty s dvojitou přesností. Tyto výpočty však neprobíhají v ALU jednotlivých proudových procesorů, ale v jednotce pro speciální výpočty na úrovni multiprocesoru. Výkon těchto operací je ve srovnání s operacemi s jednoduchou přesností několikanásobně nižší.

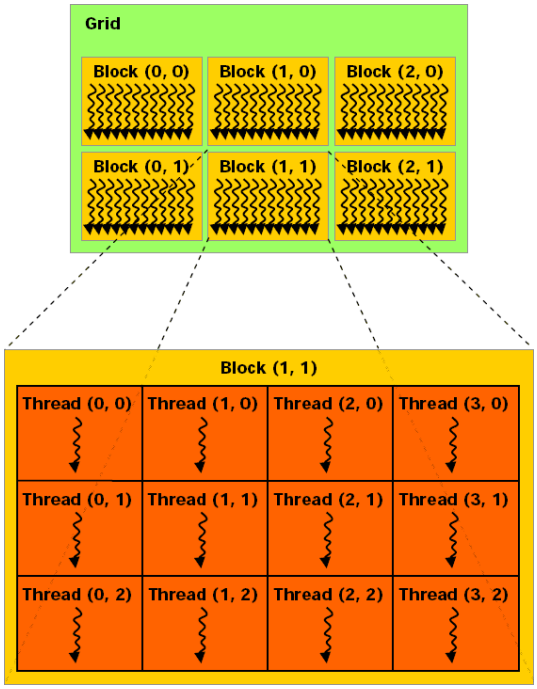
Podpora celých čísel, v předchozích generacích grafických karet nedostatečná, je v tomto případě na dobré úrovni. Procesor podporuje bitové logické operace, bitové posuny, dělení, zbytek po dělení a dokonce „bit scan“ instrukce. Implementace operací dělení a zbytku dělení jsou však velmi pomalé a měly by se proto používat jen zřídka. Rovněž násobení s plnou (32 bitovou) přesností je pomalé, přestože násobení je možné zrychlit použitím speciální instrukce s přesností 24 bitů (odvozeno z přesnosti mantisy čísla s jednoduchou přesností).

2.2.2.2 Organizace výpočtu

Základní jednotkou výpočtu je *vlákno*, přičemž princip výpočtu je obdobný jako u vláken používaných na CPU. Každé vlákno má svoje privátní lokální data, přičemž vlákna mohou libovolně číst a zapisovat data do paměti, tzn. že z hlediska taxonomie paralelních architektur se jedná o CRCW PRAM architekturu.

Vlákna se sdružují do *bloků*, přičemž blok může vykazovat podobu 1D, 2D nebo 3D pole. Volba počtu vláken v bloku závisí na množství vláken, jejichž spolupráce je nevyhnutelná anebo výhodná. Organizace vláken v bloku má pouze informativní charakter a samotný výpočet neovlivňuje. Vlákna bloku běží vždy na jednom multiprocesoru a mohou mezi sebou navzájem komunikovat pomocí sdílené paměti. Při zápisu do sdílené paměti je nutno použít operaci bariérové synchronizace, která zajistí viditelnost zápisu do paměti ostatním vláknům bloku. Značně problematická je synchronizace v podmíněném kódu, protože pokud není provedena všemi vlákny bloku, dojde k zacyklení výpočtu.

Bloky jsou součástí mřížky, která má podobu 1D nebo 2D pole. Maximální velikost strany mřížky je 65535 a z tohoto důvodu není počet bloků v mřížce nikterak omezující. Velikost mřížky a samotná poloha bloku v ní samotný výpočet nijak neovlivňují, slouží pouze k určení dat, na kterých má blok pracovat. Hierarchie organizace výpočtu je znázorněna na obrázku 2.4.



Obrázek 2.4: Organizace výpočtu. Převzato z [2]

2.2.2.3 Organizace při běhu programu

Jak již bylo uvedeno výše v sekci o architektuře procesoru, instrukce a plánování se dekódují na úrovni multiprocesoru. Dekódování a plánování trvají (v současné implementaci) čtyři takty, tj. všechny proudové procesory vykonávají stejnou instrukci po dobu čtyř taktů.

Při vykonávání programu je blok rozdělen na tzv. *warpy*. Warp je základní jednotkou bloku, s níž pracuje plánovač procesoru. Z počtu procesorů v multiprocesoru a doby trvání plánování je odvozena velikost warpu (32 vláken), přičemž všechna vlákna jednoho warpu musí vykonávat stejnou instrukci. Nespornou výhodou je skutečnost, že plánovač *zaručuje* korektnost programu při jeho větvení. Větvením kódu uvnitř warpu je ovlivněna „pouze“ výkonností procesoru. Vlákna warpu se při větvení rozdělí na aktivní a neaktivní, přičemž jejich instrukce musí být vykonány u obou typů vláken, ale jenom aktivní vlákna uloží výsledky. Z výše uvedeného vyplývá, že doba vykonávání větve nezávisí od počtu vláken, které se dané větve účastní. Z této vlastnosti lze vyvodit závěr, že pokud není počet vláken v bloku násobkem velikosti warpu, nebudou některé takty procesoru nikdy využity.

Při přístupech do paměti se používá menší jednotka půl-warp. Půl-warp je první nebo druhá polovina warpu. Vlákna jednoho půl-warpu vzájemně ovlivňují rychlost přístupu do paměti.

2.2.2.4 Využití více karet

Pokud systém obsahuje více grafických karet s podporou CUDA, záleží pouze na programátorovi, jak tuto možnost prakticky využije. Pro každou kartu na hostiteli je nevyhnutné vytvořit jedno vlákno, pro něž se pomocí knihovních funkcí CUDA nastaví aktivní zařízení. Na takto vybraném aktivním zařízení lze následně provést samotné spuštění kernelů. Rozdělení programu na více karet a rozdělení zátěže musí zajistit programátor.

2.2.3 Optimalizace výkonu

2.2.3.1 Sdružování přístupu do paměti

Pro hlavní paměť neexistuje cache, což vede k nutnosti vypracovat specifické postupy za účelem zajištění rychlého přístupu do paměti. Latence přístupu dosahuje stovek cyklů, takže plnění požadavků jednotlivých vláken samostatně by bylo značně zdlouhavé. Navíc je přístup do paměti optimalizován pro přesun větších bloků, takže by se značné množství dat přenášelo zcela zbytečně. Z těchto důvodů procesor umožňuje sdružit požadavky několika vláken na přístup do paměti do požadavku jednoho. Uvedený postup je principem sdružování přístupu do paměti (Memory Coalescing).

Pro dosažení rychlého přístupu do paměti musí vlákna bloku spolupracovat při čtení spojitého bloku paměti. V současné implementaci se sdružuje přístup šestnácti vláken z jednoho „půl-warpu“. Požadavky na sdružení se liší v závislosti od výpočetních schopností grafické karty, např. pro výpočetní schopnost 1.0 se vyžadují následující podmínky:

- Data musí byt zarovnána na velikost čteného bloku.
- Jednotlivá vlákna musí přistupovat k sekvenčně se zvyšujícím adresám.

Při jejich dodržení se uskuteční jeden přístup do paměti, v opačném případě se uskuteční 16 přístupů, tj. pro každé vlákno jeden přístup.

Požadavky pro sdružování jsou u nejnovějších karet s výpočetní schopností 1.2 značně mírnější:

- Procesor automaticky sdružuje požadavky do jednoho bloku.
- Permutace adres v rámci bloku je libovolná.
- Velikost transakce se automaticky zmenšuje podle počtu zúčastněných vláken.

V případě, že vlákna vyžadují jiný než sekvenční přístup, je nutné použít spolupráci za pomoci sdílené paměti. Kombinace sdružování se sdílenou pamětí je vlastně uživatelem řízená cache. Tento princip je velmi důležitý a v některým případech může znamenat i desetinásobný rozdíl ve výkonu [9].

2.2.3.2 Konflikty sdílené paměti

Sdílená paměť o velikosti 16 KB je rozdělena do 16ti zcela nezávislých bank o velikosti 1 KB, z nichž každá může obsloužit jeden požadavek každé dva takty. Adresový prostor je mapován tak, že po sobě jdoucí paměťové adresy se nacházejí v různých bankách. Pokud tedy vlákna jednoho půl-warpu přistupují na adresy v rozdílných bankách, je sdílená paměť stejně rychlá jako registry. V opačném případě jsou požadavky serializovány a skutečná doba přístupu závisí na maximálním počtu požadavků na jednu banku.

Finální krok v optimalizaci algoritmů, pracujících se sdílenou pamětí, představuje omezení konfliktů přístupu do paměti. Používají se dvě nejběžnější metody optimalizace:

Vkládání nevyužitého prvku Do pole dat ve sdílené paměti se každých n prvků vloží jeden nevyužívaný prvek a tím dojde k posunutí adres následných prvků a tedy k jejích umístění do jiných bank [9].

Změna pořadí operací Algoritmus, přistupující k paměti s postupně se měnícím offsetem, může odstranit konflikt tím, že provede operace v opačném pořadí. Například paralelní redukce, začínající sečtením sousedních prvků, je dvakrát pomalejší než redukce provádějící v posledním kroku sečtení sousedních prvků [11].

2.2.3.3 Obsazenost procesoru

Jak již bylo uvedeno na jiném místě, každý multiprocesor má 8 skalárních jader a může mít přiřazeno velké množství vláken. Například u karet s výpočetní schopností 1.0 a 1.1 je to až 768 aktivních vláken. Poměr maximální možné hodnoty a skutečně využité hodnoty udává tzv. obsazenost (occupancy) procesoru. Její hodnota může mít značný vliv na rychlost programu.

Jednotlivá vlákna mohou z různých důvodů čekat na dokončení instrukcí (např. se může jednat o přístup do hlavní paměti, konflikt bank sdílené paměti nebo dokonce o přístup do registrů). V tomto případě může procesor jednoduše vybrat jedno z dalších vláken a nemusí docházet ke vzniku nevyužitých cyklů. Nízká hodnota obsazenosti proto ještě nemusí automaticky snižovat výkon. Obsazenost má malý vliv na „compute intensive“ programy a velký vliv na programy vyžadující častý přístup do paměti.

Skutečný počet aktivních vláken je omezen dostupnými prostředky, tzn. že počet registrů a sdílené paměti je pevně daný. Pokud jich však vlákna vyžadují velké množství, nemůže být procesoru přiřazeno maximální možné množství vláken. Při vývoji programu je potřeba vzít v úvahu požadavky programu na přístup do paměti a snažit se optimalizovat obsazenost (tj. počet použitých registrů a sdílené paměti) pro programy náročné na přístup do paměti. Dále lze překladači nastavit horní limit počtu registrů, ale v tom případě je následně použita lokální paměť. Za této situace je potřeba ověřit, zda není vliv na výkon naopak negativní.

2.2.4 Ladění programu

V době vypracování této diplomové práce existovala jediná možnost krokování programu a to emalace na CPU. Program pro prostředí CUDA je vytvořen v jazyce C s několika specifickými rozšířeními. Překladač může zdrojový kód automaticky upravit a specifické funkce grafické karty nahradí voláním funkcí, a umožnit překlad kódu do instrukcí CPU.

Určitý problém představuje skutečnost, že emulace nesimuluje přesné chování grafické karty. Například pro matematické operace se používá nativní přesnost CPU a výsledky výpočtů se tedy mohou lišit. Za zásadní problém lze označit fakt, že některé programy, které neprošly úpravou, mohou mít při emulaci zcela jiný průběh. V emulovaném kódu je pro každé lehké vlákno bloku vytvořeno jedno skutečné vlákno operačního systému. Proměnné, které jsou sdílené vlákny, mají pro každé vlákno vlastní kopii. Při zápisu do proměnné vidí ostatní vlákna změnu až po provedení synchronizace. Na grafické kartě není vlákna v jednom warpu vůbec potřeba synchronizovat [11]. Ostatní warpy uvidí změnu po nějakém čase i bez synchronizace. Z uvedeného plyne, že program při provádění emulace je potřeba přidání dodatečných instrukcí synchronizace.

Při emulaci se vyskytuje i další problém, a tím je rychlost. Použití velkého množství vláken s nutností časté bariérové synchronizace značně zpomaluje emulovaný kód, a to až o několik řádů v porovnání s nativním spuštěním na grafické kartě, např. u kódu sledování paprsku byl zjištěn poměr i 100 ms/10 min.

Kvůli nepřesnosti matematických operací může nastat situace, při níž bude chování emulovaného a nativního kódu odlišné. V takovém případě lze použít druhou metodu ladění, při které se program spustí na grafické kartě a informace o průběhu programu se uloží do paměti. Tímto postupem se vytvoří struktura popisující potřebná data, alokuje se pro ni na grafické kartě alokuje dostatečný prostor a prostřednictvím vybraného vlákna se do ní ukládají potřebné informace. Po ukončení běhu kernelu se data zkopírují do paměti počítače a následně se zjišťují jejich hodnoty. Postup je zdlouhavý a náročný, pro některé případy ladění však nezbytný.

Samotné ladění programu je názorným příkladem směru, kterým se ubírá vývoj platformy CUDA. V průběhu zpracování této diplomové práce nebylo ladění programu spuštěného na grafické kartě proveditelné. Podle dostupné dokumentace firmy Nvidia bude nová verze CUDA 2.1 (v současné době v beta verzi) obsahovat upravený dubugger GDB (zatím pouze pro operační systém Linux) s možností krokování programu na grafické kartě. Z výše uvedeného lze usoudit, že možnosti ladění programu budou v blízké budoucnosti značně vyspělejší.

2.2.5 Alternativy

2.2.5.1 Alternativní programovací prostředí

ATI Stream V době, kdy se na trhu objevila platforma CUDA, tehdejší firma ATI (nyní sekce AMD, Sunnyvale, USA) reagovala na tuto skutečnost poněkud rozpačitě. Narychlo vyrobené rozhraní CTM (Close To Metal) svým stavem, podporou a funkcemi nebylo důstojnou alternativou nové platformy CUDA a relativně brzo bylo nahrazeno novým rozhraním ATI Stream postaveném na jazyku Brook+. Firma ATI se rozhodla povolit toto rozhraní pouze na profesionálních kartách FireStream a nebylo možno použít klasické karty pro domácí počítače. S rozmachem rozhraní CUDA a blížícím se uvedením OpenCL své rozhodnutí změnila a počátkem roku 2009 by rozhraní mělo být přístupné na všech odpovídajících kartách. Otázkou je, do jaké míry bude, po zveřejnění OpenCL, tato funkce využívána.

OpenCL Open CL je programovací jazyk pro obecné výpočty na grafické kartě, vyvinutý firmou Apple a zamýšlený jako doplněk knihoven OpenGL a OpenAl. Základními principy se podobá rozhraní CUDA. Dokumentace finální verze 1.0 byla zveřejněna 8. 12. 2008. Firmy AMD i Nvidia již vyjádřily tomuto projektu podporu a slíbily, že ho v budoucích verzích svých produktů podpoří. Podpora multiplatformních obecných výpočtů na GPU by tak do budoucnosti mohla dostat zelenou.

2.2.5.2 Procesorové architektury

IBM Cell IBM Cell je výkonná procesorová architektura zahrnující vyšší počet výpočetních jader, přičemž řídící jádro je založené na architektuře PowerPC. Řídící jádro rozděluje úkoly jednotlivým výpočetním jádrům, z nichž každé může vykonávat jiný program a má také vlastní lokální paměť. Ke komunikaci s hlavní pamětí se používá DMA přenos z lokální paměti, který umožňuje vysokou flexibilitu, ale zároveň zvyšuje složitost programování a nároky na programátora. Vysoká výkonnost procesoru jej předurčuje i pro ray-tracing [4]. V současnosti je využíván v konzoli Playstation3 a do budoucna se uvažuje o jeho využití i pro akcelerační karty do PC.

Intel Larrabee Intel Larrabee je novým GPU procesorem firmy Intel, založeným na x86 architektuře. Svojí architekturou se zásadně liší od jiných současných karet. Tvoří ho volitelný počet jader, založených na architektuře P54C (tj. původní Pentium I z doby před téměr patnácti lety) doplněné o širokou SIMD jednotku (skládající se ze 16ti prvků) a o fixní funkcionalitu pro zacházení s texturami. Tato architektura se částečně podobá architektuře procesoru Cell, největším rozdílem oproti němu je virtuální cache coherent paměťový model. Díky nové architektuře by mohla grafická karta vyhovovat jak pro klasickou grafickou pipeline, tak pro obecné výpočty. Hodnotit její skutečný přínos a výhodnost, navíc před uvedením této architektury na trh, lze jen ztěží. Její předpokládané uvedení na trh lze očekávat na přelomu roku 2009/2010.

3 Metoda sledování paprsku

3.1 Zobrazovací rovnice

S vývojem počítačové grafiky postupně vznikalo velké množství algoritmů se zaměřením na zobrazování prostorových dat. Mnohé z nich byly prostými empirickými modely, velmi hrubě aproximujícími skutečnost. S příklonem k realističtější grafice se ukázala potřeba porovnat vlastnosti algoritmů cestou exaktního matematického aparátu, popisujícího vykreslování obrazu jako výpočet toku světla scénou. K tomuto účelu slouží tzv. *zobrazovací rovnice* [14]. Řešení rovnice udává pro každý bod povrchu a pro každý směr vycházející radianci. Tato je rovna součtu radiance emitované bodem a radiance přicházející z ostatních částí scény, která je odražena. Rovnice je vlastně jiným popisem fyzikálního zákona zachování energie a má tvar

$$L_r(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_\Omega f(x,\vec{\omega},\vec{\omega}_i) L_i(x,\vec{\omega}_i) \cos\Theta \; d\vec{\omega}_i \;, \tag{3.1}$$

kde x je poloha bodu ve scéně, $\vec{\omega}$ je směr odchozí radiance, $f(x,\vec{\omega},\vec{\omega}_i)$ je tzv. BRDF funkce udávající pro každý příchozí a odchozí směr množství odražené radiance, $L_i(x,\vec{\omega}_i)$ udává intenzitu příchozí radiance pro každý směr, Θ je úhel sevřený normálou povrchu a směrem příchozí radiance $\vec{\omega}_i$.

Analytické řešení rovnice je pro reálné scény nemožné, a proto je nutné použít aproximační řešení. Možností takových řešení je celá řada, od klasických postupů řešení složitých rovnic (např. metoda konečných prvků) až po hrubé aproximace nahrazující integrál pouhým jedním vzorkem ve směru bodového světla. Všechny metody počítačové prostorové grafiky lze popsat jako částečné řešení zobrazovací rovnice a kvalitu jejich zobrazení objektivně posuzovat s využitím přesností této aproximace.

3.2 Klasický zobrazovací řetězec

Real-timová grafika je v drtivé většině případů vytvářena grafickými akcelerátory pomocí grafických rozhraní DirectX a OpenGL. Základní princip obou těchto rozhraní je stejný. Vstup na rozhraní představuje scéna reprezentovaná sítí trojúhelníků, na jejichž vrcholy je aplikována série transformací. Po modelové, zobrazovací a projektivní transformaci je určena poloha trojúhelníku ve výsledném obrazu a ten je následně „rasterizován“ na jednotlivé pixely. Viditelnost je řešena pomocí paměti hloubky (Z-buffer). Zobrazovací řetězec s rasterizací je možno jednoduše implementovat a paralelizovat.

Jednoduchá hardwarová implementace byla jedním z hlavních důvodů rozšíření. Zásadním omezením využití rasterizace z hlediska obrazové kvality představuje skutečnost, že použitý osvětlovací model je čistě lokální. Vykreslování trojúhelníku není žádným způsobem ovlivněno ostatními trojúhelníky ve scéně a tato metoda je tedy velmi hrubou aproximací zobrazovací rovnice. Sama o sobě nevytvoří ani základní efekty, jakými jsou stíny a odrazy, ale tyto efekty lze pomocí různých algoritmů doplnit. Většinou se jedná o metody založené na vícenásobném vykreslování scény. V pomocném průchodu se shromáždí informace potřebné pro daný efekt, a ty jsou následně použity v hlavním vykreslovacím průchodu.

Ve většině případů jsou efekty jen hrubě aproximovány a projevuje se u nich řada problémů (nedostatečné vzorkování scény, výrazné zjednodušení v důsledku použití jednoduché geometrie ap.). Navíc musí být všechny efekty vytvořeny dodatečně samotným programátorem. Přesto tato metoda vyvolala značný zájem, vyplývající z nedostupnosti jiných relevantních alternativ. I když jsou efekty jen aproximované, vykazují stále vyšší kvalitu. Metoda např. umožňuje implementovat další efekty globálního osvětlení (např. ambient occlusion). Z jednoduché lo-

kální metody se tak vyvinula kvalitní metoda, která může v real-time zobrazování v mnohém konkurovat metodě sledování paprsku.

3.3 Princip metody sledování paprsku

Metoda sledování paprsku vychází z jednodušší verze algoritmu, kterou je vrhání paprsku (Ray casting), a byla původně vytvořena pro zobrazování modelů vzniklých metodou *Konstruktivní geometrie těles* (Constructive Solid Geometry). Pro každý pixel je vyslán z oka jeden (primární) paprsek, a ten je následně testován vůči objektům ve scéně. Poté následuje vyhledávání nejbližšího průsečíku paprsku s objektem a nakonec je možno určit barvu pixelu.

Uvedenou metodu lze rozšířit a použít pro více reprezentací scény, a to prakticky pro jakoukoli reprezentaci, u které je možné určit průsečík s paprskem. Použití této metody pro reprezentaci, u níž je určení průsečíku obtížnější, je také možné, a to tím způsobem, že se adaptivně převádí na vhodnou reprezentaci přímo za chodu algoritmu. Pro potřeby této diplomové práce bude nadále počítáno s jediným typem reprezentace, a to se sítí trojúhelníků.

Metoda sledování paprsku, v porovnání s klasickou rasterizací, neposkytuje po vizuální stránce žádnou výhodu. Rozšíření této metody o sledování paprsků vyšších řádů je podstatou metody u *Sledování paprsku*. Pod pojmem sledování paprsku (Ray Tracing) se většinou rozumí algoritmus, který popsal T. Whitted [25]. Po nalezení průsečíku primárního paprsku oka s objektem se vytvoří řada dalších (sekundárních) paprsků, které lze zařadit do jedné z následujících kategorií:

- Stínové paprsky.
- Odražené paprsky.
- Lomené paprsky.

Stínové paprsky zjišťují přímé osvětlení objektu světlem. Stínový paprsek je vyslán od průsečíku primárního paprsku s objektem směrem ke světlu a je opět testován proti objektům ve scéně. Pokud je nalezen průsečík stínového paprsku s objektem a tento průsečík je blíže než světlo, nachází se objekt ve stínu, v opačném případě jej světlo osvětluje. Touto metodou lze získat přesné a ostré stíny z bodového zdroje světla.

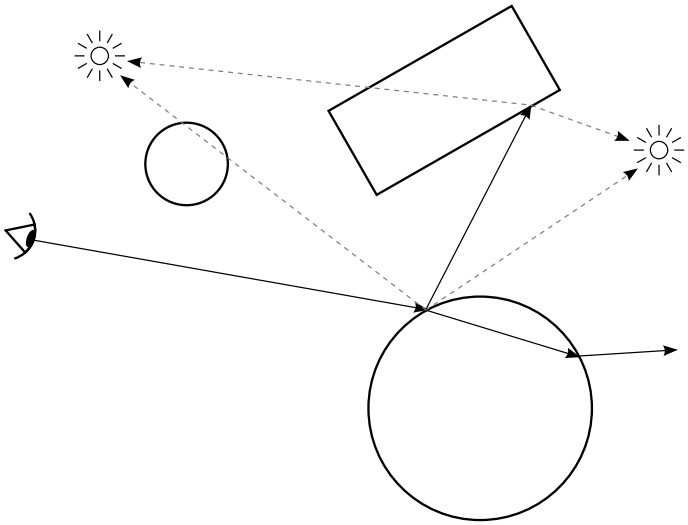
Odražené paprsky umožňují zobrazit zrcadlový odraz. Směr odraženého paprsku je odvozen dle pravidel optiky ze směru primárního paprsku a normály povrchu. Odražený paprsek se po vyslání chová stejně jako paprsek primární. Po případném nalezení průsečíků paprsku s tělesem jsou pro tento průsečík opět vyslány stínové, odražené a lomené paprsky. Tento proces umožňuje zobrazení vzájemného zrcadlení lesklých těles. Vysílání odražených paprsků by vlastně mohlo pokračovat do nekonečna, a z tohoto důvodu je stanoven horní limit počtu odražení.

Lomené paprsky simulují průhledné objekty. Princip zobrazení je obdobný jako u odražených paprsků, pouze místo směru odraženého paprsku se dle zákonů optiky odvozuje lomený paprsek.

Pro jeden primární paprsek vyslaný z oka může nastat situace, při které bude nutné sledovat velké množství sekundárních paprsků. Proces generování sekundárních paprsků je znázorněn na obrázku 3.1.

3.3.1 Vylepšení kvality obrazu

Některé předpoklady, použité v základní metodě sledování paprsku, představují v porovnání se skutečným stavem určitá zjednodušení (bodové zdroje světla, dokonalé zrcadlové odrazy, okamžitá expozice scény a další). Ve srovnání se zobrazovací rovnicí je silně zjednodušena BRDF funkce na prosté zrcadlové odrazy a také integrál je nahrazen několika vzorky ve směru světel. Zlepšení kvality zobrazení zavedením skutečné BRDF funkce a plošných zdrojů světla si klade za cíl metoda *Distributed ray tracing*. Algoritmus vysílá pro každý pixel velké množství paprsků, které při dopadu generují mírně odlišné sekundární paprsky. Stínový paprsek je vždy



Obrázek 3.1: Ukázka sledování paprsku.

vyslán směrem do mírně odlišné části plošného zdroje světla, čímž vznikají kvalitní měkké stíny. Zavedení BRDF funkce pro potřebu generování odražených paprsků umožňuje realistické ztvárnění lesklých povrchů, které však nemají dokonale zrcadlový odraz. Metodu lze využít i pro další efekty, jako je např. simulace hloubky ostrosti nebo rozmazání pohybu (motion blur).

3.3.2 Alternativní metody

Metoda sledování paprsku se jako jedna z prvních přibližuje fotorealistické grafice. Je podstatně dokonalejším řešením zobrazovací rovnice než rasterizace[1], nadále však neimplementuje celou rovnici. Hlavní nevýhodou této metody je její faktické omezení na sledování přímého šíření světla (je uvažován pouze odraz světla od objektu do oka). Pokud není mezi objektem a světlem přímá viditelnost, světlo objekt nikterak neovlivňuje.

Ve skutečnosti má prakticky každý skutečný objekt difúzní odraz, tzn. že světlo se před dopadem do oka často odrazí od velkého množství objektů. V základním ray-tracingu (a rasterizaci) je tato skutečnost aproximována tzv. ambientní složkou v osvětlovacím modelu. Ta však představuje pouze velmi hrubé zjednodušení. Pro skutečně realistické obrazy je potřeba přiblížit se principům šíření světla v daleko větší míře. Jako příklad takových algoritmů lze uvést následující dvě metody, vycházející ze sledování paprsku:

Path tracing Jedním z možných řešení výše uvedených problémů je rozšíření metody sledování paprsku o Monte Carlo náhodnou metodu. Řešení složitého integrálu v zobrazovací rovnici je v tomto případě aproximováno stochastickým vzorkováním. Metoda *Path tracing* [14] umožňuje vysílání více paprsků pro každý jeden pixel, přičemž při každém nalezení průsečíku s tělesem se paprsek od objektu odrazí. Pro určení rozložení odražených paprsků je použit pravděpodobnostní model v závislosti od vlastností materiálu. Paprsek putuje scénou a v každém jeho průsečíku s objektem je vyhodnocen osvětlovací model. Procházení paprsků scénou končí buď dosažením vrchní hranice možného počtu kroků nebo dopadem paprsku do světelného zdroje. Path tracer dále dokáže simulovat kompletní zobrazovací rovnici. Nevýhodou této metody jsou její velmi vysoké nároky na výpočetní výkon. Další nevýhodu představují náhodné odrazy paprsků, které nutně vedou k zanesení šumu do obrazu, přičemž kvalitní zobrazení vyžaduje vyslání velkého množství paprsků pro každý jeden pixel a v případě složitějších scén i v

[1] Dodatečné algoritmy tento rozdíl částečně zmenšují.

řádu desetitisíců.

Alternativní náhradu za výše uvedenou metodu představuje řešení, založené na zcela opačném postupu. V takovém případě jsou paprsky vystřelovány ze zdroje světla, putují scénou a snaží se dorazit do oka. Tato metoda je výhodnější v případě malých a bodových zdrojů světla, protože lépe vykresluje kaustiky. Ve srovnání s klasickým Path tracingem je její nevýhodou větší náchylnost na šum.

Fotonové mapy Kombinací obou zmíněných postupů vznikají obousměrné metody. Jednou z nich je metoda *Fotonových map*, skládající se ze dvou fází. V první fázi jsou ze světelného zdroje vystřelovány fotony, při dopadu na těleso je zaznamenána jejich intenzita a se sníženou intenzitou jsou opětovně vystřeleny zpět do scény. Druhá fáze je prakticky totožná se základním raytracingem, navíc je z uložených fotonů odvozena hodnota intenzity světla v bodě průsečíku. Metoda fotonových map poskytuje velmi kvalitní fotorealistické obrazy.

3.4 Akcelerační struktury

Nejjednodušší implementace sledování paprsku testuje průsečík paprsku se všemi trojúhelníky ve scéně, což je výpočetně velmi náročný proces (Složitost $O(n)$). Urychlení výpočtu spočívá v omezení počtu testů na průsečíky paprsku s trojúhelníky za použití některé z *akcelerakčních struktur*. Tyto struktury dosahují omezení počtu testů tím, že rozdělí prostor nebo objekty na menší části a při procházení paprsku scénou je nejdříve proveden test průsečíku paprsku s malou oblastí. Pokud je tento prostor paprskem zasažen, teprve poté je paprsek testován s trojúhelníky vyskytujícími se v této oblasti. Použití akceleračních struktur má zásadní vliv na rychlost sledování paprsku. Na druhou stranu přináší problém rychlosti stavby, která má zásadní význam při zobrazování dynamických scén.

V současné době existuje velké množství akceleračních struktur, vzájemně se od sebe lišících rychlostí stavby, procházením, způsobem dělení a dalšími parametry. Jedním z cílů této diplomové práce je právě provedení rozboru a charakteristik jednotlivých konkrétních struktur - viz. následující kapitoly.

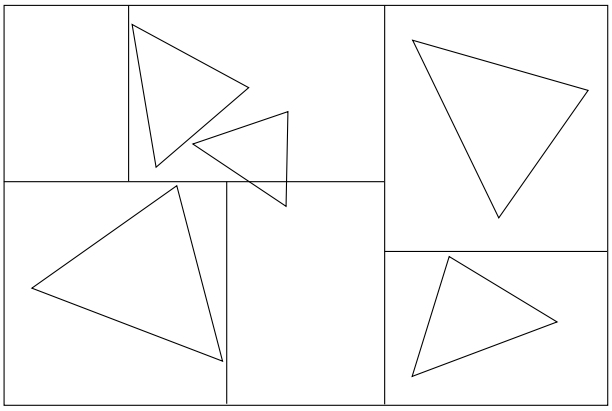
3.4.1 Uniformní mřížka

Uniformní mřížka představuje nejjednodušší způsob dělení prostoru na vzájemně se nepřekrývající buňky. Jednotlivé trojúhelníky mohou být umístěny v několika různých buňkách. Použití uniformní mřížky v grafice popsal jako první Fujimoto [7]. Procházení paprsku mřížkou je realizováno jednoduchým 3D-DDA algoritmem [3].

Výhodou mřížky je jednoduchost a rychlost stavby, jakož i jednoduchost procházení paprsku. Proto se mřížka používá pro dynamické scény v případě, že ostatní struktury neumožňují dostatečně rychlou stavbu nebo úpravu. Nevýhodou je absence jakéhokoli přizpůsobení geometrii scény, s výjimkou počtu buněk. Uniformní mřížka nejlépe funguje pro uniformní geometrii, v tomto případě je počet trojúhelníků v každé buňce přibližně stejný. Naopak velmi nevhodná je uniformní mřížka při řešení problému typu „konvička na stadiónu“ (Teapot in the stadium). Pokud představuje scéna převážně méně podrobné prostředí a současně obsahuje na některých místech velmi podrobný objekt, potom je rozdíl ve využití jednotlivých buněk značný.

3.4.2 Kd-strom

Kd-strom je speciální variantou binárního dělení prostoru (BSP - binary space partition), která dělí prostor na dvě poloviny pomocí obecné roviny. Kd-strom omezuje možnosti orientace dělící roviny podmínkou, že dělící rovina musí být vždy kolmá na některou z prostorových os. Stejně jako i pro jiná dělení prostoru, platí pro kd-strom podmínka, že trojúhelníky mohou být umís-



Obrázek 3.2: Ukázka 2D kd-stromu. Trojúhelníky mohou být umístěny ve více buňkách. Některé buňky mohou být prázdné.

těny v nekolika buňkách a že některé buňky mouhou být prázdné. Ukázka kd-stromu pro 2D je zobrazena na obrázku 3.2.

Způsob stavby stromu výrazně ovlivňuje rychlost procházení paprsku tímto stromem. Existují různé způsoby určení osy dělící roviny a její polohy. Nejjednodušším z nich je varianta, při níž se pravidelně střídají osy použité pro výběr dělící roviny. Rovina je umístěna tak, aby buď rozdělila prostor na dvě stejně velké oblasti nebo aby oblasti na obou stranách roviny obsahovaly stejný počet objektů. Tento postup však není vhodný pro stavbu stromu za účelem sledování paprsku.

Existuje řada postupů produkujících stromy s vyšší kvalitou. V současné době nejlepším a nejpoužívanějším postupem je tzv. SAH (Surface area heuristic) [22], která je založena na možnosti odhadnout předem cenu procházení paprsku stromem. Při určení této ceny se předpokládají následující skutečnosti:

- Paprsky jsou v prostoru uniformně distribuované nekonečné čáry.
- Ceny traverzačního kroku a výpočtu průsečíku jsou předem známy.
- Cena výpočtu průsečíku s n trojúhelníky je rovna ceně n jednotlivých výpočtů.

Za předpokladu platnosti těchto pravidel je možno určit cenu průchodu paprsku stromem. Pro uniformně distribuované paprsky a konvexní obálky je možno z geometrické teorie pravděpodobnosti určit v případě zásahu buňky V také pravděpodobnost zásahu buňky v ní zcela obsažené V_{sub}

$$\mathcal{P}_{[V_{sub}|V]} = \frac{\mathcal{SA}(V_{sub})}{\mathcal{SA}(V)} \tag{3.2}$$

Cenu průchodu lze tedy v daném kroku dělení určit jako cenu průchodu obou potomků, patřičně upravenou v závislosti na pravděpodobnosti jejich průchodu

$$\mathcal{C}_V(p) = \mathcal{K}_t + \mathcal{P}_{[V_l|V]}\mathcal{C}(V_l) + \mathcal{P}_{[V_r|V]}\mathcal{C}(V_r) \tag{3.3}$$

Pomocí této rovnice lze pro každý strom stanovit cenu jeho procházení. Množství různých stromů, které lze pro danou scénu sestrojit, však se vzrůstajícím počtem trojúhelníků roste exponenciální řadou. Pokoušet se nalézt strom se skutečně nejmenší možnou cenou je prakticky nemožné. Proto je při stavbě stromu heuristika použita jako základ rozhodování lokálního hladového algoritmu. Při výpočtu ceny průchodu pro konkrétní umístění dělící roviny jsou oba

vzniklí potomci považovány za listy, které se již dále nedělí. Určení ceny dalšího průchodu v daném kroku je dáno následující rovnicí

$$\mathcal{C}_V(p) = \mathcal{K}_t + \mathcal{P}_{[V_l|V]}\mathcal{K}_i N_l + \mathcal{P}_{[V_r|V]}\mathcal{K}_i N_r \ , \tag{3.4}$$

kde $\mathcal{K}_t$ je cena jednoho traverzačního kroku, $\mathcal{K}_i$ cena výpočtu průsečíku paprsku s jedním trojúhelníkem a N_l, N_r počty trojúhelníků v levém a pravém potomkovi. Cena procházení výsledného stromu není nejmenší možná, ale doba stavby nyní dosahuje prakticky použitelných časů. I když podmínky pro výpočet ceny neodpovídají zcela přesně skutečnému procházení, dosahuje SAH velmi dobrých výsledků a v současnosti není známa metoda, která by poskytovala na obecných scénách lepší výsledky.

Algoritmus procházení znamená upravené prohledávání stromu do šířky. V každém uzlu se pomocí dělící roviny zjistí průnik paprsku s jeho potomky a následně procházení stromem pokračuje pouze v potomcích, které paprsek protíná. Pokud paprsek protíná oba potomky, pak pokračuje procházení nejdříve v potomku bližším k počátečnímu bodu paprsku, a dále až do listu stromu, kde je proveden test paprsek/trojúhelník. Procházení je pak ukončeno nalezením průsečíku nebo vyčerpáním všech relevantních větví stromu. Pro konkrétní implementace může být zvolena jak rekurze, tak použití vlastního zásobníku.

3.4.3 BVH

Pro urychlení různých typů výpočtů se často využívají obálky objektu. V případě složitější operace může být výpočet znatelně urychlen tím, že se otestuje nejdříve obálka a jenom v případě úspěšného testu se provede test na skutečném objektu. Existuje několik druhů obálek, které se navzájem liší ve dvou (protichůdných) vlastnostech. Jedná se o to, jak snadno se obálky vytvářejí a jak složité jsou testy, a dále „těsnost“, se kterou se obálky přizpůsobují objektům.

Používání obálek pro jednotlivé trojúhelníky není výhodné, velkého zrychlení je naopak možno dosáhnout sdružením obálek do *hierarchie obálek* (BVH). Hierarchie má podobu n-árního stromu, většinou se používá klasický binární strom. V každém uzlu stromu je uložena obálka obklopující všechny obálky a objekty svých potomků. Obálky potomků jsou tedy vždy obsaženy kompletně v obálce rodiče, mohou se však navzájem protínat. Naopak trojúhelník je vždy umístěn pouze v jedné buňce. Pro hierarchii tedy platí přesně opačná pravidla než pro dělení prostoru. Hierarchie obálek si nárokuje méně prostoru než kd-strom, na každý trojúhelník může být odkazováno pouze jednou a počet buněk je rovněž řádově roven počtu trojúhelníků.

Stavbu hierarchie je možno provádět shora dolů i zdola nahoru a lze přitom opět úspěšně použít SAH. Procházení hierarchie paprskem je obdobné jako v případě kd-stromu, rozdíl je pouze v tom, že se testuje průsečík paprsku s obálkami potomků namísto s dělící rovinou.

3.5 Paketové procházení

Pro real-time zobrazování je sledování paprsku za použití akceleračních struktur nedostatečně výkonné. Dalšího urychlení je možné dosáhnout použitím *paketového procházení*. Základní myšlenka tohoto postupu spočívá v tom, že se společně prochází vícero paprsků najednou. Podnětem k jeho vytvoření byl pokus o využití SIMD jednotky rozšíření SSE procesorů Pentium III [20]. Tato jednotka je schopna pracovat najednou se čtyřmi hodnotami a procházení akcelerační struktury se tedy provádí se čtyřmi paprsky najednou. Pro tento paket se společně projdou všechny buňky akcelerační struktury, které by dané paprsky navštívily při individuálním procházení. Může se stát, že se jednotlivé paprsky neshodnou na dalším kroku v procházení. V tomto případě je nutno projít postupně všechny možnosti a některé paprsky po dobu vykonávání maskovat. Výkon metody tak závisí na koherenci paprsků. Při dodržení této podmínky

může metoda rychleji vykonávat matematické operace a omezit požadovanou propustnost paměti. Autoři *paketového procházení* jako akcelerační strukturu používali původně kd-strom, v následujících letech byl stejný princip rozšířen i na uniformní mřížku [23] a BVH stromy [21].

Přirozeným rozšířením paketového procházení je pokus o další zvýšení počtu společného procházení. Výkonnostní přínos v tomto případě tkví v tom, že se provádějí operace pouze pro nutný počet paprsků. Testování průsečíku s obálkou v akceleračních strukturách je možno provádět pro celý svazek paprsků v jednom celku. Rovněž průsečíky s trojúhelníkem je možno nejdříve vyloučit pro celý paket a teprve po tom, co test zjistí, že průsečík může existovat, provede se test pro všechny paprsky jednotlivě.

Paketové procházení je schopno několikanásobně zrychlit výpočet. Toto zrychlení se však liší pro každý typ paprsků. Největšího zrychlení se dosáhne logicky pro vysoce koherentní primární paprsky, rovněž stínové paprsky mají zpravidla vysokou koherenci. Problém nastává u odrazů a lomu, kdy s každým dalším odrazem se výkonnostní přínos paketů zmenšuje. U scén s velkým počtem odrazů nebo při použití pokročilé techniky (Path tracing, fotonové mapy) není přínos paketového procházení nikterak významný.

3.6 Implementace metody na grafickém akcelerátoru

3.6.1 Dosavadní výzkum

Požadavek vysokého výpočetní výkonu pro metodu sledování paprsku vedl ke hledání vhodné hardwarové architektury pro dosažení real-time rychlosti. Použití grafické karty pro sledování paprsku bylo testováno už v první generaci programovatelných grafických akcelerátorů. Purcell [16] navrhl implementaci sledování paprsku za pomocí uniformní mřížky na tehdy ještě neexistujících grafických akcelerátorech, vycházejíc ze znalosti navrhovaných vlastností DirectX 9.

Uniformní mřížka se stala díky své jednoduchosti zajímavou volbou právě pro grafický akcelerátor. Samotná mřížka se přirozeně reprezentuje 3D texturou, rovněž ostatní data jsou uložena do textur. Implementace abstrahuje proudový procesor, základní kernel simuluje jeden krok v uniformní mřížce a další kernel počítá průsečíky paprsku s trojúhelníky. Neaktivní paprsky jsou maskovány pomocí stencil bufferu.

Při tomto postupu použitá generace GPU neumožňuje provádět větvení a smyčky a rovněž uniformní mřížka není nejlepším řešením akcelerační struktury. Akcelerační strukturu se pokusil vyřešit Foley [6] a navrhl použití kd-stromu. Při implementaci kd-stromu se však objevil základní problém, a to absence zásobníku na GPU. Z tohoto důvodu byla použita speciální verze algoritmu bez zmíněného požadavku. V okamžiku, kdy by se jinak další postup uskutečnil pomocí zásobníku, je celý algoritmus restartován a procházení paprsků strukturou pokračuje opět z kořene stromu. Potřebný průchod strukturou po restartu algoritmu je umožněn změnou minimální možné vzdálenosti průsečíku. Nevýhoda tohoto postupu je zřejmá - počet kroků se oproti klasickému algoritmu podstatně zvýší.

Absenci zásobníku se pokoušel řešit Popov [15] použitím speciální verze kd-stromu doplněného o „provazy“. Každý uzel stromu obsahuje ukazatele na uzly sousedící se šesti stranami jeho obálky. Při procházení paprsků nevyžaduje tato verze stromu zásobník, její nevýhodou je však několikanásobné zvětšení paměťových nároků na uložení stromu.

Horn [12] se naopak vrátil ke klasickému kd-stromu a implementoval zásobník na grafické kartě. Každý paprsek má svůj vlastní malý zásobník o velikosti pouhých několika prvků. Při vyprázdnění zásobníku nedochází k ukončení procházení, ale naopak se použije předchozí „restart“ algoritmus. Použití byť malého zásobníku snižuje znatelně počet kroků traverzace.

Günther [10] využil možnosti nového hardwaru s novým rozhraním CUDA pro implementaci

paketového procházení BVH.

Výše zmíněné předchozí implementace slouží jako východisko pro implementaci algoritmů pro tuto diplomovou práci.

3.6.2 Problémy

Použití grafického akcelerátoru pro sledování paprsku je výzvou nejen pro namapování základního postupu do specifického prostředí, ale klade také vyšší nároky na paměť akcelerátoru. Nejde jen o dodatečnou paměť, kterou požaduje akcelerační struktura, ale také o to, že při použití rasterizace *není* nutné mít v paměti grafické karty uloženu veškerou potřebnou geometrii a textury. Při vykreslování objektu se mohou potřebná data načítat postupně hlavní paměti. Při sledování paprsku může být zasažen kterýkoli trojúhelník ve scéně a proto je nutné, aby se veškerá geometrie a případné textury umístily do paměti najednou. Případné rozdělování scény na části a postupné nahrávání dat na grafickou kartu by bylo značně náročné (došlo by ke zpomalení celého procesu).

Jako komplikované se jeví i samotné použití textur. Grafický akcelerátor obsahuje texturovací jednotky, které jsou vytvořeny pro použití pomocí klasických grafických rozhraní. Texturovací jednotky počítají s omezeným počtem aktivních textur a pro souběžné používání všech textur ve scéně nejsou uzpůsobeny. Při sledování paprsku se tak grafický procesor může paradoxně dostat do stejné situace jako klasický procesor a je nucen funkce texturovací jednotky provádět softwarově.

Definování uživatelských efektů podobných shaderům grafických rozhraní se jeví rovněž problematickým. Program pro grafickou kartu nemá možnost použít ukazatele na funkci, jako je tomu v případě jazyka C, a je proto nucen přeložit veškeré efekty najednou a společně s celým zbytkem programu. To znamená, že psaní efektů by v současné podobě vyžadovalo přítomnost kompletního vývojového prostředí.

Z výše uvedeného vyplývá, že problém plně funkčního sledování paprsku na grafické kartě není zatím zcela vyřešen. Celkem jednoduchá myšlenka „přerostla“ v obtížný problém, vyžadující řadu složitých algoritmů.

4 Implementace

4.1 Nahrávání scény

Prvním krokem na cestě k vykreslení obrazu je nutně načítání dat modelů ze souboru. Načítání modelů ze souboru je převážně rutinní záležitostí v podstatě spočívající pouze ve správné interpretaci hodnot uložených v souboru. Z tohoto důvodu nebylo k popisu implementace, pro jednotlivé formáty souborů, přistoupeno. Pro účely práce jsou implementovány následující formáty:

Stanford PLY Uvedený jednoduchý formát využívá Stanfordská univerzita k uložení výsledků skenování 3D těles.

Open Inventor Pro složitější scény je potřeba pokročilejšího formát souboru, obzvláště pro jednoduché používání materiálů povrchu. Část testovacích scén byla dostupná ve formátu MGF pro který existují nástroje pro převod do formátu Open Inventor. Z tohoto důvodu byl za formát podporující materiály zvolen právě tento.

BART Projekt BART [1] obsahuje vlastní formát pro reprezentaci scén, obsahující podporu pro popis animovaných objektů. Na stránkách projektu je k dispozici kostra parseru pro tento formát, která byla využita pro implementaci převodu scény do interní reprezentace.

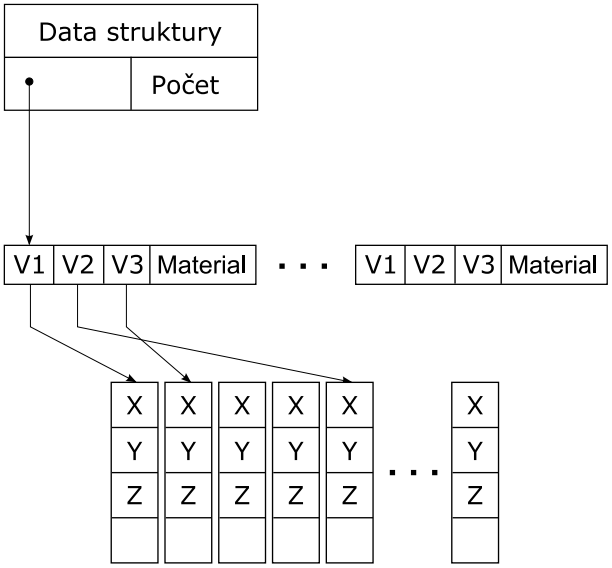
Interní formát Nahrávání obsáhlejších modelů z textových souborů je zdlouhavé. Pro rychlé nahrání modelu, zejména pro účely ladění programu, byl vytvořen vlastní (velmi jednoduchý, binární) interní formát souborů. V podstatě se jedná o prosté uložení interní reprezentace modelu do souboru.

Výchozím objektem v hierarchii objektů implementujících nahrávání souborů je velmi jednoduchá třída `FileLoader`, obsahující jedinou virtuální funkci. Tato funkce načte jí zadaný soubor a vrátí zpět nahraný objekt, přičemž nahrávání scény je následně zastřešeno jednotným rozhraním. Singleton třída `SceneLoader` obsahuje rozhraní pro správu existujících objektů, jež nahrávají soubory. Rozhraní na základě přípony zadaného souboru určí, který z objektů se použije pro načtení souboru. Registrace jednotlivých formátů se provádí automaticky prostřednictvím statické inicializace. V souboru, implementujícím daný formát, je obsažena zástupná globální proměnná, která se staticky inicializuje na základě výsledku volání funkce, jež přidává daný formát. Tímto způsobem je zajištěna možnost nahrání souboru bez nutnosti určování typu souboru uživatelem.

4.1.1 Interní reprezentace scény

Pro účely sledování paprsku je použita klasická reprezentace trojúhelníkové sítě. Vrcholy trojúhelníků jsou uloženy v jednom souvislém poli a trojúhelníky jsou tvořeny indexováním vrcholů v tomto poli. Každému vrcholu je přiřazena přesně jedna normála, není potřeba přidávat dodatečné údaje do indexů vrcholů a reprezentace tak velmi dobře funguje pro hladké povrchy. Nevýhodou je nutnost několika kopií vrcholů pro geometrii obsahující ostré hrany.

Tato reprezentace je použita jak pro načítání modelů, tak v kombinaci s akcelerační strukturou. V případě použití reprezentace akcelerační strukturou obsahuje každá z buněk nebo uzlů, použitých akceleračních struktur, údaj o počtu trojúhelníků v buňce a odkaz do pole trojúhelníků. Každá buňka má svůj vlastní seznam trojúhelníků. Pokud se trojúhelník umístí do několika buněk struktury, dojde k vícenásobnému umístění trojúhelníku v daném poli. Při velikosti indexů se jeví tato duplikace výhodnější než přidávání další úrovně do reprezentace a každý trojúhelník



Obrázek 4.1: Reprezentace geometrie scény.

je navíc doplněn o identifikaci jeho materiálu. Schéma reprezentace je znázorněna na obrázku 4.1.

Do úvahy přichází i provedení úpravy tím, že se vynechá indexovací úroveň. Z akcelerační struktury vede poté odkaz přímo do pole vrcholů, kde jsou vrcholy trojúhelníku umístěny za sebou. Výhodou tohoto řešené je zajištění koherence při přístupu do paměti, naopak nevýhodou je několikanásobné zvýšení požadavků na velikost paměti. Zvětšení koherence je navíc negativně ovlivněno paralelním zpracováním, při kterém blízké paprsky často procházejí mírně odlišnými buňkami a rovněž často se trojúhelníky dostávají při použití dělení prostoru do několika buněk. Při použití této reprezentace jsou v každé buňce uloženy kopie vrcholů. Sousední vlákna by tedy navzájem vytlačovala data z cache.

Další možná varianta reprezentace nabízí rychlejší výpočet průsečíku trojúhelníku s paprskem. Tato *projekční metoda* [19] využívá skutečnosti, že po projekci trojúhelníku a paprsku do roviny se barycentrické souřadnice nezmění. Při výpočtu je při každém testu opětovně prováděna řada stejných operací, proto se tyto operace předpočítávají a ukládají se namísto standardní reprezentace. Tento algoritmus umožňuje rychlejší výpočet průsečíku, ale za cenu zvětšení nároků na paměť a její propustnost. Ani tato reprezentace není použita z důvodu zvýšených požadavků na paměť.

4.2 Stavba akcelerační struktury

4.2.1 Uniformní mřížka

Implementace stavby uniformní mřížky používající jedno jádro procesoru je velmi jednoduchá. Pro zjištění rozsahu obálky scény a rozlišení mřížky je pro každý trojúhelník určena jeho poloha v mřížce a do patřičných buněk je uložena reference. Jediným problémem tak zůstává určení rozlišení mřížky. Stejným způsobem jako v [16] je nejdříve určen počet buněk na ose nejkratší strany modelu a jejich počet pro ostatní osy se určí tak, aby byly buňky krychlové. Počet buněk je možno zadat pro každý model ručně, nebo nechat rozhodnutí na jednoduché heuristice. Při jejím použití je počet buněk určen podle předpisu $4 \cdot \sqrt[4]{n}$, kde n je počet trojúhelníků. Přesná podoba vzorce byla určena empiricky, z měření rychlosti pro různé rozlišení mřížky.

4.2.2 Kd-strom

Pro rychlou stavbu stromu je použit algoritmus popsaný v [17], který vychází ze skutečnosti, že ve vyšších úrovních stromu není důležité zcela přesné umístění roviny. Pro každou osu je určen konečný počet pozic, na které může být umístěna dělící rovina. Pro správné určení počtu těles na obou stranách dělící roviny je použito min-max rozdělování trojúhelníku do tříd.

Pro každou osu jsou použity dvě sady tříd. Jedna z nich uchovává informace o počtu trojúhelníků v ní začínajících a druhá o počtu trojúhelníků v ní končících. Z této informace lze jednoduše určit počty trojúhelníků na obou stranách dělící roviny. Jedním průchodem pole tříd je možno za pomoci lokální hladové SAH heuristiky vypočítat a najít dělení s nejmenší cenou. Plocha obálky je efektivně vypočítávána inkrementálně. Při jakémkoli umístění dělící roviny se plocha stran rovnoběžných s dělící rovinou nemění. Protože je rovina postupně posouvána po stále stejných krocích, je možné aktuální hodnotu obsahu obálky vypočítávat inkrementálně pomocí jediné operace součtu nebo rozdílu. Při výpočtu jsou použity SSE instrukce a výpočet tedy probíhá pro všechny osy zároveň.

Od určité úrovně stromu je nezbytné opustit aproximaci výpočtu a použít klasické přesné určení dělící roviny. Implementace přesného výpočtu ceny používá algoritmus se složitostí $O(N \log^2 N)$ popsaný v [22]. Algoritmus využívá skutečnosti, že funkce ceny je po částech lineární. Cena může nabývat minima pouze na pozici rovin obálky trojúhelníku. Pro každou osu je tedy vytvořen seznam „událostí“ představujících začátek, konec trojúhelníku nebo trojúhelník rovnoběžný s dělící rovinou. Tento seznam je seřazen a jeho procházením je možno zjistit počet trojúhelníků na obou stranách dělící roviny a je možno nalézt pozici s nejmenší cenou.

Při rozdělování trojúhelníků do potomků může docházet k chybám vlivem numerické nepřesnosti. Pokud je poloha dělící roviny vypočítána obrácením vztahu pro výpočet třídy, nemusí být výsledek dostatečně přesný a po porovnání obálky trojúhelníku s touto rovinou je možné ho chybně umístit na špatnou stranu dělící roviny. Nabízí se řešením, při kterém je pro rozdělování používána stejná hodnota jako pro rozdělení do tříd. Během rozdělování jsou z obálky trojúhelníku opětovně vypočítány indexy tříd a ty jsou následně porovnány s indexem třídy s nejlepší cenou. Výsledkem je spolehlivé rozdělování trojúhelníků do potomků a to za cenu několikanásobného zvětšení počtu matematických operací ve fázi rozdělování.

Plná podpora celočíselných typů a jejich operací procesorem G80 umožňuje použít úspornou reprezentaci používanou na CPU. Uzel stromu je možné vyjádřit pouhými 8 byty. Při použití operací bitového posunu a logického součinu je možnou pro každou hodnotu použít přesně potřebný počet bitů a omezit paměťové nároky. Ukazatel na levého potomka je implicitní, levý potomek je vždy umístěn v paměti ihned za rodičem. Umístění uzlů v paměti odpovídá pořadí uzlů při průchodu stromu do hloubky, a proto je vždy alespoň jeden potomek umístěn blízko rodičů a zlepšuje koherenci přístupu do paměti. Pro efektivní přístup k uzlům stromu během procházení jsou tyto, na grafické kartě, umístěny ve formě textury.

Alokace paměti Počet uzlů výsledného stromu není možné určit předem, paměť je potřeba při stavbě dynamicky alokovat. Použití akcelerační struktury na grafické kartě vyžaduje, aby uzly stromu a indexy trojúhelníků byly uloženy ve spojitém poli. Tento požadavek je splněn použitím knihovní funkce `realloc`, pro snadnější použití zastřešené třídou `GrowArray`. Uvedená funkce umožňuje změnit velikost již alokovaného bloku paměti. Pokud je v paměti dostatek volného místa za již alokovaným blokem, je pouze zvětšena jeho velikost a není potřeba provádět kopírování na jiné místo. Zastřešující objekt umožňuje žádat i o jednotlivé prvky pole. Po vyčerpání kapacity je velikost pole zdvojnásobena. Při použití knihovní funkce nelze využít přímé ukazatele do pole, po realokaci může být pole umístěno do jiné části paměti a ukazatel je neplatný.

Pro alokaci paměti pro indexy trojúhelníků, předávaných do potomků, lze úspěšně využít pořadí jejich použití. Při kroku na nižší úroveň stromu je vyžadována nová paměť, po návratu do rodiče je možno paměť opětovně uvolnit. Provádět při každém kroku explicitní alokaci a dealokaci paměti se jeví jako zbytečné, alokace by šlo nahradit jedním velkým polem, v němž by se postupně posouval ukazatel na první volný blok. Velikost tohoto pole však není možno určit předem. Řešením je jednoduchá implementace správy paměti známé pod názvem *memory pool*. Memory pool je implementován ve třídě `TailPool`, která alokuje paměť ve velkých blocích a ukládá si je do zřetězeného seznamu. Po vyčerpání volného místa bloku je použit další blok v seznamu a pokud již seznam nemá další prvek, je alokován nový blok. Při uvolňování paměti je pouze zvětšována velikost volného místa v bloku a postupuje se na začátek seznamu. Paměť je dealokována až při zrušení objektu. Použitím těchto nástrojů se minimalizuje režie alokace paměti.

4.2.3 BVH

Stavba hierarchie obálek má mnoho společného se stavbou kd-stromu. Opět se pro odhad ceny úspěšně používá lokální hladová SAH heuristika a urychlení výpočtu dělení použitím umisťování trojúhelníků do tříd. Protože hierarchie na rozdíl od stromu dělí objekty a ne prostor, je potřeba poněkuď odlišný přístup. Pro určení dělení trojúhelníků je použit algoritmus používající četnost těžiště trojúhelníků [10]. Implementace opět využívá pro urychlení SSE instrukce. Správa paměti je totožná s postupem použitým při stavbě kd-stromu.

Při stavbě každého uzlu jsou v něm umístěné trojúhelníky rozděleny na základě svého těžiště do tříd. Po shromáždění těchto informací se určí nejlepší umístění hranice dělení a na jejím základě se přidělí trojúhelníky potomkům.

Pro všechny trojúhelníky v uzlu je určeno těžiště a podle jeho pozice je trojúhelník umístěn do jedné ze tříd. Třídy jsou rovnoměrně rozděleny pro každou z prostorových os. Pro zvýšení rozlišení tříd nejsou tyto rozprostřeny po celé délce obálky uzlu, ale po známém rozsahu těžišť trojúhelníku v uzlu. Třída uchovává informace o počtu umístěných trojúhelníků i obálku přes všechny tyto trojúhelníky. Uchované informace postačují k určení ceny při umístění dělící roviny na rozhraní tříd. Pro urychlení výpočtu se pomocí SSE instrukcí počítají najednou souřadnice třídy ve všech třech osách.

Z rozdělení trojúhelníků do tříd je možné ve dvou průchodech vypočítat ceny dělení. Každý průchod postupně sčítá počty trojúhelníků ve třídách a slučuje dohromady obálky. První průchod postupuje z levé strany tříd a výsledky ukládá do pomocného pole. Druhý průchod postupuje zprava a za pomoci informací z prvního průchodu vypočítává cenu dělení. V implementaci jsou použity SSE instrukce pro slučování obálek a výpočet obsahu povrchu obálek.

Při rozdělování trojúhelníků do potomků je nutno vyřešit zcela stejný problém jako při stavbě stromu. Určení pozice dělící hranice z indexu dělící třídy je numericky nestabilní. Použité řešení problému se principiálně liší od řešení použitého v kd-stromu. Každá třída si uchovává minimální pozici těžiště, relevantní osy, pro všechny v ní umístěné trojúhelníky. Tento údaj je tedy potřeba aktualizovat při rozdělování trojúhelníku do tříd. Použití této hodnoty zaručuje správnost rozdělení trojúhelníků.

4.3 Sledování paprsku

4.3.1 Traverzace

Implementace traverzace tří použitých akceleračních struktur má společné rysy. Ve všech případech se celý proces traverzace včetně stínování a generování sekundárních paprsků jeví jako jediný kernel v prostředí CUDA. Každý paprsek má svoje vlastní vlákno, vlákna jsou pak orga-

nizována do bloku o počtu 64 vláken. Uvedená hodnota je v dokumentaci [9] doporučována jako minimální počet vláken pro plné využití procesoru. Vláknům jsou přiřazeny paprsky patřící v obraze bloku 8 x 8 pixelů. Organizace paprsků do čtverce zvyšuje koherenci mezi paprsky, čímž se zvětšuje pravděpodobnost společného průchodu paprsků scénou. Zvětšení velikosti bloku je možné, nepřináší však žádné výhody, v případě paketového procházení může mít naopak za následek zmenšení koherence paprsků paketu a zvětšení počtu traverzačních kroků.

Výpočet průsečíku paprsku s obdélníkem reprezentujícím obálku je rovněž společný pro všechny tři případy. Implementace je upravenou verzí algoritmu popsaného v [18]. Pro každou prostorovou osu se nalezne interval paprsku mezi rovinami umístěnými v minimálním a maximálním rozsahu obdélníku. Pokud je průsečík tří intervalů prázdný, paprsek obdélník neprotíná, v opačném případě ho naopak protne. Velkou výhodou je skutečnost, že implementace neobsahuje větvení kódu. Veškeré porovnávání a následné větvení je úspěšně nahrazeno použitím instrukcí `min` a `max`.

4.3.1.1 Uniformní mřížka

Algortimus procházení paprsku strukturou je implementován podle algoritmů popsaných v [7, 3] a prakticky se nijak od nich neliší ani není výrazně upraven pro specifické vlastnosti GPU.

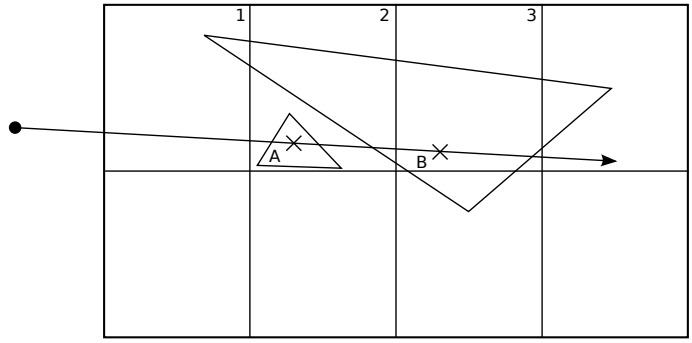
Kostra algoritmu je velmi jednoduchá. Po určení startovací buňky je v každém kroce proveden test paprsku vůči trojúhelníkům obsažených v dané buňce. Poté se algoritmus přesune do další buňky na cestě paprsku. Výpočet končí nalezením průsečíku nebo opuštěním mřížky.

Pro průchod mřížkou je použita obdoba 3D kreslení čáry za použití digitálního diferenciálního analyzéru. Použitý algoritmus musí na rozdíl od kreslení čáry zajistit průchod skutečně všemi buňkami mřížky, kterými paprsek prochází. Pro správný průchod stačí znát hodnotu parametru přímky v průsečíku s rovinou kolmou na osu posunu a hodnotu o kterou se tento parametr změní v případě posunu. Pro další postup je jednoduše použita osa, pro kterou je tato hodnota parametru nejnižší. Nevýhodou je velký počet větvení v kódu určujícím příští buňku traverzace.

Pro inicializaci algoritmu je využito nalezení průsečíku paprsku s obdélníkem. Z minimální hodnoty parametru přímky v průsečíku je určena startovací buňka. Pokud je počátek paprsku uvnitř mřížky je omezením hodnoty parametru na kladné hodnoty správně určena počáteční buňka uvnitř mřížky. Inicializace je dokončena výpočtem změny parametru pro posun o jednu buňku pro každou z os a výpočtem parametru přímky pro první přechod.

Při vyhodnocování průsečíku je potřeba zjistit, zda leží v aktuální buňce. Pokud trojuhelník protíná více buněk, může způsobit falešný průsečík a tedy chybu ve vykreslování. Tato situace je znázorněna na obrázku 4.2. V uvedeném případě bude při procházení buňky 1 úspěšně otestován trojúheník B, i když ve skutečnosti nejbližším průsečíkem je průsečík paprsku s trojúhelníkem A v buňce 2. Tomuto problému se zamezí, pokud se testuje přítomnost průsečíku v aktuální buňce.

Pro snížení počtu použitých registrů je použito paměti konstant. Algoritmus průchodu potřebuje znát pro každou z prostorových os směr pohybu a konečnou buňku v tomto směru, což vyžaduje dohromady šest registrů. Při této implementaci převyšuje počet použitých registrů hodnotu 32 a negativně ovlivňuje obsazenost procesoru. Každá z těchto proměných může přitom ze dvou hodnot nabývat jenom jedinou. Při zohlednění závislosti průchodu na směru paprsku je tedy uskutečnitelných jen 8 kombinací, připravených pro použití v paměti konstant. Každé vlákno si poté vypočítá ze směru paprsku index do těchto hodnot, takže se ušetří celkem 5 registrů. To představuje dostatečné množství na zvýšení obsazenosti procesoru. Paměť konstant je při jednotném přístupu stejně rychlá jako registry a nutnost výpočtu adresy z indexu je zanedbatelná vůči zrychlení díky vyšší obsazenosti. Zajímavé je, že větvení v kódu pro procházení, které snižuje výkon, umožňuje v tomto případě jednotný přístup do paměti konstant.



Obrázek 4.2: Falešný průsečík paprsku s trojúhelníkem.

Spolupráce vláken Jedním z problémů základní verze algoritmu je smyčka testující jednotlivé trojúhelníky. Počet opakování testování průsečíku je dán maximálním počtem trojúhelníků v buňkách, které jsou vlákny v daném kroku navštíveny. Problém nastává v případě, když mají buňky výrazně odlišný počet trojúhelníků. Jedno jediné vlákno může totiž způsobit prodlevu testování tím, že ostatní vlákna na něj čekají. Prvním pokusem o zrychlení celého testování bylo rozprostření testování průsečíku přes všechna vlákna.

Hodnoty každého vlákna jsou zkopírovány do sdílené paměti, která poté obsahuje údaje jak o paprsku (počátek, směr) tak i o nejlepším nalezeném řešení (zasáhnutý trojúhelník, vzdálenost průsečíku). Díky tomu může jakékoli vlákno počítat průsečík trojúhelníků s libovolným paprskem v bloku.

Každé vlákno se pokusí najít další neprázdnou buňku na cestě svého paprsku. Pokud některé paprsky již našly průsečík nebo opustily mřížku, dochází k řešení zhušťovacího problému. Po jeho provedení je možno ukončit výpočet, pokud jsou již všechny paprsky neaktivní. V opačném případě je možno uložit do paměti informace o počtu trojúhelníků v nalezené buňce a index paprsku.

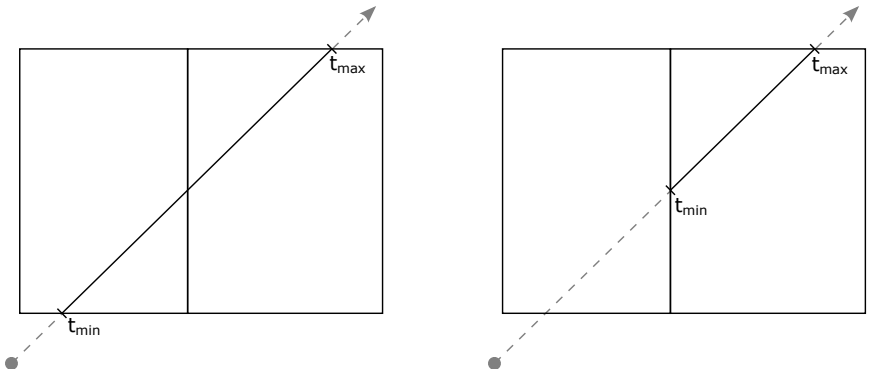
Z informací uložených do pole v předešlém kroku je možno rozdělit testy průsečíků vláknům. Vlákna lineárně procházejí pole, sečítají počet trojúhelníků a porovnávají ho se svým indexem v bloku, čímž zjistí, který paprsek mají otestovat se kterým trojúhelníkem.

Každé vlákno nyní provede test jednoho paprsku s jedním trojúhelníkem. Nad vzdálenostmi průsečíků se provede segmentová paralelní redukce. Jejím výsledkem je minimální nalezený průsečík pro každé vlákno. Pokud se výsledek redukce rovná hodnotě vlákna, může vlákno porovnat tuto hodnotu s nejlepší dosud dosaženou hodnotou pro daný paprsek.

Algoritmus se následně vrací do předchozího kroku, kde ověří, zda již otestoval všechny paprsky. V takovém případě se algoritmus vrací do první části, v opačném případě je opět rozdělen zbytek testů.

Problémem tohoto algoritmu v porovnání se základní verzi je velké množství dodatečně prováděných operací. Ukázalo se, že testování u tohoto algoritmu bylo několikrát pomalejší než u standardní verze. Použití algoritmu s rozprostřením testování průsečíku přes všechna vlákna bylo tedy zavrhnuto a jeho další vývoj byl zastaven.

Paketové procházení Dalším pokusem o zrychlení bylo použití paketového procházení dle [23]. Tento algoritmus prochází mřížku po řezech, ze směrů paprsků se určí převažující směr a ten je použit jako hlavní osa procházení. V každém kroku se algoritmus pohne o jednu buňku dále ve směru hlavní osy. V každém kroku se inkrementálně určuje rozsah buněk, které paprsky paketu protínají v ostatních osách (v řezu rovinou kolmou na hlavní osu). Paprsky paketu jsou



(a) Prvotní průchod buňkou (b) Průchod po restartu

Obrázek 4.3: Princip algoritmu *restart*. Po restartu je správného procházení dosaženo modifikací aktivního rozsahu paprsku

testovány oproti trojúhelníkům všech buněk řezu.

Tento algortimus nebyl zamýšlen pro použití na GPU. Je tedy problematické implementovat jeho vlastnosti, navíc nelze využít dvou jeho zásadních přínosů. Podstatného zvýšení výkonu je dosaženo rychlým zamítnutím testování trojúhleníku pro celý paket, na grafické kartě, kde má každý paprsek přiřazeno jedno vlákno, je však tento test irelevantní. Provádět jej několika vlákny, zatím co ostatní čekají na výsledek, nepřináší výkonnostní zlepšení.

Při paketovém procházení mřížky se několikanásobně zvýší počet testování průsečíku paprsku s jedním trojúhelníkem. Tomuto navýšení lze zamezit prostřednictvím tzv. MailBoxingu. Jeho aplikace na GPU je však velmi obtížně proveditelná, protože každý paket by vyžadoval vlastní paměť pro Mailboxing a celkové nároky by byly neúnosně velké.

Ze všech pozitiv zbývá již jen uniformní přístup do paměti a společné větvení. Lze říci, že celkově výhody nestačí vyvážit nevýhodu procházení značně vetším počtem buněk.

V testovací implementaci byly vyzkoušeny dvě velikosti paketu. První z verzí využila všech 64 vláken bloku pro jeden paket o velikosti 8 x 8. Druhá verze použila celkem 4 pakety o velikosti 4 x 4, přitom nebylo možno jednoduše používat instrukce synchronizace. Testovací implementace byla pro několik zkoušených scén výrazně pomalejší v porovnání s původní verzí algoritmu i pro primární paprsky. Použití i tohoto algoritmu tedy bylo zavrhnuto a jeho další vývoj byl taktéž ukončen.

4.3.1.2 Kd-strom

Základem procházení kd-stromu je algoritmus *restart* [6], nevyžadující k procházení žádnou formu zásobníku. Po návštěvě listu stromu, ve kterém nebyl nalezen průsečík, je procházení spuštěno opět z kořene stromu. Průchodu rozdílnými buňkami je docíleno změnou aktivního rozsahu parametru paprsku. Tento postup je znázorněn na obrázku 4.3. Při restartu je zachována hodnota t_{min} a při opětovném průchodu buňkou se na základně rozsahu paprsku pokračuje ve vzdálenější buňce.

Základní verzi může mírně vylepšit modifikace *pushdown* [12], při níž není procházení spouštěno znova od kořene stromu, nýbrž procházení pokračuje od uzlu, ve kterém je potřeba ho větvit. Do okamžiku větvení je v každém kroku uložen aktuální uzel jako uzel restartu, změnu restartovacího uzlu je možno opětovně obnovit po provedení restartu. Vliv této modifikace na výkon je velmi nejednoznačný. Možné zrychlení je ovlivněno geometrií scény a pohledem, v nejhorším

případě je nutno provést větvení již v uzlu stromu a rozdíl ve výkonnosti může být tedy nulový.

Nevýhodou algoritmu *restart* je značné navýšení počtu traverzačních kroků. Tento nedostatek se snaží zmírnit metoda *short-stack*[12]. Při této metodě je algoritmus kombinací algoritmu restart a klasického použití zásobníku. Pro traverzaci je použit malý zásobník s omezeným počtem prvků a traverzace není ukončena při vyprázdnění zásobníku, nýbrž pokračuje podle algoritmu restart.

Algoritmus *restart* má jeden zásadní nedostatek, že neumožňuje spolehlivě procházet stromy s „plochými“ buňkami (s nulovým objemem). Modifikace algoritmu *restart* považuje situaci, kdy je t_{hit} roven t_{min}, za opětovný průchod po restartu a pokračuje do vzdálenějšího potomka. Nelze tedy tento případ odlišit od případu buněk s nenulovým objemem. Trojúhelníky umístěné v plochých buňkách nejsou otestovány na průsečík s paprskem. Tomuto problému u stromu s plochými buňkami je potřeba předejít již při stavbě stromu, a ne ho řešit až během procházení.

Implementace zásobníku Zásobník je umístěn ve sdílené paměti, která jediná dokáže rozumně splnit požadavky vláken procházejících nezávisle na sobě. Velikost zásobníku je ovlivněna velikostí dostupné sdílené paměti. Je potřeba docílit toho, aby množství použité sdílené paměti neomezovalo obsaditelnost procesoru. Základním faktorem omezení obsaditelnosti procesoru je počet použitých registrů. Předpokladem toho, aby velikost použité sdílené paměti nezvyšovala toto omezení obsaditelnosti procesoru, má každý blok vláken k dispozici místo pro zásobník o čtyřech prvcích. Velikost zásobníku je tedy velmi malá, přesto jeho použití vede ke zvýšení výkonu více než o 100 %.

Zásobník je implementován pomocí kruhové fronty. Rychlost operace zbytku po dělení je u současné verze velmi malá, proto je velikost zásobníku účinně omezena na mocniny dvou. U nich lze operaci zbytku po dělení nahradit rychlým bitovým součinem. Počet prvků na zásobníku se uchovává v samostatné proměnné, jejž rozsah je jednoduše určován instrukcí max.

Na zásobník je potřeba ukládat adresu uzlu, minimální a maximální rozsah parametru paprsku pro tento uzel. Každá z těchto hodnot je ukládána do vlastního pole ve sdílené paměti. Vzhledem k nezávislému chování vláken nelze úplně zabránit konfliktům bank sdílené paměti. Při zásobníku o čtyřech prvcích a 16ti vláknech v půl-warpu by například při použití zásobníku v prvním kroku došlo ke čtyřnásobnému konfliktu sdílené paměti. Proto je pro každé další čtyři vlákna posunut začátek zásobníku o jeden prvek pole, čímž se ve výše zmíněném případě eliminují veškeré konflikty. Divergací průchodu paprsků určité konflikty vznikají, celkově je jich ale méně.

Indexace polí Implementace výpočtu průsečíku paprsku s dělící rovinou naráží na problém, že nelze indexovat proměnné, umístěné v registrech a obsahující počátek a směr paprsku (na základě indexu udávajícího orientaci dělící roviny). Je nevyhnutné zajistit přístup k těmto proměnným. Při pokusu o jejich indexaci dochází ke kopírování proměnných do lokální paměti, kterou je možno indexovat. Lokální paměť není cachovaná a je velmi pomalá. Výsledná rychlost algoritmu je pak nepřijatelně nízká.

Tento problém lze řešit několika způsoby. Jedním z nich je zkopírování proměnných do sdílené paměti, kde mohou být indexovány. Tento přístup však v tomto případě zabírá příliš mnoho sdílené paměti. Další řešení se zakládají na kopírování potřebných hodnot do dočasných proměnných. Nevýhodou je zvýšení počtu potřebných registrů. Výběr proměnné pro kopírování lze rovněž provést více způsoby. Nejjednodušší možností je větvení programu, což má však negativní vliv na rychlost programu. Dalším způsobem je použití vektorových operací, konrétně skalárního součinu. Vynásobením proměnných vektorem, obsahujícím na potřebných pozicích hodnoty 1 a 0, je možno získat správnou hodnotu bez větvení kódu. Vektory pro násobení mohou být umístěny ve sdílené paměti nebo v paměti konstant, kde mohou být indexovány.

Algoritmus 1 Procházení kd-stromu

```
R = (O, D)
t ← ∞
U, U_restart ← kořen kd-stromu
pushdown ← true
T_ID ← Pozice vlákna v bloku
shared Z[] // zásobník traverzace

(t_min, t_max) ← průsečík (R,obálka stromu)
t_global ← t_max

loop
  if t_min ≥ t_max then break
  if je list(U) then
    Otestuj trojúhelníky na průsečík s R
    Uprav t
    if není prázdný(Z) then
      (U, t_min, t_max) ← čti (Z, T_ID)
    else
      t_min ← t_max
      t_max ← t_global
      U ← U_restart
    end if
  else
    (o, d) ← vyber osu(R, U.osa)
    (P_b, P_v) ← seřaď(o, d, P_l, P_p)
    t_hit ← (U.rovina − o)/d
    if t_hit ≥ t_max ∨ t_hit < 0 then
      U ← P_b
    else if t_hit ≤ t_min then
      U ← P_v
    else
      pushdown ← false
      vlož(Z, T_ID, P_v, t_hit, t_max)
      U ← P_b
      t_max ← t_hit
    end if
    if pushdown then U_restart ← U
  end if
end loop
```

Nevýhodou je vznik konfliktů při prístupu do těchto pamětí. V implementaci byly vyzkoušeny oba zmiňované přístupy a rozdíly v jejich rychlosti byly na hranici chyby měření. V konečné implementaci je tedy použito syntakticky čistšího větvení kódu.

4.3.1.3 BVH

BVH procházení je implementací algoritmu popsaného v [10]. Na rozdíl od předchozích dvou diskutovaných algoritmů BVH strom využívá i paketového procházení. Procházení paketu ve výsledku přináší výhodu menších nároků na přístup do paměti a zároveň snižuje počet větvení kódu s rozdílným výsledkem. Kromě toho má použití BVH stromu ještě další výhodu - není nutno si pamatovat pro každý paprsek jeho rozsah v aktuálním uzlu ani jej ukládat na zásobník, postačí ukládat na zásobník pouze adresu uzlu. Jelikož všechna vlákna procházejí stejnými uzly, může být zásobník společný pro všechna vlákna. Nároky na velikost sdílené paměti, použité na zásobník, jsou tedy dramaticky nižší než při implementaci kd-stromu. Do sdílené paměti se tak vejde zásobník o velikosti dostatečující pro celý průchod stromem a není nutno provádět restart procházení.

Základní dělení výpočtu je obdobné jako u předchozích algoritmů. Velikost paketu je tedy 8 x 8 paprsků, což může způsobit již značné snížení koherence sekundárních paprsků. Problémy lze očekávat především u odražených paprsků na zakřivených plochách.

Základní kostra algoritmu je totožná s kd-stromem. Pokud je aktuální uzel listem, jsou testovány jeho trojúhelníky na průsečík s paprskem. Pokud tomu tak není, pak algoritmus rozhoduje, který potomek uzlu bude navštíven. Podmínkou návštěvy daného potomka vláknem je protnutí obálky potomka vláknem a dále to, aby oblast daná obálkou byla blíže k počátku paprsku než současný nejbližší nalezený průsečík.

Pro implementaci rozhodování je úspěšně využito možnosti současného zápisu do sdílené paměti. Sdílená paměť obsahuje malé pole o čtyřech prvcích, z nichž každý reprezentuje jednu z možností, kterého potomka vlákno naštíví - žádného, levého, pravého či oba potomky. Zmíněné pole se inicializuje na hodnotu 0. Každé vlákno zapíše do prvku, který reprezentuje jeho rozhodnutí, hodnotu 1 a tím je umožněno vzájemné informování vláken o dalším postupu - bez nutnosti použít např. paralelní redukci. Při zápisu do prvků může dojít ke konfliktům a serializaci zápisů. Zjistit, zda je tento způsob rozhodování rychlejší než např. použití paralelní redukce, je těžké a při zvolené velikosti bloku změřit tuto rychlost prakticky nemožné.

Pokud všechny vlákna chtějí pokračovat do stejného potomka, změní se pouze aktuální uzel a pokračuje se v hlavní smyčce. Pokud žádné vlákno nepotřebuje pokračovat, je další uzel získán ze zásobníku. Po vyprázdnění zásobníku se výpočet ukončí. V případě, že chtějí různá vlákna pokračovat do různých potomků, pokračuje se do potomka, kterého chce navštívit více vláken. Toto rozhodnutí je provedeno pomocí paralelní redukce. Vlákno uloží svůj požadavek do pole ve sdílené paměti, kde je reprezentován hodnotami -1, 0 a 1, nad polem je pak provedena paralelní redukce. Znaménko výsledné hodnoty určuje, kterého potomka chce navštívit většina vláken jako prvního, druhý potomek se uloží na zásobník.

V této části se implementace mírně odlišuje od algoritmu popsaného v [10], který nebere v tomto místě v úvahu, že některé paprsky nechtějí navštívit žádného z potomků. Testuje se podmínka, zda chce paprsek navštívit pravého potomka a zda je tento blíže než potomek levý. Nesplnění uvedených podmínek se vyhodnotí jako požadavek pro navštívení levého potomka. Paprsky, které již dále procházet nepotřebují, ovlivňují výsledek rozhodování a zvyšují počet kroků traverzace. Upravený kód ukládá do pole hodnotu 0, pokud vlákno nepotřebuje navštívit ani jednoho potomka. Popsaná modifikace zrychlila procházení sekundárních paprsků až o 20 % ve scénách s vysokým počtem lesklých povrchů.

Výhoda paketového procházení se projevuje v lepší optimalizaci přístupu do paměti. K uzlům

stromu není, na rozdíl od předchozích případů, přistupováno pomocí textur. Velikost uzlu BVH stromu je čtyřikrát větší než je velikost uzlu mřížky i kd-stromu. Nejmenší možnou velikostí uzlu BVH stromu je 28 bytů. Při této velikosti by bylo nutno rozdělit data do dvou textur (max. velikost prvku textury je 16 bytů). Jelikož jednotlivá vlákna nepotřebují přistupovat k rozdílným uzlům stromu, mohou vzájemně efektivně spolupracovat.

Vzhledem k tomu, že je vždy nutno otestovat paprsek s obálkami obou potomků, jsou tyto obálky v paměti umístěny za sebou. Velikost uzlu je zvětšena na 32 bytů a oba potomci tedy zabírají 64 bytů, což je přesně velikost bloku při sdružování přístupu do paměti. Všechna data potřebná pro pokračování traverzace jsou načtena pouze jedním přístupem do paměti. Pro další redukci přístupů do paměti je navýšen počet informací ukládaných na zásobník. Při paketovém procházení jsou nároky na zásobník minimalizovány a ve sdílené paměti je dostatek místa pro zásobník. Po vybrání uzlu ze zásobníku je potřeba pouze odkázat na potomky, v případě listu se odkazuje na trojúhelníky, případně na počet trojúhelníků v uzlu. Je zbytečné načítat pro získání uvedených hodnot celý blok o 64 bytech, proto jsou obě tyto hodnoty ukládány na zásobník. I po dvojnásobném zvětšení nutné velikosti zásobníku je ve sdílení paměti stále dostatek místa. Při častém používání zásobníku přitom klesne počet přístupů do paměti.

Paralelní redukce V procházení BVH stromu je použit alogritmus paralelní redukce. Grafická karta se chová jako CRCW PRAM počítač, takže lze využít algoritmu běžně uváděného v literatuře. Pro optimální výkon je však nutno samotnou implementaci upravit. Následující optimalizace jsou převzaty z ukázkového příkladu CUDA [11]. Základem úpravy paralelní redukce je rozbalení smyčky, která provádí redukci. Počet opakování je (díky fixní velikosti bloku a tedy počtu vláken) předem znám. Počet aktivních vláken se po každém kroku sníží na polovinu. Používání podmínky pro výběr aktivních vláken v každém kroku by bylo neefektivní. Pro zamezení větvení kódu provádějí výpočet i ta vlákna, která již nejsou aktivní. Korektnost algoritmu není touto úpravou ovlivněna.

Při zvolené velikosti bloku není nutná synchronizace po každém kroku. Již v prvním kroku je použito pouze 32 vláken, tedy jeden warp. Vlákna v rámci jednoho warpu jsou SIMD synchronní, vlákna stačí synchronizovat před a po provedením paralelní redukce.

Klasický algoritmus přistupuje do paměti v prvním kroku se stride 2. Tento přístup způsobuje konflikt bank sdílené paměti. Jak bylo popsáno dříve, počet aktivních vláken není v žádném kroku omezován, počet konfliktů je tedy v každém kroku stejný. Řešením problému je změna pořadí, v kterém se sčítají prvky a vlákna se stávají neaktivními. Rozdíl v přístupu do paměti je znázorněn na obrázku 4.4. Jednolivá vlákna nyní přistupují k lineárně se zvyšujícím adresám a nedochází k žádnému konfliktu bank.

4.3.2 Generování sekundárních paprsků

Výpočet průsečíku paprsku s objektem ve scéně je možno spustit pro každou z akceleračních struktur v celkem třech konfiguracích, lišících se typem používaných sekundárních paprsků a následně i výpočetní náročností. Pro každou z těchto konfigurací existuje z důvodů optimalizace vlastní funkce spouštějící výpočet. Do úvahy přicházejí následující konfigurace:

Primární konfigurace Každé vlákno se pokusí najít průsečík paprsku s objektem ve scéně, v případě úspěšného nálezu je vypočtena barva pomocí osvětlovacího modelu.

Primární a stínové konfigurace Každé vlákno se pokusí najít průsečík paprsku s objektem ve scéně, v případě úspěšného nálezu je vyslán stínový paprsek směrem ke každému světlu. Výsledky osvětlovacího modelu pro každé nezastíněné světlo jsou následně sečteny do výsledné barvy.

Algoritmus 2 Paketové procházení BVH stromu

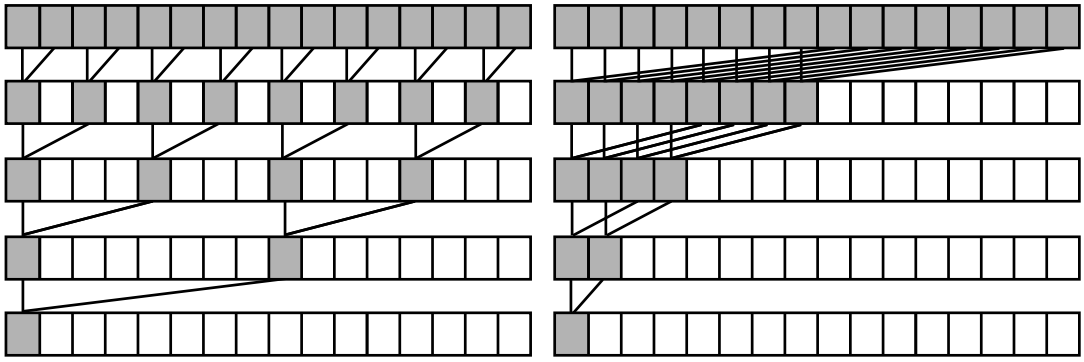
$R = (O, D)$
$t \leftarrow \infty$
$U \leftarrow$ kořen BVH stromu
$T_{ID} \leftarrow$ Pozice vlákna v bloku

shared P_l, P_r // Sdílené úložiště pro potomky uzlu
shared $M[\,]$ // Pole pro paralelní redukci
shared Z // Sdílený zásobník traverzace

loop
  if je list(U) **then**
    Otestuj trojúhelníky na průsečík s R
    Uprav t
    if je prazdny(Z) **then break**
    $U \leftarrow$ čti(Z)
  else
    if $T_{ID} \leq 16$) **then**
      $(P_l, P_r) \leftarrow$ načti potomky(U)
    end if

    $(l_{min}, l_{max}) \leftarrow$ průsečík(R, P_l)
    $(r_{min}, r_{max}) \leftarrow$ průsečík(R, P_r)
    $p_l \leftarrow (l_{min} \leq l_{max}) \wedge (l_{min} \leq t) \wedge (l_{max} \geq 0) \wedge$ *aktivní*
    $p_r \leftarrow (r_{min} \leq r_{max}) \wedge (r_{min} \leq t) \wedge (r_{max} \geq 0) \wedge$ *aktivní*

    if $P_{ID} \leq 4$ **then** $M[P_{ID}] \leftarrow$ **false**
    $M[2p_l + p_r] \leftarrow$ **true**
    if $M[3] \vee M[1] \wedge M[2]$ **then**
      $M[T_{ID}] = p_r \wedge (r_{min} \leq l_{min} \vee \neg p_l) - p_l \wedge (l_{min} \leq r_{min} \vee \neg p_r)$
      paralelní redukce (M)
      $(P_b, P_v) \leftarrow \begin{cases} (P_l, P_p) & , \textbf{if } M[0] \leq 0 \\ (P_p, P_l) & , \textbf{else} \end{cases}$
      if $T_{ID} = 0$ **then** vlož(Z, P_v)
      $U \leftarrow P_b$
    else if $M[1]$ **then**
      $U \leftarrow P_l$
    else if $M[2]$ **then**
      $U \leftarrow P_p$
    else
      if je prazdny(Z) **then break**
      $U \leftarrow$ čti(Z)
    end if
  end if
end loop



(a) Klasická redukce (b) Optimalizovaná redukce

Obrázek 4.4: Porovnání klasické a optimalizované paralelní redukce. Zvýrazněné buňky představují aktivní vlákna.

Primární, stínové a zrcadlové konfigurace Obdobně jako v přechozích případech je nalezen průsečík paprsku s objektem a jsou vyslány stínové paprsky, následně je pro paprsky, jež zasáhly lesklý povrch, vypočítán směr odraženého paprsku. S těmito paprsky je následně zacházeno jako s primárními. Celý postup se nyní zopakuje a procházení končí při zásahu nelesklého objektu paprskem nebo po dosažení maximálního počtu odrazů.

Stínové paprsky mají počátek ve světle a směřují k průsečíku primárního paprsku. Opačná možnost (směr od průsečíku ke světlu) byla vyzkoušena, poskytovala však pro všechny testované scény a akcelerační struktury horší výsledky. Důvodem je pravděpodobně vyšší koherence paprsků vycházejících z jednoho bodu.

V současné verzi algortimu nejsou podporovány lomené paprsky, nelze tudíž zobrazit průhledné předměty. Vhodným rozšířením současného algoritmu by byla úprava definice materiálu, která by definovala průhledný předmět a ve které by pro takové těleso místo odražených paprsků byly použity paprsky lomené. Plně podporovat odraz a lom na jednom materiálu je však složité, protože v tomto případě již není po každém zásahu generován jen jeden paprsek, který pokračuje v procházení, nýbrž následující traverzace odpovídá regurélnímu procházení binárního stromu. Pro další průchod je tedy nutno přidat zásobník obsahující paprsek.

U paketového procházení, použitého u BVH, dochází ke komplikaci. Jelikož jednotlivá vlákna během procházení stromu spolupracují, vyžaduje se od nich účast na výpoču i v případě, když jej sami již nepotřebují. Vlákna, jejichž paprsky nezasáhly žádný objekt či objekt s neodrazivým materiálem, musí tedy spustit traverzaci.

V místech, kde se algoritmy pro ostatní struktury rozhodují o ukončení výpočtu, se nejprve určuje, jestli alespoň jedno vlákno bloku potřebuje pokračovat ve výpočtu. V kladném případě pokračují v traverzaci všechny vlákna, přičemž je traverzace následně spuštěna s příznakem určujícím, zda je vlákno aktivní. Tím se zamezí ovlivňovnání rozhodování o dalším postupu v traverzaci aktivních vláken.

4.3.3 Stínování

Pro stínování je použit klasický Phongův osvětlovací model s výpočty ambientních, difúzních a spekulárních složek.

Za účelem snížení počtu potřebných registrů se z funkce procházení scény vrací pouze zasažený trojuhelník, opětovně se vypočítá hodnota parametru průsečíku paprsku i souřadnice průsečíku na trojúhelníku. Tímto je snížen počet potřebných registrů. Pro stínování je potřeba získat index

materiálu a normálu povrchu, k čemuž je potřeba (stějne jako pro výpočet průsečíku) přečíst indexy vrcholů. Cena přepočítání se tedy snižuje tím, že načtení stejné části dat se současně využije pro více účelů.

Stínovým paprskům však postačuje pouze hodnota parametru paprsku, kterou je možno vypočítat samozřejmě stejným postupem jako v předchozím případě. Jelikož nepotřebujeme ostatní takto získané hodnoty, je to postup zbytečně zdlouhavý. Nabízí se řešení, při kterém se funkce traverzace vytvoří ve formě šablony. Na základě hodnoty parametru šablony vrací funkce traverzace buď zasažený trojúhelník nebo hodnotu parametru. Tím je urychleno vyhodnocování stínových paprsků a zároveň je snižován počet nutných registrů.

4.4 GUI

Po splnění zadání, tedy porovnání jednotlivých akceleračních struktur na různých scénách, plně vyhovuje provádění testů bez uživatelského rozhraní. Aby však bylo možno rychle ověřit správnost funkce sledování paprsku a získat přehled o jeho výkonu na použité grafické kartě bylo implementováno jednoduché uživatelské rozhraní umožňující nahrát, zobrazit a interaktivně procházet testovacími scénami. Za běhu je možno měnit rozlišení obrazu a použitou akcelerační strukturu.

Pro implementaci uživatelského rozhraní byla použita knihovna wxWidgets. Důvodem pro tento výběr byla podpora více operačních systémů a jednoduchá možnost použití OpenGL jako součásti uživatelského rozhraní. Pro rychlé zobrazení výsledků je úspěšně využita možnost spolupráce prostředí CUDA a OpenGL. V prostředí CUDA je možno jednoduše umožnit přímý zápis do PBO objektu rozhraní OpenGL a umožnit tedy zobrazení výsledku bez nutnosti jeho kopírování do hlavní paměti počítače a zpět na grafickou kartu.

Funkce a ovládání uživatelského rozhraní je podrobně popsána v uživatelské příručce v příloze D.

5 Testování

5.1 Popis testovacích scén

Za příklad uniformní geometrie jsou vybrány modely z depozitáře Stanfordské univerzity, které byly vytvořeny skenováním skutečných objektů a následně byly převedeny do trojúhelníkové reprezentace. Každý ze zmíněných modelů reprezentuje jeden objekt, jsou tedy velmi uniformní. Pro testování jsou použity následující modely:

- Bunny
- Dragon
- Budha

Dalším zdrojem modelů jsou scény z benchmarku BART [1], vytvořené pro účely testování výkonu sledování paprsku na animovaných scénách. Pro každou scénu je vybrán jeden ze snímků, scény jsou tedy statické. V plné míře je však přebrán pohyb kamery. Scény jsou komplexní, obsahují více světel a velké množství lesklých ploch a obsahují rovněž velmi podrobnou část modelu. Měřeny jsou následující scény:

- Robots
- Museum
- Kitchen

Poslední trojice scén je rovněž často používaná pro měření výkonu sledování paprsku. Původně byly tyto scény vytvořeny pro demonstraci možností formátu souboru MGF použitelného pro výpočet globálního osvětlení. Jedná se o následující scény:

- Theatre
- Office
- Conference room

Zobrazení jednotlivých scén jsou umístěna do přílohy C.

5.2 Metodika testování

V testech je řada měření prováděna s různou konfigurací sekundárních paprsků. Pro přehlednost a jednoduchost jsou tyto konfigurace uváděny jako zkratky z anglických názvů typů paprsků.

P Scéna je vykreslována pouze pomocí primárních paprsků.
PS Scéna je vykreslována pomocí primárních a stínových paprsků.
PSR Scéna je vykreslována pomocí všech typů paprsků, primárních, stínových a odražených.

	počet trojúhelníků	počet vrcholů	počet světel
Bunny	69451	35947	1
Dragon	871414	437645	1
Budha	1087716	543652	1
Robots	71708	136640	1
Museum	14380	17889	2
Kitchen	110559	68889	4
Theatre	53832	62925	2
Office	35916	22844	3
Conference	298866	229642	2

Tabulka 5.1: Parametry scén.

	výpočetní schopnost	počet procesorů	frekvence procesoru	velikost paměti	propustnost paměti
Nvidia 8600GT	1.1	32	540Mhz	256MB	22,4GB/s
Nvidia GTX280	1.3	240	600Mhz	1GB	141,7GB/s

Tabulka 5.2: Porovnání vlastností testovaných karet.

Ve scénách obsahující uniformní geometrii a představujících jeden objekt jsou měření prováděna se dvěma konfiguracemi paprsků - P a PS. Kolem objektu rotuje kamera nebo světlo. U primárních paprsků rotuje kamera okolo tělesa a míří vždy na střed tělesa. V testech se stínovými paprsky je kamera fixní, kolem tělesa se naopak pohybuje světlem, a to obdobným způsobem, jako s kamerou u paprsků primárních. Celkem je vykresleno 400 snímků pro každé měření.

V ostatních scénách prochází kamera scénou. U scén z benchmarku BART je tento průchod již jejich součástí, u ostatních bylo potřeba tento průchod vytvořit. Tyto testy jsou prováděny se všemi konfiguracemi paprsku - P,PS a PSR. U obou typů měření se zaznamenávají minimální, maximální a průměrná doba trvání vykreslování jednotlivých snímků.

Měření jsou pro srovnání prováděna na dvou kartách, Nvidia 8600GT a Nvidia GTX280. Porovnání vlastností uvedených karet je uvedeno v tabulce 5.2. Tyto karty se navzájem značně liší jak výkonem tak architekturou. Navýšením počtu procesorů a frekvence jejich taktu je dosaženo více něž osminásobného teoretického zvýšení výkonu. Toto zvýšení je podpořeno sedminásobným zvětšením propustnosti paměti. Výkonnější karta obsahuje navíc procesor s novou revizí architektury doplněné o nové funkce. Porovnání výsledků měření testovaných karet je zajímavé jak z hlediska ověření skutečného nárůstu výkonu oproti teoretickému předpokladu a zjištění vlivu změn v architektuře na výkon jednotlivých algoritmů, tak i z hlediska očekávaných výkonnostních změn budoucích generací karet.

5.3 Testování parametrů

5.3.1 Parametry akceleračních struktur

5.3.1.1 Rozlišení mřížky

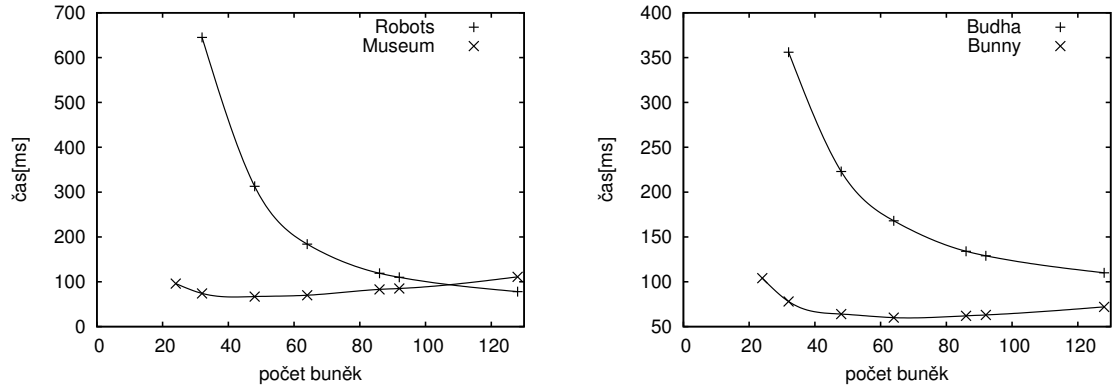
Cílem měření je prokázat, že rychlost traverzace závisí na rozlišení mřížky. Test je prováděn na dvou scénách s uniformní geometrií a na dvou scénách z BART benchmarku. Výsledky testování jsou znázorněny na obrázku 5.1.

Z naměřených hodnot je patrné, že předpis pro rozlišení mřížky úspěšně funguje pro uniformní geometrii. Dobrých výsledků je dosaženo i pro scénu Museum, která je rozměrově malá a má jistou uniformitu. Pro skutečně neuniformní geometrii není takové jednoduché pravidlo účinné. Pro scény obsahující problém „konvička na stadiónu“, jako např. Robots, je lépe zvolit podstatně větší rozlišení, než jaké poskytuje heuristika.

Pro skutečné scény by tedy bylo vhodné volit nejlepší rozlišení ručně či použít výkonnější heuristiku s rozložením geometrie ve scéně, což by však zásadně ovlivnilo rychlost stavby a k nevýhodám uniformní mřížky by se přidala ještě potřeba ručně optimalizovat rozlišení.

5.3.1.2 Rychlost stavby

Rychlost stavby je důležitou vlastností zejména pro dynamicky se měnící scény. Testování těchto scén není předmětem této diplomové práce, přesto je tento údaj zajímavý. Měření jsou prováděna na procesoru AMD Athlon64 X2 2Ghz s jedním využitým jádrem a je zjišťována doba stavby pro všechny testované scény a akcelerační struktury. Výsledky testování jsou shrnuty



(a) Neuniformní geometrie (b) Uniformní geometrie

Obrázek 5.1: Závislost rychlosti procházení na rozlišení mřížky. Počet buněk je udáván pro nejkratší stranu mřížky.

	počet trojúhelníků	čas[ms]		
		mřížka	kd-strom	BVH
Museum	14380	12	219	31
Office	36310	21	593	63
Theatre	53832	83	1031	94
Bunny	69451	39	1249	125
Robots	71708	44	1203	157
Kitchen	110559	68	1390	203
Conference	298866	307	5468	578
Dragon	871414	362	9015	1797
Budha	1087716	393	11827	2406

Tabulka 5.3: Doba stavby akceleračních struktur.

do tabulky 5.3 a odpovídají předpokladům. Nejrychleji probíhá stavba uniformní mřížky se složitostí $O(n)$ a s malým počtem operací na každý trojúhelník.

Nejpomalejší stavbu vykazuje kd-strom. Prostor lze dělit teoreticky donekonečna, doba stavby zásadně závisí na parametrech stavby. Rychlost je negativně ovlivňována použitím algoritmu, jehož asymptotická složitost není nejlepší známá a navíc neobsahuje modifikaci „perfect splits“ (v této verzi algoritmu je příliš náročná). Rychlost je snižována také vyšším počtem trojúhelníků, pro které se přepíná z aproximace na exaktní výpočet ve snaze zvýšit kvalitu stromu. V současnosti lze již dosáhnout rychlejší stavby stromu, která navíc produkuje kvalitnější strom.

Ukazuje se, že stavba BVH stromu je pomalejší než stavba mřížky, avšak výrazně rychlejší než u kd-stromu. Maximální počet kroků je předem daný počtem trojúhelníků, díky tomu stavba závisí bez významnějších odchylek prakticky pouze na počtu trojúhelníků. Mřížka i kd-strom mají pro některé scény výrazné výkyvy. Například ve scéně „Conference“ je doba stavby mřížky i kd-stromu výrazně delší než se předpokládá, z čehož lze usuzovat na přítomnost trojúhelníků velkých rozměrů. Stavba mřížky požaduje vyšší počet operací na trojúhelník, v případě kd-stromu jsou trojúhelníky při dělení pravděpodobně umístěny do velkého počtu buněk.

5.3.1.3 Paměťové nároky

Výsledné velikosti jednotlivých akceleračních struktur a scén (včetně kompletní geometrie) jsou uvedeny v tabulkách 5.4, 5.5 a 5.6. Uvedená velikost použité paměti již zahrnuje i požadavky na

	velikost mřížky	max. velikost buňky	reference	velikost[MB]
Bunny	83x82x64	24	3.69	8.33
Dragon	273x193x122	121	3.06	103.10
Budha	128x312x129	174	2.79	102.13
Robots	128x209x268	1363	19.23	79.91
Museum	71x43x106	111	9.50	5.10
Kitchen	254x128x256	343	14.06	89.32
Theatre	172x135x60	568	22.95	31.40
Office	93x55x93	2126	6.49	7.88
Conference	387x246x93	1220	10.25	121.32

Tabulka 5.4: Vlastnosti uniformní mřížky pro testovací scény.

	poč. listů	poč. prázdných listů	max. velikost listu	reference	velikost[MB]
Bunny	153698	29837	11	5.45	9.22
Dragon	977816	225337	23	2.27	58.50
Budha	1264821	296340	33	2.45	76.48
Robots	82110	11880	85	6.61	12.66
Museum	25828	5074	32	4.88	2.01
Kitchen	164409	28903	67	3.93	11.24
Theatre	123955	18144	91	8.63	10.90
Office	55133	8029	105	6.42	5.06
Conference	338092	42339	128	8.65	51.63

Tabulka 5.5: Vlastnosti kd-stromu pro testovací scény.

	poč. uzlů	velikost[MB]
Bunny	45814	2.84
Dragon	589110	35.83
Budha	737222	44.71
Robots	50170	6.08
Museum	9196	0.90
Kitchen	72790	4.88
Theatre	35474	3.27
Office	22218	1.54
Conference	195794	14.48

Tabulka 5.6: Vlastnosti BVH stromu pro testovací scény.

reprezentaci geomterie scény. Z výsledků vyplývá, že největší nároky na paměť má uniformní mřížka a následně kd-strom, nejmenší paměťové nároky má BVH strom. Tabulky rovněž obsahují další údaje, které vypovídají o vlastnostech akceleračních struktur.

Počty trojúhelníků v buňkách mřížky dokládají její nevýhodu pro neuniformní geometrii. Příkladem nechť je scéna Office, ve které i při velmi malém počtu trojúhelníků a při relativně vysokém rozlišení mřížky, existují buňky s velmi vysokým počtem trojúhelníků.

Nepříjemným zjištěním je přítomnost listů kd-stromu s velmi vysokým počtem trojúhelníků. Důkladnější analýza za použití histogramu poukazuje na nezanedbatelné množství velkých listů, vyskytujících se zejména ve scénách s dlouhými úzkými trojúhelníky. Při stavbě kd-stromu se používá pouze obálka, ne však ořezávání. Výsledkem je přiřazení trojúhelníku do uzlu, do něhož

	počet registrů	obsazenost [%]	Robots		Kitchen		Museum	
			čas[ms]	zrychl. [%]	čas[ms]	zrychl. [%]	čas[ms]	zrychl. [%]
Mřížka	59	16	261,35		1393,56		590,42	
	32	33	240,96	8	1158,29	17	471,59	21
Kd-strom	56	16	116,97		623,25		504,98	
	32	33	90,70	23	489,54	22	381,61	25
BVH	53	16	221,75		1455,50		633,31	
	32	33	163,66	27	1171,57	20	482,43	24

Tabulka 5.7: Vliv obsazenosti procesoru na rychlost výpočtu při použití karty 8600GT.

	počet registrů	obsazenost [%]	Robots		Kitchen		Museum	
			čas[ms]	zrychl. [%]	čas[ms]	zrychl. [%]	čas[ms]	zrychl. [%]
Mřížka	59	25	430.67		907.71		266.19	
	32	50	350.24	19	704.78	23	220.45	18
Kd-strom	56	25	110.68		429.54		240.52	
	32	25	128.97	-18	496.17	-15	278.18	-15
BVH	53	25	151.01		1064.75		274.72	
	32	50	129.90	25	762.39	29	214.37	22

Tabulka 5.8: Vliv obsazenosti procesoru na rychlost výpočtu při použití karty GTX280.

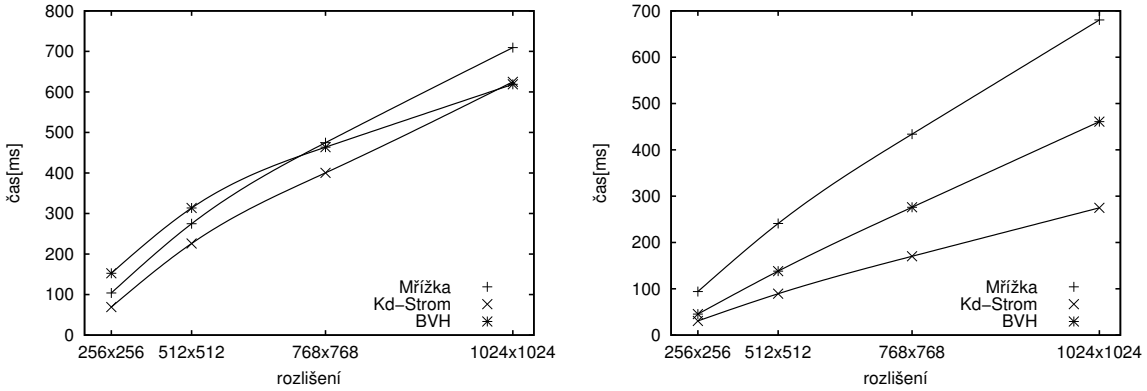
zasahuje jeho obálka, on samotný však nezasahuje. Další dělení takových uzlů je obtížné.

5.3.2 Obsazenost grafického procesoru

Kernel sledování paprsku v konfiguraci se sekundárními paprsky používá v porovnání s verzí pro primární paprsky značné množství registrů. Velký počet použitých registrů snižuje obsazenost procesoru a může zpomalovat výpočet. Samotný algoritmus procházení je přitom stejný, liší se pouze způsob jeho použití. Tato skutečnost vede k domněnce, že většina využitých registrů není v tomto případě při samotném procházení použita. Bylo by zajímavé zjistit, zda nelze výpočet zrychlit pomocí parametru překladače, který může počet použitých registrů snížit. Některé proměnné jsou poté umístěny do pomalé lokální paměti, ale vzhledem k vyslovené domněnce by četnost jejího používání měla být malá a celkově by změna měla mít pozitivní vliv na výkon. Měření má za cíl ověřit premisu, zda tomu tak skutečně je.

Pro zjištění takto postaveného cíle jsou použité scény z BART benchmarku. Použití těchto tří scén je dostačující, protože ze své podstaty by vliv omezení počtu registrů u jednotlivých scén měl být prakticky stejný. Měření probíhalo pro konfiguraci PSR. Počet registrů byl omezen na hodnotu rovnající se počtu registrů použitých ve verzi pro primární paprsky. Výsledky pro kartu Nvidia 8600 při použití rozlišení 512 x 512 pixelů jsou uvedeny v tabulce 5.7, pro kartu Nvidia GTX280 při použití rozlišení 1024 x 1024 pixelů pak v tabulce 5.8. Rozdílné rozlišení nikterak neovlivní výsledky testování. Měření si neklade za cíl přímé porovnání výkonu obou karet.

Ukázalo se, že při použití karty starší generace má omezení počtu registrů jednoznačně pozitivní vliv na samotnou rychlost výpočtu. U novější generace tomu tak není. Karty s výpočetní schopností 1.3, k nimž GTX280 patří, mají oproti předcházející generaci dvojnásobný počet



(a) Dragon (b) Robots

Obrázek 5.2: Závislost doby výpočtu na rozlišení obrazu při použití karty Nvidia 8600GT.

použitelných registrů, zatímco množství sdílené paměti zůstalo stejné. V důsledku těchto změn není obsazenost procesoru při použití algoritmu procházení kd-stromu již omezena počtem registrů, ale velikostí sdílené paměti. Omezení počtu registrů přineslo nevýhodu spočívající v nutnosti přistupovat do lokální paměti a výsledek na rychlost je negativní. Zároveň tento výsledek poukazuje na skutečnost, že procházení ostatních struktur je možno dále zrychlit omezením počtu použitých registrů na úrovni algoritmu. Tento způsob, na rozdíl od použití parametru překladače, nepřináší žádná negativa spojená s použitím lokální paměti.

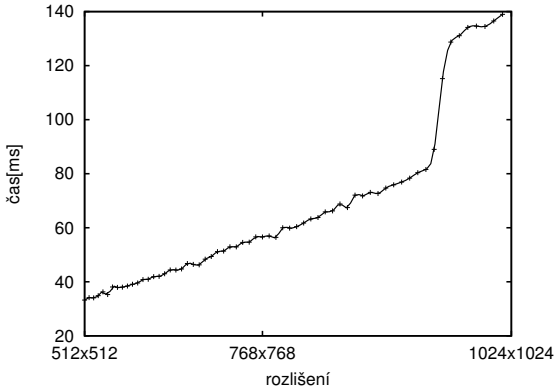
Výsledky měření prokázaly důležitost optimalizovat obsazenost procesoru pro aplikace náročné na přístup do paměti, jakou je metoda sledování paprsku. Zrychlení se pohybovalo okolo 20 %. Příklad procházení kd-stromu zároveň potvrdil nutnost zjištění vlivu omezení počtu registrů na výkon, zvlášť pro každý jeden algoritmus.

5.3.3 Rozlišení obrazu

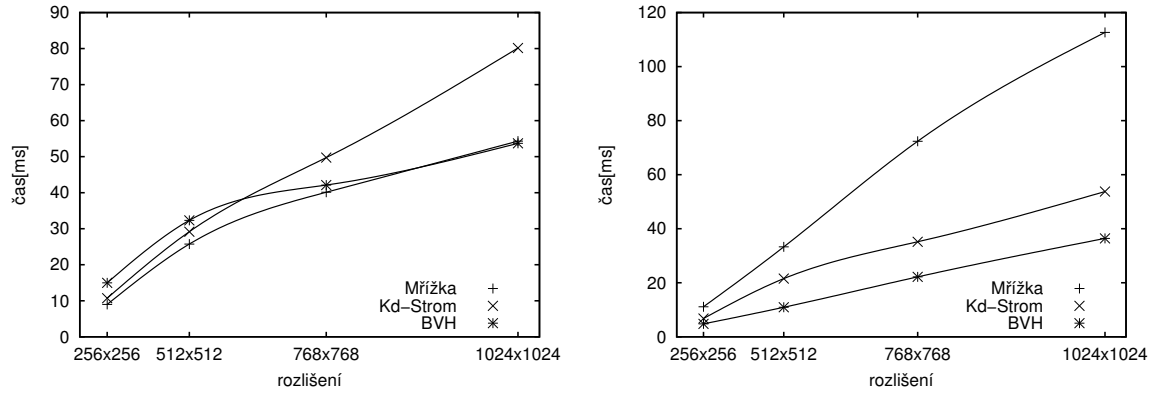
Cílem následujícího testování je určení závislosti rychlosti procházení na rozlišení obrazu. Základním předpokladem je lineární růst doby výpočtu s rostoucím počtem pixelů. Nabízí se alternativní řešení, při kterém závislost doby výpočtu na počtu pixelů bude lepší než lineární. Zvyšováním rozlišení se prostorový úhel svíraný 64 paprsky bloku zmenšuje, pravděpodobnost společného průchodu se naopak zvyšuje a následně se snižuje počet větvení kódu jakož i přístupů do paměti. Kompletní zdroje údajů pro grafy v této části jsou uvedeny v příloze B.

Výsledky měření na kartě Nvidia 8600GT, zobrazené na obrázku 5.2, plně odpovídají předpokladům a ve všech provedených měřeních nerostla doba výpočtu se zvyšujícím se rozlišením lineárně. Doba výpočtu přepočtená na jeden paprsek klesá s rostoucím rozlišením. Zvětšená koherence paprsků skutečně způsobuje rychlejší průchod scénou. Nejvíce se změna podle očekávání projevuje u paketového procházení BVH. V extrémním případě, tj. v případě měření scény Dragon, se při zvětšování rozlišení mění BVH procházení z nejpomalejšího na nejrychlejší.

Při použití karty Nvidia GTX280 se v nižších rozlišeních opakovala tendence již pozorovaná u předchozího měření, ale při použití rozlišení 1024 x 1024 došlo k výraznému zhoršení doby výpočtu. Ve snaze zjistit příčinu této výkonnostní regrese byla provedena řada měření. Testování velkého množství postupně se zvyšujících rozlišení, znázorněno na obrázku 5.3, ukazuje skokové prodloužení doby výpočtu. Zajímavým zjištěním je skutečnost, že tento skok je pozorován ve větší či menší míře u všech akceleračních struktur a scén. Byla provedena řada testů zaměřených na zjištění důvodu tohoto zpomalení, která bohužel nepřinesla žádný výsledek. Poslední z těchto testů je zaměřen na zjištění zcela identických podmínek pro všechna rozlišení. Pro výpočet celé



Obrázek 5.3: Výkonnostní regrese výpočtu na kartě Nvidia GTX280.



(a) Dragon (b) Robots

Obrázek 5.4: Závislost doby výpočtu na rozlišení obrazu při použití karty Nvidia 280GTX.

scény je použito pouze čtyř rozdílných paprsků. Tyto jsou přiřazeny čtveřici sousedních bloků, přičemž všechna vlákna jednoho bloku používají jeden z těchto paprsků. Tento vzorec je poté opakován po celé ploše obrazu, čímž zaručuje identické podmínky pro všechny bloky obrazu a eliminuje vedlejší vlivy změny rozlišení na dobu výpočtu. Výsledky měření nejenže obsahují skokovou změnu výkonu, ale tato změna je nyní opravdu skoková s lineární změnou rychlosti před a po tomto skoku. Z této skutečnosti je možno učinit závěr, že prodloužení doby výpočtu je způsobeno chybou na straně platformy CUDA. Zhoršení výkonu je pravděpodobně způsobeno buď chybou ovladače karty, který špatně přiřazuje výpočet kartě, nebo chybou v implementaci plánovače v procesoru grafické karty.

Posledním pokusem o vyřešení zjištěného problému je rozdělení vykreslování obrazu na vykreslování několika menších. Obraz je rozdělen na několik menších obdélníkových oblastí, které jsou vykresleny postupně. Z výsledků měření na obrázku 5.4 vyplývá, že po provedení popsaného postupu se doba výpočtu mění se změnou rozlišení již předpokládaným způsobem, i když lze pozorovat mírné kolísání růstu doby výpočtu.

5.3.4 Výkon grafické karty

Jak bylo popsáno v popisu architektury, je možno vyrábět grafické karty s různým výkonem v závislosti na počtu multiprocesorů, na frekvenci procesoru i na frekvenci paměti. Cílem dalšího měření je ověření lineárního růstu rychlosti sledování paprsku s rostoucím počtem multiprocesorů. Lze také očekávat větší nárůst rychlosti v závislosti na rozdílech ve frekvenci procesoru,

či naopak negativní ovlivnění výkonu propustností paměti. Výsledky jsou částečně zkresleny rozdílnou architekturou čipů použitých v jednotlivých kartách. Kompletní výsledky měření jsou uvedeny v příloze B.

Změny architektury jsou pro uniformní mřížku velmi prospěšné. Pro některé scény je růst výkonu dokonce vyšší než teoretický. Lze předpokládat, že je tento nárůst zapříčiněn podstatným zmenšením požadavků na sdružování přístupu do paměti, lepší funkcí cache paměti textur, zvýšením počtu registrů (zvětšujícím obsazenost procesoru) a lepším skrýváním latence přístupu do paměti. Tato domněnka je podpořena chováním algoritmu na různých konfiguracích sekundárních paprsků. K nejvyššímu zrychlení dochází při použití primárních paprsků, k menšímu zrychlení při použití stínových paprsků. Při zapojení odražených paprsků rychlost opět klesá. S klesající koherencí paprsků a tedy i klesající koherencí přístupu do paměti je obtížnější maskovat její latenci. V závislosti na použité scéně,rozlišení a konfiguraci paprsků bylo zjištěno 7 až 12násobné zrychlení oproti starší kartě.

Algoritmus BVH využívající paketové procházení není zásadně ovlivněn změnami architektury. Všechny přístupy algoritmu do paměti jsou ideálně sdruženy nebo používají cache textur. Zrychlení při použití rychlejší karty odpovídá zvýšení propustnosti paměti, čímž je dokázáno že algoritmus je omezen přístupem do paměti. Vysoká obsazenost procesoru úspěšně maskuje latence přístupů. Navýšení výkonu při použití BVH se pohybovalo v rozmezí 6 až 8násobného zrychlení.

Algoritmu procházení kd-stromu prospěly změny architektuře nejméně. Důvodem je absence změny velikosti sdílené paměti, kterou algoritmus ve velké míře používá. Důsledkem je, že při použití algoritmu procházení kd-stromu je obsazenost procesoru poloviční oproti ostatním algoritmům a výsledky rychlosti jsou tímto značně ovlivněny. Pro primární paprsky je relativní propad výkonu vůči ostatním strukturám značný. Při použití sekundárních paprsků není rozdíl natolik výrazný, protože dosažení vysoké obsazenosti je u ostatních struktur vyváženo používáním lokální paměti. Rozdíl v rychlosti kd-stromu byl ze všech struktur nejmenší a představoval pouze 3 až 6násobné zrychlení.

5.4 Detailní výsledky

Dosavadní testování mělo za cíl odhalit chování traverzace při změně některého z volitelných parametrů. V této kapitole dochází k hlavnímu porovnání celkového výkonu jednotlivých akceleračních struktur na různých scénách při různých konfiguracích sekundárních paprsků. Veškerá měření jsou prováděna v rozlišení 1024 x 1024 pixelů na grafické kartě Nvidia GTX280. Kompletní výsledky měření pro různá rozlišení a grafické karty jsou uvedeny v příloze B.

Počty traverzačních kroků a testovaných průsečíků pro různé konfigurace sekundárních paprsků jsou uvedeny v tabulkách 5.9,5.10 a 5.11. Pro zřetelnější vliv koherence paprsků každé měření obsahuje počet kroků pro nově přidaný typ paprsků. Změny v počtu kroků při použití různých konfigurací jsou většinou dány tím, že paprsky prochází jinými částmi scény. U BVH je jasně zřetelný nárůst počtu kroků při použití odražených paprsků s nízkou koherencí. Zlepšení kd stromu pro stínové paprsky je dáno možností použít první nalezený průsečík mezi světlem a objektem, včetně průsečíků umístěných mimo obálku buňky.

5.4.1 Primární paprsky

Při použití primárních paprsků je jasným vítězem algoritmus packetové procházení BVH. Kd-strom a mřížka dosahují podobných výsledků. Špatné výsledky kd jsou způsobeny změnami architektury procesoru, které podstatně více vyhovují ostatním metodám. Na kartě Nvidia 8600GT byly výsledky kd stromu ve srovnání s ostatními algoritmy podstatně lepší a blížili se spíše BVH než mřížce.

	poč. kroků			poč. testů primitiv		
	P	PS	PSR	P	PS	PSR
Bunny	27.64	47.39		10.34	37.22	
Dragon	79.84	117.31		11.42	45.54	
Budha	70.75	100.79		18.07	43.07	
Robots	36.96	40.43	18.58	154.61	66.45	594.76
Museum	66.36	60.19	35.04	28.05	33.56	46.43
Kitchen	235.74	154.76	76.30	20.46	12.71	24.20
Theatre	90.62	56.60	41.80	30.21	37.90	112.01
Office	92.01	73.87	8.87	40.80	71.56	32.88
Conference	264.59	165.70	85.59	29.62	31.53	108.37

Tabulka 5.9: Počet provedených kroků při procházení mřížky přepočtený na jeden paprsek.

	poč. kroků			poč. testů primitiv			poč. restartů		
	P	PS	PSR	P	PS	PSR	P	PS	PSR
Bunny	24.96	53.97		3.63	12.31		0.62	0.05	
Dragon	31.25	68.81		3.05	10.38		0.62	0.16	
Budha	36.57	64.31		4.03	8.83		0.52	0.14	
Robots	42.39	30.49	60.65	9.95	8.75	26.00	0.10	0.21	0.42
Museum	55.26	19.20	44.60	14.22	6.65	14.76	0.10	0.13	0.29
Kitchen	41.54	6.26	34.76	10.24	1.04	9.93	0.12	0.13	0.19
Theatre	45.28	17.39	56.72	10.82	5.41	14.61	0.01	0.01	0.32
Office	29.50	11.20	27.74	15.51	4.36	10.41	0.03	0.09	0.04
Conference	42.63	14.30	58.06	15.37	4.78	50.15	0.09	0.11	0.36

Tabulka 5.10: Počet provedených kroků při procházení kd stromu přepočtený na jeden paprsek.

	poč. kroků			poč. testů primitiv		
	P	PS	PSR	P	PS	PSR
Bunny	21.77	52.14		3.50	8.88	
Dragon	56.28	114.13		14.88	27.99	
Budha	92.84	130.93		26.86	30.89	
Robots	30.63	30.69	62.70	2.72	3.51	11.48
Museum	43.57	40.06	131.01	4.57	3.90	22.99
Kitchen	44.85	40.40	152.13	4.30	5.19	25.89
Theatre	34.36	33.13	102.70	3.40	3.66	17.72
Office	24.86	30.47	29.01	4.80	4.85	4.60
Conference	35.49	34.62	113.20	5.46	5.86	25.23

Tabulka 5.11: Počet provedených kroků při procházení BVH přepočtený na jeden paprsek.

	čas[ms]								
	Mřížka			Kd-strom			BVH		
	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Bunny	12.17	20.50	16.03	29.91	58.02	41.58	11.42	17.24	13.76
Dragon	26.1	54.31	40.19	32.82	80.17	55.87	23.42	53.70	39.04
Budha	16.3	48.42	34.59	21.82	66.72	45.59	16.52	53.84	36.82
Robots	4.33	112.65	27.32	6.26	53.76	20.52	12.78	36.41	25.89
Museum	4.61	45.94	24.99	4.03	67.04	46.19	6.43	28.54	20.03
Kitchen	18.39	68.36	41.64	7.38	66.85	40.46	16.55	49.44	29.34
Theatre	6.97	104.57	43.07	7.49	97.94	42.34	16.63	71.86	34.30
Office	46.96	62.42	52.92	37.67	52.54	44.12	19.98	26.2	22.68
Conference	58.48	97.99	83.19	40.00	109.06	83.73	20.68	34.84	28.90

Tabulka 5.12: Rychlost procházení pro primární paprsky.

	čas[ms]								
	Mřížka			Kd-strom			BVH		
	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Bunny	22.84	31.83	27.40	51.16	78.00	61.74	23.22	30.28	26.98
Dragon	54.97	92.28	73.32	68.00	117.41	85.98	66.39	93.39	79.97
Budha	44.18	89.16	69.12	48.90	92.43	73.49	56.59	96.82	81.77
Robots	9.41	200.40	53.78	13.18	79.04	35.60	22.92	70.16	49.96
Museum	35.66	92.69	68.27	24.05	116.00	86.19	23.65	64.94	53.15
Kitchen	133.15	251.43	209.64	37.56	225.84	130.04	81.65	191.31	138.78
Theatre	48.13	208.10	119.70	26.35	177.42	87.29	50.93	159.73	93.61
Office	202.49	249.43	218.62	108.11	130.41	116.12	82.61	96.30	87.94
Conference	161.21	275.60	228.15	77.14	180.93	152.96	58.01	99.20	85.20

Tabulka 5.13: Rychlost procházení při použití stínových paprsků.

	čas[ms]								
	Mřížka			Kd-strom			BVH		
	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Robots	10.82	381.21	88.97	13.35	111.43	43.76	23.84	128.9	64.31
Museum	35.91	219.5	168.54	24.15	247.63	184.17	25.76	219.34	163.70
Kitchen	135.7	727.99	442.8	37.84	438.18	244.38	86.77	785.87	403.92
Theatre	48.81	856.55	379.72	26.43	547.61	201.56	54.48	611.28	292.07
Office	203.02	254.39	223.97	111.73	134.87	120.13	89.54	103.41	94.20
Conference	165.04	376.68	292.78	—	—	—	61.22	149.95	114.15

Tabulka 5.14: Rychlost procházení při použití odražených paprsků.

5.4.2 Stíny

Naměřené časy pro konfiguraci primární + stínové paprsky jsou uvedeny v tabulce 5.13. Poměr výkonu jednotlivých algoritmů je obdobný předchozímu měření. Stínové paprsky nemají zpravidla zásadně horší koherenci než primární, přesto je možno pozorovat mírné zlepšení kd-stromu oproti ostatním algoritmům.

5.4.3 Odrazy

Výsledky měření pro konfiguraci primární + stínové + odražené paprsky jsou uvedeny v tabulce 5.14. Maximální počet odrazů byl nastaven na dva. Po přidání výpočtu odrazů se naplno projevilo paketové procházení BVH. Pokud scéna obsahuje více než velmi malé procento zrcadlových objektů, jeho výkon vůči ostatním algoritmům klesá. Při této konfiguraci, pro všechny měřené scény splňující popsanou podmínku, nastává stejné pořadí výkonnosti. Rychlost akceleračních struktur je následující: kd-strom > BVH > Mřížka. Ztráta výkonu BVH je jasně dána reakcí paketového procházení na snížení koherence paprsků. Důvod poklesu výkonu mřížky již tak zřejmý není. Pokles pravděpodobně způsobuje průchod různými buňkami již prakticky od začátku traverzace sekundárních paprsků. Traverzace kd-stromu začíná vždy od uzlu a rozdíly v traveraci paprsků začínají až na nižších úrovních.

5.4.4 Shrnutí

Algoritmus paketového procházení BVH je velmi ovlivněn absencí koherence u sekundárních paprsků. Paket o 64 paprscích je v tomto případě příliš velký. Zmenšování paketu je vzhledem k omezením architektury rovněž problematické, při použití primárních a stínových paprsků je však pro běžné scény jednoznačně nejrychlejší.

Největší odolnost vůči zhoršování koherence sekundárních paprsků prokázal jednoduchý algoritmus procházení kd-stromu (bez paketového procházení). Rychlost procházení kd a BVH stromů je značně závislá na kvalitě stavby stromu. Měření parametrů stavby prokázalo suboptimální výsledky stavby kd-stromu a proto není tedy vyloučeno, že zlepšením kvality stavby stromu by se algoritmus procházení kd-stromu mohl vyrovnat BVH i pro primární paprsky.

Uniformní mřížka byla ve všech testech, používajících běžné scény s neuniformní geometrií až na jeden případ(scéna s nejmenším počtem trojúhelníků), vždy nejpomalejší. Jedinou výhodou mřížky tak zůstává rychlost stavby.

Ukazuje se, že vlastnosti procesoru současné generace grafických karet nejsou pro sledování paprsku ideální. V případě samostatného procházení paprsků se vyskytuje problém s nutností uchovávat zásobník s velkým počtem informací pro každý paprsek. Množství rychlé sdílené paměti je pro uložení zásobníku nedostatečné a použití hlavní paměti příliš pomalé. Tato situace se pravděpodobně v nejbližší době nezmění. Nejnovější úprava architektury procesoru plně odpovídá jejich primárnímu zaměření na klasickou rasterizaci. Stále složitější shadery vyžadují pro svůj plný výkon velké množství registrů, zatímco sdílená paměť není při běhu shaderu používána.

Je zřejmé že při použití paketového procházení nastává problém vyvolaný omezením výkonu při větvení programu. Velikost warpu vyvolává potřebu používat velké pakety se značným počtem paprsků, které následně způsobují pokles výkonu při použití sekundárních paprsků. Aby bylo dosaženo efektivní implementace sledování paprsku na grafické kartě je třeba provést změnu architektury, která vyřeší tyto dva problémy. V budoucnu je tedy potřeba dosáhnout zvětšení velikosti rychlé paměti pro uložení zásobníku a zmenšit warp za účelem snadnějšího vytváření menších paketů.

6 Závěr

V rámci práce byl úspěšně implementován celý postup nutný pro vytvoření obrazu metodou sledování paprsku na grafickém akcelerátoru. Jednotlivé části implementace je možno jednoduše upravovat a rozšiřovat, např. o podporu pro další formát souborů a že implementace poskytuje dobrý základ pro provádění dalších optimalizací a testů. Mezi další možné postupy umožňující zvýšení výkonu lze zařadit např. snížení počtu použitých registrů, pokusy s rozložením datových struktur v paměti a další zlepšování kvality stavby akceleračních struktur. Za dlouhodobé cíle lze považovat podporu textur a vytvoření prostředí pro tvorbu efektů ve stylu shader programů známých z grafických API.

Pro ověření vlastností algoritmů byla provedena řada měření na dostatečném počtu scén. Měření byla zaměřena jak na zjištění ovlivnění algoritmů vlastnostmi použité procesorové architektury, tak na přímé porovnání akceleračních struktur mezi sebou. Toto porovnání nemá „jasného“ vítěze, pouze je možno konstatovat, že z hlediska rychlosti procházení se jako nejpomalejší (prakticky ve všech případech) prokázala uniformní mřížka. Výsledky kd-stromu a BVH jsou rozdílné v závislosti na použitém typu paprsků. U primárních a stínových paprsků je „jasným“ vítězem BVH, které navíc prospěly změny v nejnovější generaci grafického akcelerátoru. Při použití odražených paprsků s nízkou koherencí, se vlivem velikosti paketu BVH, stává vítězem kd-strom.

Ukázalo se, že současná generace grafického hardware není zcela ideální pro implementaci výše popsaných typů akceleračních struktur. Metody sledující paprsky procházející samostatně mají příliš veliké nároky na uložení zásobníku, nedostatečně koherentní přístup do paměti a nekonzistentní větvení kódu. Při použití paketového procházení, nutí penalizace výkonu v případě větvení kódu, používat pakety s příliš velikým počtem paprsků pro získání dobrého výkonu pro pokročilé efekty. Pokud architektura příští generace grafických karet nepřinese řešení těchto problémů, bude nutno nejspíše zaměřit úsilí do algoritmů používajících paralelismus na zrychlení sledování jednoho paprsku [5],[24].

Měření rovněž prokázalo, že současná generace grafického hardware stále není schopna nabídnout dostatečný výkon pro použití metody sledování paprsku v reálném čase. Navíc zatím chybí komplexní řešení celého řetězce s jasně definovaným způsobem použití a tvorbou efektů. Zda-li bude pro účely sledování paprsku v budoucnosti používána architektura podobná současným grafickým kartám, klasickým procesorům, nebo na novém speciálním hardwaru je nyní těžké předpovědět. Na poli klasických procesorů dochází k neustálému zvyšování počtu jader jako pravděpodobně jediné současné možnosti zvyšování výkonu. Při dosažení dostatečného počtu jader může být metoda jednoduše implementována klasickou cestou a to bez problému způsobených specifickými vlastnostmi grafických karet. Použití metody sledování paprsku na grafickém akcelerátoru představuje stále relativně mladý obor. Pro možnost jeho obecného rozšíření je potřeba vyřešit velké množství problémů jak v oblasti změny architektury karet, tak v oblasti vývoje nových efektivních algoritmů.

7 Literatura

[1] Bart: A benchmark for animated ray tracing. `http://www.ce.chalmers.se/research/group/graphics/BART`.

[2] Nvidia cuda compute unified device architecture - programming guide, 2008. Version 2.1 Beta.

[3] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

[4] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. *Technical Report, inTrace Realtime Ray Tracing GmbH, No inTrace-2006-001 (submitted for publication)*, 2006.

[5] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *Comput. Graph. Forum*, 27(4):1225–1233, 2008.

[6] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM.

[7] A. Fujimoto, Takayuki Tanaka, and K. Iwata. Arts: accelerated ray-tracing system. pages 148–159, 1988.

[8] Dominik Göddeke. Languages and programming environments. `http://www.mathematik.uni-dortmund.de/~goeddeke/arcs2008/A2_Languages.pdf`.

[9] Simon Green. Introduction to cuda, cuda performance, cuda toolchain. `http://www.mathematik.uni-dortmund.de/~goeddeke/arcs2008/C1_CUDA.pdf`.

[10] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Real-time ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007.

[11] Mark Harris. Optimizing parallel reduction in cuda. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[12] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[13] Thiago Ize, Ingo Wald, Chelsea Robertson, and Steven G Parker. An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 27–55, 2006.

[14] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, 1986.

[15] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Eurographics 2007*, volume 26 of *Computer Graphics Forum*, page 415–424, Prague, Czech Republic, September 2007. European Association for Computer Graphics, Blackwell.

[16] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[17] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast KD-tree construction fory interactive ray tracing of dynamic scenes. In *"Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007"*, 2007.

[18] Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3:1–14, 1998.

[19] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[20] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001.

[21] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[22] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006.

[23] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, pages 485–493, 2006. (Proceedings of ACM SIGGRAPH 2006).

[24] Ingo Walk, Carsten Benthin, and Solomon Boulos. Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs.

[25] Turner Whitted. An improved illumination model for shaded display. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 4, New York, NY, USA, 2005. ACM.

[26] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–11, New York, NY, USA, 2008. ACM.

A Seznam použitých zkratek

ALU	Arithmetic Logic Unit
API	Aplication Programing Interface
BART	a Benchmark for Animated Ray Tracing
BRDF	Bidirectional Reflectance Distribution Function
BVH	Boundary Volume Hierarchy
CPU	Central Processing unit
CRCW	Concurent Read Concurent Write
CTM	Close To Metal
CUDA	Compute Unified Device Architecture
DDA	Digital Differential Analyzer
DMA	Direct Memmory access
FMAD	Fused Multiply ADd
GCC	GNU Compiler Collection
GDB	GNU Debugger
GLSL	GL Shading Language
GPU	Graphical proccesing unit
GUI	Graphical user interface
HLSL	High Level Shading Language
PBO	Picture Buffer Object
PRAM	Paralel Random Access Machine
PTX	Parallel Thread Execution
RAW	Read After Write
RF	Register file
SAH	Surface Area Heuristic
SFU	Special function unit
SIMD	Single Instruction Multiple Data
SP	Streaming Processor
SSE	Streaming SIMD Extension

B Výsledky měření

		čas[ms]								
		Mřížka			Kd strom			BVH		
scéna	rozlišení	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Bunny	256x256	1.44	3.37	2.33	3.56	7.54	5.65	2.32	4.71	3.54
	512x512	4.17	9.09	6.42	11.40	22.89	16.75	4.61	8.62	6.63
	768x768	8.25	14.74	11.08	18.08	35.22	25.63	7.66	12.84	9.82
	1024x1024	12.17	20.50	16.03	29.91	58.02	41.58	11.42	17.24	13.76
Dragon	256x256	3.66	8.98	5.93	3.76	10.71	6.52	5.74	14.97	10.59
	512x512	9.43	25.74	16.82	10.67	29.17	20.12	11.99	32.33	22.44
	768x768	18.71	40.12	28.60	19.98	49.75	34.15	18.39	42.12	30.91
	1024x1024	26.10	54.31	40.19	32.82	80.17	55.87	23.42	53.70	39.04
Budha	256x256	2.36	9.79	6.26	3.22	8.73	6.03	4.95	16.59	10.50
	512x512	6.48	28.77	17.77	9.73	29.27	20.27	9.58	36.28	23.78
	768x768	12.03	35.74	24.73	12.62	38.25	27.03	12.05	41.42	28.10
	1024x1024	16.30	48.42	34.59	21.82	66.72	45.59	16.52	53.84	36.82
Robots	256x256	0.31	11.11	4.34	0.44	6.77	2.51	0.82	4.78	2.57
	512x512	0.97	33.28	10.10	1.53	21.54	8.24	2.98	10.93	7.15
	768x768	2.63	72.36	18.15	3.75	35.17	12.99	7.45	22.18	15.85
	1024x1024	4.33	112.65	27.32	6.26	53.76	20.52	12.78	36.41	25.89
Musem	256x256	0.31	3.79	2.45	0.28	7.97	5.46	0.41	3.04	2.31
	512x512	1.07	11.59	7.11	0.94	27.14	18.94	1.44	8.14	5.86
	768x768	2.78	27.92	15.59	2.48	42.36	28.96	3.83	17.68	12.45
	1024x1024	4.61	45.94	24.99	4.03	67.04	46.19	6.43	28.54	20.03
Kitchen	256x256	1.36	7.68	4.55	0.54	12.93	5.53	1.06	6.46	3.67
	512x512	4.74	19.71	12.36	1.79	35.21	17.16	3.76	12.54	9.09
	768x768	11.85	43.33	26.29	4.46	45.07	26.29	9.74	28.94	18.72
	1024x1024	18.39	68.36	41.64	7.38	66.85	40.46	16.55	49.44	29.34
Theatre	256x256	0.48	10.26	4.47	0.56	15.60	5.24	1.07	5.84	3.02
	512x512	1.71	33.28	12.76	1.91	45.99	16.58	3.91	19.28	9.42
	768x768	4.14	65.83	27.54	4.48	62.95	27.14	9.62	42.00	21.24
	1024x1024	6.97	104.57	43.07	7.49	97.94	42.34	16.63	71.86	34.30
Office	256x256	9.64	12.45	10.64	6.18	8.94	7.55	2.69	4.29	3.55
	512x512	19.16	26.17	22.35	18.02	25.89	22.30	7.58	10.48	9.11
	768x768	33.57	44.33	37.60	24.09	34.86	28.23	13.82	17.55	15.63
	1024x1024	46.96	62.42	52.92	37.67	52.54	44.12	19.98	26.20	22.68
Conference	256x256	5.13	18.67	12.92	5.32	17.08	10.78	2.48	6.23	4.34
	512x512	15.21	38.31	28.06	16.43	49.14	35.46	6.68	11.89	9.43
	768x768	36.65	69.75	59.47	27.33	70.57	55.83	13.47	22.60	19.48
	1024x1024	58.48	97.99	83.19	40.00	109.06	83.73	20.68	34.84	28.90

Tabulka B.1: Výsledky měření primárních paprsků na kartě Nvidia GTX280

		čas[ms]								
		Mřížka			Kd strom			BVH		
scéna	rozlišení	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Bunny	256x256	3.33	5.49	4.31	5.86	12.02	7.87	5.80	9.09	7.37
	512x512	7.85	12.33	10.19	18.43	30.82	22.75	9.60	13.58	11.66
	768x768	15.50	21.88	18.86	31.48	48.62	38.70	16.90	22.40	19.40
	1024x1024	22.84	31.83	27.40	51.16	78.00	61.74	23.22	30.28	26.98
Dragon	256x256	6.83	14.00	9.28	6.29	12.64	8.62	13.32	24.11	18.02
	512x512	18.25	33.03	25.33	19.87	36.19	25.99	28.82	42.82	35.64
	768x768	40.14	65.43	52.77	41.87	74.70	53.68	51.66	72.56	63.06
	1024x1024	54.97	92.28	73.32	68.00	117.41	85.98	66.39	93.39	79.97
Budha	256x256	8.59	17.93	13.84	7.02	14.13	10.72	16.11	29.35	23.75
	512x512	23.59	48.46	38.25	22.94	44.39	35.50	33.94	63.30	51.73
	768x768	31.20	61.51	49.50	29.05	53.17	43.82	41.43	73.07	61.42
	1024x1024	44.18	89.16	69.12	48.90	92.43	73.49	56.59	96.82	81.77
Robots	256x256	0.65	25.62	9.38	0.87	9.64	4.25	1.50	10.33	5.19
	512x512	2.10	61.98	21.65	3.16	31.95	13.97	5.36	21.89	14.25
	768x768	5.50	129.84	37.02	7.62	52.59	22.29	15.28	43.86	31.00
	1024x1024	9.41	200.40	53.78	13.18	79.04	35.60	22.92	70.16	49.96
Musem	256x256	2.34	8.54	6.61	1.61	12.94	9.95	1.55	8.67	6.42
	512x512	8.62	25.01	19.73	5.94	42.70	32.79	5.59	20.54	16.25
	768x768	20.44	57.60	42.96	13.89	76.00	53.93	13.70	42.12	33.53
	1024x1024	35.66	92.69	68.27	24.05	116.00	86.19	23.65	64.94	53.15
Kitchen	256x256	11.62	29.01	20.68	3.22	27.37	16.09	5.89	25.25	17.20
	512x512	34.87	75.10	59.91	11.29	81.69	51.43	20.75	61.82	43.77
	768x768	77.06	167.21	131.93	22.68	146.36	82.93	49.58	129.41	89.45
	1024x1024	133.15	251.43	209.64	37.56	225.84	130.04	81.65	191.31	138.78
Theatre	256x256	3.12	29.55	13.48	2.16	24.43	10.79	3.45	15.71	8.91
	512x512	11.89	67.41	37.29	7.16	71.21	33.25	12.42	45.65	26.78
	768x768	28.73	140.58	78.70	16.40	114.60	55.82	29.78	96.26	59.20
	1024x1024	48.13	208.10	119.70	26.35	177.42	87.29	50.93	159.73	93.61
Office	256x256	33.41	45.29	39.13	16.28	20.97	18.39	10.50	15.27	12.73
	512x512	77.84	97.22	86.48	49.03	61.40	55.78	28.36	36.14	33.21
	768x768	139.25	184.29	152.67	68.19	84.56	74.73	54.96	64.26	59.27
	1024x1024	202.49	249.43	218.62	108.11	130.41	116.12	82.61	96.30	87.94
Conference	256x256	14.73	53.12	35.14	11.08	27.95	19.11	7.37	18.60	12.90
	512x512	44.56	106.71	80.02	33.97	76.33	61.52	20.87	35.97	28.37
	768x768	101.06	193.21	161.53	50.10	120.62	100.93	38.47	67.90	58.62
	1024x1024	161.21	275.60	228.15	77.14	180.93	152.96	58.01	99.20	85.20

Tabulka B.2: Výsledky měření při použití stínových paprsků na kartě Nvidia GTX280

		čas[ms]								
		Mřížka			Kd strom			BVH		
scéna	rozlišení	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Robots	256x256	0.73	51.09	16.92	0.88	15.14	5.53	1.59	18.26	8.04
	512x512	2.46	135.97	38.57	3.13	49.90	17.79	5.52	42.54	20.50
	768x768	6.31	260.80	64.65	7.74	71.88	27.68	13.90	83.49	41.75
	1024x1024	10.82	381.21	88.97	13.35	111.43	43.76	23.84	128.90	64.31
Musem	256x256	2.36	27.83	20.46	1.61	29.66	21.86	1.61	36.37	26.04
	512x512	8.65	81.32	60.66	5.96	95.12	72.07	5.78	89.76	65.54
	768x768	20.63	153.38	113.96	13.96	161.53	115.36	14.30	160.78	111.82
	1024x1024	35.91	219.50	168.54	24.15	247.63	184.17	25.76	219.34	163.70
Kitchen	256x256	11.66	76.88	49.42	3.23	49.74	29.40	6.06	93.08	53.46
	512x512	34.96	233.34	152.28	11.33	157.80	98.28	21.27	262.28	153.10
	768x768	77.41	498.36	293.08	22.85	278.68	153.86	51.37	534.33	270.72
	1024x1024	135.70	727.99	442.80	37.84	438.18	244.38	86.77	785.87	403.92
Theatre	256x256	3.14	93.09	38.68	2.17	65.66	21.24	3.96	70.49	29.06
	512x512	11.99	299.23	125.38	7.17	220.16	74.33	13.75	217.80	93.79
	768x768	29.09	551.80	254.07	16.44	328.16	125.95	31.94	396.26	192.33
	1024x1024	48.81	856.55	379.72	26.43	547.61	201.56	54.48	611.28	292.07
Office	256x256	33.55	45.29	39.13	16.57	20.91	18.61	10.98	15.78	13.29
	512x512	77.98	98.11	87.14	50.44	62.59	56.62	29.96	38.69	35.07
	768x768	141.69	187.35	155.52	71.40	86.10	77.48	59.15	68.69	63.00
	1024x1024	203.02	254.39	223.97	111.73	134.87	120.13	89.54	103.41	94.20
Conference	256x256	15.00	76.81	45.50	—	—	—	8.58	32.80	19.67
	512x512	45.17	151.67	106.71	—	—	—	21.76	61.11	41.80
	768x768	103.44	276.15	215.85	—	—	—	40.33	103.59	82.92
	1024x1024	165.04	376.68	292.78	—	—	—	61.22	149.95	114.15

Tabulka B.3: Výsledky měření při použití odražených paprsků na kartě Nvidia GTX280

		čas[ms]								
		Mřížka			Kd strom			BVH		
scéna	rozlišení	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Bunny	256x256	13.16	25.26	18	16.12	31.48	21.42	17.18	33.06	23.59
	512x512	32.93	61.07	45.19	46.79	88.53	64.84	34.07	55.07	43.3
	768x768	58.11	102.03	77.34	86.44	158.72	117.18	52.51	79.64	65.37
	1024x1024	87.15	149.21	114.61	129.06	233.01	175.43	75.33	111.22	91.37
Dragon	256x256	29.35	68.69	47.86	15.37	37.99	27.44	40.43	98.33	70.88
	512x512	75.5	158.19	116.57	50.68	116	83.6	84.67	190.77	140.53
	768x768	134.24	271.53	198.1	96.66	221.13	160.71	115.28	267.43	199.11
	1024x1024	200.92	393.23	291.4	155.43	347.16	251.7	157.96	339.94	254.96
Budha	256x256	18.5	65.36	44.41	10.65	32.1	23.23	24.19	91.5	66.17
	512x512	45.72	151.58	105.01	28.67	95.85	67.78	50.22	184.77	129.6
	768x768	72.9	251.2	170.75	54.85	184.66	127.5	73.95	276.65	187.38
	1024x1024	105.5	353.61	239.07	88.35	270.28	194.07	96.81	333.45	231.55
Robots	256x256	3.15	112.51	28.2	3.47	32.93	12.64	7.28	26.14	17.29
	512x512	11.13	384.22	77.87	13.15	88.57	37.34	27.9	81.6	56.99
	768x768	26.18	824.45	153.54	28.78	165.17	72.71	61.98	168.45	120.3
Musem	256x256	3.56	38.31	20.91	3.02	41.84	27.93	4.65	22.24	17.02
	512x512	13.6	131.53	64.02	11.88	122.02	85.33	18.3	65.7	49.55
	768x768	30.88	275.12	128.14	26.5	220.94	157.68	41.27	134.9	99.97
Kitchen	256x256	10.1	60.68	34.72	6.61	38.7	25.54	12.03	36.63	25.57
	512x512	34.4	165.88	99.06	25.64	114.48	74.78	46.95	95.57	74.51
	768x768	73.29	320.5	194.05	57.8	214.74	143.11	105.27	183.56	149.91
Theatre	512x512	20.06	349.94	124.99	20.16	204.25	81.48	41.94	152.87	77.75
Office	512x512	127.71	160.55	144.22	70.15	98.27	83.5	51.85	64.25	57.22
Conference	512x512	124.59	226.08	186.67	68.13	176.03	137.59	52.40	81.41	69.30

Tabulka B.4: Výsledky měření primárních paprsků na kartě Nvidia 8600GT

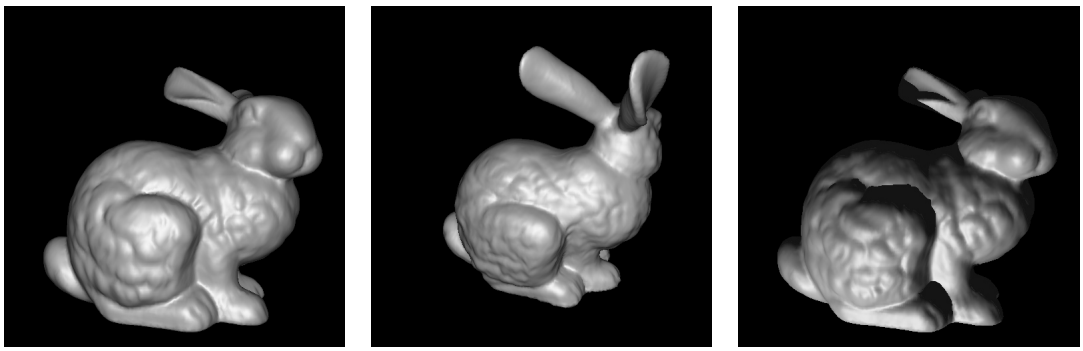
		čas[ms]								
		Mřížka			Kd strom			BVH		
scéna	rozlišení	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Bunny	256x256	24.32	39.13	32.42	31.53	51.31	40.87	36.79	49.28	43.97
	512x512	63.57	94.42	79.12	93.86	141.60	115.41	71.04	90.36	80.83
	768x768	111.37	163.63	137.01	166.20	259.80	206.93	108.56	139.82	123.92
	1024x1024	166.42	249.14	205.05	248.43	389.28	313.40	154.16	194.99	175.96
Dragon	256x256	60.08	103.93	82.31	39.03	69.24	51.05	110.16	152.59	133.19
	512x512	166.93	274.85	217.66	127.79	226.12	159.39	230.75	313.67	278.23
	768x768	287.64	474.75	376.31	247.20	400.34	307.41	341.45	463.67	408.12
	1024x1024	443.15	709.47	568.71	393.76	624.95	485.46	455.36	618.76	544.68
Budha	256x256	58.70	119.21	95.69	30.05	60.20	47.77	93.08	172.34	142.04
	512x512	152.21	303.93	232.25	89.72	167.01	133.47	203.13	335.97	286.00
	768x768	239.97	493.39	373.64	168.98	319.30	254.88	298.18	505.25	428.49
	1024x1024	331.54	677.42	514.81	253.33	473.12	380.31	368.78	609.16	524.30
Robots	256x256	5.98	213.85	59.46	7.91	60.62	25.20	12.84	51.77	33.78
	512x512	22.61	652.17	155.15	31.47	156.32	74.64	49.72	157.56	109.09
	768x768	52.53	1390.24	294.84	70.69	293.37	145.45	111.04	321.46	228.52
Musem	256x256	15.95	83.51	59.84	14.91	88.98	63.83	13.32	58.81	46.94
	512x512	59.10	262.59	180.55	57.72	260.97	190.48	52.37	160.72	129.83
	768x768	134.60	532.65	359.90	124.73	478.63	356.99	115.05	311.58	254.20
Kitchen	256x256	97.89	249.00	180.04	26.56	170.79	97.88	49.26	186.51	123.94
	512x512	321.19	640.42	515.24	88.72	458.80	272.41	183.32	464.43	333.25
	768x768	675.63	1212.48	1003.11	193.69	858.07	512.00	393.81	881.86	643.53
Theatre	512x512	110.17	656.67	349.93	58.20	413.58	194.16	112.23	349.35	213.95
Office	512x512	587.26	694.72	620.75	235.09	281.92	257.25	195.52	229.05	208.56
Conference	512x512	391.87	716.80	587.89	158.73	364.87	311.86	144.84	242.45	206.06

Tabulka B.5: Výsledky měření při použití stínových paprsků na kartě Nvidia 8600GT

		čas[ms]								
		Mřížka			Kd strom			BVH		
scéna	rozlišení	min.	max.	prům.	min.	max.	prům.	min.	max.	prům.
Robots	256x256	6.91	406.39	93.97	8.34	83.20	30.64	13.68	101.24	45.35
	512x512	26.05	1070.50	240.98	32.67	217.85	89.39	52.59	274.02	138.16
	768x768	60.81	2165.67	433.88	73.98	393.09	170.16	117.74	508.41	275.96
Musem	256x256	16.62	237.30	174.66	14.57	187.69	133.30	14.07	243.24	174.14
	512x512	60.73	671.59	493.16	56.23	535.85	393.55	55.51	566.08	429.71
	768x768	140.93	1253.69	929.72	121.12	990.48	743.50	121.11	996.40	765.75
Kitchen	256x256	100.42	613.42	392.94	26.19	295.20	164.97	50.73	613.95	346.52
	512x512	329.94	1850.11	1103.48	89.99	836.84	460.49	188.92	1808.61	913.98
	768x768	693.45	3550.14	2064.27	191.57	1576.22	860.99	405.50	3372.23	1654.01
Theatre	512x512	123.88	2255.90	949.80	64.31	1077.97	390.65	125.39	1431.70	635.11
Office	512x512	599.36	702.66	636.14	242.61	285.30	263.55	209.43	243.26	224.08
Conference	512x512	414.55	905.56	744.89	—	—	—	151.76	345.34	274.24

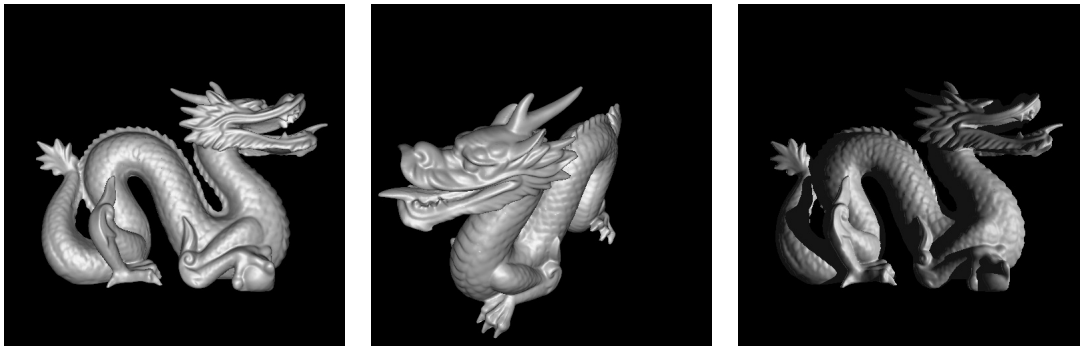
Tabulka B.6: Výsledky měření při použití odražených paprsků na kartě Nvidia 8600GT

C Obrazová příloha

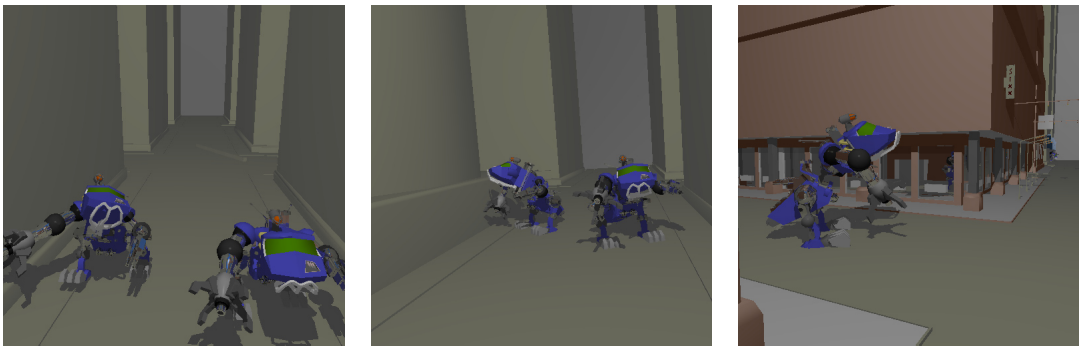


Obrázek C.1: Stanford Bunny.

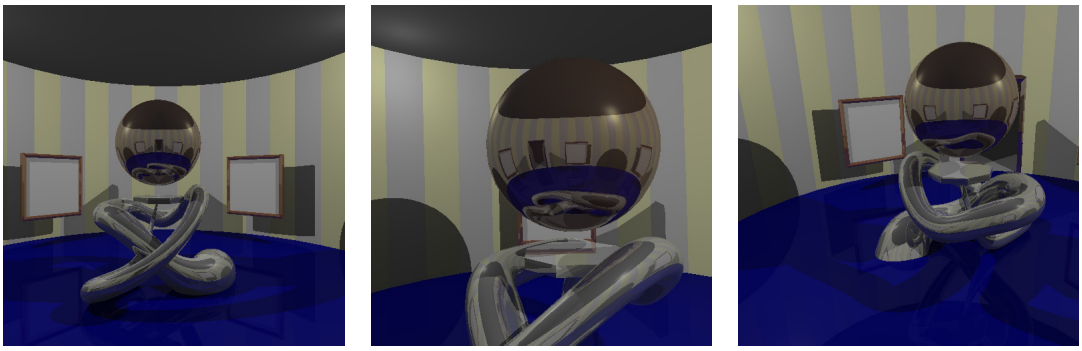
Obrázek C.2: Stanford Budha.



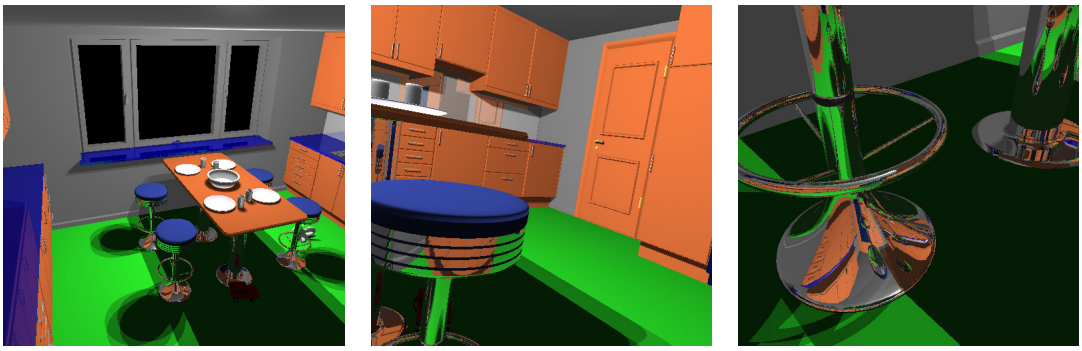
Obrázek C.3: Stanford Dragon.



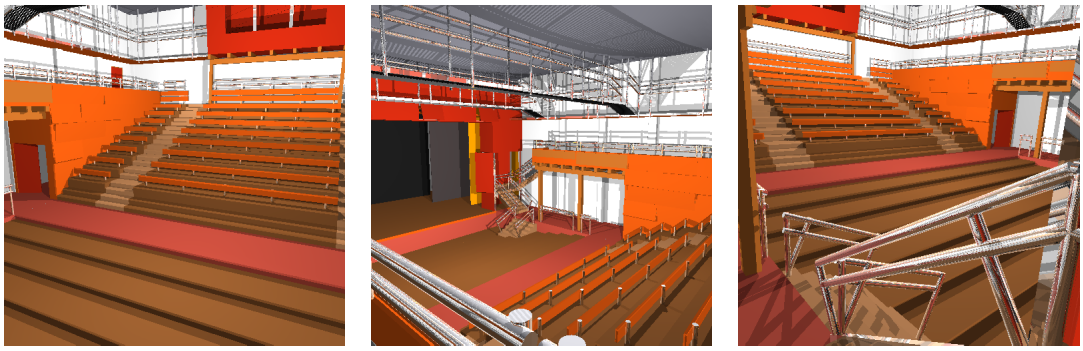
Obrázek C.4: BART Robots.



Obrázek C.5: BART Museum.



Obrázek C.6: BART Kitchen.

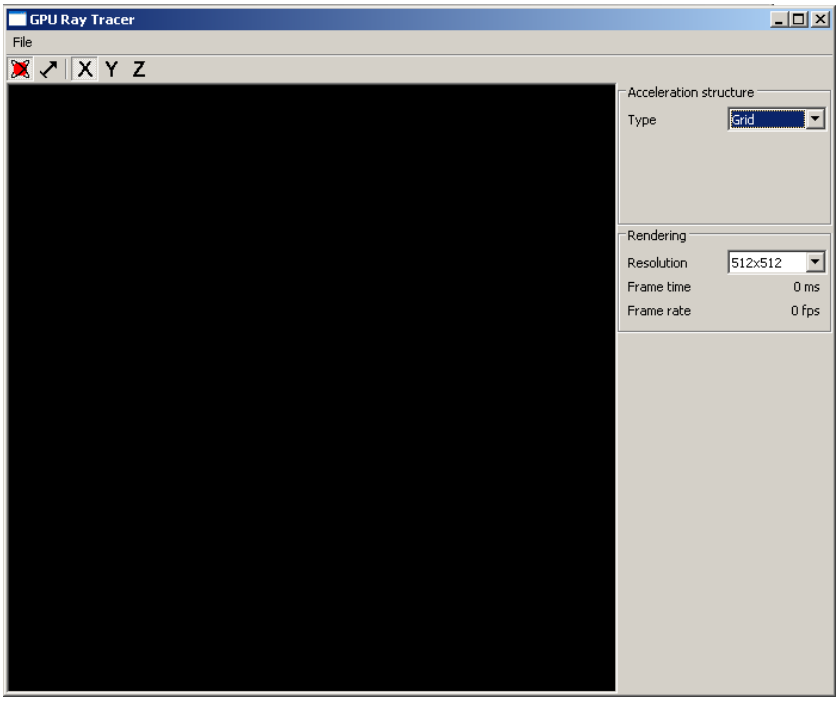


Obrázek C.7: Theatre.

Obrázek C.8: Office.

Obrázek C.9: Conference.

D Uživatelská příručka



Obrázek D.1: Uživatelské rozhraní programu.

Prvky uživatelského rozhraní

	Přepne ovládání pohledu do módu rotace
	Přepne ovládání pohledu do módu procházení scénou.
	Umožňuje výběr hlavní osy pohledu. Testovací scény pocházejí z mnoho zdrojů a může se u nich odlišovat směr pohledu. Pomocí těchto přepínačů je možnost nastavit pro prohlíženou scénu správné nastavení
Type	Umožňuje nastavit typ používané akcelerační struktury
Resolution	Umožňuje nastavit použité rozlišení pro vykreslování obrazu.

Ovládaní myší

Pomocí myši je možno ovládat pohled na scénu. Pohled je měněn umístěním myši do hlavního okna, stiskem levého tlačítka a následným tahem myši. Konkrétní reakce na ovládání myší závisí na zvoleném módu ovládání pohledu.

Mód rotace	Slouží pro zobrazení scén obsahující jediný objekt. Ten je umístěn do středu obrazu a tahem myši je možno rotovat kamerou okolo tělesa
Mód procházení	a Slouží pro procházení rozsáhlých scén. Při tahu myší se kamera otáčí okolo svého středu. Pro pohyb ve scéně jsou použity klávesy popsané v další části.

Ovládání klávesnicí

V módu procházení scénou je možno pohybovat kamerou pomocí následujících kláves.

W Pohyb kamerou směrem dopředu.
S Pohyb kamerou směrem dozadu.
A Pohyb kamerou směrem doleva.
D Pohyb kamerou směrem doprava.
Q Rotace kamery kolem osy pohledu proti směru hodinových ručiček.
E Rotace kamery kolem osy pohledu po směru hodinových ručiček.

E Obsah přiloženého CD

EXE	Program pro interaktivní zobrazení scén pomocí metody sledování paprsku.
HTML	Dokumentace zdrojových kódy ve formátu Doxygen.
SRC	
.GIT	Repozitář systému správy verzí GIT obsahující historie vývoje zdrojových kódu.
SRC	Zdrojové kódy aplikace.
VCPROJECTS	Projekt pro program Visual Studio 2005 použitelný k překladu zdrojových kódů.
TEXT	Zdrojový kód textu diplomové práce v formátu LaTeX.
VIDEO	Video sekvence zaznamenávající průchody testovacích scén.