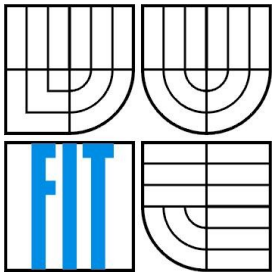


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

STREAMING RAYTRACER NA GPU

STREAMING RAYTRACER ON GPU

DIPLOMOVÁ PRÁCE

AUTOR PRÁCE — Bc. Jakub Dvořák
AUTHOR

VEDOUCÍ PRÁCE — Ing. Adam Herout, Ph. D.
SUPERVISOR

BRNO 2008

Abstrakt

Současné GPU je možné snadno použít jako vysoce výkonné stream procesory a představují tak lákovou platformu pro implementaci raytracingu. V první části práce stručně přibližuji základy raytracingu, programovatelnou pipeline moderních GPU a možnosti jejího využití. V druhé části popisuji algoritmy využité pro implementaci jednoduchého raytraceru a rozebírám experimenty s ním provedené.

Klíčová slova

Raytracing, Stream processing, GPGPU, CUDA

Abstract

Current consumer GPUs can be used as high performance stream processors and are a tempting platform to be used to implement raytracing. In this paper I briefly present raytracing principles and methods used to accelerate it, modern GPUs programmable pipeline and examples of its use. I describe stream processing in general and available interfaces enabling the usage of GPU as stream processor. Then I present my GPU raytracer implementation, used algorithms and experiments I have made.

Keywords

Raytracing, Stream processing, GPGPU, CUDA

Citace

Dvořák, Jakub: Streaming Raytracer na GPU. Brno, 2008, diplomová práce, FIT VUT v Brně.

Streaming raytracer na GPU

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením Ing. Adama Herouta, Ph.D..
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

….....….
Jakub Dvořák
18.5.2008

Obsah

1 Úvod

Již několik let jsou GPU dobře využitelné jako stream procesory. A s každou novou generací karet roste nejen výkon, ale i programovatelnost těchto procesorů. Již delší dobu se proto na grafické karty upírá pozornost výzkumu v oblasti raytracingu. Nová architektura DirectX 10 kompatibilních GPU spolu s novými rozhraními pro přístup k nim dále zlepšují využitelnost GPU pro negrafické aplikace (přesněji řečeno pro aplikace nevyužívající rasterizační pipeline) a tedy i raytracing.

Raytracing je poměrně oblíbenou technikou užívanou pro zobrazování trojrozměrných scén. Základům jeho činnosti a možnostem jeho akcelerace, která je nezbytná pro jakoukoliv netriviální implementaci se věnuji v druhé kapitole. Obecné principy a vlastnosti stream processingu popisuji na začátku třetí kapitoly. Dále se zabývám vlastnostmi moderních GPU s důrazem na možnosti využití programovatelné grafické pipeline z hlediska GPGPU. S tím úzce souvisí popis a kategorizace dostupných rozhraní pro využití GPU jako stream procesoru v kapitole čtvrté. Těchto rozhraní není mnoho, ale výběr je přesto poměrně pestrý: od využití grafických rozhraní OpenGL či DirectX v kombinaci se zavedenými shading jazyky až po nejnovější specializovaná rozhraní NVIDIA CUDA a ATI CTM.

Na práci převážně studijního rázu a několik experimentů provedených v rámci semestrálního projektu jsem navázal při implementaci. Raytracing na GPU je poměrně mladou oblastí výzkumu v počítačové grafice, proto existuje pouze několik významných prací, které větší měrou ovlivnili následující výzkum problematiky. Většinu z nich stručně popisuji v rychlém přehledu současného stavu výzkumu na začátku páté kapitoly a po načrtnutí hlavních problémů, kterým raytracing na GPU čelí se již věnuji pouze vlastní práci na implementaci raytraceru a provedeným měřeních. Rozebírám přitom nejdůležitější využité algoritmy, vztahy mezi nimi a výsledky experimentů a možná řešení zjištěných omezení. V šesté kapitole se stručně zabývám některými implementačními detaily projektu, organizací zdrojových kódů a využitými technikami.

2 Sledování paprsku

Obecně vzato je sledování paprsku či-li raytracing způsob modelování světla[1] založený na sledování jednotlivých paprsků a jejich interakce s okolím. V rámci geometrické optiky je využíván například i pro návrh objektivů, mikroskopů či dalekohledů. V počítačové grafice je raytracing označením pro rendrovací algoritmus využívající tuto techniku pro vizualizaci 3D scén.

2.1 Princip

2.1.1 Raycasting

Nejjednodušší formou sledování paprsku je raycasting. Poprvé byl popsán roku 1968 A. Appelem. Při raycastingu jsou jednotlivými pixely obrazu „vystřelovány" paprsky a hledány nejbližší objekty jim stojící v cestě. Výpočet trajektorie paprsku zastaví na tomto prvním/nejbližším objektu a není tedy simulován odraz světla či refrakce. Oproti *scanline* algoritmu ale umožnil reprezentaci nerovinných útvarů jako jsou například koule či kužele.

Raycasting byl využit například v počítačových hrách *Wolfenstein3D, v* enginu *Build* (*DukeNukem3D, Blood*) či v na *voxelech* založeném *Commanche*. [13]

2.1.2 „Whitted" raytraycing

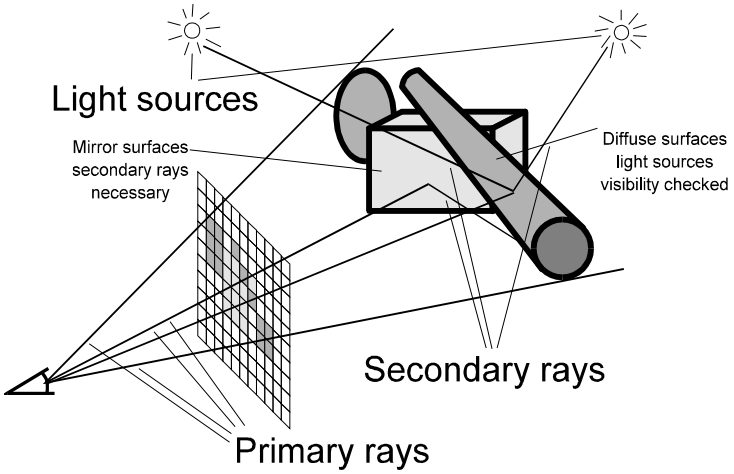
„Whitted" raytracing je od doby svého vzniku nejpoužívanějším a zřejmě i nejznámějším typem raytracingu. Algoritmus, který navrhl v roce 1979 Turner Whitted pracuje takto: Na rozdíl od předchozích algoritmů se paprsek po dopadu na povrch tělesa může odrazit – vygenerovat až tři nové paprsky – odražený, refraktovaný a stínový.

Odražené paprsky se vytváří po dopadu na lesklý povrch a pokračují v odrážení dále. První objekt, který tento paprsek protne bude vidět v odrazu. Odrážení ale může pokračovat dál. Počet odrazů, které paprsek provede je ale vhodné z výpočetních důvodů omezit – takováto omezení bývají různá: od zcela náhodného „zabíjení" paprsků až po stanovení maximální počtu odrazů, které paprsek může provést.

Refraktované paprsky jsou generovány při průniku primárního (nebo i sekundárního) paprsku s průhledným tělesem. Dráha paprsku je na styku dvou rozhraní s rozdílným indexem lomu zalomena a paprsek pokračuje dál. Typicky se při refrakci vytváří i odražený paprsek a omezení počtu paprsků se tak řídí stejnými pravidly.

Stínové paprsky jsou využívány k ověření, zda je bod na povrchu přímo viditelný z nějakého světla ve scéně. Pokud je viditelný, tak se toto světlo použije pro zobrazení daného bodu. Pokud je však mezi bodem odrazu a světlem jakýkoliv neprůhledný objekt, světlo pro výpočet osvětlení bodu použitu není. [13]

1 Sledování paprsku se využívá i pro modelování šíření zvukových vln v oceánu či elektromagnetických vln v ionosféře či v seismologii.



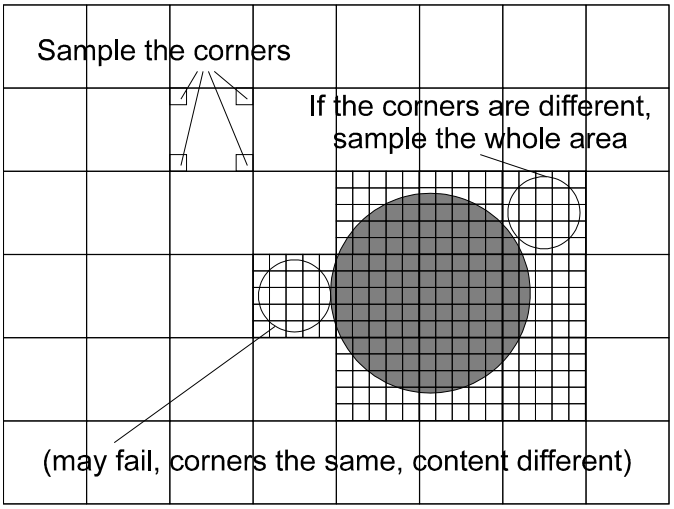
Obrázek 2.1: Princip Whitted Ray tracingu [14]

2.2 Akcelerace raytracingu

Naivní či brute-force přístup k raytracingu trpí velice špatným výkonem a tak vzniklo několik typů optimalizací.

2.2.1 Redukce počtu rendrovaných pixelů

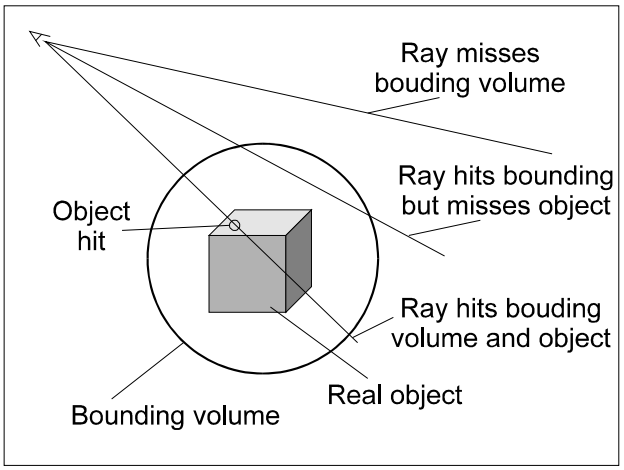
Nejednodušším způsobem snížení náročnosti raytracingu je zmenšení počtu rendrovaných pixelů. Nejpoužívanější metodou je adaptivní subsampling. Při jeho vhodném užití lze dosáhnout zrychlení v řádu desítek procent.



Obrázek 2.2: Redukce počtu rendrovaných pixelů pomocí adaptivního subsamplingu[14]

2.2.2 Využití obalových těles

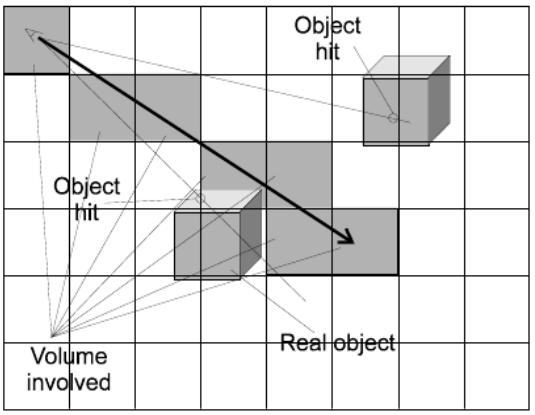
Aby obalová tělesa dobře fungovala, musí obsahovat celý objekt při minimálním vlastním obje-



Obrázek 2.3: Vyu~ití obalových těles [14]

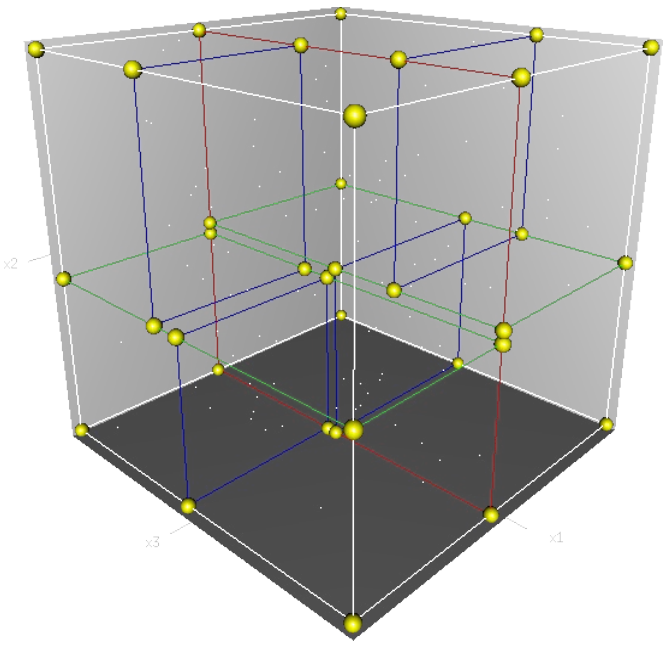
mu. Z výpočetního hlediska je nelepším obalovým tělesem koule. Pro posouzení zda paprsek kouli protíná totiž stačí zjistit počet řešení rovnice pro výpočet průsečíku. Obalová tělesa ve tvaru koule se však relativně obtížněji vytvářejí. U všech obalových těles platí, že je třeba, aby byla připraven předem. Ať již ručně autorem scény nebo softwarem, který scénu generuje. Pomocí obalových těles lze dosáhnout více než desetinásobného zrychlení rendrování scény.

2.2.3 Rozdělení prostoru



Obrázek 2.4: Uniform grid [14]

Dělení prostoru je metodou užívanou v mnoha oblastech pracujících s prostorem jako takovým (mimo jiné v GIS či obecněji prostorových databázích) a existuje množství algoritmů pro něj využívaných. Ne všechny jsou však vhodné pro raytracing. Nejpoužívanější metodou dělení prostoru při raytracingu je Uniform Grid (ilustrace 2.4) a různé varianty prostorových stromů (kd-tree, oct-tree či bsp-tree).



Obrázek 2.5.: 3D kd-strom. Plocha dělící kořenový uzel je znázorněna červeně, zelené jsou dělící plochy uzlů/buněk druhého řádu a modře uzlů řádu třetího. [12]

3 Programovatelné GPU a stream processing

3.1 Princip stream processingu

Stream processing je programovací princip který, při použití na vhodném typu úloh, značně ulehčuje využití paralelizace. Proces zpracování data je popsán sérií výpočetně intenzivních operací – jader (kernel) na vstupních / výstupních datech – proudech (stream). Hlavními výhodami tohoto typu zpracování jsou:

- vysoká výpočetní intenzita – převážná část instrukcí prováděných jádrem jsou výpočty a ne například přístup do paměti či řízení běhu programu
- lokalita – jádro zpravidla pracuje s malým množstvím dat, pro která něž stačí poměrně malé množství (rychlé) on-chip paměti a značně se tak (společně s předchozím bodem) snižují nároky na rychlost přístupu paměti systémové. Rychlost přístupu do paměti je u všeobecných procesorů v poslední době řešena rostoucí velikostí vyrovnávací paměti, která ale zabírá stále více místa v čipech. Výrazně menší velikost vyrovnávací paměti ve stream procesorech umožňuje věnovat větší část čipu výpočetním jednotkám a dále tak zvýšit výpočetní výkon
- abstrakce – pro programátora odpadá nutnost synchronizace přístupu k datům, považovaná za hlavní úskalí a zdroj chyb při vícevláknovém zpracování. Naproti tomu překladač (a následně hardware) takto získává mnohem více informací o vztazích a závislostech mezi daty díky čemuž může lépe plánovat běh vláken vzhledem k přístupu k paměti.

Za jistou formu stream processingu se dají považovat i *SIMD* instrukce moderních všeobecných procesorů.

3.2 Programovatelné GPU

Prvním programovatelným GPU byla NVIDIA GeForce 3 uvedená na trh v roce 2001. Od té doby proběhl značný vývoj. Možnosti shaderů[2], či-li programovatelných součástí grafického pipeline, značně vzrostly po všech stránkách: zvětšilo se množství použitelných instrukcí, řádově vzrostla maximální délka programů a množství registrů, obrovsky vzrostl výkon grafických procesorů…

2 Pojem „shader“ se vžil pro pojmenování programovatelných jednotek GPU a v následujícím textu jej budu používat výhradně v tomto významu.

3.2.1 Vertex Shader

Vertex shader je programovatelná jednotka operující na atributech příchozích vertexů, tedy barvě, pozici, osvětlení, texturovacích souřadnicích a dalších. Vstupem a výstupem je pouze jeden vertex a jeho atributy.

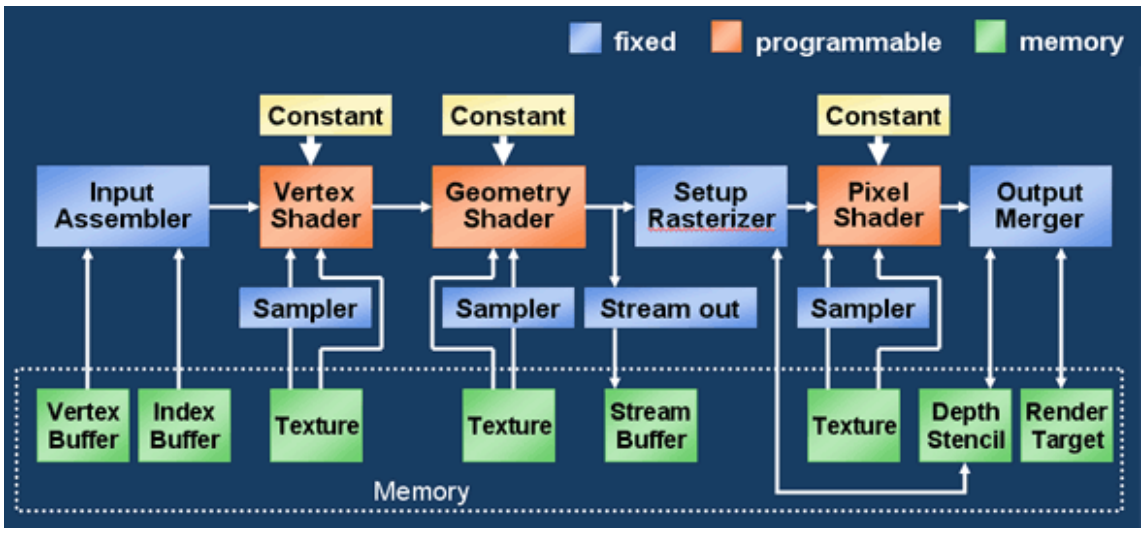
Programy běžící na vertex shaderech tedy nemají k dispozici žádné topologické informace. Vertex programů se tak většinou využívá pro přípravu či přímo výpočet dat pro fragment shadery (která jsou následně interpolována pro jednotlivé fragmenty).

3.2.2 Fragment Shader

Fragment nebo pixel shadery slouží ke zpracování již rasterizovaných částí obrazu. Vstupem jsou informace o poloze fragmentu, texturovací souřadnice, informace o osvětlení a jakákoliv další data předaná sem vertex shaderem. Z důvodu snadné paralelizace může FS zapsat pouze jednu hodnotu na předem dané souřadnice – a to barvu výsledného pixelu.

3.2.3 DirectX 10 / Shader model 4.0

Po delší době spíše evolučního vývoje přinesla DX10 / SM 4.0 specifikace velké změny. Z nich nejvýznamnější jsou USA (Unified Shader Architecture) a rozšíření pipeline o novou programovatelnou část – Geometry shader.



Obrázek 3.1: DirectX 10 rendrovací pipeline

Z hlediska stream processingu (a tedy mé práce) je nejvýznamnější již zmíněná USA architektura. Namísto dosavadních oddělených / specializovaných jednotek pro zpracování vertex a pixel programů obsahují nové GPU takzvané unifikované shadery na kterých běží *vertex*, *pixel* a (nové) *geometry* programy. Výhodou by mělo být hlavně lepší využití výkonu GPU – rozdílné typy scén mívají rozdílné nároky na výkon pixel a vertex shaderů. Proto docházelo k situacím, kdy jeden typ jeden typ shaderů zahálel zatímco druhý výkonově nedostačoval. V DX10 GPU

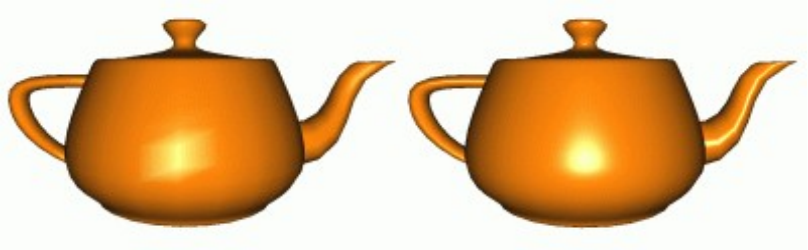
by zjednodušeně řečeno měl hardwarový arbitr rozdělovat jádra mezi běžící vlákna dle aktuální potřeby a maximalizovat tak využití dostupného výpočetního výkonu.

3.3 Příklady použití

Shadery se staly tak běžnou součástí rendrovací pipeline, že se bez nich neobejde prakticky žádná soudobá počítačová hra. Stále větší část zpracování scény se odehrává právě v shaderech a očekává se, že tento trend bude pokračovat (DX 10 je toho důkazem[3]). Pro demonstraci možností shaderů jsem vybral následující jednoduché ukázky:

3.3.1 Per fragment osvětlení

Standardní pevná pipeline počítá osvětlení ve pouze pro vertexy / vrcholech tělesa a pro jednotlivé fragmenty ho interpoluje. Pokud v pixel shaderu tuto interpolovanou hodnotu normalizujeme, kvalita zobrazení méně detailních modelů se značně zvýší.



Obrázek 3.2: Per vertex (vlevo) a per pixel osvětlení

3.3.2 Nerealistické zobrazování

V některých počítačových hrách se naopak pro navození atmosféry snižuje realismus zobrazování. Příkladem muže být prahování obrazu, které má za výsledek „komiksový“ vzhled zobrazovaných objektů.

Obrázek 3.3: Cartoon shader

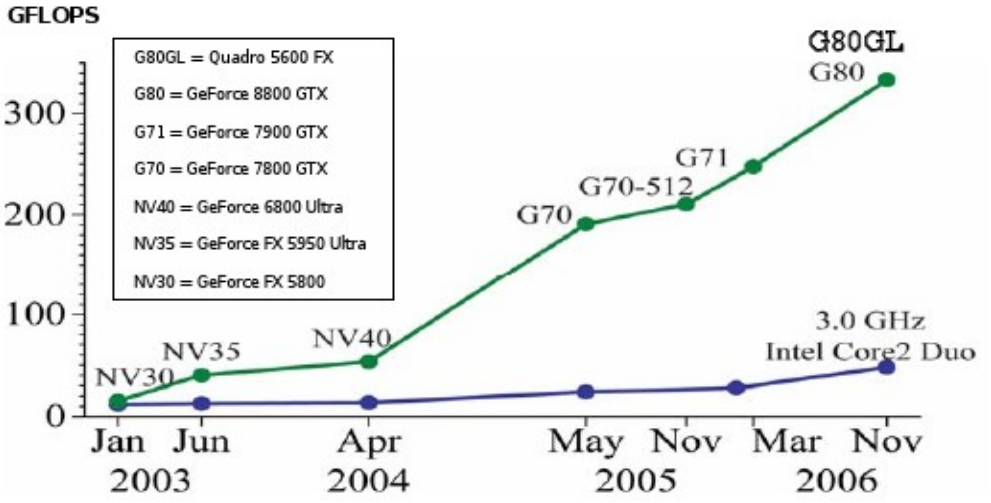
3 V Direct3D 10 byla kompletně vypuštěná pevná pipeline a vše je tedy třeba implementovat pomocí shaderů. Odlišný přístup volí OpenGL 3.0, zde je fixní pipeline size také odstraněna, ale v režimu zpětné kompatibility je pomocí shaderů emulována.

3.4 Srovnání výkonu CPU a GPU

Přímé srovnání GPU a CPU je poměrně obtížné. Soudobé GPU jsou velice paralelizované a hodí se jen pro omezený typ úloh. CPU jsou naproti tomu schopné „přiměřeně“ dobře řešit široké spektrum úloh. Ale i CPU lze v posledních letech pozorovat příklon ke stále větší paralelizaci. Tento trend je však spíše vynucen možnostmi výrobní technologie, která neumožňuje další zvyšování výkonu jednojádrových procesorů při zachovaní rozumných energetických a tepelných vlastností. Dále je nutnou podotknout, že soudobé GPU pracují na rozdíl od CPU pouze s jednoduchou (32bit) přesností desetinných čísel, což především pro vědecké aplikace může představovat významné omezení. To by však měla změnit následující generace GPU – tedy NVIDIA 9xxx a ATI/AMD 4xxx.

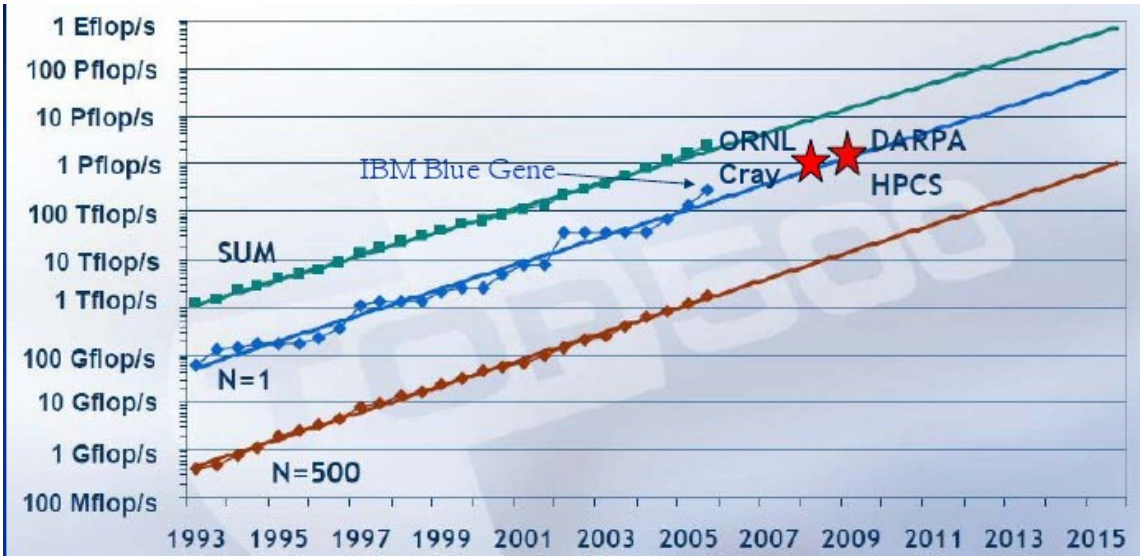
Obrázek 3.4: Srovnání architektury CPU a GPU [2]

GPU jsou konstruovány jako *stream procesory* se všemi z toho plynoucími výhodami, tedy vysokou výpočetní intenzitou, lokalitou dat a z toho plynoucími nízkými nároky na rychlost (latenci) přístupu do systémové paměti. Tomu také odpovídá architektura čipů, schematicky zobrazená na ilustraci 3.4. A při vhodném typu úloh a správném využití tomu odpovídá i podávaný výkon (ilustrace 3.5).



Obrázek 3.5: Srovnání výkonu CPU a GPU [2]

V současné době začala NVDIA nabízet GPGPU karty Tesla a na nich postavené servery s výkonem až 2 TFLOP. Cluster sestavený pouze z několika Tesla serverů by tedy v současné době mohl aspirovat na zápis mezi Top500 superpočítačů, kde vede IBM Bluegene s výkonem přes 1 PFLOP a na konci žebříčku se pohybují stroje/clustery s výkonem kolem 4-5 TFLOP.



Obrázek 3.6.: Trend růstu výkonu superpočítačů

3.5 Intel Larrabee

V poslední době si stále více pozornosti získává vyvíjený grafický/stream processor Intel Larrabee představený veřejnosti na jaře 2006. Od ostatních (v současné době dostupných) GPU se liší především tím, že využívá jen mírně upravenou x86 instrukční sadu. To by mělo oproti

graficky orientovaným sadám stávajících přinést větší flexibilitu. Procesor je také od samého počátku konstruován pro stream processing.

V současné době Intel intenzivně pracuje na DirectX a OpenGL rozhraních pro tento procesor. Navzdory mnoha po internetu kolujícím mýtům toto rozhraní využívá „klasickou“ rasterizaci a ne raytracing (alespoň dle jednoho z vývojářů [10]). Larrabee působí rozhodně zajímavě, otázkou je, zda to tak bude i v době jeho uvedení na trh – v roce 2010. Již současná generace grafických procesorů je značně flexibilnější než tomu bývalo a tím i blíže obecným stream procesorům a jejich výrobci rozhodně bitevní pole bez boje vyklidit nehodlají.

4 Rozhraní pro stream processing na GPU

Postupné zvětšování programovatelnosti a zvyšování výkonu umožnilo využití GPU pro ne-grafické (obecné) aplikace. Využití masivně paralelní architektury soudobých GPU může přinést (a již přináší) v mnoha oblastech velký růst výkonu. Příkladem může být projekt Folding@Home.

S pomocí ATI/AMD byl během roku 2006 vyvinut nový klient F@H, který je schopen využívat grafické procesory ATI X19xx. Podle statistik projektu dosahují tyto klienty 70x většího výkonu než průměrný klient a 19x většího výkonu než klient běžící na moderním CPU (Intel Core Duo a lepší). Dlužno ale dodat, že mnohem větší význam mělo nasazení F@H na herní konzole Playstation 3. IBM/Sony procesor CELL sice nedosahuje takového výkonu jako ATI 19xx, ale rozšířenost konzolí společně se snadnějším užitím (od firmware verze 1.3 je F@H na PS3 předinstalován) udělaly své – v současné době kolem 70% výkonu F@H zajišťují právě tyto konzole.

Prostředky pro generické programování GPU byly dlouhou dobu značně omezené – malý počet registrů a velice omezená instrukční sada v Pixel Shaderech neumožňovaly větší rozšíření / nasazení této technologie (pro většinu aplikací jsou důležité především Pixel Shadery). Ze strany GPU byl průlomem Shader Model 3, který možnosti Pixel Shaderů značně rozšířil – přibyla možnost smyček a podmíněných skoků v kódu a značně se prodloužila maximální délka programů. Na straně programovacích rozhraní šel vývoj také kupředu a tak jsou pro stream processing na GPU v současné době následující možnosti:

- přímé využití grafického rozhraní OpenGL či DirectX
- použití obecného stream processing / programming jazyka či knihovny s GPU backendem
- použití specializovaného jazyka či rozhraní

4.1 Grafická rozhraní

Stále velmi používaným způsobem GPGGU je využití grafického rozhraní GPU. Je to dáno jednak tím, že velká část GPGGU využití je v počítačových hrách či jiných aplikacích, které grafické rozhraní tak jak tak využívají. A jedná se o stále nejjistější, nejkompatibilnější a nejdostupnější způsob.

Textury jsou užívány jako pole / zdroje dat, shader programy (především pixel shadery) jsou výpočetními jádry. Důležité omezení plynoucí z využití grafické pipeline je předem určená adresa návratové hodnoty (odpovídající souřadnicím pixelu / fragmentu v obraze) a nemožnost zápisu více hodnot (kromě GPU podporujících SM 4 a novější).

Zpočátku bylo nutné shadery programovat v „grafickém assembleru“, postupně se však ob-

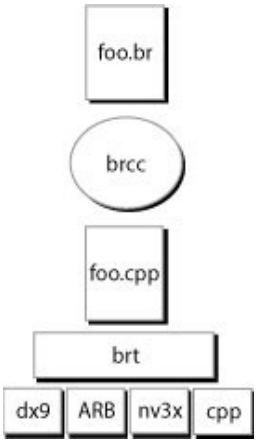
jevily jazyky vyšší úrovně:

- HLSL (High Level Shading Language, 2003)– vytvořen ve spolupráci Microsoftu a NVIDIA a vázaný na Direct3D (jehož součástí od verze DX9.0 je)
- Cg („C for Graphics“, 2002) – ač byl vytvořen v NVIDIA, není vázán na hardware, grafickou platformu ani operační systém.
- GLSL (GL Shading Language, 2001) – dostupný jako rozšíření v mnoha implementacích OpenGL od verze 1.4, součásti OpenGL je od verze 2.0

4.2 „Obecná rozhraní“

4.2.1 BrookGPU

Brook je na C založený *stream processing* jazyk. BrookGPU je implementací a překladačem tohoto jazyka pro moderní GPU. K dispozici je DirectX 9, OpenGL ARB, NV3x a referenční C++ backend, vhodný především pro debugování.



Obrázek 4.1: Brook

Překladač *brcc* je v podstatě preprocesor, který ze zdrojových kódů jazyka Brook vytvoří C++ zdrojové kódy užívající příslušný backend poskytovaný knihovnou *brt* (Brook Runtime Library).

4.2.2 LibSh

Na rozdíl od BrookGPU nevyžaduje LibSh speciální kompilátor či preprocesor. Pomocí C++ šablon a přetěžování operátorů LibSh shromažďuje požadované operace za běhu programu a následně je vykonává pomocí patřičného backendu. Je možné využít CPU a GPU backend.

Vývoj LibSh v průběhu roku 2006 prakticky ustal a na jejím základě vznikl komerční spinoff RapidMind. RapidMind Platform je oproti LibSh více zaměřena na GPGPU a je rozšířena především o podporu Cell procesorů a backend pro paralelizaci na standardních víceprocesorových systémech.

4.3 „Specializovaná rozhraní“

Nejnovější GPU jsou od základu koncipovány jako stream procesory a značnou část grafické pipeline v podstatě emulují (není to nic až tak nového: již v dobách DirectX 8 mnohé GPU emulovaly DirectX 7 T&L operace programem běžícím ve vertex shaderech). Specializovaná programová rozhraní výrobců GPU by tedy měla přinést snazší použitelnost pro GPGPU (stejně jako „obecná rozhraní“) a dále přinést znatelný nárůst výkonu především tím, že

- přímější přístup k hardware umožňuje lépe využít jeho specifických vlastností než abstrahované grafické rozhraní
- umožní rychlejší přenosy dat mezi kódem běžícím na CPU a kódem běžícím na GPU

V roce 2006 se pak objevily první beta verze NVIDIA CUDA (Compute Unified Device Architecture) a ATI CTM (Close To Metal).

4.3.1 ATI/AMD Close To Metal

ATI/AMD vytvořila rozhraní pro přístup k GPU na velice nízké úrovní, které umožňuje přístup k operační pamětí GPU a neomezené využití jeho instrukční sady. Na tomto základě by měly či mohly být postaveny nástroje vyšší úrovně jako jsou překladače, debbugery, matematické knihovny či přímo aplikační platformy. Otevřenost rozhraní umožňuje značnou flexibilitu a vznik i více konkurenčních „ekosystémů“. V březnu 2007 AMD na sourceforge.net v projektu *amdctm* uvolnila beta verzi CTM obsahující i virtuální stroj pro simulaci GPU. Projekt je ale poslední půlrok neaktivní – AMD údajně pracuje na nové verzi. Současný stav je bohužel takový, že CTM není prakticky použitelné – jedná se spíše o marketingový nástroj nežli skutečný produkt.

Do jiného světla výše zmíněné ale staví uvedení „ostré verze“ AMD Stream SDK. Jedná s v podstatě o obdobu CUDA postavenou na BrookGPU využívají CTM pro nízkoúrovňový přístup ke GPU.

4.3.2 NVIDIA CUDA

NVIDIA šla opačnou cestou. Namísto otevřené nízkoúrovňového rozhraní vytvořila kompletní uzavřenou / proprietární platformu pro GPGPU na jejích grafický procesorech. Výhodou se ukázala brzká dostupnost – již v březnu 2007 byla dostupná první, poměrně použitelná, betaverze CUDA SDK. Zprvu použití komplikovalo potřeba speciálních ovladačů, ale od prosince 2007 je vše potřebné pro běh aplikací využívajících CUDA součástí standardních ovladačů (alespoň v Linuxu).

5 Raytracing na GPU

5.1 Historie

Stále rostoucí výkon a programovatelnost grafických procesorů již delší dobu lákají pro využití pro raytracing. Mezi první počiny na tomto poli patří „The Ray Engine“ [1]. V této práci je grafický procesor využíván pouze pro výpočet průniku paprsku s trojúhelníkem. Transfer dat mezi pamětí grafické karty a pamětí systémovou je však ale slabou stránkou využití gpu pro generické výpočty a značně snižuje výkon. V roce 2002 ale nebyla jiná možnost (programovatelnost GPU nebyla ještě na dostatečné úrovni) a i přesto autoři dosáhli zrychlení cca o 30% oproti verzi běžící čistě na CPU. Pro výpočet průsečíku paprsku a trojúhelníku zde byla poprvé využita lehce upravená verze algoritmu z „Fast, Minimum Storage Ray/Triangle“[7], která je v této podobě využita ve velké části podobně zaměřených prací.

Prvním pokusem o kompletní provedení raytracingu na GPU je práce Timothy J. Purcella a kolektivu „Ray Tracing on Programmable Graphics Hardware“ [9]. Místo implementace na stávajícím hardware (která nebyla proveditelná) autoři vyšli z již dostupné specifikace GPU Shader Modelu 2 (na trh uvedeném v následujícím roce) a vytvořili implementaci využívajících jeho předpokládaných vlastnosti. Pro experimenty využili simulátor – virtuální stroj, který mimo jiné shromažďoval informace o vykonávaných instrukcí. Na základě těchto informací a výkonových charakteristik dostupných GPU pak odhadli potenciál jejich raytraceru: zhruba 56 milionů průsečíků za sekundu oproti zhruba 20 M/s, kterých dosahoval pro SSE optimalizovaný raytracer Ingo Walda z roku 2001.

V roce 2002 byly uvedeny na trh první karty se SM 2 a objevili se i první reálné implementace. Raytracing na GPU tedy byl prokázán za proveditelný, výzkum se obrátil na jeho zrychlování. Jak jsem již zmínil, algoritmus pro průsečík paprsku s trojúhelníkem zůstal ve většině prací nezměněn a hlavní těžiště akcelerace je tedy stejně jako u CPU implementací v urychlení procházení prostoru. Nejdříve a nejvíce využívaným přístupem je v [9] zavedená Uniform Grid. Výhody jsou zřejmé – pro uložení dat lze využít 3D textury, které jsou přímo podporovány grafickým hardware a tak je implementace jednak poměrně jednoduchá a také rychlá, protože hardwarově akcelerovaná. Vzhledem k možnostem a způsobu práce GPU byly jiné, pokročilejší, způsoby organizace prostoru považovány nevhodné. Alespoň do doby než Daniel Reiter Horn a kolektiv [5] modifikovali nerekurzivní verzi k-D tree algoritmu z PhD práce Ingo Walda [11]. Společně s dalšími optimalizacemi (např. využití rasterizace pro získání průsečíků primárních paprsků), větším výkonem GPU a hlavně rozšířením možnostmi shaderů (v té době již SM 3) dosáhli velikého nárůstu výkonu, který umožnil realtimový raytracing v dříve nemožné kvalitě.

5.2 Problémy

Raytracing není ideální úlohou pro stream processing z více důvodů. Hlavním důvodem časové

složitosti raytracingu je procházení prostoru při hledání průsečíků a to je i při použití pokročilých algoritmů typ kd-stromu náročné na rychlý náhodný přístup do paměti. Právě to ale není silnou stránka stream procesorů obecně. Novější GPU (i některé jiné stream procesory) jsou schopné při dostatečném množství paralelně běžících vláken tuto latenci skrývat, ale i tato schopnost není při raytracingu plně uplatnitelná.

5.2.1 Skrývání latence

Skrývání latence probíhá tak, že GPU při požadavku na přístup k paměti přeruší běh aktuálního vlákna/procesu a spustí jiné vlákno[4]. V ideálním případě jsou při probuzení původního vlákna již data k dispozici. Klíčový je ale dostatečný počet souběžně běžících vláken. Pro uložení pracovních dat a stavu procesu jsou na GPU k dispozici pouze registry, kterých je v GPU řady G8x (GF8xxx) na první pohled obrovské množství. Pro běh běžných pixel a vertex programů je jejich počet naprosto dostačující a tak GPU latenci dobře skrývá. Při raytracingu je ale registrů alokováno mnohem více a vliv přístupu do paměti je tak mnohem větší. Tím spíše, že se často jedná o přístup náhodný[5]. Řešením by se mohlo zdát rozdělení výpočtu do více jader a tím snížení počtu alokovaných registru na vlákno. Dle mnou studovaných prací ale vývoj probíhal a probíhá opačně. Dobře je to vidět na srovnání prací [3] a [5]. Sami autoři přiznávají, že veliký nárůst výkonu oproti starší práci není ani tak dílem algoritmických vylepšení, ale větším výkonem a hlavně možnostmi hardwaru – konkrétně schopností provádět smyčky odpadla nutnost mnohonásobných průchodů polí/textur. V současné době s novou řadou grafických karet a dostupností technologií typu CUDA se zdá být trendem soustředit veškeré výpočty v jednom jádře (např. [8]). Potřeba ukládání mezivýsledků při rozdělení do více jader neklesá, spíš naopak a na místo rychlých registrů je pro ně použita výrazně pomalejší paměť.

5.2.2 Procedurální texturování

Všechny popisované raytracery používaly pouze standardní Bliss-Phongův světelný model a v lepším případě i textury a tak se dosud neprojevil podle mého názoru poměrně závažný problém: praktická nemožnost využití procedurálního texturování – jedné z hlavních deviz raytracingu jako takového. GPU jsou schopné efektivně pracovat pouze když je k běhu připraveno veliké množství vláken a když všechna současně běžící vlákna vykonávají stejný kód. Zavádění nových výpočetních jader/kódu je poměrně drahou operací. Procházení prostoru a odrážení paprsků příliš nezachovává prostorovou lokalitu a tak by využití procedurálních textur na GPU (zejména při velkém počtu objektů) obnášelo časté zavedení nového kódu jen pro několik málo paprsků a tedy naprosté pohřbení výkonu. Osobně bych viděl možnost v obnovení

4 Spíše než s jednotlivými vlákny toto přerušování pracuje s bloky/skupinami vláken, protože (jak jsem již dříve zmínil pro efektivní využití výpočetních schopností GPU je třeba aby co největší počet vláken vykonával stejný kód.

5 V CUDA je při dodržení jistých „přístupových vzorů“ latenci čtení i zápisu do paměti i bez vyrovnávací paměti minimalizovat. Je však nutné aby přistup k paměti by pravidelný a tak předvídatelný. Více v [2].

„spolupráce“ s CPU – přenosy mezi pamětí grafické karty a hostitelského stroje jsou dnes díky novým nízkoúrovňovým rozhraním (CUDA, CTM) a vývoji na poli počítačových sběrnic rychlejší než dříve (v [1] byly tyto přenosy zdrojem značného úbytku výkonu). GPU by mohl kompletně spočítat průniky paprsků se scénou i osvětlení a na CPU by zbylo doplnění případných procedurálních textur (v podstatě pouze barvy). Nadějně vypadají také experimenty s grafickou kartou počítaným globálním osvětlením. To by se (byť v zjednodušené podobě) mělo objevit v několika v současné době vyvíjených hrách.

Řešení tohoto problému je ale značně nad rámec mé práce a mého průniku do problematiky. Všechny práce se kterými jsem měl možnost se seznámit jsou čistě experimentální povahy a od případného praktického nasazení na GPU založeného raytraceru je velice daleko.

5.3 Srovnání Cg a CUDA

Pro implementaci jsem bral v potaz dvě možnosti. Využití grafického rozhraní a Shading jazyka (konkrétně Cg) a nebo specializované rozhraní nižší úrovně (v mém případě CUDA). Ve prospěch grafického rozhraní hovoří snadná přenositelnost, rozšiřitelnost a také to že se jedná o již poměrně ověřené produkty. CUDA je naproti tomu platforma zcela nová a omezená na hardware firmy NVIDIA, ale v mnoha ohledech zajímavější. Volba nebyla jednoduchá a proto jsem se rozhodl provést základní experimenty v obou prostředích a na základě této krátké (ale alespoň nějaké) zkušenosti vybrat platformu pro řešení mé práce. S jazykem/platformou Cg jsem již nějaké zkušenosti měl a tak vše proběhlo bez problémů.

Implementace v CUDA byla lehce odlišná. Překvapivě zde nejsou předdefinovány aritmetické operace pro práci s vektory. Ty však lze poměrně snadno dodefinovat – CUDA je totiž spíše než rozšířením C procedurálním podmnožinou C++ a umožňuje tedy kromě přetěžování funkcí a operátorů například i využití šablon. Uživatelem definované operátory by měly být prováděny stejně rychle jako předdefinované operátory v Cg. Jedno jádro G8xxx totiž na rozdíl od předchozí generace GPU obsahuje místo dvou čtyřmístných vektorových FPU osm skalárních. Z toho vyplývá, že například „po sobě“ provedené sčítání se vykoná stejně rychle jako dříve přímo podporovaný součet vektorů. A sečtení kupříkladu 5 3-rozměrných vektorů rychleji. I zde se ukazuje jak shader jazyky a grafické rozhraní abstrahují způsob práce GPU.

Při práci s CUDA jsem narazil na poměrně zákeřnou nedokumentovanou vlastnost – funkcím volaným na GPU z hostu nelze předávat parametry odkazem (u funkcí volaných pouze na hostu nebo pouze na GPU to lze). Vhledem k tomu, jak CUDA pracuje, je to pochopitelné, ale překladač si nestěžoval a ani runtime neprotestoval. „Pouze“ to nefungovalo. A právě zde jsem poznal velkou výhodou CUDA – možnost debugování. Napsaný kód je možné přeložit jak do podoby pro běh na GPU tak pro běh na CPU a následně použít standardní ladící nástroje. Při běhu na CPU se sice neprojeví problémy potencionálně způsobené synchronizací mnoha paralelně běžících vláken – vše je vykonáváno sekvenčně, ale vše ostatní funguje identicky[6]. A tak jsem přišel na problém s referencemi a zároveň se definitivně rozhodl pro platformu CUDA.

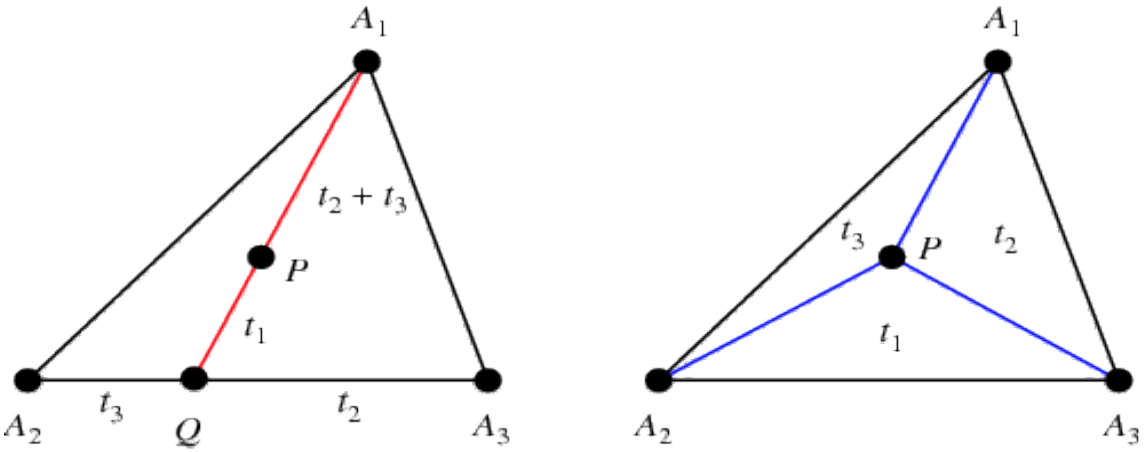
6 V průběhu práce jsem se ukázalo, že identičnost chování při emulaci a nativní běhu je spíš zbožným přáním autorů dokumentace než realitou, přesto je emulovaný běh velice užitečnou vlastností.

5.4 Průsečík paprsku s trojúhelníkem

Pro výpočet průsečíku paprsku s trojúhelníkem jsem použil dříve zmíněného algoritmu [7]. Jeho hlavní výhodou jsou minimální[7] nároky na paměť při srovnatelné rychlosti jako jiné užívané algoritmy. To představuje obzvlášť při implementaci na GPU výraznou výhodou. Další výhodou je, že průsečík je počítán v normalizovaných (homogenních) barycentrických souřadnicích, které jsou výhodné pro interpolaci normál a texturovacích souřadnic z vrcholů trojúhelníka.

5.4.1 Barycentrické souřadnice

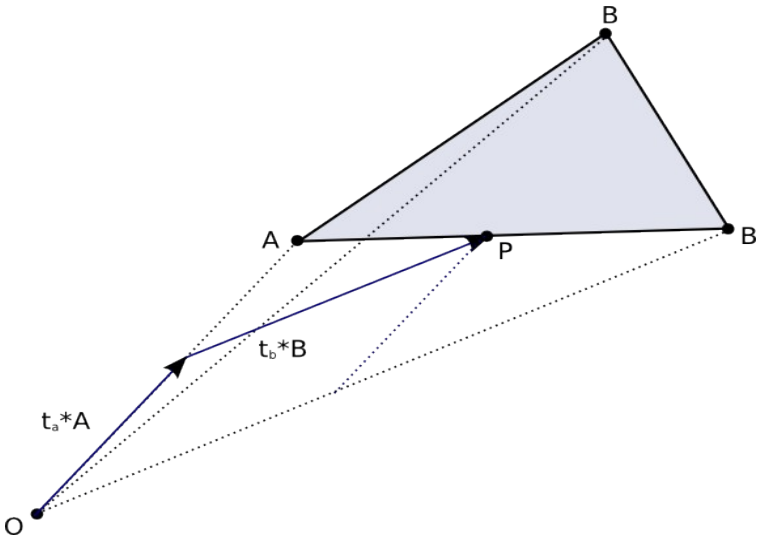
Poloha bodu *P* v trojúhelníku v homogenních barycentrických souřadnicích představuje hmotnosti, které je třeba umístit do jeho vrcholů aby daný bod *P* byl jeho těžištěm. Tyto hodnoty jsou úměrné plochám proti odpovídajícím vrcholům (vpravo na obrázku 5.1). Pro trojúhelník ABC tedy bod A má souřadnice (1, 0, 0), bod B (0, 1, 0) a bod C (0, 0, 1). Užitečnou vlastností je, že součet souřadnic $t_1+t_2+t_3=1$. Ze dvou souřadnic lze tedy třetí snadno dopočítat. Dle principu naznačeného na obrázku 5.2 ze znalosti souřadnic vrcholů (*A, B, C*) a barycentrických souřadnic (t_a, t_b a t_c) můžeme určit souřadnice daného bodu *P*.



Obrázek 5.1.: Barycentrické souřadnice [6]

$$\vec{P}=\vec{A}+t_B(\vec{B}-\vec{A})+t_C(\vec{C}-\vec{A})$$
$$\vec{P}=(1-t_b-t_c)\vec{A}+t_b\vec{B}+t_c\vec{C} \quad (5.1).$$

7 Nižší už opravdu asi být nemůžou – algoritmus nevyužívá žádné předpočítané hodnoty, pouze souřadnice trojúhelníku a směr a počátek paprsku.

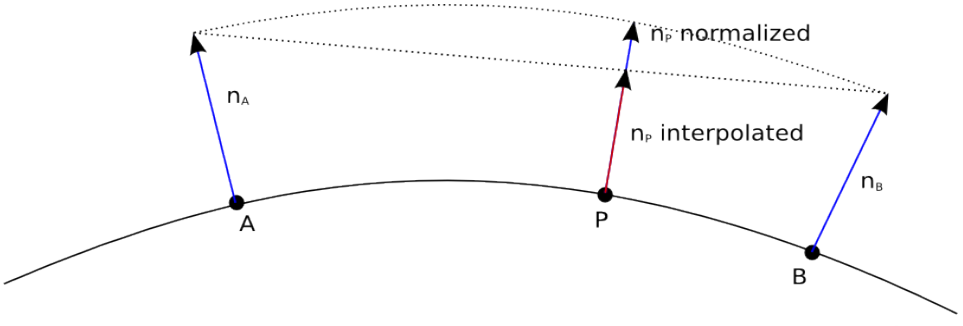


Obrázek 5.2: Výpočet polohy bodu P z barycentrických souřadnic (0.4, 0.5, 0)

Normálu v bodu *P* získáme podobně

$$\vec{n}_P=(1-t_B-t_C)\vec{n}_A+t_B\vec{n}_B+t_C\vec{n}_C$$

je ji však třeba následně normalizovat (obrázek 5.3)



Obrázek 5.3: Interpolace normály v bodu P

5.4.2 Výpočet průsečíku

Poloha bodu v trojúhelníku je určena rovnicí 5.1, polohu bodu *R na* paprsku (polopřímce) můžeme vyjádřit parametricky

$$\vec{R}=\vec{O}+k\vec{d}\ ,$$

kde $\vec{O}$ je počátek paprsku a $\vec{d}$ je jeho směr. Výpočet průsečíku odpovídá řešení rovnice

$$\vec{R}(k)=\vec{P}(t_B,t_C)$$

$$\vec{O}+k\vec{d}=(1-t_b-t_c)\vec{A}+t_b\vec{B}+t_c\vec{C}\ ,$$

kterou lze upravit na

$$\left[-\vec{d}, \vec{B}-\vec{A}, \vec{C}-\vec{A}\right]\begin{bmatrix} k \\ t_B \\ t_C \end{bmatrix}=\vec{O}-\vec{A}\ .$$

```
bool intersectTriangle (Ray ray, Triangle tri, float2 &coord, float &dist)
{
      // find vectors for two edges sharing node a
      float3 edge1 = tri.b – tri.a;
      float3 edge2 = tri.c – tri.a;

      // begin calculating determinant
      // also used to calculate U (coord.x) parameter
      float3 pvec = cross (ray.direction, edge2);

      // if determinant is near zero, ray lies in plane of tri
      float det = dot (edge1, pvec);
      if (det < Epsilon)
            return false;
      // calculate distance from node a to ray origin
      float3 tvec = ray.origin – tri.a;

      // calculate U parameter and test bounds
      coord.x = dot (tvec, pvec);
      if (coord.x < 0.0 || coord.x > det)
      return false;

      // prepare to test V (coord.y) parameter
      float3 qvec = cross (tvec, edge1);

      // calculate V parameter and test bounds
      coord.y = dot (ray.direction, qvec);
      if (coord.y < 0.0 || coord.x + coord.y > det)
      return false;

      // calculate distance, scale parameters, ray intersects tri
      dist = dot (edge2, qvec);
      if (dist < Epsilon)
            return false;

      coord /= det;
      dist /= det;

      return true;
}
```

Kód 5.1: Průsečík trojúhelníku s paprskem

5.4.3 Pozorování

Po implementaci průsečíku s trojúhelníkem jsem provedl několik výkonnostních testů s poměrně nízkým množstvím trojúhelníků (tabulka 5.1). Díky nízkému počtu trojúhelníku bylo možné snadno omezit vliv latence přístupu k paměti na výpočet. To, že test není ovlivněn rychlostí přístupů do paměti jsem ověřil tak, že jsem provedl více testů s různě taktovanými paměťmi a procesorem (GPU). Naměřený výkon téměř lineárně závisel na taktu procesoru. Oproti tomu takt paměti neměl na výkon prakticky žádný vliv. Je možné, že snížení taktu procesoru negativně ovlivnilo výkon vyrovnávací paměti, ale nepředpokládám, že by to mohlo mít tak výrazný dopad.

	cubes	5	10	15	20	25	30
	triangles	60	120	180	240	300	360
Full rate	fps	28,1	13,3	8,7	6,5	5,2	4,3
	isect * 10^6/sec	442	418	411	409	409	406
50 % GPU	fps	14,2	6,8	4,4	3,3	2,6	2,2
	isect * 10^6/sec	223	214	208	208	204	208
	ratio	51%	51%	51%	51%	50%	51%
50 % VRAM	fps	26,9	13,1	8,7	6,5	5,2	4,3
	isect * 10^6/sec	423	412	411	409	409	406
	ratio	96%	98%	100%	100%	100%	100%

Tabulka 5.1: Průsečík paprsku trojúhelníkem

Je třeba mít na paměti, že rychlost výpočtu průsečíku paprsku typicky není pro výkon raytracingu stěžejní. I v nepříliš optimalizované podobě představuje v závislosti na scéně většinou 1/6 – 1/5 času stráveného procházením akceleračních struktur (Uniform Grid, kd-tree a další, více v data scény [11].

5.5 Konstrukce kd-stromu

Konstrukci kd-stromu je možné řešit více způsoby, které se liší především ve způsobu určení dělící roviny uzlu. Nejběžnější postup vypadá takto:

- Při sestupu stromem cyklicky vybíráme osy pro určení dělících rovin. Pokud je tedy kořenový uzel rozdělen rovinou kolmou k ose *x* tak jeho potomci budou rozděleni rovinou kolmou k ose *y* a jejich potomci kolmo k ose *z* (pro trojrozměrný kd-strom).
- V každém děleném uzlu určíme medián v něm obsažených bodů a umístíme do něj dělící rovinu.
- Pokud počet uzlů poklesne požadovanou mez, nebo je dosaženo maximální hloubky stromu ukončíme dělení uzlů, vložíme list s daty a pokračujeme další větví.

Tímto způsobem získáme vyvážený kd-strom, ve kterém mají všechny listy stejnou vzdálenost ke kořenu. Použití mediánu však není nutností, je také možné jeho výpočet (lineární časové složitosti) aproximovat tak, že náhodně vybereme určitý (pevný) počet bodů a medián určíme z nich [12]. Vyvážené kd-stromy však nejsou nutně nejlepší volbou pro všechny aplikace a k dělení pomocí mediánu existuje řada alternativ. Raytracing kd-strom využívá typicky jinak než například prostorové databáze a jako výhodné se tak ukázaly například stromy s co největšími prázdnými uzly, tedy prakticky naprostý opak stromů vyvážených. Dobré výsledky přináší také dělení založené na cenovém modelu snažícím se vyvážit nikoliv počet uzlů ale cenu (složitost) hledání průsečíků paprsků v uzlech. Tento postup vychází z toho, že pravděpodobnost protnutí uzlu odpovídajícího kvádru paprskem je úměrná objemu tohoto kvádru [4]. Uzly s malým objemem tedy mohou obsahovat větší množství těles i podřízených uzlů (a tedy i více úrovní), protože nejsou tak často procházeny. Tomuto odpovídající i již zmíněné kd-stromy s velkými prázdnými uzly.

Důležitým prvkem konstrukce kd-stromu je určení ukončovacího kritéria. Nejčastěji bývá za-

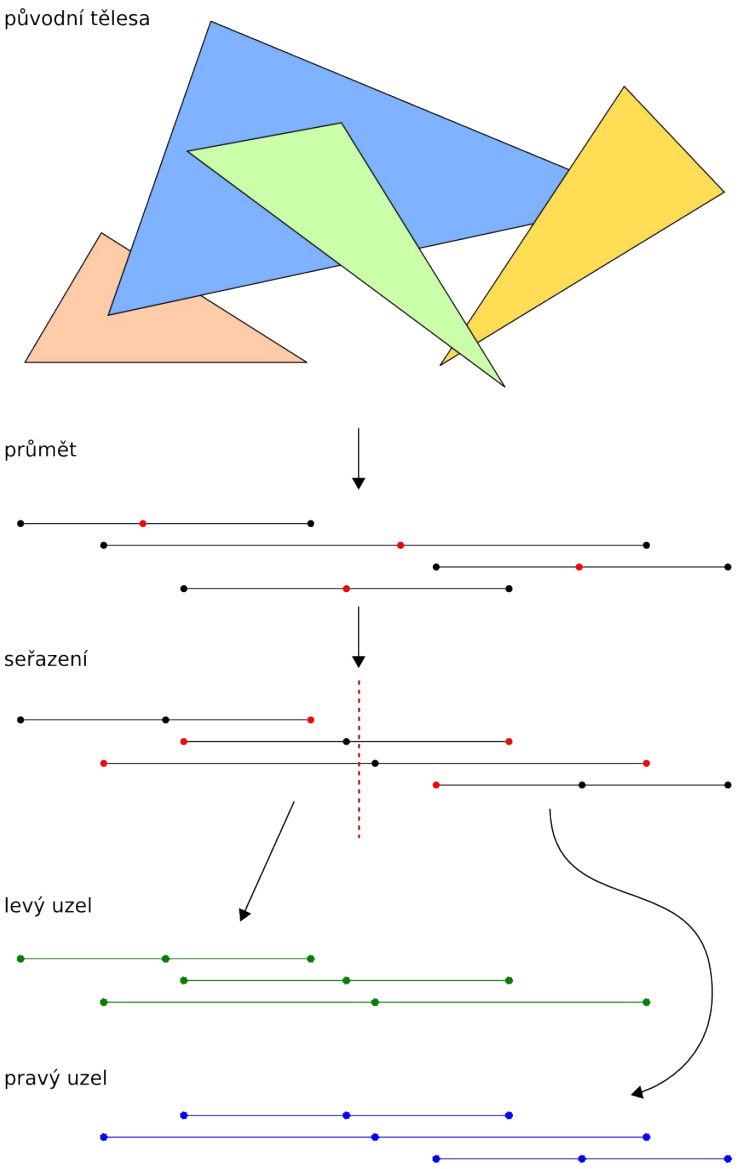
loženo na maximálním počtu prvků v uzlu (listu) N_{max} a maximální hloubce stromu d_{max}. Pokud poklesne počet objektů pod N_{max} nebo má uzel větší vzdálenost od kořene než d_{max} je uzel přeměněn na list. Nastavení těchto konstant je otázkou nalezení minima funkce složené ze složitosti hledání průsečíků těles a složitosti procházení kd-stromu. Na výkon obou procedur má vliv značné množství faktorů – druh, velikost či orientace scény, architektura raytraceru či stroje na kterém program běží a tak není velkým překvapením že v podstatě neexistuje metoda exaktně řešící tento problém. Využívané hodnoty proto bývají založeny (jen) na zkušenostech autorů a testech „typických" scén (například známy ray shooting engine Mental Ray ve výchozím nastavení využívá $N_{max}=4$ a $d_{max}=24$ bez ohledu na počet objektů ve scéně) [4]. Jisté zlepšení nabízí konstrukce ukončovacího kritéria na bázi cenového modelu zmíněné výše.

5.5.1 Postup

Konstrukce kd-stromu není těžištěm mé práce a nesnažil jsem se její rychlost nijak optimalizovat. Jedná se o jednorázový úkon a tak nízký výkon není při využití statické scény na překážku. Pro dynamické scény je kd-strom nevhodný (je obtížně / náročně aktualizovatelný) a optimalizace na tom nemůže mnoho změnit. Prostor pro zlepšení mé práce tu je značný, dobrým startem by mohl být například algoritmus popsaný v příloze práce V. Havrana [4].

Využívám dělení na základě mediánu a pokusně jsem implementoval i jednoduchý cenový model. V celé práci pracuji pouze s trojúhelníky a pro určení mediánu a rozdělení objektů (tedy trojúhelníků) do uzlů postupuji takto:

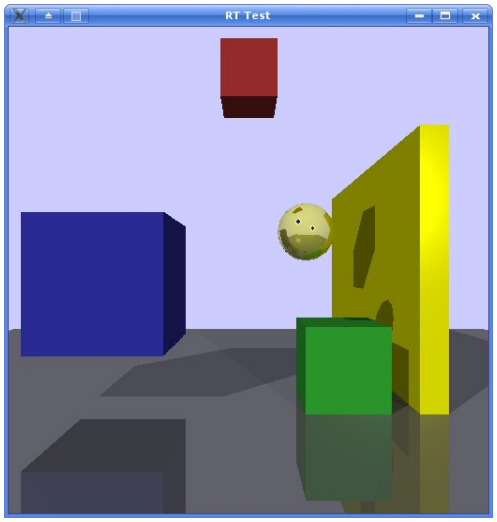
- Každému trojúhelníku přiřadím hodnotu v_n rovnou průměru souřadnic odpovídající ose uzlu (pro uzel dělený kolmo k ose *x* to tedy bude x-ová souřadnice).
- Seřadím trojúhelníky podle v_n.
- Určím hodnotu mediánu a umístím dělící rovinu mezi krajní body dvou k němu nejbližších trojúhelníků.
- Vyhodnotím ukončovací kritérium (počet uzlů a hloubku stromu)
- Vytvořím z trojúhelníků dvě skupiny (částečně se překrývající). Trojúhelník přísluší do skupiny tehdy, pokud odpovídající souřadnic některého jeho vrcholu leží před (za) dělící rovinou
- Pokračuji prvním bodem pro obě vytvořené skupiny a následující osu



Obrázek 5.4: Dělení uzlu při konstrukci kd-stromu

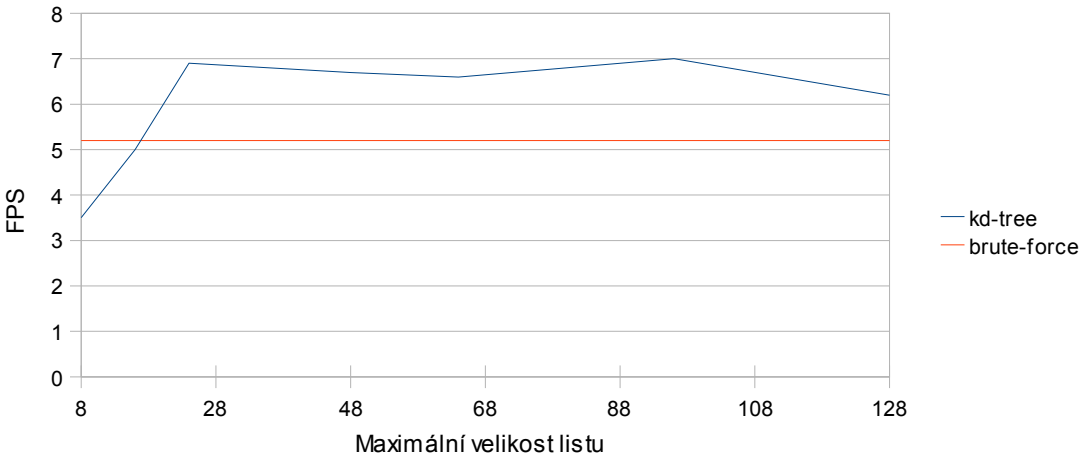
5.5.2 Pozorování

U testovaných scén docházelo poměrně často k tomu, že obsah dvou sousedních uzlů byl velmi podobný (podobně jako na obr. 5.4), protože velká část trojúhelníků zasahovala do více uzlů. Podezříval jsem tento fakt z významné degradace výkonu a tak jsem rozšířil ukončovací kritérium o kontrolu podobnosti uzlů. Výkon se tím ale zlepšil v nejlepším případě zhruba jen o 5% a to v případě relativně vysokých kd-stromů, které se celkově ukázaly jako méně efektivní – cena procházení struktury kd-stromu je vzhledem k architektuře GPU (pomalý náhodný přístup do paměti) vyšší než u CPU implementací. Oproti tomu zásadní vliv na výkon má nastavení výše zmíněných konstant N_{max} a d_{max}.



Obrázek 5.5.: Testovací scéna

Běžně se využívají velmi nízké počty těles v listech, často i pouze jedno těleso na list[8]. Proto mě překvapily testy které jsem provedl s mou GPU implementací, která nejlepších výsledků dosahovala při malých hloubkách stromu a počtu trojúhelníků v listech mezi 30 a 60. Těžko rozhodnout zda je výpočet průsečíku tak rychlý nebo průchod stromem pomalý. Výpočet průsečíku je typ úlohy který GPU svědčí:



Graf 5.1: Vliv maximální velikosti listů na výkon

- Výrazně u něj převyšují výpočty nad přístupy do paměti (také díky využitému algoritmu s minimálními nároky na velikost vstupních dat [7]).
- Pracuje s trojúhelníky, které jsou uloženy blízko vedle sebe – to je vhodné při využití texturovací jednotky pro přístup k datům.

8 Čtyři trojúhelníky v listu stromu u MentalRay jsou pravděpodobně důsledkem optimalizace pro využití SIMD instrukcí.

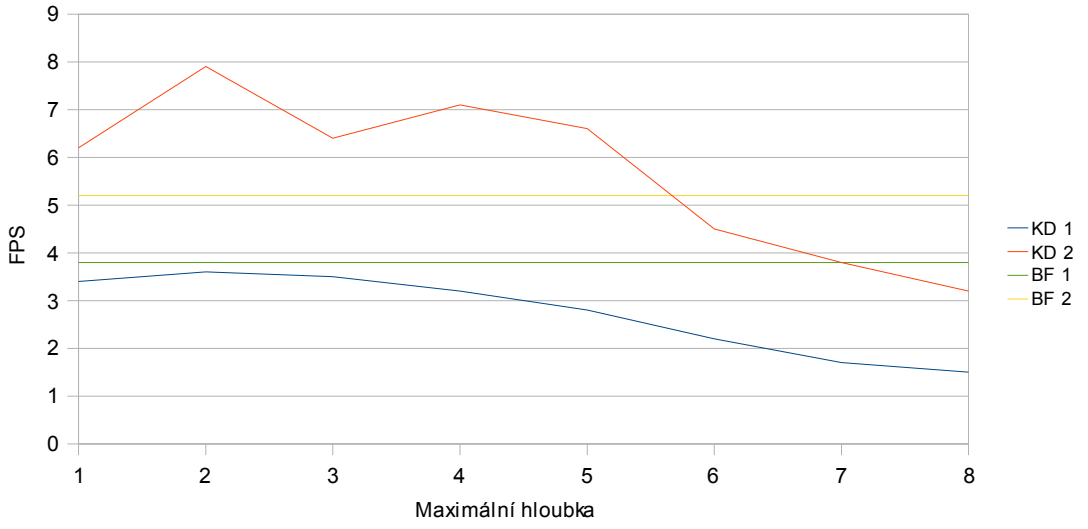
- CUDA implicitně vektorizuje aritmetické výpočty

Oproti tomu výkon průchodu stromem může být degradován z následujících důvodů:

- Adresování paměti na základě výpočtů programu – pro GPU se jedná o prakticky náhodný přístup, který má na výkon špatný vliv. Situaci částečně vylepšuje přístup přes texturovací jednotku a její vyrovnávací paměť.
- Dochází ke značným rozdílům v běhu vláken – pokud například několik vláken dorazí při procházení stromu k listu a nenajde průsečík, zastaví se dokud nedoběhne byť i jediné zbylé vlákno.
- Vzhledem k možnostem GPU je nutné využít jiný, méně efektivní, algoritmus než u CPU implementace.

Výpočet průsečíku bezpochyby rychlý je, přikláním se však spíše k druhé variantě. Nenalezl jsem ale způsob jak to experimentálně ověřit.

Zajímavým se ukázal vliv rozdílu ve velikosti trojúhelníku na rychlost průchodu stromem. Dvě testované scény o zhruba 700 trojúhelnících se lišily pouze (ne) přítomností plochy složené ze dvou trojúhelníků výrazně nadprůměrných rozměrů. Na grafu 5.2 je znázorněn výkon raytraceru pro tyto dvě scény v závislosti na maximální hloubce stromu. *KD* značí využití kd-stromu a *BF* „brute force“ přístup (bez akcelerační struktury). Rozdíl při odebrání zmíněné plochy je velký i při brute-force přístupu – je to dáno značným úbytkem sekundárních paprsků[9], ztráta výkonu není ale zdaleka tak významná jako u kd-stromu. Malý nárůst (či dokonce úbytek) výkonu při využití kd-stromu je dán především nízký počtem trojúhelníků ve scéně. Blíže se tomu věnuji ke konci následující sekce.5.1

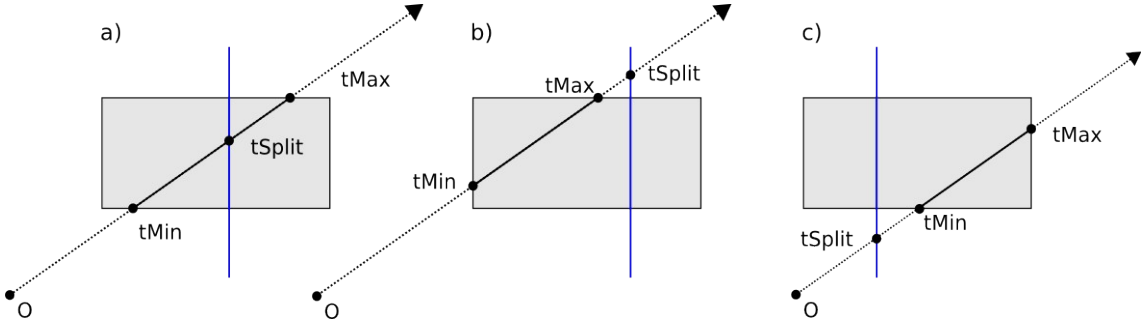


Graf 5.2: Vliv maximální hloubky stromu na výkon

9 Nárůst výkonu je o to výraznější, že že zpracování sekundárních paprsků je z hlediska GPU obtížnější než zpracování paprsků primárních. Primární paprsky jsou si navzájem „podobnější“, běh vláken se tedy méně liší i přístup do paměti má lokálnější charakter.

5.6 Procházení kd-stromu

Díky vlastnostem kd-stromu je jeho procházení poměrně jednoduché i efektivní. Pro určení průsečíku s dělící rovinou je využito pouze souřadnic odpovídajích ose uzlu. Možnosti, které mohou nastat při průchodu buňky jsou znázorněny na obr. 5.6. Proměnné *tMin*, *tMax* a *tSplit* značí vzdáleností bodu vstupu paprsku do buňky, výstup z buňky a poloho průsečíku paprsku s dělící rovinou od jeho počátku. Nejsou tedy závislé na ose uzlu. V případě a) je nutné dále zkontrolovat oba podřízené uzly a v případě b) a c) je nutné zkontrolovat pouze levou respektive pravou buňku. Standardní algoritmus pro průchod kd-stromem v nerekurzivní podobě publikované například v [11] je na 5.2.



Obrázek 5.6.: Průchod uzlem kd-stromu

Pro využití na GPU ale není tento algoritmus vhodný – pro zásobník dostatečné velikosti je třeba značné množství paměti a dá se tak očekávat značný vliv na výkon – využití registrů nepřipadá v úvahu a ukládání dat do paměti paměti se neobejde bez značné degradace výkonu.

```
stack.push(root, sceneMin, sceneMax)
tHit=infinity
while not stack.empty():
      node, tMin, tMax = stack.pop()

      while not node.isLeaf():
            a = node.axis
            tSplit = (node.value – ray.origin[a]) / ray.direction[a]
            first, sec = order (ray.direction[a], node.left, node.right)

      if tSplit ≥ tMax or tSplit < 0:
            node = first
      else if tSplit ≤ tMin:
            node = second
      else:
            stack.push(sec, tSplit, tMax)
            node = first
            tMax = tSplit

      for tri in node.triangles ():
            tHit = min(tHit, tri. Intersect (ray))

return tHit
```

Kód 5.2: Standardní průchod kd-stromem

5.6.1 Kd-restart

Jedním z možných řešení zmíněného problému je algoritmus *kd-restart* [3], který v podstatě ze standardního postupu pouze vypouští zásobník. Dokud není třeba projít oba dva podřízené uzly nic se nemění. Pokud tato potřeba nastane, tak algoritmus neuloží druhý uzel na zásobník pro následné zpracování, ale pouze posune *tMin* do průsečíku a po skončení kontroly prvního (levého) uzlu začne znovu od kořene. Posun *tMin* způsobí že je při dalším průchodu zkontrolován následující uzel. Složitost tohoto algoritmu je sice $O(n\log(n))$ oproti $O(\log(n))$ standardního provedení, ale při praktickém nasazení jsou hloubky stromů typicky malé a zvýšená složitost by se tedy neměla projevit. Naopak by měla být více než vynahrazena sníženou potřebou paměti a přístupu k ní. Oba algoritmy jsem srovnal v CPU verzi raytraceru na stromech s maximální hloubce dvaceti a rozdíly mezi oběma verzemi byly stěží měřitelné. Na GPU jsem standardní verzi algoritmu neimplementoval.

```
# stack.push(root, sceneMin, sceneMax) # removed
tMin = tMax = sceneMin
tHit = infinity
while tMax < sceneMax:
      node = root
      tMin = tMax
      tMax = sceneMax

      while not node.isLeaf():
            a = node.axis
            tSplit = (node.value – ray.origin[a]) / ray.direction[a]
            first, sec = order (ray.direction[a], node.left, node.right)

            if tSplit ≥ tMax or tSplit < 0:
                  node = first
            else if tSplit ≤ tMin:
                  node = second
            else:
                  # stack.push (sec, tSplit, tMax) # removed
                  node = first
            tMax = tSplit

      for tri in node.triangles():
            tHit = min(tHit, tri. Intersect (ray))

return tHit
```

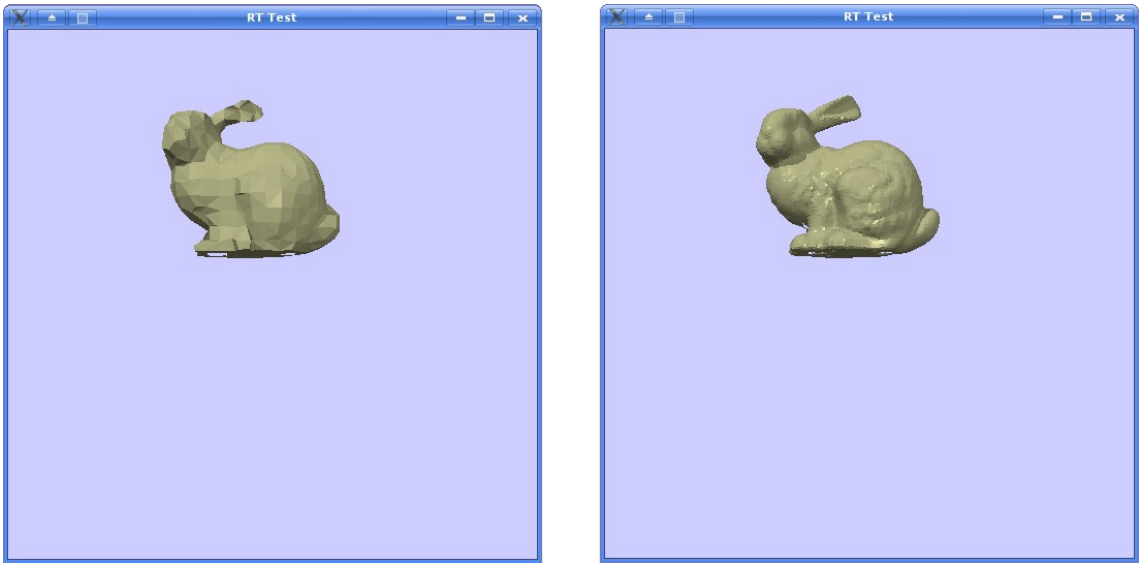
Kód 5.3: kd-restart algoritmus přidaný kód zeleně, změněný nyčerveně

Další možností je o rok mladší *kd-shortstack* [5] kombinující dva předchozí přístupy. Využívá zásobník s malou, omezenou délkou. Po vyprázdnění zásobníku postupuje stejně jako *kd-restart*. Autoři uvádějí v průměru dvojnásobný nárůst výkonu oproti využití *kd-restart*.

Nejnovějším příspěvkem k využití kd-stromů na GPU je bezzásobníkový průchod kd-stromu s pomocí „lan“ (ropes) uvedený [8]. Lana propojují uzly stejné úrovně a za cenu více než trojnásobného zvýšení paměťové náročnosti by měla urychlit procházen stromu. Autoři však neuvádějí srovnání výsledného výkonu raytraceru, ale pouze počet kroků které jsou průměrně vykonány při průchodu stromem. To však nemusí být rozhodující – při využití lan je zřejmě třeba výrazně více přístupů do paměti což by v nejhorším případě mohlo i převážit nárůst výkonu daný nižší

počtem kroků.

Implementace *kd-shortstack* jsem pouze zahájil (zásobník s omezenou velikostí jsem založil na principu kruhové fronty) a k průchodu pomocí lan jsem se bohužel nedostal vůbec.

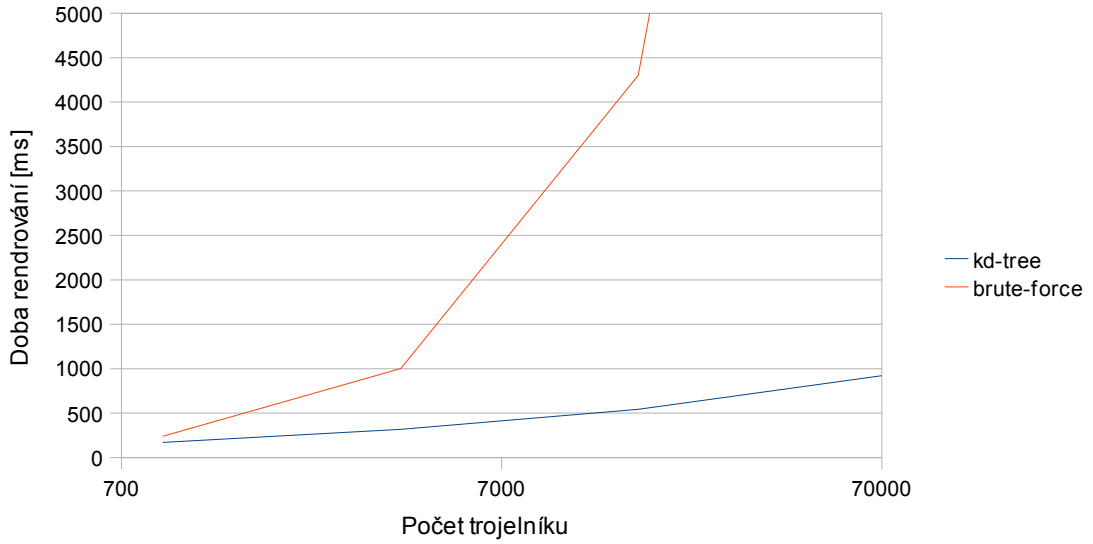


Obrázek 5.7.: Stanford bunny o 948 a 16301 trojúhelnících

5.6.2 Pozorování

Přidání kd-stromu nezlepšilo výkon raytraceru tak, jak jsem očekával. Velký vliv na to má velikost scény – testované scény byly spíše malé (několik set trojúhelníků) a tak se přínos akcelerační struktury neměl šanci dostatečně projevit. Velký vliv na výkon mělo nastavení ukončovacího kritéria při konstrukci stromu (více v sekci 7.6.2). Malý nárůst výkonu při využití kd-stromu viditelný na grafu 5.2 je dán hlavně nízkým počtem trojúhelníku (či obecně těles) ve scéně. Není jich jednoduše dost aby se mohl projevit přínos akcelerační struktury a tak nad nárůstem výkonu převažuje režie způsobená jejím využitím. Testy provedené na několika variantách modulu populárního Stanfordského králíka (obr. 5.7) to potvrdily[10]. Na grafu 5.3 je jasně vidět že přínos akcelerační struktury se projeví především při větších počtech těles. Při 70000 trojúhelnících byl již nárůst výkonu více než dvacetinásobný.

10 Nejedná se o žádné velké překvapení – je to typický příklad algoritmů s lineární a logaritmickou časovou složitostí.



Graf 5.3: Vliv počtu trojúhelníků na výkon

5.7 Reprezentace dat na GPU

Na grafických kartách s podporu CUDA je k dispozici několik druhů paměti. Zpočátku jsem pro uložení pro uložení všech dat využíval takzvanou *konstantní paměť*. Přístup k ní je veden přes vyrovnávací paměť a je velice rychlý – v ideálním případě jako přístup k registrům. Konstantní paměť je však nepříjemně malá – pouhých 64 KiB. Proto bylo třeba alespoň data trojúhelníků a kd-stromu přesunout do *globální paměti*. Ta při dodržení daných přístupových vzorů může dosahovat skoro stejné rychlosti. K datům při raytracingu je ale (obzvlášť při zpracování sekundárních paprsků) přistupováno spíše náhodně. Pro přístup k náhodně rozloženým datům je doporučováno využití texturovací jednotky v kombinaci s *CUDA poli*. CUDA pole jsou ekvivalentem textur z grafických rozhraní – data jsou uložena blíže nespecifikovaným způsobem (zarovnaným a rozloženým dle potřeb GPU) a jsou přístupná pouze prostřednictvím texturovacích funkcí. Přístup pomocí textur by měl být výrazně rychlejší – využívá mimo jiné vlastní vyrovnávací paměť – přináší ale značnou nevýhodu: nutnost využívat pouze vestavěné primitivní datové typy (*float, int, char*) a z nich složené 2 a 4 rozměrné vektory (tj. *float2*, *float4*, …).

5.7.1 Využití vestavěných datových typů

Datové typy bylo tedy třeba upravit tak, aby velikostí co nejvíce odpovídaly vestavěny vektorům a tak e předešlo zbytečným přístupům do paměti. Využívané datové struktury obsahují jak celočíselné tak desetinné členy. Celočíselné hodnoty vesměs slouží jako indexy pro přístup k dalším datům, takže by je bylo možné reprezentovat desetinnými čísly[11], problematickým se ukázalo sloučení více stavových proměnných uzlu kd-stromu do jednoho čísla pomocí bi-

11 Při využití texturovací jednotky by využití indexování pomocí desetinných čísel nemělo být doprovázeno ztrátou výkonu.

```
union KdNodeD
{
	int4 tex;
	struct {
		int flags;
		int left;
		int right;
		float splitPos;
	} inner;
	struct {
		int flags;
		int first;
		int last;
	} leaf;
}
```

Kód 5.4: Reprezentace uzlu kd-stromu pro GPU

tových operátorů. Takto vzniklé hodnoty typu *float* nebyl často validní a GPU je opravoval. Bylo tedy třeba využít celočíselný vektor, do něj uložit čísla v desetinné reprezentaci a následně je dle potřeby přetypovat[12]. Přetypování pomocí *void* ukazatelů nefungovalo na GPU stoprocentně, plně funkční se však ukázal být typ *union*. Strukturu pro uložení uzlů kd-stromu jsem takto reprezentoval celou, pro reprezentaci trojúhelníků jsem pouze přetypoval jeden člen – pro uložení trojúhelníků je třeba více vektorů a zbytečně jsem se nezaplétal do potencionálních problémů se zarovnáváním[13].

Při hledání průsečíku jsou využívány pouze vrcholy trojúhelníku a proto jsem data trojúhelníku rozdělil na dvě nezávislé části – jednu struktury s vrcholy a druhou strukturu s daty využívanými při výpočtu osvětlení. Počet přístupů k trojúhelníku při procházení scény daleko vyšší než počet přístupů při výpočtu osvětlení, uvedené rozdělení by tedy mělo jednoznačně vést k lepšímu výkonu – ať již díky snížení přístupů do paměti či zvýšení lokality dat a sní souvisejícím lepším využitím vyrovnávací paměti. Nepovažoval jsem proto za nutné toto experimentálně ověřovat.

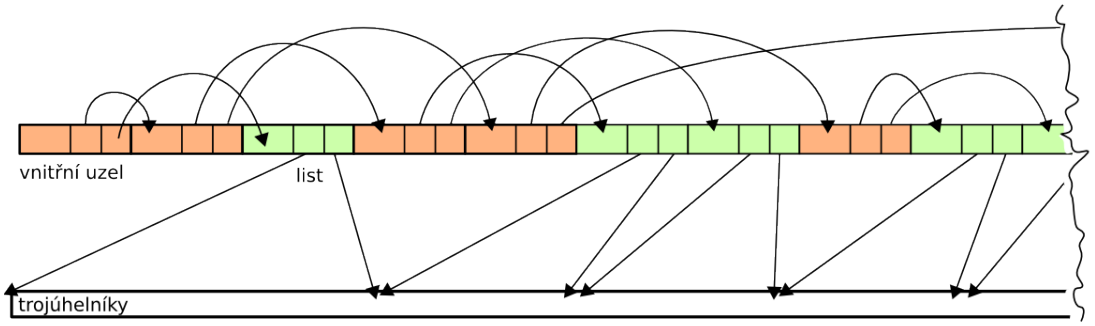
5.7.2 Reprezentace kd-stromu

Prostředí CUDA sice umožňuje využití ukazatelů, při využití *CUDA polí* a přístupu k nim pomocí texturovací jednotky to však přirozeně možné není[14]. Reprezentaci stromu pomocí dynamicky alokovaných struktur (využitou při jeho konstrukci) není možné na GPU provést a ani by to nebylo vhodné – z hlediska lokality paměti při procházení scény. Proto bylo třeba uzly uložit v polích a ukazatele nahradit indexy. Možností reprezentace binárního stromu polem prvků je řada, několik z nich je diskutováno v [11] z hlediska lokality paměti, vlivu vyrovnávací paměti a využití vektorových (SIMD) instrukcí. Pro GPU implementaci jsou toto uspořádání méně pod-

12 Reprezentace desetinných čísel na GPU NVIDIA není údajně plně IEEE kompatibilní a tak jsem pokusy zabezpečit validnost reprezentace desetinných čísel rychle vzdal.

13 Do těch jsem se přes explicitní využívání zarovnávacích atributů dostal v jiné části práce a nepodařilo se mi ji uspokojivě vyřešit, ale pouze obejít.

14 Jak jsem již dříve zmínil přístup k CUDA polím je možný pouze pomocí texturovací jednotky a jejich interní struktura není nikterak blíže specifikována.



Obrázek 5.8: Reprezentace kd-stromu na GPU

statné – vzhledem k vysoce paralelní povaze GPU a množství zpracovávaných paprsků je zvýšení lokality dat a efektivní využití (značně malé) vyrovnávací paměti při průchodu stromu prakticky nemožné či přinejmenším velmi obtížné. Vzhledem k tomu že jsem žádné testy vlivu různého uspořádání dat stromu neprovedl, nemohu potencionální příznivý vliv vyloučit. Předpokládám, ale že možnosti jsou zde daleko menší než ty které skýtá konstrukce a samotné procházení stromu (proto jsem se tomuto aspektu také více nevěnoval). Můj postup využívá levého preorder průchodu stromem a uložení uzlů je znázorněno na obr. 5.8.

5.7.3 Pozorování

Ačkoliv je CUDA popisována jako rozšíření C jedná se opravdu spíše o poměrně velkou podmnožinu C++. Příjemnou (a nedokumentovanou) vlastností je možnost využití členských funkcí u uživatelských typů i v kódu vykonávaném na GPU[15]. V kombinaci se slušnou podporou šablon se jedná o šikovný nástroj pro zvýšení čitelnosti a znovupoužitelnosti kódu. Vliv využití členských funkcích na výkon jsem otestoval na několika příkladech a nezaznamenal jsem měřitelný rozdíl.

5.8 Výkon

Všechna měření jsem prováděl na spíše slabé grafické kartě GF 86000M GS – bohužel se mi nepodařilo práci dokončit v dostatečném předstihu, tak abych mohl provést měření na výkonnějším stroji na fakultě. Rozdíl ve výkonu obou karet je propastný. Proto mají naměřené hodnoty význam především pro srovnání různých aspektů využitých algoritmů, které by se ani na podstatně výkonnější grafické kartě neměly příliš. měnit.

Provedl jsem také zcela informativní srovnání výkonu s CPU. K tomu je třeba přistupovat opatrněji. Stejně jako není naměřený výkon reprezentativní ukázkou možností grafické karty, není moje CPU implementace raytracingu ani zdaleka ukázkou možností CPU. Vhodné využití SIMD instrukcí vede k velikému růstu výkonu [11] a experimentální GPU raytracery se stále ještě nejlepším CPU implementacím nevyrovnaly.

15 Virtuální funkce se pochopitelně nekonají, současné GPU totiž neznají vůbec neznají pojem funkce a veškerý kód je inlinován. Využití virtuální funkcí v pro rychlost kritickém kódu stejně vhodné a proto nikterak neschází.

	GF 8800 Ultra	GF 8600M GS
multiprocesor count	128	16
processing power [Gigaflops]	576	57,6
bus width [bit]	384	128
max. bandwidth [GB/s]	103,7	12,8

Tabulka 5.2: Srovnání výkonu vyu~ité grafické karty

Při testech na modelech stanfordského králíka byl GPU raytracer zhruba 3-krát rychlejší při využití kd-stromu a 10-krát rychlejší bez něj. To odpovídá v sekci 7.6.2. zmíněnému poznatku, že kd-strom přináší na GPU menší urychlení než při běhu na CPU.

5.8.1 Vytížení GPU

Jak jsem již dříve zmínil, důležitým prvkem architektury (moderních) GPU je skrývání latence přístupu do paměti pomocí paralelní běhu velkého množství vláken. Tato funkce je kritická především pro aplikace s častým a/nebo náhodným přístupem do paměti. Příkladem takového algoritmu je třeba procházení kd-stromu, při kterém probíhá minimu výpočtu a přistupuje se množství nesouvislých míst v paměti. Výpočet průsečíku není sice úplně na druhé straně barikády, poměr přístupu do paměti a výpočtu je zde ale mnohem příznivější.

V daném okamžiku je aktivní pouze část vláken a pro uchování stavu spících vláken je využito registrů. Registrů je na na (moderních) GPU značný počet (u G8xxx konkrétně 8192 na multiprocesor či-li 16 procesorů). Raytracing je ve srovnání s jinými (vhodnějším) aplikacemi poměrně komplexní a tak počet registrů přestává stačit. Nikoliv tak jak bylo běžné u raných programovatelných GPU, že by se program pro nedostatek registrů nespustil, ale plánovač nemůže mít připraveno dostatečné množství spících vláken pro efektivní skrývání latence. Tím dochází k tomu že výpočetní výkon GPU není plně využit. Pro tento jev se využívá termín *registry pressure*.

V mém případě je maximální počet využitých registrů 42 a tomu odpovídá vytížení procesoru 25%[16]. Toto číslo je podstatné pro zmiňované, rychlostí paměti omezené, aplikace, pro běh výpočetně intenzivních vláken nemusí být vůbec určující. Procházení kd-stromu a v menší míře i výpočet průsečíku k paměti přistupují poměrně intenzivně a tak jsem se snažil počet registrů omezit. Nedobral jsem se ale prakticky žádného výsledku. Z toho usuzuji, že překladač využití registrů silně (a dobře) optimalizuje a manuální úpravy nemají význam. Pro dosažení lepšího vytížení (33%) by bylo třeba stáhnout využití registrů pod 32 a i zakomentování poměrně velkých částí kódu přineslo úsporu pouhých 2 až 3 registrů.

16 Ke zjištění počtu registrů složí direktiva překladače pro vytvoření cubin souborů – v podstatě symbolického assembleru doplněného. Pro výpočet vytížení se využívá „CUDA Occupancy Calculator“ – spreadsheat, který je součástí CUDA SDK.

6 Implementace

Zdrojový kód aplikace je rozdělen do několika modulů, které jsou kompilovány zvlášť a využity ve formě statických knihoven. Toto rozdělení přináší zrychlení překladu a při současném využití CUDA a C++ dochází k menšímu míchání kódu. Bylo tak kupříkladu možné snadno nahradit CUDA verzi raytraceru verzí pro CPU. Jednotlivé moduly jsou na sobě co nejméně závislé a neměl by tedy být problém využít modul pro práci se scénou samostatně.[17]

Kromě rozhraních poskytovaných CUDA jsem využil aplikačního frameworku Trolltech Qt. Ačkoliv je Qt vnímáni především jako grafický toolkit, já jsem zprvu využíval hlavně jejich (STL kompatibilní) kontejnery. Ty mají oproti standardním kontejnerům alespoň v mých očích několik výhod – především fakt, že jsou implicitně atomicky sdílené a při mnoha způsobech nasazení významně výkonnější. Také jsem ale od začátku počítal s tvorbou alespoň minimálního grafického rozhraní a volba Qt byla vzhledem k dostupnosti CUDA pro Linux i Windows a multiplatformní povaze Qt poměrně jasnou volbou.

Často jsem sdílel mezi CUDA a C++ kódem hlavičkové soubory. Sdílení hlaviček funguje bez problémů, jen je třeba důsledně dbát na to aby všechny funkce definované v hlavičkách byly inline. Pokud je totiž ta samá hlavička použita jak v C++ tak pro CUDA kód pro běh na hostiteli[18] dochází ke konfliktům symbolů při linkování. Možnost sdílení datových typů a rozhraní funkcí je deklarovanou (i faktickou) výhodou CUDA a využití speciálního typu hlaviček pr CUDA, které je možné vidět v několika příkladech v CUDA SDK proto podle mého názoru nemá opodstatnění. Stejně jako při použití čistého C++ zde platí výhody omezení počtu a rozsahu vkládaných hlaviček a deklarací v rozhraní vůbec. Zde je dobrým pomocníkem využití takzvaného PIMPL idiomu.

6.1 PIMPL

PIMPL či-li Private IMPLementation je technikou využívanou v C++ pro důkladné oddělení rozhraní a implementace, snížení počtu vkládaných hlaviček a tím (nejen) urychlení překladu a v případě knihoven usnadňuje kýžené zachovávání binární kompatibility i při velkých změnách vnitřní struktury.

Pro zachování zpětné kompatibility C++ s C byla učiněna řadu ústupků, mezi jeden hlavních patří zachování rozdělení rozhraní a implementace do různých typů souborů. To by samo o sobě nebyl takový problém, ačkoliv to přináší některé komplikace, starosti a práci navíc. Problematické je ale nutnost kompletní definice šablon v hlavičkách a nutnost kompletní znalosti struktur pro jejich uložení, tedy i deklaraci členských proměnných tříd a rozhraní funkcí tamtéž. Šablony

17 Výjimkou je využití vestavěných vektorů z CUDA po celém kódu pramenic především z mé pohodlnosti. Ty by mělo být snadné nahradit a nebo jednoduše tyto struktury dodefinovat.

18 Ku běžícího na GPU se to netýká – zde koncept funkce neexistuje a definování funkcí zde tedy je pouze jakýmsi „syntaktickým cukříčkem – veškerý kód je inlineován

```
// „klasicky“
#include „c.h“
#include „d.h“
class A {
...
private:
      C c;
      D d;
}

// PIMPL
// rozhraní
class B {
…
private:
      struct B_Private d*;
}

// implementace
#include „c.h“
#include „d.h“

struct B_Private {
      C c;
      D d;
}
```

Kód 6.1: Příklad vyu~ití PIMPL idiomu

nejsou bohužel řešitelné, částečným řešením druhého problému je PIMPL. Pro jeho využití je podstatný fakt, že pro deklarování ukazatelů, referencí a návratových hodnot funkcí úplná znalost typů nutná není. V tělu třídy je tedy pouze ukazatel na typ dodefinovaný až v souboru s implementací. Třída využívající PIMPL tedy typicky vypadá podobně jako na 6.1. Pro využití rozhraní třídy *B* není na rozdíl od třídy A třeba vkládat hlavičky „c*.h“* a „*d.h*.“

Tento přístup má několik nevýhod[19], ale při vhodném využití výhody zcela převažují. Výše zmíněná rychlost překladu či vliv na binární kompatibilitu nemají v mém případě prakticky žádnou roli. Ocenil jsem ale omezení vkládaných hlaviček – při překladu CUDA souborů (pro hostitele) jsem bez ohledu na využité direktivy měl potíže s vkládáním hlaviček některých využitých C++ kontejnerů. Problém jsem odstranil tím, že jsem je z rozhraní „skryl“ (byly třeba pouze z implementačních důvodů a v rozhraní umožnil pouze k C polím prvků převedených do podoby pro využití na GPU. Další výhodou je, že je možné definovat výchozí hodnoty všech atributů na jednom místě, což zmenší riziko zbytečných chyb při využití více konstruktorů.

Explicitní alokace a dealokace ukazatele s privátními daty je jednak otravná a především při využití více konstruktorů náchylná na opomenutí s ním spojené chyby. Proto využívám třídu typu „chytrého ukazatele“, která se stará o alokaci a dealokaci privátních dat a několika maker pro často využívané deklarace a definici s touto technikou spojené.

19 Obecně není vhodný pro objekty používané v „rychlém“ kódu – jak dynamická alokace paměti, tak přístup k datům přes odkaz navíc či nutnost přístupu zvenčí pouze pomocí členských funkcí (bez možností inlinování) mají jistou režii, která se při častém projeví. Při použití jen u „velkých“ objektů však tato režie nehraje prakticky žádnou roli.

6.2 Moduly

6.2.1 Práce se scénou – scene

Tento module sdružuje třídy pro vytváření scény. Zprvu zde také probíhal převod do formy vhodného pro GPU.

Scene

V současné podobě obsahuje již jen dvě třídy. Třída *Scene* je už v podstatě jenom kontejnerem pro data scény. Konverzi dat či vytváření kd-stromu jsem během vývoje přesunul do samostatných tříd, které jsou v případě potřeby explicitně volány. Třída *MaterialRef* zjednodušuje přístup a nastavování definic materiálů použitých ve scéně.

Builder

Builder je rozhraní pro vkládání dat do scény. Ta předpokládá plně inicializovaná data, *Builder* oproti vytváření data maximálně ulehčuje, umožňuje některé parametry vynechat ve prospěch přecházejících a implicitní hodnot či je dopočítává – například normály u zadávaných trojúhelníku. *Scene* umožňuje vkládat pouze trojúhelníky, na které *Builder* komplexnější tělesa rozkládá.

Rozhraní třídy je inspirováno OpenGL. Je tedy zachováváno poslední nastavení a znovu využíváno pro nově vkládané objekty. Podobně je možné před vkládáním objektů transformovat souřadnicový systém a stav transformace „světa“ ukládat na zásobník a obnovovat jej z něj.

Transform

Třída *Transform* slouží pro aplikování afinních transformací na body a vektory. Je využívána hlavně při konstrukci scény a například pro umístění a nastavení kamery. Transformace jsou implementovány pomocí třídy *Matrix*.

Matrix není jedna třída, ale několik šablon odvozených od společného typu. Velikost matice je staticky určena a taktéž staticky je kontrolováno, zda odpovídají rozměry při násobení a dalších operací s maticemi – odpovídající operátory jsou definovány pouze pro matice s kompatibilními rozměry. Operace s vektory (jak řádkovými tak sloupcovými) jsou optimalizovány pomocí částečné specializace. Celkem vzato je tato třída pro potřeby projektu zbytečně komplikovaná (či propracovaná?) propracovaná – její hodnota je především naučná

KdTree

Za vytváření kd-stromu odpovídá třída *KdTree* s minimalistickým rozhraním. Vytvořený strom je pouze pomocnou strukturou a pro další využití je určen vektor uzlů a vektor indexů odpovídajících trojúhelníků. Toto rozložení je dáno především dalším využitím na GPU ale i pro CPU řešení by bylo třeba vytvořit strukturu novou – vytvořený strom obsahuje množství údajů nutných

pouze pro jeho vytvoření a také je zmíněné rozdělení je výhodnější z výpočetního hlediska[11].

6.2.2 Vlastní sledování paprsku – `raytracer`

Modul obsahuje dvě implementace sledování paprsku *CudaRaytracer* a *CpuRaytracer*. Algoritmy využité v obou raytracerech jsou velmi podobné, nebylo však možné k významnějším způsobem sdílet – liší se především v detailech přístupu k datům a několika dalšími detaily. Těžištěm práce CUDA implementace. Pro přístup do paměti jsem vyzkoušel všechny možnosti které CUDA podporuje, tři z nich zůstaly pro srovnání ve zdrojovém ku zachovány: konstantní paměť, textury nad globální pamětí a textury nad CUDA poli[20]. Volba mezi způsobem přístupu do paměti je možná z výkonnostních důvodů (a pro omezení duplikování kódu) pouze prostřednictvím preprocesorových maker. Další makra slouží pro přístup k datům uvnitř výpočetních jader, která díky jsou díky tomu nezávislá na typu využívaného přístup k paměti.

Pro výstup dat v současné době využívám pouze pixel bufferu. U GPU raytraceru je to ostatně i přirozené. Třída *RaytracerOutput* zapouzdřuje operaci s výstupními daty. Její instance jsou vytvářeny daným raytracerem v souladu s jeho možnostmi. I když je využíváno pouze pixel bufferů je třeba dvou odlišných implementací – CUDA využívá pro práci odlišné rozhraní než čisté OpenGL využité pro výstup CPU implementace. Vzhledem k potřebě okamžitého zobrazování jsem další typy výstupu neimplementoval – jejich využití by bylo pouze zdrojem zatížení sběrnice a negativně by tak ovlivnil měření. V případě potřeby není problém data z pixel bufferu načíst zpět do hostiteli přístupné paměti.

20 Přímý přístup ke globální paměti se již v rané fázi projektu ukázal tak pomalým, že jsem ho dále nerozvíjel.

7 Závěr

Současné GPU jsou využitelné jako výkonné stream procesory a mají značný potenciál pro využití v negrafických aplikacích. Dlouho dobu však bylo obtížné tento potenciál plně využit. V poslední době uvolněná rozhraní (CUDA, AMD Stream SDK) tuto bariéru z větší části odstraňují a zpřístupňují tak GPGPU širšímu okruhu vývojářů. Je však třeba mít na paměti, že se nejedná o všelék, ale prostředek vhodný pro řešení poměrně omezeného spektra úloh.

Raytracing mezi tyto úlohy však spíše nepatří. Současné GPU raytracery sice dosahují srovnatelného výkonu jako optimalizované CPU řešení, implementují vesměs ale pouze velice základní funkcionalitu. Architektura současných GPU neumožňuje efektivní implementaci pokročilejší funkcionality a snahy o ní vedou k výrazné degradaci výkonu. Praktické nasazení GPU raytracerů tedy není bez větších změn architektury GPU v dohledné době pravděpodobné. Přinejmenším několik dalších let tedy zůstane co do kvality obrazu a rychlosti zapracování na grafických kartách nejlepším řešením rasterizace, tím spíše že Direct3D 10.1 obsahuje podporu globálního osvětlení. Nejvýkonnější platformou pro raytracing tak zůstává CELL procesor, ačkoliv ani pro něj nebyl ještě uveden plnohodnotný raytracer. Využití GPU by mohlo být ale výhodné pro urychlení některých částí CPU implementací. Ani takový produkt však zatím nebyl uveden.

Výše zmíněné problémy ale nepředstavují nic co bylo při rozsahu mé práce třeba řešit a tak lze prohlásit, že jsem při implementaci funkčně jednoduchého raytraceru dosáhl dílčích úspěchů. Patří mezi ně především fakt, že navzdory nízkému výkonu využité grafické karty dosahovala GPU verze výrazně lepších výsledku než obdobná CPU verze.

Možností rozšíření další práce je veliké množství. Největší potenciál je pravděpodobně ve vylepšení konstrukce kd-stromu. I jednoduché heuristiky založené na cenovém modelu dávají u klasických implementací výrazně lepší výsledky a lze předpokládat, že by tomu tak bylo i na GPU. Procházení kd-stromu bylo oproti CPU verzi výrazně pomalejší a jakékoliv jeho zlepšení by proto mělo výrazný vliv na výkon. Pro začátek by stačilo i dokončit implementaci *kd-shortstack* algoritmu, který by měl přinést přibližně dvojnásobné zrychlení. Možností pro experimenty je mnoho, protože jak se v průběhu práce ukázalo, mnoho ověřených heuristik a pravidel ze světa CPU raytracerů na GPU neplatí a proto zlepšení výkonu může přijít i z nečekaných směrů.

Literatura

[1] Carr, N. A., Hall, J. D., Hart, J. C.: The Ray Engine., 2002.

[2] NVIDIA CUDA Programming Guide 1.1. 2007.

[3] Foley, T., Sugerman, J.: KD-Tree Acceleration Structures for a GPU Raytracer. HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2005.

[4] Havran, V.:: Heuristic Ray Shooting Algorithms. [disertační práce] Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague. 2000.

[5] Horn, D., Sugerman, J., Houston, M., Hanrahan, P.: Interactive k-D Tree GPU Raytracing. I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games, 2006.

[6] Weisstein, E. W.: Barycentric Coordinates. URL http://mathworld.wolfram.com/BarycentricCoordinates.html.

[7] Moller, T., Trumbore, B.: Fast, Minimum Storage Ray/Triangle Intersection. Journal of graphics tools, 2, 1997, s. 21-28.

[8] Popov, S., Günther, J., Seidel, H., Slusallek, P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing
. Computer Graphics Forum, 2007.

[9] Purcell, T. J., Buck, I., Mark, W. R., Hanrahan, P.: Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics, 3, 2002, s. 703-712.

[10] Forsyth, T.: TomF's Tech Blog. URL http://home.comcast.net/~tom_forsyth/blog.wiki.html#[[Larrabee%20and%20raytracing]].

[11] Wald, I.:: Realtime Ray Tracing and Interactive Global Illumination. [disertační práce] Computer Graphics Group, Saarland University. 2004.

[12] Wikipedia - kd-tree. URL http://en.wikipedia.org/wiki/Kd-tree.

[13] Wikipedia - Ray tracing (graphics). URL http://en.wikipedia.org/wiki/Ray_tracing_(graphics).

[14] Zemčík, P.: Raytracing optimization. 2007.