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

PHOTON TRACING NA GPU

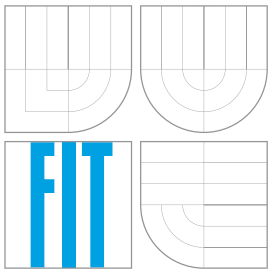
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ROMAN GALACZ

BRNO 2013

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

PHOTON TRACING NA GPU

PHOTON TRACING ON GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ROMAN GALACZ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUKÁŠ POLOK

BRNO 2013

Abstrakt

Tato práce se zabývá urychlováním metody photon mapping na grafické kartě. Jedná se o metodu výpočtu globálního osvětlení scény, jež hraničí s realismem. Samotný výpočet je časově poměrně náročný a jeho zrychlování je tedy žhavým tématem v oblasti počítačové grafiky. Photon mapping je podrobně popsán z pohledu sledování fotonů a následného vykreslování scény. Pozornost je následně věnována strukturám dělícím 3D prostor, především uniformní mřížce. V další části práce je popsán návrh a implementace aplikace provádějící výpočet photon mappingu na GPU, čehož je dosaženo spolupráci mezi OpenGL a CUDA. Aplikace je nakonec řádně otestovaná. Dosažené výsledky jsou zhodnoceny v závěru práce.

Abstract

Subject of this thesis is acceleration of the photon mapping method on a graphic card. The photon mapping is a method for computing almost realistic global illumination of the scene. The computation itself is relatively time-consuming, so the acceleration of it is a hot issue in the field of computer graphics. The photon mapping is described in detail from photon tracing to rendering of the scene. The thesis is then focused on spatial subdivision structures, especially to the uniform grid. The design and the implementation of the application computing the photon mapping on GPU, which is achieved by OpenGL and CUDA interoperability, is described in the next part of the thesis. Lastly, the application is tested properly. The achieved results are reviewed in the conclusion of the thesis.

Klíčová slova

architektura GPU, OpenGL, CUDA, globální osvětlení scény, dělení prostoru, uniformní mřížka na GPU, sledování paprsku na GPU, sledování fotonů na GPU, photon mapping na GPU, interoperabilita mezi OpenGL a CUDA

Keywords

GPU architecture, OpenGL, CUDA, global illumination, spatial subdivision, uniform grid on GPU, ray tracing on GPU, photon tracing on GPU, photon mapping on GPU, OpenGL CUDA interopearbility

Citace

Roman Galacz: Photon tracing na GPU, diplomová práce, Brno, FIT VUT v Brně, 2013

Photon tracing na GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Lukáše Poloka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Roman Galacz
21. května 2013

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce projektu panu Ing. Lukáši Polokovi za jeho cenné rady a ochotu při konzultacích. Dále bych chtěl poděkovat přítelkyni a rodině za podporu při psaní této práce. Největší dík však patří mým rodičům, kteří mi studium na vysoké škole umožnili.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Již několik desítek let se v počítačové grafice řeší problém globálního osvětlení scény. Toto osvětlení má za úkol přiblížit výstupní kvalitu obrázků vytvořených na počítači k fotorealismu. Využití je možné nalézt v mnoha odvětvích, jako je např. filmografie, kde v dnešní době vzniká nesčetné množství animovaných filmů a filmových triků renderovaných výhradně na PC. Dalším příkladem je herní průmysl, ve kterém je kladen důraz na co nejrealističtější grafiku od jeho samého počátku, nebo architektura s renderováním architektonických návrhů pro prezentaci zákazníkům. Takových příkladů využití je možné nalézt opravdu hodně.

Postupem času vznikaly různé metody a algoritmy, které globální osvětlení více či méně řeší. Univerzální algoritmus zvládající modelovat všechny optické jevy, však neexistuje. Pro co nejrealističtější zobrazení scény se naopak využívá kombinace technik aproximujících jednotlivé jevy. Jednou z nich je metoda *photon mapping*, které je věnována tato práce.

Velkou nevýhodou výše uvedených algoritmů a technik je jejich časová náročnost výpočtu pro komplexní scény, kvůli čemuž pracují tzv. *offline*. Avšak díky obecně dobré paralelizovatelnosti většiny těchto algoritmů je možné jejich výpočet značně urychlit převedením výpočtů na grafickou kartu. Programovatelné grafické akcelerátory jsou fenoménem dnešní doby a jejich dnes již masivní rozšíření je předurčuje k využití urychlování výpočtu tam, kde je to možné a vhodné. V této práci se tedy budeme zabývat možností skloubení rychlosti grafických karet a výpočtu globálního osvětlení pomocí photon mappingu.

Práce je rozdělena do několika kapitol a nejdříve se poměrně podrobně zabývá metodou photon mapping. V této kapitole je mimo jiné popsána metoda sledování paprsku, ze které photon mapping vychází. Dále je zmíněno několik způsobů urychlení photon mappingu a na konci kapitoly jsou krátce popsány některé další metody pro řešení globálního osvětlení. Třetí kapitola je věnována dělení prostoru scény, které umožňuje podstatně urychlit výpočet photon mappingu. Ve čtvrté kapitole je popsána hardwarová architektura moderních grafických karet. Tato kapitola dále poskytuje jemný úvod do problematiky programování grafických karet pomocí různých nástrojů a knihoven. V páté kapitole je rozebrána implementace photon mappingu na grafické kartě. V rámci implementace byla navržena poměrně hybridní metoda výpočtu, jež využívá možnosti propojení grafické knihovny OpenGL a architektury CUDA k dosažení co nejlepších výsledků při urychlování photon mappingu. Šestá kapitola popisuje testování vytvořené aplikace. V poslední kapitole jsou nakonec shrnuty dosažené výsledky a konzultovány možnosti dalšího vývoje vytvořené aplikace.

Kapitola 2

Photon mapping

Jedná se o jednu z metod efektivně řešící globální osvětlení 3D scény. Metodu poprvé prezentoval v roce 1996 Henrik Wann Jensen [1]. Pomocí photon mappingu lze simulovat nepřímé osvětlení, radiozitu, kaustiky, rozklad a rozptyl světla pod povrchem objektů nebo volumetrické materiály. Na rozdíl od jiných obecných Monte Carlo metod, jako je např. sledování cest (*path tracing*) nebo obousměrné sledování cest (*bidirectional path tracing*) [2], které zvládají simulovat stejné jevy, je výpočet mnohonásobně rychlejší.

Photon mapping je dvouprůchodový algoritmus. V prvním průchodu se do scény vystřelují fotony ze světelných zdrojů a sledují se, dokud nenarazí na plně difúzní povrch, který je pohltí anebo opustí scénu. Po nárazu fotonu na jakýkoliv difúzní povrch je tento foton uložen do fotonové mapy. Ve druhém průchodu, nebo-li při renderování scény, se využívá informace z fotonové mapy pro určení výsledné zářivosti v daném bodě. Pro renderování scény se obvykle využívá metoda sledování paprsku (*ray tracing*). Výhodou je taky oddělení fotonové mapy od zbytku geometrie scény, což umožňuje efektivní zpracování velmi komplexních scén [3].

Na rozdíl od již zmíněných metod sledování cest má photon mapping poměrně velkou spotřebu paměti právě díky nutnosti uložení mapy fotonů. Toto je ale vyváženo rychlostí samotného algoritmu pro většinu scén a také tím, že photon mapping produkuje šum na nízkých frekvencích, který je lidským okem méně zaznamenatelný, oproti šumu na vysokých frekvencích způsobovanému obecnými Monte Carlo metodami. Navíc výsledky photon mappingu mohou být zkresleny a nekonvergovat ke správnému řešení zobrazovací rovnice. Jelikož se jedná o konzistentní metodu, můžeme správných výsledků dosáhnout pomocí zvýšení počtu sledovaných fotonů ve scéně [3].

2.1 Sledování paprsku

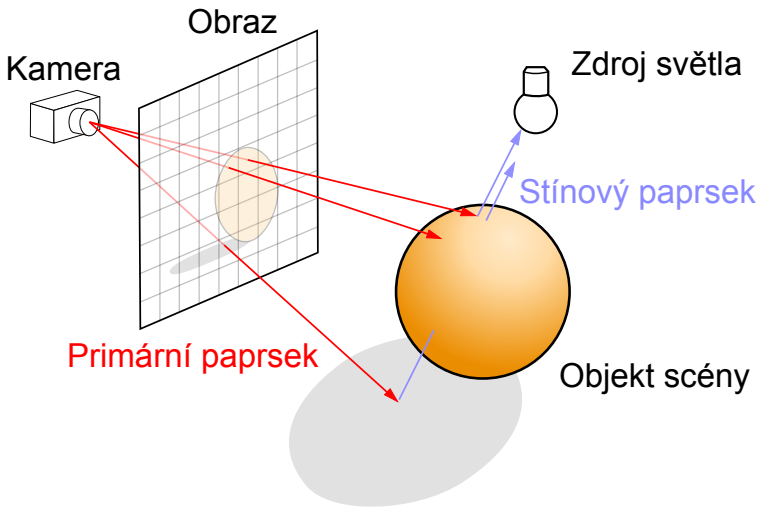
Sledování paprsku (*ray tracing*) [4] je pro tuto práci jednou z důležitých metod počítání globálního osvětlení, jelikož algoritmus photon mappingu je na sledování paprsku postaven a sledování paprsku se také často používá v druhé fázi metody photon mapping při vykreslování výsledných obrazů, proto si jej zde popíšeme.

Na rozdíl od sledování paprsku v reálném světě, kdy se paprsek šíří ze zdroje světla do oka pozorovatele (kamery), jsou paprsky vysílány z oka pozorovatele do scény a sledovány. Počet paprsků vycházejících ze světla je totiž nekonečný a není tak výpočetně možné sledovat, které paprsky dopadnou do oka pozorovatele.

Díky sledování paprsku lze simulovat odraz a lom světla na objektu s danými fyzikálními

vlastnostmi. Pokud budeme sledovat taky stínové paprsky, můžeme jednoduše zobrazit stíny objektů ve scéně. Stínový paprsek je vržen směrem ke světlu z místa, kde dopadl primární paprsek z kamery a pokud mu v cestě stojí nějaký objekt, je tento bod ve stínu. Princip renderování scény pomocí sledování paprsku je na obrázku 2.1. Algoritmicky vše funguje tak, že se z kamery vysílají přes všechny pixely vykreslovaného okna paprsky a zkoumá se jejich průsečík se scénou. Pokud paprsek narazí na objekt scény, tak je podle jeho vlastností buď zcela pohlcen (objekt je z plně difúzního materiálu) a sledování končí. Dále může být paprsek odražen (odrazivý materiál) nebo zalomen (průsvitný materiál) a pokračuje se v rekurzivním sledování nově vytvořeného paprsku (sekundární paprsek). Hloubka rekurze je dána požadavky na aplikaci a obecně platí, že pro zobrazení pouze difúzní scény pomocí primárních paprsků je potřeba hloubka 1, pro jednoduchý odraz je to hloubka 2 a pro lom světla minimálně hloubka 3 (paprsek na vstupu do objektu a paprsek při výstupu z objektu).

Největším problémem při sledování paprsku je jeho vysoký čas výpočtu pro složité scény, které se skládají z mnoha trojúhelníků. Je potřeba testovat každý paprsek se všemi trojúhelníky ve scéně. V následující kapitole si ukážeme, jak efektivně omezit počet kontrol na průsečík paprsku s trojúhelníky, a tím značně urychlit metodu sledování paprsku.



Obrázek 2.1: Renderování scény pomocí sledování paprsku (převzato z [5])

2.2 Photon tracing

Photon tracing je první fázi photon mappingu, kdy postupně vysíláme fotony ze zdroje světla do scény a sledujeme je. Fotony, které dopadly na difúzní povrch jsou uloženy do fotonových map. V této podkapitole ukážeme generování fotonů pro různé typy světla a použití projekčních map, sledování fotonů a použití ruské rulety při rozhodování o zániku fotonů. Nakonec zmíníme použití vhodné struktury pro uchování fotonů.

2.2.1 Generování fotonů

Fotony mohou být do scény vygenerovány buď jedním, nebo více světly. Popíšeme si jaké rozdíly v takovém generování jsou. Předtím se ještě zmíníme o některých tvarech světelných zdrojů.

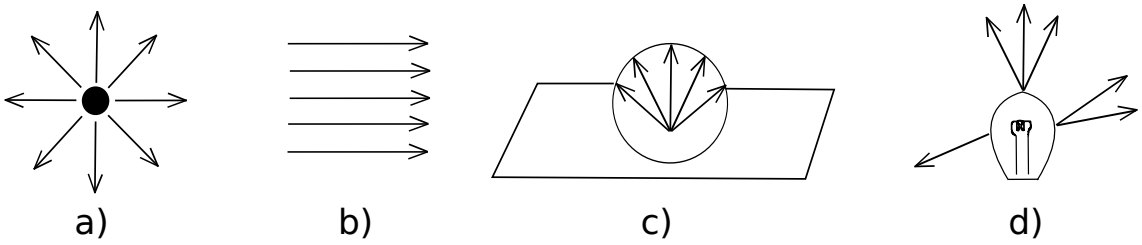
Jedno světlo ve scéně

Na obrázku 2.2 můžeme vidět různé tvary světelného zdroje, ze kterého budeme vysílat fotony. Prvním tvarem je bodové světlo, ze kterého se fotony generují náhodně s uniformním rozložením. Následuje směrové světlo, kde jsou fotony vrženy po směru světla. U plošného zdroje lze foton generovat náhodně z jakéhokoliv místa na ploše světla ve směru omezeném hemisférou. To lze zařídit pomocí kosínové distribuce [6], kdy největší pravděpodobnost vyslání fotonu je ve směru kolmém na světlo a nulová ve směru paralelním. Nakonec lze modelovat i případ obecného světla, kterým může být např. žárovka. Pokud známe počátek na povrchu světla a směr, ve kterém v tomto počátku může být foton vygenerován, pak pravděpodobnost vyslání fotonu právě z tohoto místa závisí jak na počátku, tak na směru [3].

Světlo má také svoji energii danou barevnými složkami RGB. Tuto energii je potřeba rovnoměrně rozložit mezi všechny vygenerované fotony. Fotony tedy ponesou zlomek energie světla. Na rovnici 2.1 můžeme vidět, jak tuto energii fotonu Φ_p vypočítat z energie světla Φ_s, pokud známe počet generovaných fotonů n.

$$\Phi_p = \frac{\Phi_s}{n} \tag{2.1}$$

Pokud je počet vytvořených fotonů velký, je výhodnější vysílat fotony s plnou energií, sledovat je ve scéně a teprve po ukončení sledování je podělit celkovou energií světla. Tím předejdeme zbytečným nepřesnostem při výpočtech, ke kterým by docházelo v případě velmi nízké počáteční energie fotonu.



Obrázek 2.2: Různé zdroje světla: a) bodové, b) směrové, c) plošné, d) obecné (převzato z [7])

Více světel ve scéně

Jestliže scéna obsahuje více světel, je potřeba vygenerovat fotony z každého světla. Počet fotonů generovaných daným světlem závisí na velikosti jeho energie, a proto je větší množství fotonů generováno ze silnějších světel, které více přispívají k celkovému osvětlení scény [3].

2.2.2 Projekční mapy

Jejich použití je velice vhodné pro optimalizaci generování fotonu v řídkých scénách, tedy ve scénách, kde je malý počet objektů a velké množství vygenerovaných fotonů by nezasáhlo žádný objekt. Projekční mapa je mapa geometrie z pohledu světla a je rozdělena na buňky, které jsou buď aktivní nebo neaktivní podle toho, zda se v buňce nachází nějaký objekt nebo ne. Projekční mapa nám dá odhad, ve kterých směrech je nutné fotony ze světla generovat [3].

Generování fotonů podle projekční mapy pak probíhá tak, že směr nově vytvořeného fotonu je konfrontován s odpovídající buňkou. Pokud se v této buňce žádný objekt nenachází, generuje se nový foton s jiným náhodným směrem. Dalším možným přístupem je náhodně vybírat pouze ty buňky, kde se nějaké objekty nacházejí a poté generovat fotony v náhodných směrech odpovídajícím těmto buňkám.

Při použití projekčních map je nutné upravit vzorec 2.1 pro počáteční energii fotonu. Upravený vzorec 2.2 respektuje počet buněk obsahujících nějaký objekt vůči celkovému počtu vygenerovaných fotonů [3].

$$\Phi_p = \frac{\Phi_s}{n} \frac{pocet\ ativnich\ bunek}{celkovy\ pocet\ bunek} \tag{2.2}$$

Další vhodné využití projekčních map je při sledování fotonů, které vytvářejí kaustiky. Je možné identifikovat odrazivé a průhledné objekty a v jejich směru pak generovat fotony, které se uloží odděleně od fotonů pro zbytek scény.

2.2.3 Sledování fotonů

Nyní, když máme vygenerovány fotony, můžeme je začít sledovat ve scéně. K tomu využijeme metodu sledování paprsku (*ray tracing*), kterou jsme si popsali výše. Oproti sledování paprsku se metoda liší v tom, že fotony nesou informaci o světelném toku. Pokud sledovaný foton narazí na objekt, může být zrcadlově odražen, zalomen, difúzně odražen nebo pohlcen, viz obrázek 2.3, kde:

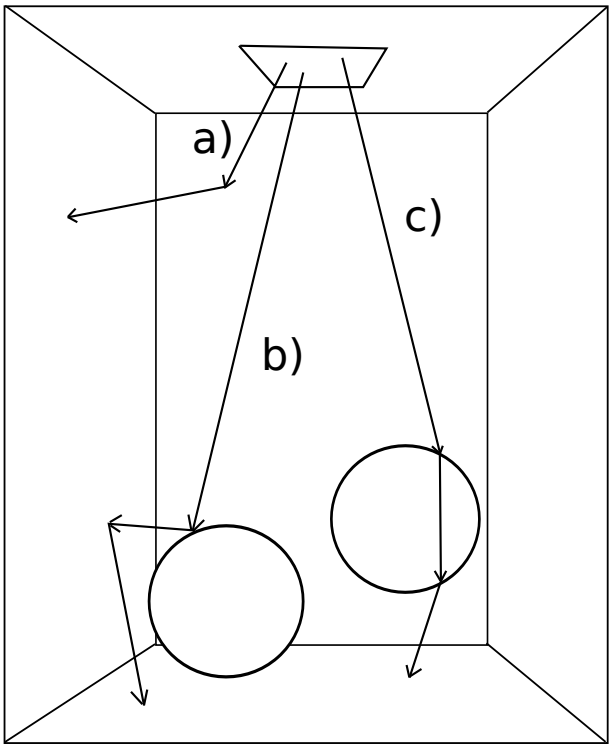
- Difúzní odraz vzniká na matných materiálech s hrubým povrchem, který rozptyluje světlo do všech směrů. Ideálně difúzní odraz má zcela náhodný odchozí směr nezávislý na příchozím směru světla (fotonu).
- Zrcadlový odraz vzniká na typicky velmi hladkých površích, jako je vyleštěný kov nebo na povrchu nekovových materiálů jako sklo či voda. Pro účely sledování fotonů lze opět využít ideálního odrazu světla (fotonu) na takovém povrchu podle fyzikálního zákona dopadu a odrazu světla pod stejným úhlem.

Pro rozhodnutí typu interakce fotonu s povrchem se používá technika známá jako ruská ruleta [8].

Ruská ruleta jednoduše slouží k tomu, že na základě pravděpodobnosti rozhodne, zda bude foton po nárazu na objekt dále sledován, nebo se sledování ukončí. Tato metoda velkou měrou sníží výpočetní a paměťové nároky na sledování fotonu. Pokud bychom ji nepoužili, pak by z fotonu po nárazu na např. difúzně odrazivý materiál vznikly dva nové fotony (odražený a difúzně vyzářený) s menší energií, které by se musely dále sledovat. Z jednoho vygenerovaného fotonu by tak mohlo průchodem scény vzniknout množství nových fotonů. Navíc by se do fotonové mapy ukládaly fotony s různou hodnotou nesené energie, což není výhodné při odhadu výsledné světelné energie ve druhém kroku photon mappingu [3].

Příklad použití ruské rulety

Předpokládejme materiál, který je zrcadlově i difúzně odrazivý a zároveň průhledný. Pro zrcadlový odraz je dán koeficient pravděpodobnosti s, pro difúzní odraz koeficient d a koeficient pro průchod fotonu materiálem t. Součet všech koeficientů dá pravděpodobnost menší nebo rovnou jedné, tedy $s + d + t \leq 1$. Následně pomocí uniformního rozložení



Obrázek 2.3: Cesty fotonu ve scéně: a) difúzní odraz, b) zrcadlový odraz následovaný difúzním odrazem, c) změna cesty fotonu zalomením a jeho následné pohlcení (převzato z [7])

vygenerujeme náhodné číslo $\xi \in [0, 1]$. Podle této hodnoty stanovíme, co se s fotonem po dopadu na tento materiál stane:

$\xi \in [0, d]$ – difúzní odraz

$\xi \in [d, s + d]$ – zrcadlový odraz

$\xi \in [s + d, t + s + d]$ – průchod se zalomením

$\xi \in [t + s + d, 1]$ – pohlcení fotonu

Ostatní typy materiálu se modelují obdobně, jen se ve výše uvedeném příkladu vynechá vlastnost, kterou materiál nedisponuje.

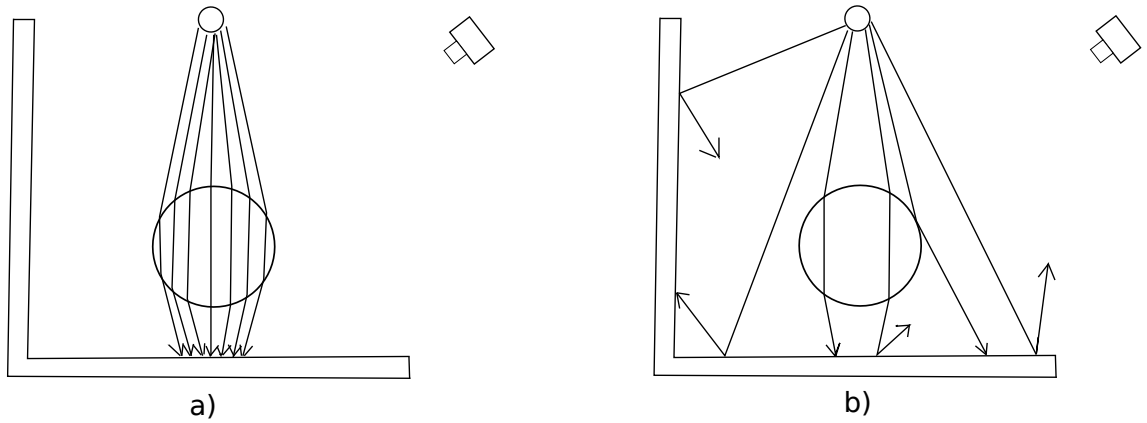
2.2.4 Ukládání fotonů

Do fotonových map se ukládají všechny fotony, které narazí na difúzní povrch. Zásadně se neukládají ty fotony, které interagují s odrazivými materiály, protože u takto odraženého světla je pravděpodobnost dopadu tohoto světla na čočku pozorovací kamery téměř nulová. Pro přesné vykreslení odrazu se využívá metody sledování paprsku. Za čas svého putování scénou může být foton uložen do mapy vícekrát.

Pro každý foton je potřeba uložit jeho pozici kolize s objektem, energii ve tvaru RGB, kterou přenáší a směr pod jakým foton na povrch dopadl. Protože se ve většině scén sleduje stovky tisíc až milióny fotonů, je potřeba, aby byla struktura uchovávající informace o fotonu

reprezentována na co nejmenším počtu bytů. Pro tuto skutečnost navrhl H. W. Jensen strukturu [3], která v paměti zabírá pouhých 20 bytů.

V důsledku efektivního renderování scény, které si ukážeme později, je dobré rozdělit ukládané fotony do dvou map, a to do mapy globální a mapy kaustik. Pro scénu provedeme dvakrát photon tracing. Poprvé generujeme fotony a hledáme průsečíky s odrazivými a průhlednými objekty, které způsobují vysokou koncentraci fotonů na jednom místě, tedy kaustiky. Pro větší efektivitu výpočtu můžeme využít projekčních map, tak jak bylo popsáno v 2.2.2. Tyto fotony uložíme do mapy kaustik. Ve druhém sledování fotonů ukládáme do globální mapy všechny fotony, které dopadly na difúzní povrch. Vše je ilustrováno na obrázku 2.4.



Obrázek 2.4: Vytvoření fotonových map: a) mapa kaustik, b) globální mapa (převzato z [7])

2.3 Renderování scény

Druhým krokem photon mappingu je renderování celé scény za využití informací z fotonových map. Jak již bylo řečeno, k renderování může být využito obyčejného sledování paprsku. Pro odhad energie záření vycházejícího z místa, kde paprsek proťal nějaký objekt scény (difúzní složka osvětlení), je možné využít fotonovou mapu. Tento přístup je ale nevhodný pro určení přímého osvětlení, ke kterému by bylo potřeba mnoho fotonů v mapě. Lepším přístupem je difúzní osvětlení rozdělit na přímé a nepřímé. Přímé osvětlení je potom vypočteno pomocí sledování paprsku a nepřímé jako odhad z fotonové mapy. Mapy kaustik jsou samozřejmě použity k zobrazení kaustik tam, kde se soustředí fotony odražené odrazivým materiálem nebo po průchodu průhledným objektem, který je zalomí.

2.3.1 Odhad zářivé energie plochy

Pro tento odhad se využívá aproximace zobrazovací rovnice (nepracujeme s vlnovou délkou světla), která je popsána vztahem 2.3 [3],

$$L_r(x,\vec{\omega}) = \int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}')(\vec{n}_x \cdot \vec{\omega}')d\vec{\omega}', \tag{2.3}$$

kde:

x je pozice plochy, kde sledovaný paprsek proťal objekt

ω je směr vyzářené energie

L_r je odchozí zářivá energie plochy v x ve směru ω

$\vec{\omega}$ je směr příchozí zářivé energie na plochu danou x

f_r je distribuční funkce BRDF (bidirectional reflectance distribution function) [9] v x

Ω_x je hemisféra nad x všech příchozích směrů zářivého toku

L_i je příchozí zářivá energie do x

$\vec{n}_x$ je normála plochy v x

Protože fotony, které jsou uloženy ve fotonové mapě, nesou informaci o energii a směru, pod kterým na povrch dopadly, poskytuje fotonová mapa informaci o přicházejícím zářivém toku. Díky tomu můžeme vyjádřit L_i jako:

$$L_i(x, \vec{\omega}') = \frac{d^2\Phi_i(x, \vec{\omega}')}{(\vec{n}_x \cdot \vec{\omega}')d\vec{\omega}'dA_i}, \tag{2.4}$$

kde Φ_i získáme odhadem z fotonové mapy tak, že sesbíráme n fotonů v požadované vzdálenosti od x. Každý foton má energii $\Delta\Phi_p$ a směr ω_p vzhledem k ploše ΔA, na které je sběr fotonů proveden. Tyto příspěvky energie sesumírujeme a dosadíme do rovnice 2.4, pak L_i dosadíme do rovnice 2.3 a dostaneme aproximaci integrálu:

$$L_r(x, \vec{\omega}) \approx \sum_{p=1}^{n} f_r(x, \vec{\omega}_p, \vec{\omega}) \frac{\Delta\Phi_p(x, \vec{\omega}_p)}{(\vec{n}_x \cdot \vec{\omega}_p)d\vec{\omega}_p\Delta A}(\vec{n}_x \cdot \vec{\omega}_p)d\vec{\omega}_p, \tag{2.5}$$

což můžeme matematicky upravit na

$$L_r(x, \vec{\omega}) \approx \sum_{p=1}^{n} f_r(x, \vec{\omega}_p, \vec{\omega}) \frac{\Delta\Phi_p(x, \vec{\omega}_p)}{\Delta A}. \tag{2.6}$$

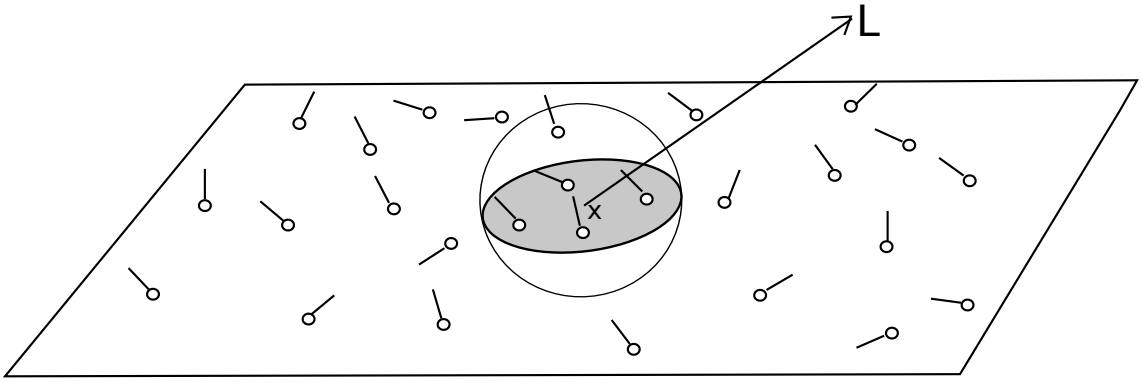
Sběr fotonů probíhá tak, že kolem x vytvoříme kouli s požadovaným poloměrem r. Požadované fotony leží uvnitř koule, viz obrázek 2.5. Je-li počet fotonů n, který potřebujeme sesbírat, větší než počet fotonů v kouli, můžeme poloměr této koule zvětšovat. Předpokládáme-li, že lokální oblast kolem x je plocha, můžeme pak ΔA v rovnici 2.8 nahradit za výpočet obsahu kružnice, která je tvořena danou kouli:

$$\Delta A \ = \pi r^2. \tag{2.7}$$

Dosazením 2.7 do 2.8 a vyjádřením zlomku před sumu získáme výslednou rovnici pro výpočet výstupní zářivé energie v jakémkoliv bodě scény x:

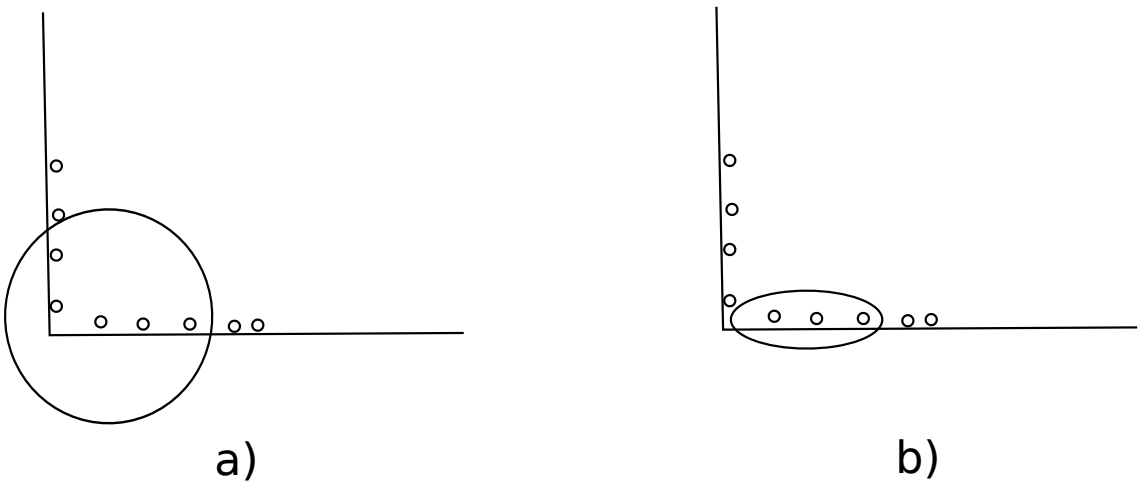
$$L_r(x, \vec{\omega}) \approx \frac{1}{\pi r^2} \sum_{p=1}^{n} f_r(x, \vec{\omega}_p, \vec{\omega})\Delta\Phi_p(x, \vec{\omega}_p). \tag{2.8}$$

Pro co nejlepší odhad energie záření potřebujeme dostatečné množství fotonů, čehož dosáhneme větším počtem generovaných fotonů ze světelného zdroje. Díky použití koule pro sběr fotonů může být odhad zatížen chybou, a to v případě, kdy se do sběru zahrnou fotony s drastickou změnou směru (normály) dopadu na povrch. To se může stát např. tam, kde spolu sousedí dva objekty a vytvářejí roh jako na obrázku 2.6 a). K výpočtu



Obrázek 2.5: Odhad zářivé energie L na základě N nejbližších fotonů z fotonové mapy (převzato z [7])

zářivé energie jednoho objektu se tak zahrnou i fotony patřící druhému objektu. Proto je výhodné pro sběr fotonů využít např. elipsoid, jak naznačuje obrázek 2.6 b). Tento elipsoid lze vytvořit na základě normály v bodě x. V případě použití jiného tělesa je potřeba změnit výpočet ΔA v rovnici 2.7 podle zvoleného tvaru tělesa. Posledním problémem, který se při odhadu může vyskytnout, je přítomnost fotonu na místě, kde nepatří. Tento případ se dá částečně vyřešit pomocí váhového filtrování, kdy fotonu blíže počítané pozice x je přiřazena větší váha.



Obrázek 2.6: Sběr fotonů a) pomocí koule, b) pomocí elipsoidu (převzato z [3])

2.3.2 Řešení osvětlení scény pomocí zobrazovací rovnice

Nyní, když umíme odhadnout příspěvek zářivé energie v daném bodě, můžeme přejít k výpočtu osvětlení pro celou scénu. Zobrazovací rovnice pro řešení globálního osvětlení scény tak, jak byla představena Kajiyou [10]:

$$L(x, \omega_0) = L_e(x, \omega_0) + \int_H L(r(x, \omega_i), -\omega_i) \cdot f_r(x, \omega_i \rightarrow \omega_0) \cdot cos(\theta_i) d\omega_i, \qquad (2.9)$$

může být zjednodušeně zapsána jako:

$$L_r = L_d + L_s + L_c + L_i, \tag{2.10}$$

kde:

L_d je přímé osvětlení

L_s je odraz na zrcadlovém a lesklém povrchu

L_c jsou kaustiky

L_i je nepřímé osvětlení

Dále si ukážeme, jak se jednotlivé příspěvky osvětlení vypočítají za pomocí fotonových map a využití odhadu zářivé energie.

Přímé osvětlení

Jak již bylo řečeno, pro přímé osvětlení vycházející ze zdrojů světla se využívá metody sledování paprsku, která poskytuje výborné výsledky. Metoda je navíc poměrně rychlá pro výpočet ostrých stínů, pokud jsou ve scéně pouze bodové zdroje světla, a tedy stačí vyslat pouze jeden stínový paprsek ke světlu z místa průsečíku sledovaného paprsku se scénou. Když se ale ve scéně nachází nějaké plošné světlo a je potřeba zobrazit měkké stíny, musí se z místa průsečíku vyslat několik paprsků k tomuto světlu, aby bylo možné efektivně integrovat příspěvek osvětlení tohoto světla. Díky tomu však narůstá čas výpočtu, protože jak již bylo řečeno v odstavci 2.1, je nutné všechny paprsky testovat s objekty scény.

Pokud nám stačí pouze aproximace, můžeme využít přímého odhadu zářivé energie z globální fotonové mapy. V tomto případě není potřeba testovat paprsek se světly ve scéně, a to ani v případě stínů.

Odraz na zrcadlovém a lesklém povrchu

Již jsme zmínili, že fotony po nárazu na zrcadlový nebo lesklý materiál nejsou uloženy do fotonové mapy. Místo odhadu z fotonové mapy je využito sledování paprsku, jak bylo popsáno v podkapitole 2.1.

Kaustiky

Příspěvek kaustik se odhaduje výhradně z fotonové mapy, která je pro ně vyhrazena. Protože kaustiky mohou obsahovat detaily, o které nechceme přijít, je potřeba dostatečně velký počet fotonů uložených v mapě. Pro zaostření okrajů můžeme využít např. již zmíněné váhové filtry.

Nepřímé osvětlení

Nepřímé osvětlení je příspěvek energie fotonů, které byly po vygenerování ze světla minimálně jednou difúzně vyzářeny a poté uloženy do mapy. Aproximaci nepřímého osvětlení dostaneme odhadem zářivé energie z globální fotonové mapy. Pro dobrý výsledek je potřeba mít větší množství fotonů v mapě. Pro velmi přesné zobrazení je možno využít techniky *final gathering* [3], která ale drasticky prodlouží čas výpočtu.

2.4 Možnosti urychlení photon mappingu

Jelikož je photon mapping postaven na sledování paprsku, naskýtá se několik možností urychlení této metody:

1. Sledování fotonu je stejně jako sledování paprsku absolutně nezávislé na sledování ostatních fotonů. Proto lze u photon mappingu oba jeho průchody poměrně jednoduše paralelizovat buď na CPU anebo na GPU. Akcelerace výpočtů pomocí GPU bude ukázaná v kapitole 4.

2. Dalšího podstatného zrychlení lze dosáhnout pomocí dělení prostoru, popsaného v následující kapitole. Toto dělení prostoru opět urychlí jak první průchod, tak druhý průchod metody. V prvním průchodu takto omezíme počet primitiv, na které může foton dopadnout, a tím se sníží počet testování. U druhého průchodu je tomu obdobně při sledování paprsku. Navíc lze pomocí akceleračních struktur pro dělení prostoru reprezentovat fotonové mapy, což se projeví v rychlosti sběru požadovaných fotonů.

3. Jako poslední zmíníme rozdělení fotonů do dvou map, tedy do globální fotonové mapy a mapy kaustik. U scén, kde se vyskytují kaustiky, tak snížíme počet fotonů v globální mapě, a tím zmenšíme velikost případné akcelerační struktury, což může vést k rychlejšímu vyhledání fotonů v dané mapě.

2.5 Přehled dalších metod řešících globální osvětlení

V této podkapitole si představíme některé další metody řešící globální osvětlení scény a ukážeme, které efekty zobrazování rovnice 2.9 umí řešit.

2.5.1 Distribuované sledování paprsku

Distribuované sledování paprsku (*Distributed ray tracing*) [11] je rozšíření základního sledování paprsku, které umožňuje vytvářet pouze ostré obrazy scény. Je to dáno tím, že pracuje s ideálním odrazem, zalomením paprsku a zvládne simulovat pouze bodové zdroje světla, čímž vznikají ostré stíny objektů ve scéně. Distribuované sledování paprsku naopak využívá distribuční funkci, pomocí které jsou paprsky do scény vrhány. Distribuční funkce tedy určuje pravděpodobnost, že paprsek bude vržen daným směrem. Pro určení distribuční funkce je potřeba spočítat integrál, který je možné nahradit pomocí metody Monte Carlo, kdy je distribuce vzorkována náhodně zvolenými pokusy a výsledkem je vážený průměr všech pokusů.

Tato metoda řeší stejně jako základní sledování paprsku pouze přímé osvětlení, odraz a zalomení paprsku. Pomocí distribuovaného sledování paprsku lze úspěšně simulovat následující jevy:

- Antialiasing pomocí vyslání více paprsků jedním pixelem.
- Matný odraz daných materiálů na základě variace normály v bodě dopadu paprsku.
- Matně průhledné materiály jako je matné sklo. Opět lze využít variaci normály.
- Měkké stíny pro plošné zdroje světla.
- Hloubka ostrosti scény (*depth of field*).

- Rozmazání obrazu (*motion blur*) v závislosti na čase.

2.5.2 Radiozita

Radiozita [12] vychází ze zákona zachování energie v uzavřené scéně. Metoda je založena na předpokladu, že difúzní plochy mohou světlo odrážet a zároveň mít vlastní zářivost. Nelze pomocí ní simulovat průchod světla průhlednými objekty nebo zrcadlový odraz světla, tedy kaustiky. Naopak umí zvládnout šíření difúzního odrazu světla a měkké stíny.

Ještě před výpočtem radiozity v dané scéně je potřeba tuto scénu rozdělit na konečný počet plošek, u kterých budeme zkoumat vliv každé plošky na každou ostatní plošku. Tento vliv se nazývá konfigurační faktor (*form factor*) a plošky, které se nevidí, mají tento faktor roven nule. Vlastní výpočet radiozity je buď iterační, kdy se na počátku považují za zdroje záření pouze světla ve scéně a postupně se světlo šíří a vytváří sekundární zdroje světla, anebo řešitelný pomocí soustavy rovnic, tedy matice. Ve většině scén je tato matice silně diagonální, protože hodně plošek má konfigurační faktor nulový, nevidí na sebe. Rovnice radiozity pro jednu plošku je definována takto:

$$B_i = E_i + \rho_i \Sigma B_j F_{ij}, \tag{2.11}$$

kde:

- B_i je záření (radiozita) z plošky i.
- E_i je vyzařování energie plošky i.
- ρ_i je odrazivost plošky i.
- F_{ij} je konfigurační faktor mezi ploškou i a ploškou j.

Výsledný výpočet osvětlení je poté třeba zobrazit např. pomocí rasterizace nebo sledování paprsku.

2.5.3 Sledování cest

Sledování cest (*path tracing*) je metoda řešící všechny části zobrazovací rovnice 2.9 a byla představena Kajiyou spolu s touto rovnicí [10]. Metoda funguje tak, že pro každý pixel plátna se do scény vrhá mnoho paprsků z kamery a rekurzívně se hledají jejich průsečíky s objekty scény. Daný paprsek se ve scéně po dopadu na nějaký materiál může odrazit a pokud se odrazí do světelného zdroje, pak sledování paprsku končí a ve všech místech, kde se po cestě odrazil, vypočítáme hodnotu osvětlení. Na konci sledování všech paprsků pro jeden pixel se zprůměrují jejich hodnoty, čímž je výpočet barvy pro daný pixel dokončen.

Pro co nejlepší výsledky je potřeba vyslat jedním pixelem obrovské množství paprsků, protože ne každý se trefí do zdroje světla. Jedná se tak o metodu, jejíž konvergence je velmi pomalá. Při použití obrovského množství paprsků hraničí generované obrázky s fotorealismem a často se pak používají jako referenční obrázky pro měření kvality jiných renderovacích algoritmů. Metoda je takto přesná díky zpětné simulaci cesty světla jako v reálném světě, kdy světlo putuje ze zdroje do oka (kamery).

Pro urychlení metody sledování cest bylo vytvořeno několik modifikací, z nichž nejznámější je obousměrné sledování cest (*bi-directional path tracing*) [2], kde se sleduje paprsky jak z kamery, tak ze světelných zdrojů.

Kapitola 3

Dělení prostoru

Základní metoda sledování paprsku, kde je potřeba testovat průsečík paprsku s každým objektem scény (ve většině případů se jedná o trojúhelníky, ze kterých se scéna skládá), bývá výpočetně, a tedy i časově velmi náročná (složitost $O(n)$). V této kapitole si ukážeme několik akceleračních struktur, které je možné použít při sledování paprsku. Na konci kapitoly se ještě podíváme na rychlé vyhodnocení průniku trojúhelníku s osově zarovnanou buňkou a na test průsečíku paprsku s trojúhelníkem.

Akcelerační struktury fungují na principu dělení prostoru scény na menší oblasti, kde se nejprve testuje průsečík paprsku s touto malou oblastí a teprve pokud je test pozitivní, tak následuje test na průsečík s trojúhelníky, příp. objekty, které v dané oblasti leží. Výběr požadované struktury hodně záleží na vlastnostech scény, jako např. zda se jedná o dynamickou scénu a je potřeba, aby byla přestavba struktury co nejrychlejší. Některé ze struktur spolu s procházením paprsku těmito strukturami si popíšeme níže.

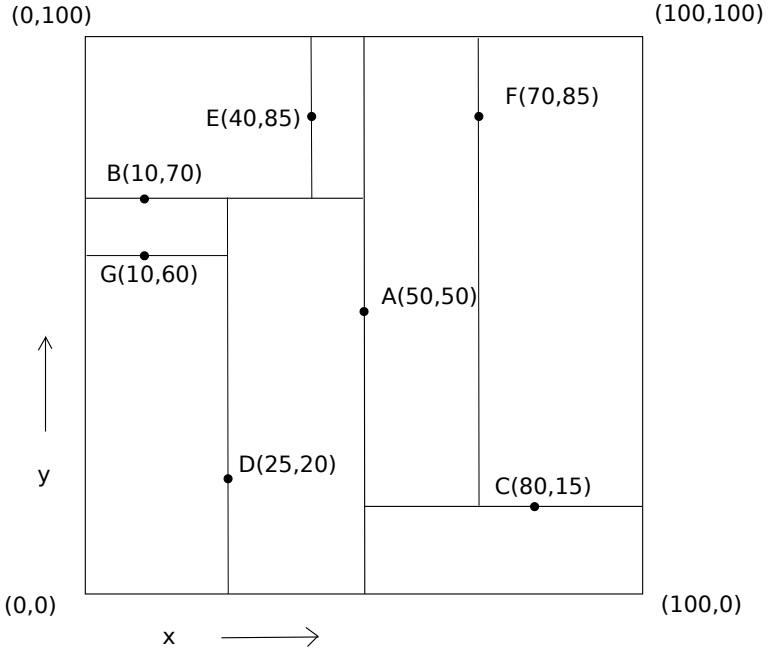
Pozn.: Struktury popisované v následujícím textu pracují obecně s polygony, ale my se omezíme pouze na popis práce s trojúhelníky, neboť dále budeme využívat tyto struktury pouze pro práci se scénou složenou z trojúhelníku, příp. bodů, které reprezentují jednotlivé fotony ve scéně, viz níže.

3.1 KD strom

KD strom je založen na binárním vyhledávacím stromu a umožňuje rekurzivně dělit k dimenzionální prostor. Každý uzel stromu je dělící nadrovinou dané dimenze, kdy jsou nadroviny kolmé na jednu z prostorových os [13]. Na obrázku 3.1 jsou znázorněny dělící nadroviny pro body ve 2D prostoru. Pro trojúhelníky vypadá strom obdobně. Rozdíl je v tom, že se dělící nadrovina vybírá podle osově zarovnaných obálek trojúhelníků (*Axis-aligned bounding box*). Některé trojúhelníky mohou ležet ve více uzlech stromu, a naopak některé uzly mohou být prázdné.

Vybalancovaný strom se dá sestavit tak, že postupně procházíme všechna data např. x, y, z v 3D prostoru, kdy v prvním kroku nalezneme v ose x medián, který bude tvořit kořen stromu. Tím vzniknou dvě podmnožiny, jedna s daty menšími než medián - tvoří levý podstrom a druhá s daty většími jak medián - tvoří pravý podstrom stromu. Ve druhém kroku obě podmnožiny seřadíme podle osy y a opět hledáme v každé podmnožině medián, který nám tyto podmnožiny rozdělí na pravé a levé podstromy. Analogicky postupujeme pro osu z. Dále se pokračuje znovu podle první osy x. Vyhledávání v takto vybalancovaném stromu má složitost $O(logN)$ [13], kde N je počet dat uchovávaných ve stromu. K vylepšení

vlastností naivně pomocí mediánu sestaveného stromu se často využívá SAH (*Surface area heuristic*) [14], která určuje cenu průchodu (vyhledání) ve stromu.



Obrázek 3.1: K-D strom pro 2D prostor (převzato z [13])

Jedním z algoritmů pro procházení paprsku KD stromem je tzv. zásobníkově závislý algoritmus [15]. Při hledání průsečíku algoritmus v daném uzlu vybírá, v jakém pořadí mají být jeho potomci procházení, a jestli nemá být některý z prohledávání vyřazen. Potomci se dělí na bližší a vzdálenější v závislosti na poloze počátku paprsku a dělící nadroviny v daném zpracovávaném uzlu. V tomto případě se pak jako bližší bere levý potomek, pokud počátek paprsku leží v levé polovině dělící nadroviny a pravý jako vzdálenější. Stejně je tomu i naopak, kdy počátek paprsku leží v pravé polovině. Pro takto klasifikované potomky mohou nastat tři případy průchodu:

- Navštiv pouze bližšího potomka.
- Navštiv pouze vzdálenějšího potomka.
- Navštiv nejdříve bližšího a pak vzdálenějšího potomka.

Algoritmus prochází stromem, dokud nenarazí na listový uzel a tam testuje trojúhelníky na průsečík, pokud průsečík není nalezen a zároveň je zásobník prázdný, průsečík se scénou neexistuje, jinak pokud není zásobník prázdný, zpracovává se další uzel ze zásobníku.

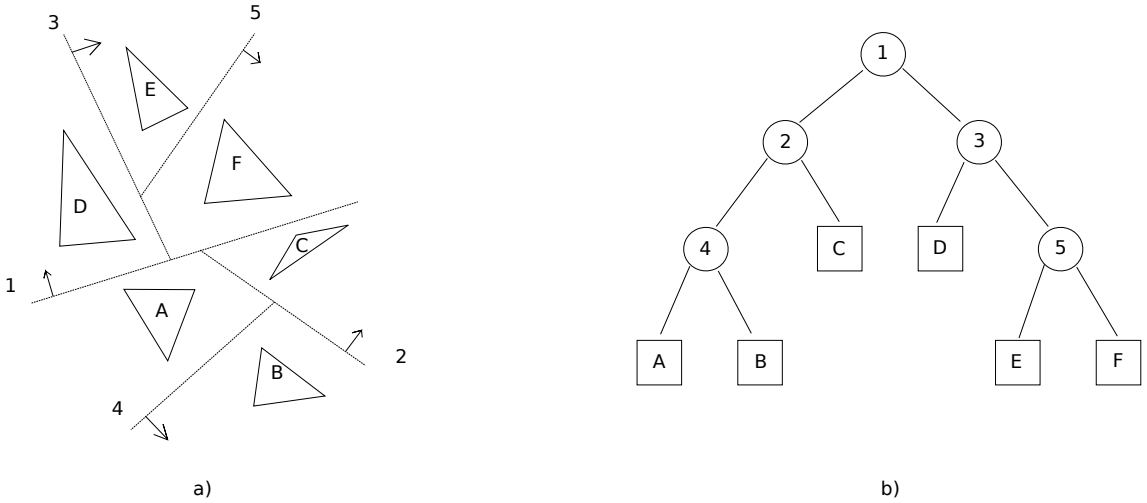
3.2 BSP strom

Opět se jedná o strom založený na binárním vyhledávacím stromu, dělící prostor na podprostory pomocí nadroviny, který byl představen kolem roku 1980 [16]. Na rozdíl od KD stromu je obecnější, protože jeho dělící nadrovina nemusí být rovnoběžná s žádnou prostorovou osou a může tedy mít jakoukoliv orientaci.

Dělení pomocí BSP probíhá rekurzivně, a to tak, že nejdříve zvolíme dělící nadrovinu a objekty scény (trojúhelníky) rozdělíme na dva seznamy. Jeden seznam obsahuje pouze trojúhelníky ležící před dělící nadrovinou a druhý trojúhelníky ležící za dělící nadrovinou. Tyto seznamy se uloží do dvou uzlů, které jsou potomky uzlu s dělící nadrovinou. Dále se pokračuje zpracováním těchto dvou nově vytvořených uzlů pomocí rekurze až do té doby, dokud není celá scéna (trojúhelníky) rozdělena do jednotlivých uzlů stromu. Proces dělení je znázorněn na obrázku 3.2. Při dělení je potřeba řešit tyto čtyři případy:

- Trojúhelník leží celý před nadrovinou, zařadíme jej do seznamu s předními trojúhelníky.
- Trojúhelník leží celý za nadrovinou, zařadíme jej do seznamu se zadními trojúhelníky.
- Trojúhelník se protíná s dělící nadrovinou a je třeba jej uložit do obou seznamů.
- Trojúhelník leží v dělící nadrovině, zařadíme jej do právě zpracovávaného uzlu, který odpovídá dělící nadrovině.

Algoritmus pro nalezení průsečíku paprsku se scénou je prakticky totožný s algoritmem popsaným u struktury KD strom, viz 3.1.



Obrázek 3.2: BSP strom dělící 2D prostor, a) ukázka postupného výběru dělících nadrovin, b) sestavení BSP stromu (převzato z [17])

3.3 Octree

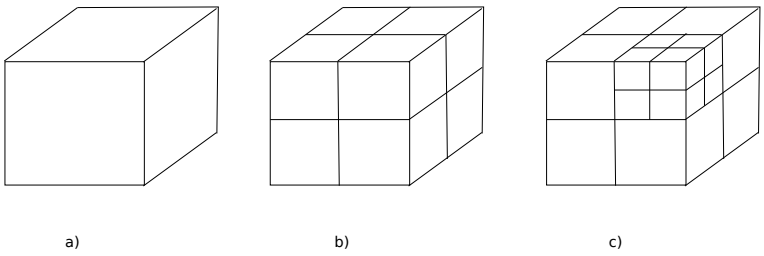
Další strukturou pro dělení prostoru je oktalový strom, nebo-li octree prezentovaný D. Meagherem [18]. Octree pracuje pomocí osově zarovnaných krychlí (voxely), kdy kořen stromu je krychle obalující celou scénu. Tento kořen se dále dělí na osm vzájemně se nepřekrývajících stejných krychlí (z toho název octree), které jsou potomky (uzly) kořene. Dále se provádí rekurzivní dělení těchto uzlů na přesně osm potomků, pokud se v nich nacházejí nějaké objekty scény (trojúhelníky). Trojúhelníky se nacházejí až v listových uzlech a odkaz na ně může být uložen ve více uzlech, pokud je překrývá několik krychlí. Podobně jako u KD stromu mohou být nějaké listy prázdné, a to z důvodu tvorby konstantního počtu

nových krychlí z daného uzlu. Postup dělení prostoru pomocí octree je patrný z obrázku 3.3.

Problémem, který je nutné řešit při vytváření octree, zůstává ukončení rekurzivního dělení tak, aby neprobíhalo do nekonečna. Pro zajištění tohoto ukončení existuje několik přístupů:

- Pevná hloubka rekurze - pokud po skončení rekurze zbývají ještě nějaké trojúhelníky, tak jsou přiděleny do koncových uzlů.
- Maximální počet trojúhelníků v uzlu - pokud je počet trojúhelníků menší než tato konstanta, dělení končí.
- Test na maximální počet uzlů.

Pro procházení paprsku stromem existuje několik přístupů. Nejčastěji se používá buď metoda zdola nahoru nebo metoda shora dolů. U metody zdola nahoru se začíná v prvním listovém uzlu, který má průsečík s paprskem. Poté se pomocí algoritmu nejbližších sousedů hledají další listové uzly, kterými paprsek prochází [19]. Metoda shora dolů využívá prohledávání stromu do hloubky a začíná v kořenovém uzlu. Poté se z aktuálního uzlu najdou jeho potomci, kteří jsou zasaženi paprskem a na tyto potomky se rekurzivně využije tohoto hledání dalších potomků. Rekurzivní proces pokračuje tak dlouho, dokud není dosaženo listových uzlů [20].



Obrázek 3.3: Dělení prostoru pomocí octree, a) kořen, b) první dělení kořene, c) dělení jednoho z uzlů, vytvořeného v kroku b) (převzato z [21])

3.4 Uniformní mřížka

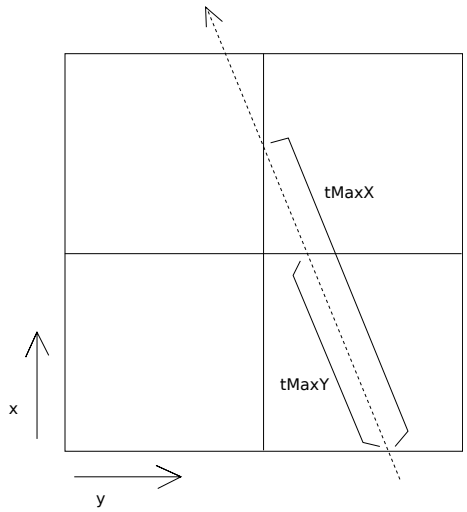
Jedná se o jednu z nejjednodušších akceleračních struktur pro dělení prostoru. Uniformní mřížka dělí prostor na stejně velké navzájem se nepřekrývající buňky, které mohou obsahovat odkazy na geometrii scény (trojúhelníky), pokud s nimi kolidují. První použití mřížky v počítačové grafice popsal Fujimoto [22]. Pro svoji jednoduchost a rychlost přestavby se uniformní mřížka často používá při urychlování metody sledování paprsku v dynamicky se měnících scénách. Její velkou nevýhodou je nemožnost se jakkoliv přizpůsobit geometrii scény. Lze pouze ovlivnit počet buněk mřížky. Mřížka je proto nevhodná např. pro scény typu „detailní míč uprostřed fotbalového stadionu“. Zde je zřejmý problém v tom, že některé buňky překrývají velmi detailní objekt a jejich využití je oproti buňkám ve zbytku scény poměrně značný a při nalezení průsečíku s takovou buňkou je potřeba testovat všechny trojúhelníky, které v ní leží.

Metoda, která umožňuje velmi efektivní průchod paprsku uniformní mřížkou, se nazývá 3D-DDA algoritmus [23], který je založen na Bresenhamově algoritmu DDA pro rasterizaci

přímky [24]. Tento 3D-DDA algoritmus funguje tak, že paprsek $\vec{u} + t\vec{v}$, $t \geq 0$ rozdělí na intervaly t, kde každý interval vyplňuje jednu danou buňku. Poté začínáme na počátku paprsku a procházíme každou buňkou v pořadí vypočítaných intervalů.

3D-DDA si popíšeme na 2D verzi, ze které se pak dá jednoduše odvodit verze pro 3D. Algoritmus má dvě fáze, kdy v první se inicializují počáteční hodnoty a ve druhé probíhá samotný inkrementální průchod mřížkou. V inicializační části je potřeba nejdříve určit pozici buňky, ve které má paprsek počátek $\vec{u}$. Pokud leží počátek paprsku mimo buňku, určíme jako výchozí buňku, do které paprsek vstupuje. Následně podle znaménka x a y komponenty směrového vektoru $\vec{v}$ určíme směr pohybu paprsku po buňkách, který může na obou osách (x a y) nabývat hodnot 1 nebo -1. Dále určíme hodnotu t, která nám říká o kolik může paprsek postupovat v ose x a y tak, aby ještě stále zůstal v aktuální buňce. Pro osu x tedy nalezneme první vertikální hranici buňky a pro osu y naopak první horizontální hranici. Přírůstek v ose x (tMaxX) a y (tMaxY) je naznačen na obrázku 3.4. Nakonec určíme jak daleko se musíme posunout po paprsku, aby paprsek proťal právě jednu buňku.

Druhá část, tedy průchodová část, je již jen jednoduchý posun paprsku do další buňky právě o nejnižší hodnotu parametru t ve směru osy x nebo y. Tento postup se opakuje ve smyčce. Pokud narazíme na buňku, která obsahuje nějaké trojúhelníky, provedeme test průsečíku paprsku s těmito trojúhelníky. Zde si však musíme dát pozor, zda testovaný trojúhelník leží pořád ve stejné buňce jako paprsek, protože jeden trojúhelník může být uložen ve více buňkách, a tím mohou vznikat chyby při určení tzv. falešného průsečíku. Pokud je průsečík nalezen, anebo paprsek vystoupí z uniformní mřížky, a tedy není průsečík nalezen, algoritmus končí.



Obrázek 3.4: Průchod paprsku uniformní mřížkou, určení parametru tMaxX a tMaxY (převzato z [23])

3.5 Test na překrytí trojúhelníku s osově zarovnanou buňkou

Umět rychle rozhodnout, zda trojúhelník patří do zkoumané buňky (v našem případě se omezíme pouze na osově zarovnanou buňku), je velmi důležitým kritériem při stavbě jak octree tak uniformní mřížky. Pro účely detekce je vhodný tzv. *Separating Axis Theorem* (SAT) [25]. SAT umí rozhodnout, zda se dva konvexní tělesa, ať už ve 2D nebo 3D, prostoru

protínají. K tomu využívá projekci těles na osy kolmé k jejich stěnám. Obě tělesa promítneme na jednu osu kolmou ke straně jednoho tělesa a kontrolujeme, zda se jejich projekce protínají. Testujeme tedy průnik dvou intervalů v 1D. Pokud se projekce neprotínají, pak zde existuje osa, kterou lze obě tělesa oddělit, tedy *separating axes* a tělesa se určitě neprotínají. Naopak, pokud se projekce překrývají, pokračuje s projekcemi pro všechny ostatní kolmé osy k hranám těles. Teprve po konci testování všech projekcí a pokud se všechny projekce překrývají, víme, že se oba tělesa protínají. Urychlení při testování průsečíku spočívá v tom, že jakmile najdeme první osu rozdělující oba tělesa, tak můžeme s naprostou jistotou říci, že se neprotínají.

Jelikož jsme si uvedli, že se omezíme pouze na detekci průniku trojúhelníku a osově zarovnané buňky, využijeme rychlého 3D algoritmu od T. Akenine-Möllera [26], který je derivací metody SAT. V rámci tohoto algoritmu přesuneme oba testovaná tělesa, tedy buňku i trojúhelník do počátku souřadného systému podle středu buňky, čímž zjednodušíme testy. Následně provedeme 13 testů SAT:

1. 3 testy mezi osově zarovnanou buňkou a minimální osově zarovnanou obálkou trojúhelníku.
2. 1 test mezi normálou trojúhelníku a buňkou, kdy lze využít algoritmu pro rychlou detekci překrytí roviny a osově zarovnané buňky [27].
3. 9 testů mezi hranami trojúhelníku a hranami buňky, tedy provedeme projekci trojúhelníku podle rovnice

$$\vec{a_{ij}} = \vec{e_i} \times \vec{f_i}, i, j \in 0, 1, 2, \tag{3.1}$$

kde $\vec{e_i}$ jsou normály buňky posunuté do počátku souřadnicového systému a $\vec{f_i}$ jsou hrany taktéž posunutého trojúhelníku spočítané pomocí jeho vrcholů jako $\vec{f_0} = \vec{v_1} - \vec{v_0}, \vec{f_1} = \vec{v_2} - \vec{v_1}, \vec{f_2} = \vec{v_0} - \vec{v_2}$. Následně promítneme vrcholy $\vec{v_0}, \vec{v_1}, \vec{v_2}$ na a_{ij} tak, že uděláme skalární součin:

$$\begin{aligned} p_0 &= \vec{a_{ij}} \cdot \vec{v_0} \\ p_1 &= \vec{a_{ij}} \cdot \vec{v_1} \\ p_2 &= \vec{a_{ij}} \cdot \vec{v_2} \end{aligned} \tag{3.2}$$

Po této projekci spočítáme „poloměr“ buňky r promítnutý na $\vec{a}$ jako

$$r = h_x|a_x| + h_y|a_y| + h_z|a_z|, \tag{3.3}$$

kde $\vec{h}$ je polovina délky stran buňky. Pokud se jedná o uniformní buňku, která má všechny strany stejně velké, pak $h_x = h_y = h_z$.

Nakonec provedeme test, zda existuje separační osa mezi buňkou a trojúhelníkem, tedy $min(p_0, p_1, p_2) > r \;||\; max(p_0, p_1, p_2) < -r$. Pokud se ani v jednom z 13 testů nenalezne separační osa, pak se trojúhelník a buňka překrývají.

Kapitola 4

Akcelerace výpočtů pomocí grafických karet

V této kapitole se zaměříme na urychlování výpočtů pomocí grafických karet. Ve zkratce popíšeme architekturu grafických čipů (budeme uvažovat karty NVIDIA) a ukážeme si rozdíly v programování pro GPU oproti CPU. Také zmíníme několik základních programovacích API grafických karet a to OpenGL, CUDA a OpenCL.

4.1 Architektura GPU

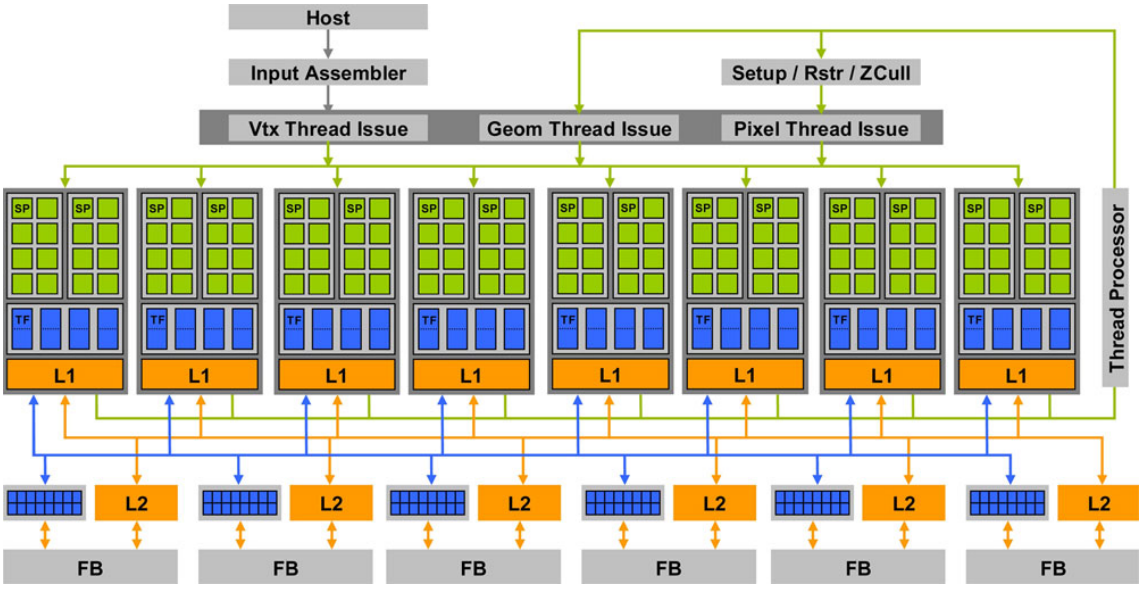
Za několik posledních let prošly grafické karty bouřlivým vývojem. Již neslouží jen pro urychlování vykreslování grafiky, ale staly se plnohodnotnými aritmetickými koprocesory ke klasickým CPU, na rozdíl od kterých mají mnohonásobně vyšší výpočetní výkon. Je však nutné podotknout, že nejsou plně univerzální a existují určitá omezení, na jaké výpočty mohou být použity. Od CPU se odlišují v architektuře hardwaru, kterou je potřeba poměrně precizně znát, aby bylo možné plně využít jejich potenciální výkon.

Zatímco CPU sestává z jednotek až maximálně desítek jader, GPU jich obsahuje stovky až tisíce. Na GPU tedy lze provádět masivní paralelizaci výpočtů. Základní stavbu GPU si ukážeme na unifikované architektuře NVIDIA G80 [28], viz obrázek 4.1, ze které pak vycházejí další architektury grafických karet NVIDIA.

GPU obsahuje několik *streaming multiprocesorů* (SM). Tyto SM sdružují bloky skalárních výpočetních jader, nebo-li *streaming procesorů* (SP), umožňujících provádět aritmetické operace sčítání a násobení. Detailnější pohled na multiprocesor je na obrázku 4.2. Zde můžeme vidět, že jeho součástí jsou mimo SP i *special function unit* (SFU) sloužící k provádění složitějších aritmetických operací jako dělení, sin, cos… Počet SFU je značně menší než počet procesorů. K multiprocesoru dále patří sdílená paměť *(shared memory)* o velikosti řádově desítek kB, kde mají přístup všechny procesory daného multiprocesoru. Každý SP má navíc svoji vlastní sadu registrů pro ukládání proměnných opět o velikosti v řádech kB.

Výše jsme uvedli, že SP v rámci jednoho multiprocesoru mohou navzájem sdílet data pomocí sdílené paměti. Tato sdílená paměť ale není viditelná pro ostatní SM. Sdílení dat mezi SM je možné pomocí globální paměti GPU, jejíž velikost se v dnešních zařízeních pohybuje v rozmezí jednotek GB. Součástí globální paměti je dále lokální paměť pro proměnné SP, které se nevejdou do registrů a kešovatelné paměti textur a konstant [28].

Přístupová doba do jednotlivých pamětí GPU není stejná [31]. Nejrychlejší je přístup



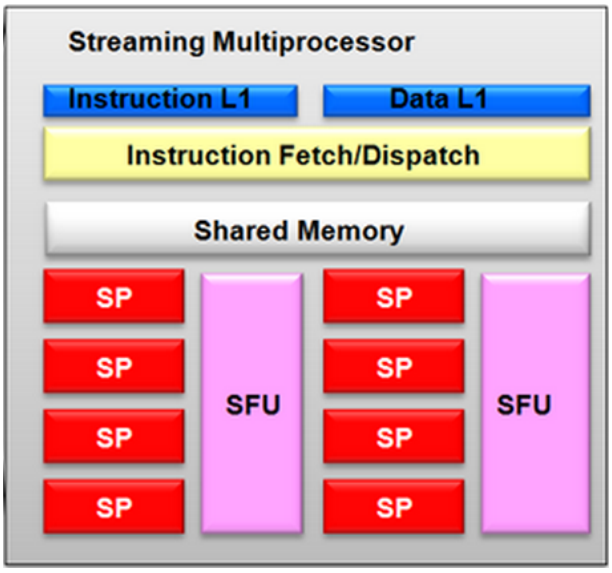
Obrázek 4.1: Blokové schéma architektury NVIDIA G80 (převzato z [29])

do registrů a sdílené paměti ležící přímo na čipu GPU. Tato paměť je rozdělena na skupiny nazývané banky, do kterých může být přistupováno zároveň. Vícenásobné přístupy do stejných bank vedou ke konfliktu a tento přístup musí být serializován, čímž dochází ke zpomalení. O něco pomalejší je cache pro paměť konstant a texturovací paměť. Přístup do těchto dvou pamětí je tedy velmi rychlý, ale to pouze za předpokladu, že se požadovaná data v cache nachází. Pokud tam přítomna nejsou, je potřeba je získat z globální paměti. Globální paměť se nachází mimo čip GPU a přístup do ní je řadově stokrát pomalejší, než přístup do registrů nebo sdílené paměti. Pro dosažení plné propustnosti této paměti musí být přístup do ní zarovnán [31]. Poslední lokální paměť opět leží mimo čip, a tudíž má stejnou přístupovou dobu jako paměť globální.

4.2 Programování pro GPU

Nyní když známe architekturu GPU, můžeme se velmi zjednodušeně podívat na psaní programů pro grafické karty. To se totiž poměrně liší od programování na CPU. Architektura *streaming multiprocesoru* je typu SIMD (*single instruction multiple data*), nebo-li provádí zpracování jedné instrukce nad množstvím dat. Tato data by ideálně měla být na sobě nezávislá. Dále je výpočet rozdělen do velkého množství vláken SIMT (*single instruction multiple thread*). Tato vlákna se v rámci jednoho SM vykonávají po balících (*warp*), jejichž latence je při přepínání zanedbatelná. Na GPU musíme rozdělit výpočet na co největší počet vláken, čímž překryjeme zpoždění při přístupu do paměti, protože zatímco některé *warpy* čekají na data, další mohou pracovat. Zde je potřeba dávat si pozor na divergenci vláken, která může nastat při zpracování podmínky *if* nebo vyhodnocení *case*. Všechna vlákna v rámci *warpu* jsou ve stejném stavu zpracování, jakmile však dojde k divergenci a některá vlákna jdou jinou cestou, musí *warp* čekat na tyto vlákna, což značně brzdí výpočet.

Problémem způsobujícím zpomalení výpočtu je taky přístup do paměti. Jak jsme si ukázali výše, globální paměť je nejpomalejší. Proto pracujeme co nejvíce se sdílenou pamětí, kde nahrajeme podmnožinu dat z globální paměti a provedeme nad nimi co nejvíce operací.



Obrázek 4.2: Základní bloky streaming multiprocesoru (převzato z [30])

Výsledky se poté zapíšou opět do globální paměti. Navíc musíme dbát na zarovnaný přístup do paměti a správné čtení s rozestupem [32].

4.2.1 OpenGL

OpenGL (*Open Graphics Library*) [33] je nízkoúrovňová multiplatformní knihovna určená k vykreslování 3D grafiky. Jedná se o průmyslový standard spravovaný konsorciem ARB (Architecture Review Board). Od prvního vydání knihovny v roce 1992 již vzniklo několik nových verzí, které byly vždy zpětně kompatibilní až do verze 3.0, kdy se začaly odstraňovat zastaralé funkce knihovny. Od verze 2.0 je fixní pipeline nahrazená programovatelnou pipeline pomocí shaderů. Shadery jsou programovatelné pomocí jazyka GLSL, který svojí syntaxí vychází z jazyka C. Vykreslování probíhá tak, že vrcholy scény jsou nejprve uloženy do paměti grafické karty, poté jsou nad každým vrcholem zvlášť provedeny operace definované vertex shaderem a výsledný fragment je obarven pomocí fragment shaderu. Nyní (prosinec 2012) se knihovna nachází ve verzi 4.3.

4.2.2 CUDA

CUDA (*Compute Unified Device Architecture*) [34] představuje hardwarové a softwarové rozhraní pro programování GPU, vytvořené firmou NVIDIA a je použitelná výhradně pro grafické karty NVIDIA. První verze 1.0 byla představena v roce 2006 a první SDK bylo uvolněno o rok později pro GPU architekturu G80. S vývojem GPU přicházely další verze, které přidávaly novou funkčnost, nebo-li rozšiřovaly *compute capability*. Jelikož je každá verze pevně svázaná s danou architekturou, není možná zpětná kompatibilita. Nyní (prosinec 2012) je poslední stabilní verzí CUDA 5.0.

Zjednodušeně pomocí CUDA programujeme tak, že nejdříve na CPU *host* načteme potřebná data, inicializujeme GPU *device*, překopírujeme data z *host* na *device*, provedeme výpočet a přeneseme výsledek zpět z *device* na *host*. Program pro GPU lze psát pomocí jazyka C, do kterého jsou vloženy rozšiřující funkce. CUDA poskytuje také rozšíření pro

C++ a Fortran a pomocí řešení třetích stran ji lze napojit na Python, Perl, Java, Ruby, Lua, Haskell, MATLAB, IDL.

4.2.3 OpenCL

OpenCL (*Open Computing Language*) [35] je průmyslový standard určený pro paralelní programování různých zařízení. Standard je spravován konsorciem Khronos a první specifikace byla vydána v roce 2008. Nyní (prosinec 2012) je poslední verze 1.2.

Na rozdíl od CUDA není OpenCL závislá na výrobku konkrétní firmy a lze pomocí něj programovat více druhů zařízení, jako jsou GPU (od verze ATI HD řady 4xxx a NVIDIA GeForce řady 8xxx), CPU (s rozšířením SSE3), DSP, mobilní čipy nebo procesory typu Cell. Pro programování daného zařízení se využívá podmnožina standardu jazyka C (C99), do kterého jsou přidána některá rozšíření a omezení. Aby bylo možné použít OpenCL na různých zařízeních, je definován čtyřmi abstraktními modely [35]:

- Model platformy – specifikuje jeden procesor řídící výpočet *host* a jedno nebo více zařízení *device* spouštějící OpenCL kód.
- Exekuční model – definuje, jak je OpenCL prostředí na *host* nastaveno a jak bude spouštěn kód na *device*.
- Paměťový model – definice abstraktní paměťové hierarchie, která je OpenCL využívána.
- Programovací model – popisuje, jak je konkrentní model namapován na fyzické zařízení.

4.2.4 Interoperabilita OpenGL a CUDA

Nyní si ve zkratce popíšeme, jak lze zajistit spolupráci CUDA, která nám něco spočítá a OpenGL, jež umožní vizualizaci dosažených výsledků. To je samozřejmě možné provést i s využitím OpenCL, ale práce se dále bude zabývat pouze technologií CUDA.

Ačkoliv CUDA umožňuje provádění obecných výpočtů na grafické kartě, je GPU pořád primárně určena pro akceleraci vykreslování grafiky za použití např. OpenGL. Proto je v CUDA podpora pro mapování některých prvků OpenGL do jejího adresního prostoru, a tím umožňuje předejít velkým časovým pokutám při přenosu dat mezi CPU a GPU. Mapovat lze textury a buffer objekty. Pro nastavení OpenGL a registraci jejich bufferů v CUDA postupujeme v několika krocích. Nejdříve vytvoříme vykreslovací okno, poté OpenGL kontext a CUDA kontext. Vygenerujeme buffer objekty nebo textury, které budeme sdílet a nakonec je zaregistrujeme v CUDA [36]. Tyto zaregistrované prvky pak lze podle potřeby přidávat nebo odebírat. Po registraci můžeme v daném adresním prostoru GPU provádět výpočty pomocí CUDA. Po ukončení výpočtu a uvolnění prostředků lze k vypočteným hodnotám přistupovat díky OpenGL.

Kapitola 5

Implementace

V této části poměrně podrobně popíšeme implementaci metody photon mapping na GPU a zaměříme se na možnost vykreslování statické scény v několika snímcích za sekundu.

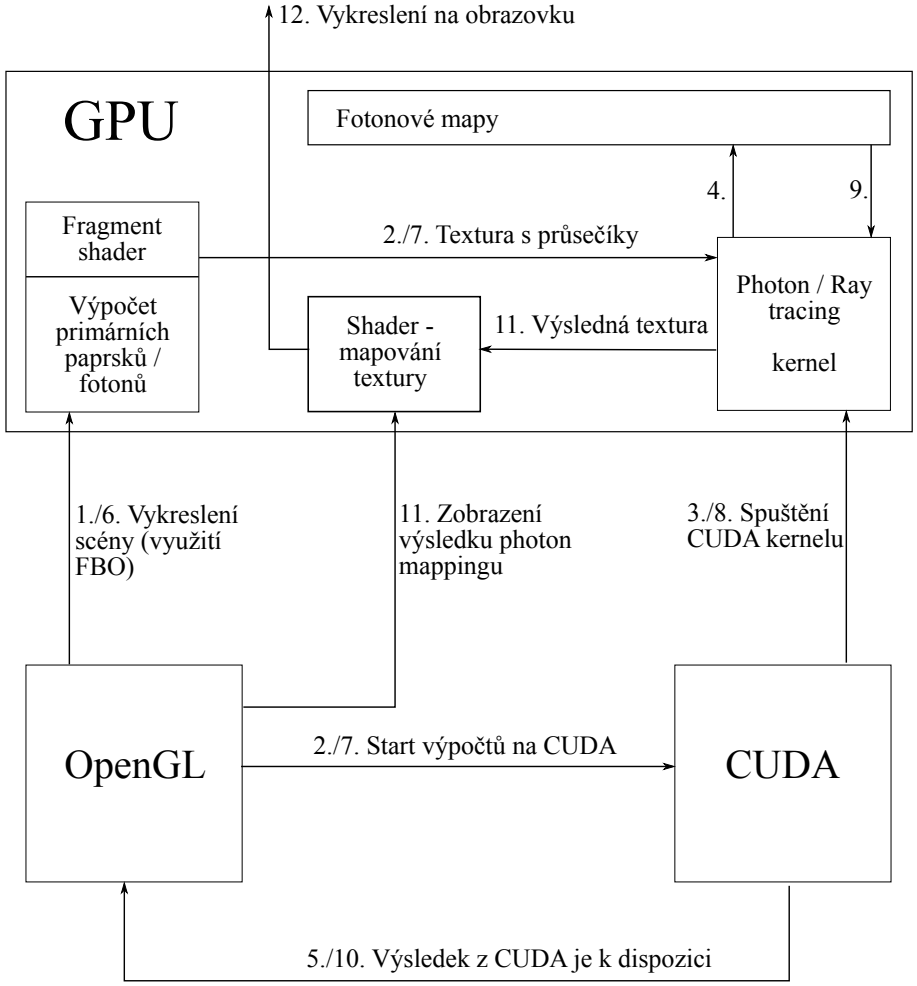
Při implementaci aplikace využijeme spolupráci knihovny OpenGL a CUDA pro výpočet sledování paprsků a sledování fotonů, kdy si průsečíky primárních paprsků, resp. fotonů se scénou předpočítáme pomocí rasterizace OpenGL. Následné sekundární paprsky a fotony se již budou sledovat pomocí CUDA. Jedná se o poměrně jiný přístup k řešení tohoto problémů, neboť OpenGL je primárně určeno pouze k rasterizaci scény a samotné urychlování metody sledování paprsku nebo fotonů se většinou provádí kompletně pomocí CUDA. Příkladem může být třeba knihovna OptiX, což je ray tracer vytvořený společností NVIDIA [37]. Rasterizace pro primární paprsky a fotony přináší značné výhody v rychlosti, ale také má několik úskalí, se kterými se také seznámíme.

Jednou z důležitých funkcionalit OpenGL pro implementaci bude rozšíření *frame buffer objekt* (FBO) [38]. FBO umožňuje vykreslovat obraz do bufferu nebo textury ještě před vykreslením na obrazovku. My FBO využijeme pro zápis potřebných informací do textur, které budeme následně zpracovávat v CUDA.

5.1 Návrh řešení

Jak již bylo zmíněno, klíčovou vlastností aplikace bude spolupráce OpenGL a CUDA znázorněnou na obrázku 5.1, který si nyní blíže popíšeme a v dalším textu se budeme zabývat jednotlivými částmi viděných na tomto obrázku.

Jednotlivé kroky prováděné programem jsou očíslovány a kompletní výpočet osvětlení pomocí photon mappingu probíhá ve dvou iteracích. Nejprve načteme 3D scénu ze souboru, což není z obrázku patrné a inicializujeme OpenGL. Scénu v podobě vrcholů a normál uložíme na GPU tak, aby bylo možné ji zobrazovat pomocí OpenGL a následně inicializujeme FBO, ke kterému připojíme požadované textury, např. pro zápis průsečíku primárních paprsků se scénou. První iterace programu je určena pro sledování fotonů a vytvoření fotonových map. Proto v prvním kroku (1.) provedeme vykreslení scény z pozice světla a ve *fragment shaderu* spočítáme průsečíky primárních fotonů se scénou. Ve druhém kroku (2.) namapujeme textury získané pomocí FBO do adresního prostoru CUDA a zavoláme obslužnou funkci pro spuštění photon tracingu v CUDA. CUDA se postará o provedení výpočtu photon tracingu na GPU pomocí příslušného kernelu (3.). Při výpočtu se využívají průsečíky z textury namapované v předešlém kroku a postupně se naplňují fotonové mapy (4.), pro které je alokován požadovaný prostor v globální paměti GPU. Tyto fotonové mapy



Obrázek 5.1: Návrh implementace photon mappingu na GPU pomocí spolupráce OpenGL a CUDA

jsou perzistentní po celý běh programu, a jelikož je aplikace určena pro vykreslování statických scén, tak je spočítáme pouze jednou. O ukončení výpočtu a vytvoření požadovaných fotonových map je následně informován hlavní program s OpenGL (5.).

Ve druhé iteraci programu provedeme vykreslení scény spolu s osvětlením spočítaným v první iteraci. Vše funguje obdobně jako v prvním případě. Nyní však vykreslíme scénu z pohledu kamery (6.) a opět ve fragment shaderu vypočteme průsečíky primárních paprsků se scénou. Tyto průsečíky uložíme do požadované textury díky FBO, namapujeme texturu do adresního prostoru CUDA a předáme ji výpočet (7.). CUDA spustí kernel pro sledování paprsku (8.). V tomto kernelu použijeme vytvořené fotonové mapy pro odhad osvětlení v daném bodě (9.). Výsledné barvy jednotlivých pixelů se ukládají do výstupní textury a po skončení výpočtu v CUDA je hlavnímu programu oznámeno, že jsou výstupní data připravena v textuře (10.). Výsledný obrázek nakonec vykreslíme pomocí OpenGL a výstupní textury na obrazovku (11., 12.).

5.2 Načítání a reprezentace 3D scény

Jelikož pomocí aplikace budeme renderovat obrázky 3D scén, musíme umět nějak načíst požadované modely. Tyto 3D modely mohou být v několika formátech a pro možnost načtení co největšího počtu formátů využijeme volně dostupnou knihovnu Assimp[1] (*Open Asset Import Library*). Tato knihovna umožňuje zpracovávat formáty jako **3ds**, **obj**, **ply** a mnoho dalších [39]. Její důležitou vlastností je načítání materiálů spolu s modelem.

Model scény se většinou skládá z několika objektů, které jsou reprezentovány trojúhelníkovou sítí. Pro reprezentaci takovýchto objektů je vytvořena menší třída `Mesh`, která uchovává informaci o jednotlivých vrcholech trojúhelníků a jejich normálách spolu s definicí materiálu daného objektu. Vlastnosti materiálu, které třída uchovává jsou:

- Ambientní složka - všudypřítomné světlo, což můžeme použít jako barvu objektu např. při rasterizaci v OpenGL.
- Difúzní složka - difúzní vlastnosti a odraz na povrchu objektu.
- Spekulární složka - zrcadlový odraz objektu.
- Transparentní složka - průhlednost objektu.
- Index lomu - hodnota zalomení světelného paprsku průchodem průhledného objektu.

Ambientní, difúzní a spekulární složky jsou reprezentovány třemi hodnotami pro každý kanál RGB. Transparentní složka, stejně jako index lomu, je pouze jedna hodnota, kdy průhlednost určuje, kolik světla materiálem projde. Složitější materiály, jako třeba textury, nebyly implementovány a zůstávají tak možností pro případné rozšíření aplikace.

5.3 Inicializace OpenGL, FBO a textur

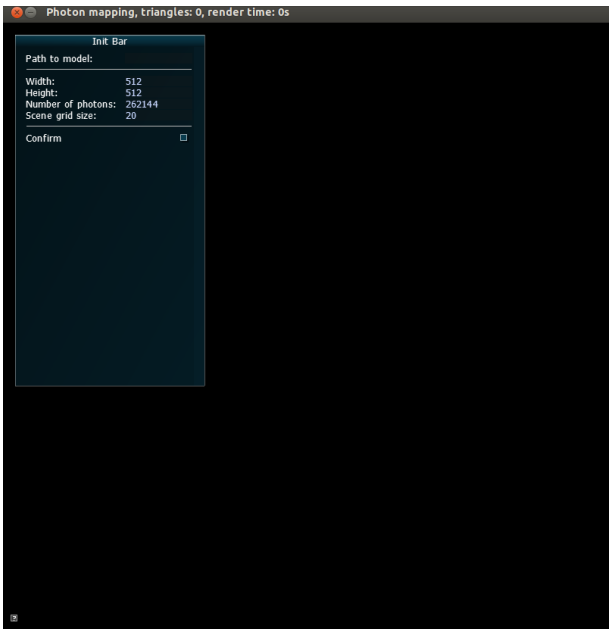
Pro práci s OpenGL je vytvořena třída `OGLRenderer`, která jednoduše zaobaluje celou aplikaci z pohledu vykreslování. Protože OpenGL poskytuje pouze nízkoúrovňové API a tvorba vykreslovacího okna by byla poměrně náročná, necháme tuto práci na volně dostupné knihovně GLFW[2]. Pomocí GLFW vytvoříme OpenGL kontext a zaregistrujeme tzv. „callback“ funkce pro získávání vstupních údajů od uživatele, jako je např. pozice kurzoru myši v okně nebo stlačení klávesy. Aby mohl uživatel interaktivně měnit hodnoty aplikace za běhu, využijeme další volně dostupnou knihovnu AntTweakBar[3]. Za použití této knihovny vytvoříme uživatelské menu a předáme mu předtím zaregistrované callback funkce, čímž tedy vytvoříme možnost pro pohodlné ovládání aplikace. Takto vytvořené okno je zobrazeno na obrázku 5.2.

Výše jsme uvedli, že každý objekt ve scéně má vlastní trojúhelníkovou síť spolu s normálami. Pro uložení daných vrcholů trojúhelníků a jejich normál do paměti grafické karty vytvoříme každému objektu jeho vlastní VBO (*vertex buffer object*). Data do nich nahrajeme v pořadí 3 hodnot reprezentujících daný vrchol a 3 hodnot značících normálu v tomto vrcholu. Informace o materiálu objektu připojíme do vertex shaderu až při vykreslování jednotlivých VBO, a tím pádem nemusí být uloženy s vrcholy trojúhelníků na GPU.

[1] http://assimp.sourceforge.net/index.html
[2] http://www.glfw.org/
[3] http://anttweakbar.sourceforge.net/doc/

Dalším krokem je vytvoření a inicializace FBO. Jedná se o jedno z rozšíření OpenGL a nemusí tak být přítomno na všech grafických kartách. Pro načtení tohoto rozšíření využijeme knihovnu GLEW[4] (*The OpenGL Extension Wrangler Library*). FBO vygenerujeme pomocí funkce `glGenFramebuffers()` a dále vygenerujeme textury pro uložení informací o průsečících, normálách paprsků dopadlých na daný povrch, normálách daného povrchu, difúzních a spekulárních vlastnostech materiálu. Textury mohou uchovávat až 4 složky RGBA a pomocí `GL_RGBA32F` lze do textury zapsat čtyři 32-bitová čísla datového typu `float`. Tuto možnost zvolíme i my a k difúzním vlastnostem materiálu, které jsou reprezentovány třemi hodnotami typu `float`, do té samé textury přidáme hodnotu indexu lomu. Podobně k textuře uchovávající informaci o spekulárních vlastnostech materiálu připojíme informaci o průhlednosti objektu. Takto ušetříme místo, které by bylo potřeba pro alokaci dalších dvou textur.

Vytvořené textury nakonec připojíme k FBO funkci `glFramebufferTexture2D()` a hodnoty `GL_COLOR_ATTACHMENTi` specifikující buffer (texturu), do kterého fragment shader zapisuje. Pro vykreslení hodnot do textury slouží funkce `glDrawBuffers()`.



Obrázek 5.2: OpenGL okno spolu s menu vytvořeném pomocí knihovny AntTweakBar

5.4 Shadery pro výpočet primárních paprsků a fotonů

Pro výpočet primárních paprsků a fotonů byla zvolena možnost rasterizace scény pomocí OpenGL. Do fragment shaderu vstupují interpolované souřadnice rasterizovaných primitiv. Můžeme do něj však předat taky uživatelské hodnoty z vertex shaderu, které se také interpolují podle zpracovávaného primitiva. Této možnosti využijeme a do fragment shaderu pošleme reálné souřadnice vrcholů daného modelu, které jsou interpolovány, čímž získáme průsečíky se scénou. Dále mu předáme pozici kamery nebo světla, normály a materiálové vlastnosti objektu a můžeme začít provádět další výpočty.

[4] http://glew.sourceforge.net/

Nejprve ve fragment shaderu přiřadíme lokace pro zápis hodnot do jednotlivých textur, které jsme si specifikovali výše při inicializaci FBO. Provedeme to pomocí `layout(location = i)`, kde `i` odpovídá číslu v `GL_COLOR_ATTACHMENTi` přiřazenému dané textuře. Operace je naznačena v následující části kódu.

```
// průsečíky se scenou
layout(location=1)out vec4 fragIntersect;
// normály dopadnuvších paprsků
layout(location=2)out vec4 fragHitNormal;
// normály povrchů objektů
layout(location=3)out vec4 fragNormal;
// difúzní vlastnosti materiálu + index lomu
layout(location=4)out vec4 fragDiffuse;
// spekulární vlastnosti materiálu + průhlednost
layout(location=5)out vec4 fragSpecular;
```

Kód 5.1: Namapování výstupních hodnot z fragment shaderu do textur

Po přiřazení lokací zapíšeme do textur interpolované průsečíky se scénou, normálu povrchu v daném průsečíku, difúzní, spekulární a transparentní vlastnosti a index lomu. Směr dopadu paprsku, příp. fotonu lze jednoduše dopočítat z pozice kamery a pozice průsečíku. Barva primárních paprsků a energie fotonů při dopadu na povrch objektu je ovlivněna difúzní složkou materiálu, jedná se tedy o první tři hodnoty uložené v textuře `fragDiffuse`.

Při takovémto výpočtu průsečíku pomocí interpolace ale narážíme na jeden velký problém, který se týká sledování fotonů, alespoň při generování z bodového světla, kterému se budeme věnovat posléze. Jak jsme ukázali v kapitole 2.2.1, fotony ze světla jsou generovány ve směrech s náhodným uniformním rozložením, což však při naší implementaci není možné. Směr primárních fotonů bude vždy ovlivněn rasterizací. Sledování sekundárních paprsků v CUDA bude již probíhat s uniformní pravděpodobností. Znamená to tedy, že primární fotony, které se neodrazí a zůstanou přímo ve fotonové mapě, mohou vytvářet pravidelnou texturu. Tato textura bude viditelná především při menším počtu fotonů a malém poloměru sběrné koule. Ukázku nalezneme v předposlední kapitole spolu s testováním aplikace.

Dalším problémem vztaženým k uniformnímu generování fotonů ze všesměrového světla je samotná krychlová cubemapa. Při uniformním generování fotonů se jejich směr určuje pomocí vzorkování povrchu jednotkové koule. Abychom toto uniformní rozložení fotonů alespoň aproximovali v cubemapě, zmenšíme energii těm fotonům, které leží dále od středu hran cubemapy, jelikož tam jsou fotony po promítnutí na kouli hustěji u sebe. Hodnotu pro zmenšení energie vypočítáme na základě pozice rasterizovaného objektu v *screen space* souřadnicích, což jsou 2D souřadnice vykreslovacího zařízení s počátkem ve středu zařízení a rozsahem hodnot [-1,1]. Výpočet hodnoty, kterou vynásobíme energii fotonu, provedeme ve fragment shaderu následovně:

```
float compesation = abs(cos(screenSpace.x) * cos(screenSpace.y));
```

Kód 5.2: Výpočet hodnoty pro kompenzaci energie fotonů v cubemapě

5.5 Tvorba světel pro generování fotonů

V kapitole Generování fotonů 2.2.1 jsme uvedli několik tvarů světelných zdrojů. Protože jsme pro generování fotonů vybrali OpenGL, není možné všechny tvary dobře modelovat. My se budeme zabývat pouze tvorbou všesměrového a jednosměrového bodového světla, které lze rasterizací simulovat.

5.5.1 Všesměrové bodové světlo

Všesměrové bodové světlo vytvoříme pomocí techniky známé jako „cubemap“ texturování. Jedná se o vytvoření šesti různých čtvercových textur z pohledu světla, které zachycují celou scénu okolo tohoto světla. Princip je takový, že v OpenGL vytvoříme kameru v pozici světla, použijeme perspektivní projekci a nastavíme FOV (*field of view*) na 90° a postupně měníme směr pohledu kamery a pořizujeme textury, celkově tedy šest. Tyto textury uložíme ve dvou řadách po třech do jedné velké textury, zvané též atlas, která je obdélníkového tvaru a její obsah je přibližně roven počtu fotonů (přibližně proto, protože při výpočtu délek stran obdélníku se využívá zaokrouhlování na celočíselné hodnoty velikosti textury). Pro výpočet velikosti stran atlasu ze zadaného počtu fotonů slouží třída `Counter` implementující jednoduché počítadlo, které využijeme ještě při tvorbě více světel ve scéně. Jednotlivé čtvercové textury do atlasu zapíšeme na správnou pozici pomocí funkce

```
glViewport(x, y, sirka, vyska);
```

Kód 5.3: Funkce pro mapování textur do atlasu

kde `x` a `y` udává počáteční pozici a `sirka`, `vyska` jsou stejné a udávají délku strany čtvercové textury. Výsledná podoba atlasu je naznačena na obrázku 5.3.

Levá strana	Vepředu	Pravá strana
Vzadu	Dole	Nahoře

Obrázek 5.3: Pozice jednotlivých stěn cubemapy v atlasové textuře

5.5.2 Směrové bodové světlo

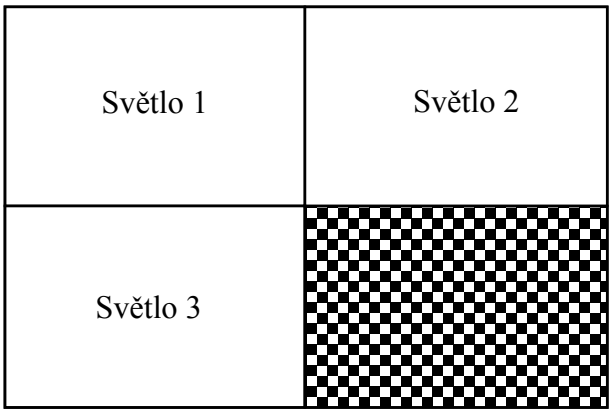
Druhým typem světla, které v aplikaci půjde využít, je směrové světlo. U tohoto světla není potřeba vytvářet cubemapu, ale stačí nastavit směr pohledu kamery v pozici světla a scénu vykreslit. Při vykreslování použijeme ortogonální projekci. Textura, kterou pomocí vykreslování vytvoříme, bude opět obdélníková a výpočet jejich stran proběhne stejně jako u atlasu pro všesměrové světlo. Osvětlení tedy bude připomínat hranol se čtvercovou nebo

obdélníkovou podstavou, což záleží na volbě rozměrů světla. To, že je výpočet velikosti textury, do které se světlo uloží, stejný jako pro všesměrové světlo, má své opodstatnění při vkládání více světel do scény.

5.5.3 Více světel ve scéně

V zobrazované scéně může být několik světel, proto je potřeba nějak efektivně světla spravovat. Přestože mohou mít světla různou energii, která se modeluje pomocí množství fotonů vygenerovaných daným světlem, tak my se omezíme pouze na světla se stejnou energií, tudíž každé světlo bude produkovat stejný počet fotonů. U různých energií bychom museli využívat různě velké textury, což by podstatně ztížilo výpočty offsetů pro uložení jednotlivých textur do jedné velké textury (atlasu), která bude mapována do CUDA.

V již zmíněné třídě `Counter` se uchovávají informace o všech vytvořených texturách. Světla se ve scéně nastavují před začátkem výpočtu osvětlení a jakmile jsou všechna světla nastavena, nelze již jejich počet měnit. Po nastavení světel se tedy velikost hlavní textury, ve které budou uloženy všechny textury od jednotlivých světel, přepočítá. Příklad výsledné textury je naznačen na obrázku 5.4, kde výsledná textura obsahuje 3 světla. Šachovnicovým vzorem je naznačena oblast, která neobsahuje žádné světlo, s čímž je pak nutné počítat při zpracování textury pomocí CUDA. Informaci o tomto prázdném místě udržuje počítadlo.



Obrázek 5.4: Celkově 3 světla zabalené v jedné textuře (atlasu)

5.6 Vytvoření interoperability mezi OpenGL a CUDA

Základní koncept spolupráce OpenGL a CUDA jsme naznačili v kapitole 4.2.4. Nejdříve je potřeba po inicializaci OpenGL kontextu vytvořit CUDA kontext, což zajišťuje funkce `cudaGLSetGLDevice()`. Tato funkce taky inicializuje zařízení s CUDA tak, aby sdílelo prostředky s OpenGL. Dále je potřeba v CUDA zaregistrovat textury, které jsme vytvořili, viz 5.3. Využijeme funkci `cudaGraphicsGLRegisterImage()`, které předáme ukazatel na danou texturu a hodnotu, zda bude textura jen pro čtení nebo pro zápis. Pro zápis inicializujeme výstupní texturu, do které bude v CUDA vykreslena závěrečná scéna i s vypočítaným osvětlením.

Nakonec do CUDA namapujeme výše zaregistrované prostředky použitím funkce `cudaGraphicsMapResources()` a pomocí `cudaGraphicsSubResourceGetMappedArray()` zajistíme odkaz na `cudaArray`. Pro čtení hodnot z textur musíme provést ještě poslední

krok, a to je mapování `cudaArray` do texturovací paměti CUDA, což uděláme díky funkci `cudaBindTextureToArray`. Ukazatel na texturu je v globálním prostoru zdrojového CUDA kódu a jeho formát je:

```
texture<float4, cudaTextureType2D, cudaReadModeElementType> refTex;
```

Kód 5.4: Formát odkazu na texturovací paměť

kde datový typ `float4` značí, že budeme používat čtyři složky textury zmíněné výše, `cudaTextureType2D` specifikuje dimenzi textury a `cudaReadModeElementType` říká, že při čtení z textury nebude probíhat žádná konverze hodnot.

5.7 Implementace kernelů pro photon mapping v CUDA

Po popisu OpenGL části programu přistoupíme k vytváření kernelů pro paralelní výpočty v CUDA. Jako výpočetní rozhraní této architektury byla zvolena verze 2.0. Aplikace tak nemusí být zpětně kompatibilní s nižší verzí. CUDA kód se nachází ve vlastním souboru s příponou `.cu` a není možné jej psát jinde.

Protože budeme pracovat pouze se statickou scénou, nejdříve pomocí `cudaMalloc()` na GPU alokujeme místo pro celou scénu, tzn. pro jednotlivé vrcholy trojúhelníků následované jejich normálami. Poté data nahrajeme do globální paměti GPU (`cudaMemcpy()`), kterou namapujeme do texturovací paměti. Přístup k datům přes texturovací paměť je o něco rychlejší než přístup z globální paměti.

Podobně jako scénu uložíme do paměti GPU i vytvořenou strukturu s vlastnostmi materiálů jednotlivých objektů scény. Jelikož se jedná o strukturu, není možné ji namapovat do texturovací paměti, která je striktně omezena jen pro jednoduché datové typy jako je `integer`, `float` a 1, 2, 3 nebo 4 složkové vektorové typy.

5.7.1 Dělení prostoru pomocí uniformní mřížky

V kapitole 3 jsme ukázali některé struktury pro dělení prostoru. Tři z těchto struktur jsou založeny na binárních vyhledávacích stromech. Tyto stromy se většinou na CPU jednoduše sestaví pomocí rekurze a rekurze se využívá i pro průchod stromy. Podpora rekurze byla do architektury CUDA přidána poměrně nedávno a většina autorů, např. Popov [40], se zabývá stavbou stromu na CPU a pak nerekurzívní procházení na GPU. Stavba stromu přímo na GPU, popsána např. Zhouem [41], je již značně složitější.

Pro naši implementaci jsme zvolili uniformní mřížku, která má poměrně jednoduchou implementaci na GPU a její výstavba je velmi rychlá. Značně upravená implementace vychází z článku Koljanova [42], který se zabývá právě stavbou uniformní mřížky na GPU.

Algoritmus paralelní stavby uniformní mřížky

Při stavbě mřížky je potřeba si nejdříve uvědomit, že trojúhelníky mohou patřit do více buněk. Jelikož není možné na GPU dynamicky alokovat paměť a my potřebujeme alokovat dostatečné místo pro odkazy trojúhelníků na buňky, musíme nejdříve spočítat počet těchto odkazů a až pak přistoupit k samotné tvorbě mřížky.

Pro uniformní mřížku postavenou na GPU potřebujeme dvě pole ukazatelů. Jedno pole je pro uložení počátku a konce dané buňky, neboť k jedné buňce může být přiřazeno několik trojúhelníků a druhé pole slouží pro ukazatele na jednotlivé trojúhelníky přiřazené

Algoritmus 1 Stavba uniformní mřížky pro scénu složenou z trojúhelníků. Kernel je značen pomocí <<<>>>.

1: $r \leftarrow$ `spočítej rozměry mřížky`
2: $p \leftarrow$ `alokuj paměť o velikosti počtu trojúhelníků`
3: $t \leftarrow$ `ukazatel na paměť se scénou`
4: $U \leftarrow$ `počet ukazatelů jednotlivých trojúhelníků na buňku<<<>>>(r, p, t)`
5: $c \leftarrow$ `redukce<<<>>>(U)` {celkový počet ukazatelů}
6: $pb \leftarrow$ `alokuj paměť pro ukazatele na buňky o velikosti` c
7: $pt \leftarrow$ `alokuj paměť pro ukazatele na trojúhelníky o velikosti` c
8: $B \leftarrow$ `přiřaď trojúhelníky do buněk<<<>>>(pb, pt, r, t)`
9: $S \leftarrow$ `seřaď ukazatele na buňky<<<>>>(B)`
10: $pzk \leftarrow$ `alokuj paměť o velikosti` $2r$ `pro uložení začátku a konce buněk`
11: $ZK \leftarrow$ `spočítej začátek a konec jednotlivých buněk<<<>>>(S, pzk)`

k buňce určené svým počátkem a koncem. Vytvoření takovéto uniformní mřížky je popsáno algoritmem 1, který si blíže vysvětlíme.

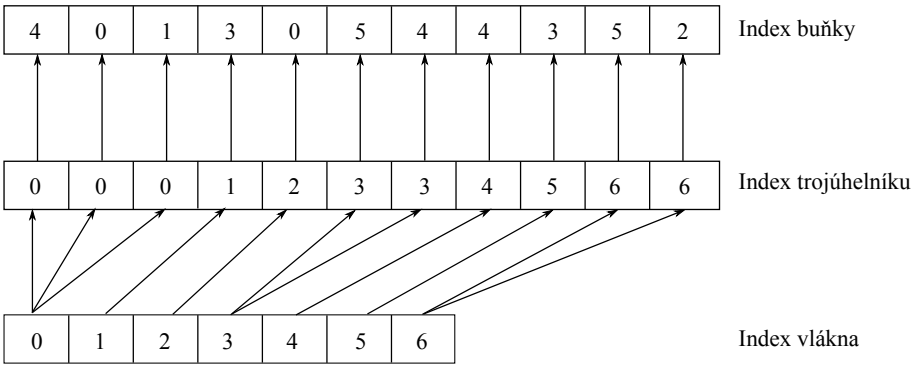
V prvním kroku spočítáme rozměry mřížky ze zadané obálky scény a velikosti jedné buňky, kterou určuje uživatel. Následně alokujeme paměť na GPU o velikosti celkového počtu trojúhelníků ve scéně. Do paměti si uloží každý trojúhelník počet buněk, které překrývá. Tento výpočet je proveden pomocí kernelu uvedeného v kroku 4. Kernel je spuštěn s počtem vláken odpovídajícím počtu trojúhelníků ve scéně a jedno vlákno tak reprezentuje jeden trojúhelník. Každé vlákno spočítá ohraničující obálku pro svůj trojúhelník a nalezne odpovídající buňky, které se s touto obálkou překrývají. Pro přesnější určení, zda se trojúhelník překrývá s nalezenými buňkami, využijeme metody SAT popsanou v kapitole 3.5. Implementace SAT je podobná jako na CPU s tím rozdílem, že vyhodnocení o tom, zda trojúhelník překrývá buňku nebo ne, provedeme až na konci. Tím předejdeme divergenci vláken v místě, kde by byla nalezena separační osa a další vyhodnocování by bylo ukončeno. Vlákna tedy zapíšou na dané místo v globální paměti počet buněk, které trojúhelník překrývá. Pokud by všechny trojúhelníky překrývaly stejný počet buněk, byl by přístup do globální paměti zarovnaný, neboť nalezení odpovídajících buněk pomocí ohraničujících obálek je realizován operacemi sčítání a odčítaní a funkcemi `min()`, `max()`. V drtivé většině scén tomu tak ale nebude.

Následuje krok 5, kdy je pomocí redukce na GPU spočítán celkový počet ukazatelů získaných v předešlém kernelu. Pomocí metody redukce se paralelně provádí součet všech hodnot v poli, čímž nakonec vznikne jedna hodnota. Pro výpočet redukce využijeme knihovny Thrust[5], která realizuje mnoho vysoce optimalizovaných paralelních algoritmu, jež se často používají při programování na GPU.

V 6 a 7 kroku alokujeme místo pro ukazatele na buňky a pro trojúhelníky, které na tyto buňky ukazují. Spustíme kernel z kroku 8 opět s počtem vláken stejným jako počet trojúhelníků. Každé vlákno hledá překryv svého trojúhelníku s buňkami obdobně, jak bylo popsáno výše. Avšak nyní vlákno zapisuje na více pozic globální paměti. Do paměti pro ukazatele na trojúhelníky zapíše index svého trojúhelníku tolikrát, kolik buněk překrývá a do paměti pro ukazatele na buňky vloží právě index dané buňky. Vše je možné vidět na obrázku 5.5.

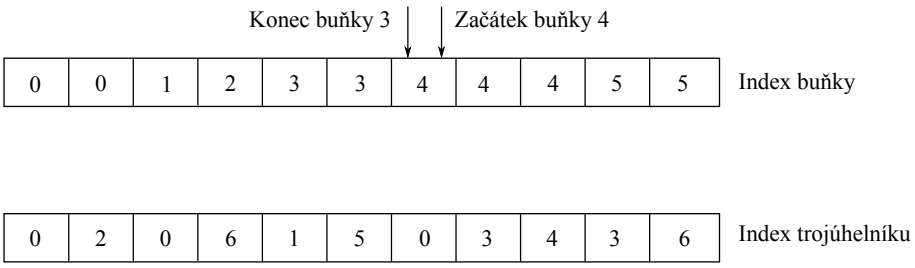
Poslední částí algoritmu je krok 9, 10 a 11. Ukazatele na buňky a trojúhelníky získané v předchozím kroku seřadíme podle indexu buněk. Jedná se tedy o tzv. hash tabulku s klíčem

[5] http://code.google.com/p/thrust/



Obrázek 5.5: Tvorba hash tabulky pro uniformní mřížku

indexu buňky a hodnotou reprezentovanou indexem trojúhelníku. K řazení využijeme opět knihovny Thrust umožňující rychlé řazení na GPU. Poté alokujeme pole pro ukazatele na začátek a konec všech buněk mřížky a vynulujeme jej. Nalezení počátku a konce buněk provedeme v kernelu 11, který je spuštěn s počtem vláken rovnajícím se počtu odkazů na buňky v seřazeném poli. Každé vlákno si načte hodnotu z paměti a dále následující hodnotu v paměti. Pokud se tyto dvě hodnoty liší, jedná se o konec buňky, viz obrázek 5.6. Buňky, které neobsahují ukazatel na žádný trojúhelník, mají počátek a konec nulový.



Obrázek 5.6: Seřazení hash tabulky podle indexu buněk a naznačení získání počátku a konce dané buňky

Uvedeným postupem jsme na GPU vytvořili uniformní mřížku pro celou scénu. Mřížka je nyní reprezentována polem s počátkem a koncem jednotlivých buněk a k němu seřazeným polem indexů trojúhelníků. Trojúhelníky, které leží v dané buňce, získáme procházením pole indexů trojúhelníků od pozice určené začátkem buňky až po pozici danou koncem buňky. Mřížka pro fotony se sestaví podobně, s tím rozdílem, že není potřeba spouštět kernel 4 a 5, protože foton bude patřit vždy pouze do jedné buňky.

Procházení uniformní mřížky pomocí 3D-DDA

V kapitole 3.4 jsme ukázali efektivní algoritmus na procházení uniformní mřížky. Algoritmus na GPU naimplementujeme obdobně jako na CPU. Avšak taková implementace obsahuje mnoho podmínek `if`, kde vzniká velká divergence vláken. Tuto divergenci lze efektivně vyřešit nahrazením všech podmínek logickými operacemi `and` a `or`. Vlákna tak budou muset zpracovat všechny příkazy ve funkci realizující 3D-DDA algoritmus. Jedná se však pouze o několik rychlých aritmetických a logických operací. Část původního a upraveného kódu je naznačena v 5.5, resp. 5.6.

```
if (tMax.x < tMax.y)
{
  if (tMax.x < tMax.z)
  {
    x = x + stepX;
    if (x == outX) return false;
    tMax.x += tDelta.x;
  }
  else
  {
    z~= z~+ stepZ;
    if (z~== outZ) return false;
    tMax.z += tDelta.z;
  }
}
```

Kód 5.5: Ukázka části kódu se značnou divergencí vláken

```
mult = (tMax.x < tMax.y);
mult2 = (tMax.x < tMax.z);

x = (x + stepX*mult*mult2);
mult3 = (x == outX && mult && mult2);
isEnd |= mult3;
tMax.x += (tDelta.x)*mult*mult2;

z~= (z~+ stepZ*mult*(!mult2));
mult3 = (z~== outZ && mult && !mult2);
isEnd |= mult3;
tMax.z += (tDelta.z)*mult*(!mult2);
```

Kód 5.6: Ukázka části kódu již bez divergence vláken

5.7.2 Photon tracing na GPU

Další částí implementace je sledování fotonů pomocí CUDA. Pro sledování využijeme textury se světly, jejichž tvorbu jsme popsali v kapitole 5.5. Celkem se jedná o pět textur obsahujících:

- Průsečíky primárních fotonů se scénou.
- Normalizovaný směr dopadu fotonu v místě průsečíku.
- Normalizovanou normálu povrchu v daném průsečíku.
- Difúzní vlastnosti materiálu spolu s indexem lomu.
- Spekulární vlastnosti materiálu spolu s hodnotou průhlednosti.

Textury namapujeme do texturovací paměti GPU tak, jak jsme uvedli v kapitole o interoperabilitě.

Poté alokujeme na GPU místo pro globální fotonovou mapu a mapu kaustik. Velikost těchto map musí být známá ještě před samotným sledováním, protože předem nevíme, kolik fotonů bude do map uloženo a na GPU nelze dodatečně alokovat další místo. Velikost map

je tedy dána počtem fotonů vygenerovaných jednotlivými světly, který lze získat z objektu třídy `Counter` vytvořeného při modelování světel v OpenGL. Struktura jednotlivých fotonů v mapě je následující:

```
typedef struct
{
    float3 position; // pozice fotonu
    float3 direction; // směr dopadu v~dané pozici
    float3 energy; // energie fotonu
    float3 surfaceNormal; // normála povrchu v~místě dopadu
} cuPhoton;
```

Kód 5.7: Struktura fotonu

Generátor náhodných čísel

Jelikož pro rozhodování o dalším sledování fotonu využijeme ruské rulety a difúzního odrazu, který má směr daný uniformním rozložením pravděpodobnosti na základě normály povrchu, potřebujeme generátor náhodných čísel. V každém vlákně tedy vytvoříme jednoduchý kongruentní generátor náhodných čísel, jehož vstupem je počáteční semínko (*seed*) a výstupem je náhodné číslo v intervalu $< 0, 1 >$. Semínko musí být pro každé vlákno unikátní, aby generátor negeneroval stejné hodnoty pro různá vlákna. Musíme proto počáteční hodnoty vygenerovat ještě před spuštěním kernelu a uložit do globální paměti. Vygenerování lze provést klasicky na CPU, což ale pro velký počet fotonů může být značně pomalé. Z tohoto důvodu využijeme opět knihovny Thrust, která umožňuje vygenerovat náhodná čísla přímo na GPU. Jelikož takto vygenerovaná náhodná čísla jsou již uložena v globální paměti GPU, stačí kernelu pro sledování fotonů předat pouze ukazatel na tuto paměť a vyhneme se velkým ztrátám výkonu při kopírování hodnot z RAM paměti na GPU. Vlákna si tyto počáteční hodnoty uloží do jednoho registru. Jelikož vlákno může potřebovat spočítat náhodné číslo několikrát (ruská ruleta, difúzní odrazy), bude doba přístupu k semínku v registru menší, než by tomu bylo u globální paměti. Po výpočtu náhodného čísla pomocí kongruentního generátoru se semínko v registru aktualizuje.

Implementace kernelu pro photon tracing

Vytvořený kernel pro sledování fotonů má 2D dimenzi a je spouštěn pro každé světlo v textuře zvlášť. Jedno vytvořené vlákno odpovídá jednomu fotonu, který bude sledovat. Každé vlákno si načte hodnoty ze všech uvedených pěti textur a uloží si je do registru. Načtení lze obstarat funkcí `tex2D`. Přístup do texturovací paměti bude rychlý, pokud nebude docházet k výpadkům v texturovací cache.

V ukázce algoritmu 2 je naznačena implementace kernelu pro photon tracing. Po načtení potřebných dat z textur vytvoříme strukturu fotonu, který budeme sledovat. Energie tohoto fotonu je na počátku rovná 1. Následuje získání náhodného čísla ξ a určení průměrných hodnot $\Delta d, \Delta s, \Delta t$ z vlastností materiálu pro ruskou ruletu. Na základě těchto hodnot zjistíme, jak se bude foton dále sledovat, anebo jestli sledování ukončíme.

Příklad sledování vidíme na difúzním odrazu. Pokud má být foton difúzně odražen, spočítáme nejdříve směr jeho náhodného odrazu a zmenšíme jeho energii o průměrné Δd. Poté vyhledáme průsečík tohoto fotonu se scénou uloženou v uniformní mřížce pomocí výše popsaného algoritmu 3D-DDA. Pokud není průsečík nalezen, uloží vlákno foton do globální

paměti představující jak globální mapu fotonu, tak mapu kaustik. Energie tohoto fotonu bude záporná, čímž určíme, že při stavbě mřížky nebudeme s tímto fotonem pracovat. Je-li průsečík nalezen, aktualizuje se normála povrchu a vlastnosti materiálu podle vlastností aktuálního materiálu v místě průsečíku. Obdobně funguje sledování zrcadlově odraženého nebo zalomeného fotonu.

V posledním kroku algoritmu dojde k uložení fotonu do jedné ze dvou fotonových map podle toho, na jakém povrchu se v průběhu sledování odrazil. Pokud byl alespoň jednou zrcadlově odražen nebo zalomen díky průchodu průhledným materiálem, tak je uložen do mapy kaustik, jinak do globální mapy. K fotonu přidáme informaci o normále povrchu, na kterém jeho sledování končí. Pomocí této normály pak při sledování paprsku a výpočtu zářivé energie určíme, zda foton patří danému povrchu.

Již na první pohled má sledování fotonů pomocí algoritmu 2 několik nedostatků, které v podstatě nelze efektivně vyřešit. Hlavním problémem bude vysoká divergence vláken při rozhodování ruské rulety. Dále pak smyčka `while`, protože ta má u každého vlákna jiný počet iterací, který je opět ovlivněn ruskou ruletou a ukončením sledování fotonu. Zápis do globální paměti je tak značně chaotický a propustnost této paměti je drasticky snížena. Dalším významným ničitelem výkonu je náhodné odrážení fotonu a následné testování na průsečík se scénou. Vlákna totiž přistupují k různým buňkám uniformní mřížky, které mohou obsahovat různý počet trojúhelníků a doba testování se tedy výrazně liší (vadí to především v rámci vláken jednoho warpu).

5.7.3 Sledování paprsku na GPU s využitím fotonových map

Ještě před samotným sledováním paprsku je potřeba vytvořit uniformní mřížku nad fotonovými mapami. Vytvoření mřížky pro scénu je popsáno v kapitole 5.7.1 a její tvorba pro fotony je podobná, což je v této kapitole taky naznačeno. Fotony jež mají zápornou hodnotu se do mřížky neuloží, ale fyzicky ve fotonových mapách stále budou. Velikost jedné buňky mřížky se rovná hodnotě poloměru sběrné koule a lze ji v aplikaci interaktivně měnit uživatelem. Dále alokujeme místo v globální paměti GPU pro uložení hodnot do výstupní textury, na kterou máme odkaz získaný pomocí interoperability. Jakmile je mřížka a výstupní paměť připravena, můžeme přejít k samotnému sledování paprsku.

Pro sledování využijeme úplně stejné textury jako v případě photon tracingu, celkově tedy pět textur. V texturách jsou informace pouze o scéně z jedné kamery a počet vláken 2D kernelu se rovná rozměrům textury, tedy šířka $\times$ výška. Každé vlákno si opět načte informace z textur do registrů. Na rozdíl od klasického sledování paprsku na CPU, které je rekurzívní, budeme pracovat pouze s jedinou vlastností materiálu. Materiál jednoho objektu tak může být pouze difúzní, spekulární nebo průhledný. Pro ukázku funkčnosti photon mappingu to bude dostačující.

Implementace kernelu sledování paprsku

Algoritmus 3 popisuje kernel vytvořený pro sledování paprsku. Po načtení hodnot z textur provedeme sběr fotonů v místě průsečíku a určíme zářivou energii plochy. Ke sběru využijeme kouli, jejíž poloměr zadal uživatel. V uniformní mřížce s fotony nalezneme požadovanou buňku, ve které leží daný průsečík se scénou. Jelikož je poloměr sběrné koule stejně velký jako hrana jedné buňky, vybereme celkem 27 buněk ležících kolem místa průsečíku, do kterých může sběrná koule zasahovat. Výběr je naznačen na obrázku 5.7. Pokud bychom postavili mřížku s délkou hrany jedné buňky $2r$, tedy o velikosti průměru sběrné koule, vybrali bychom pouze 8 buněk, do kterých by mohla koule zasahovat. Problémem však je,

Algoritmus 2 Kernel pro photon tracing na GPU.

```
p ← průsečík z textury
n ← normála povrchu z textury
hn ← směr dopadu fotonu z textury
d ← difúzní vlastnost materiálu z textury
s ← spekulární vlastnost materiálu z textury
i ← index lomu z textury
t ← průhlednost materiálu z textury
foton ← p, n, hn, energie(1)
loop
  ξ ← náhodné číslo ∈ [0..1]
  Δd ← průměr RGB složek d
  Δs ← průměr RGB složek s
  Δt ← t
  if ξ ∈ [0, Δd] then
    // difúzní odraz fotonu
    foton ← p, náhodný směr odrazu(n), energie(energie/Δd)
    if !(3DDDA(foton, scena)) then
      // průsečík se scénou nenalezen
      foton ← energie(−1)
      f_mapa ← foton
      break;
    end if
    aktualizuj(n, d, s, i, t)
  else if ξ ∈ [Δd, Δd + Δs] then
    // zrcadlový odraz fotonu
  else if ξ ∈ [Δd + Δs, Δd + Δs + Δt] then
    // zalomení fotonu průchodem průhledného povrchu
  else
    // uložení fotonu do mapy
    foton ← energie(energie ∗ d) / celkový počet fotonů
    foton ← normála povrchu
    f_mapa ← foton
    break;
  end if
end loop
```

že pro poloměr sběrné koule r je objem těchto 8 buněk větší než objem výše uvedených 27 buněk, viz rovnice 5.1 a 5.2. Z toho vyplývá, že pro stejnou scénu bude počet fotonů větší v 8 buňkách a při sběru všech těchto fotonů by bylo potřeba více přístupů do globální paměti s fotony, čímž by došlo k prodloužení výpočtu. Zda tomu tak je, ukážeme v kapitole 6.

$$o_8 = (2r + 2r)^3 = 64r^3 \tag{5.1}$$

$$o_{27} = (r + r + r)^3 = 27r^3 \tag{5.2}$$

Z vybraných 27 buněk tedy sesbíráme všechny fotony, ale k výslednému výpočtu energie budou přispívat pouze ty, které leží ve vzdálenosti od průsečíku p menší než r. Pokud bychom při určování vzdálenosti používali podmínky `if`, vznikala by při sběru fotonů divergence vláken. Této divergenci předejdeme tak, že výsledek porovnání vzdálenosti fotonu a poloměru koule uložíme do proměnné. Proměnná může nabývat hodnoty 0, pokud je foton od průsečíku vzdálen více, než je poloměr koule, resp. 1, pokud je foton uvnitř sběrné koule. Následně touto hodnotou vynásobíme energii daného fotonu a přičteme k celkové zářivé energii. Abychom do odhadu zářivé energie plochy nezapočítali i fotony, které danému objektu v místě průsečíku nepatří, využijeme normálu povrchu, kterou jsme u photon tracingu uložili spolu s fotonem. Pokud tato normála svírá s normálou v místě průsečíku větší úhel, než je nastavený práh, pak foton náleží jinému povrchu a není do výsledného odhadu započítán. Rozdíl mezi sběrem bez a s touto modifikací je patrný na obrázku 5.8.

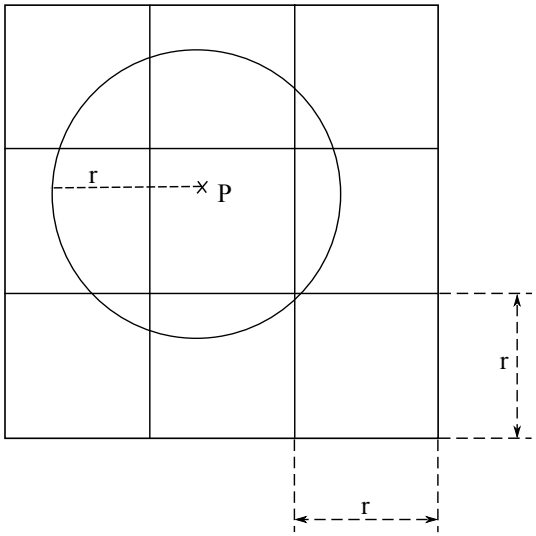
Sledování sekundárních paprsků, tedy odražených nebo zalomených, je prováděno ve smyčce `while`. Pro paprsek hledáme průsečík se scénou pomocí 3D-DDA procházení uniformní mřížky s trojúhelníky. Je-li průsečík nalezen, provedeme odhad zářivé energie stejně, jak bylo popsáno v předchozím odstavci. Není-li průsečík nalezen, nastavíme barvu paprsku na barvu pozadí. Po ukončení sledování každé vlákno uloží vypočítanou barevnou hodnotu svého paprsku do daného místa v globální paměti. Tímto získáme výslednou texturu, kterou můžeme po ukončení kernelu a uvolnění prostředků zobrazit v OpenGL.

Podobně jako u photon tracingu obsahuje smyčka `while` několik podmínek `if`, na kterých může docházet k divergenci kódu. Avšak u kompletně difúzních scén je dobrá pravděpodobnost, že paprsky reprezentované vlákny budou v rámci warpu ukončovány současně. Je to dáno tím, že v textuře již máme vypočítané průsečíky a pokud warp zpracovává průsečíky ležící ve stejné buňce patřící jednomu objektu (trojúhelníku), pak všechna vlákna warpu sesbírají fotony ze stejných buněk. Díky tomu se sníží počet větvení kódu, a taky počet přístupů do globální paměti. Při zvyšujícím se rozlišení obrazu se zmenšuje prostorový úhel svíraný paprsky a pravděpodobnost společného průchodu je pak ještě větší. Je proto možné, že čas výpočtu neporoste lineárně se zvyšujícím se rozlišením obrazu, což ověříme v následující kapitole.

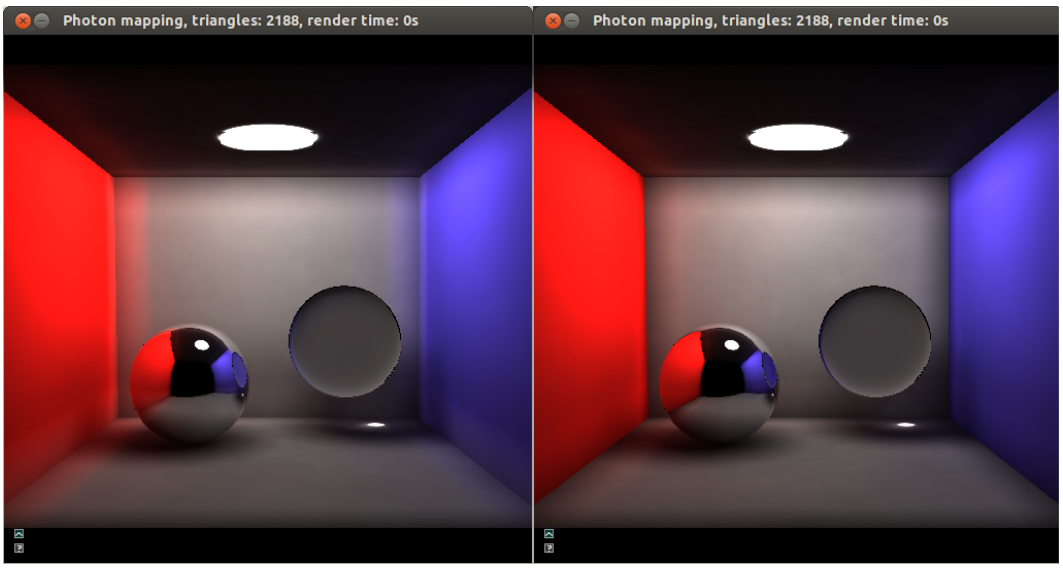
Výše uvedené předpoklady již úplně neplatí pro scénu se zrcadlovými nebo průhlednými objekty. Směr odrazu nebo lomu paprsku na takovém objektu je již dost náhodný, stejně jako průchod takových paprsků scénou. Nastává tedy různé větvení kódu pro vlákna stejného warpu, které tak čtou jiný počet fotonů z různých buněk.

Algoritmus 3 Kernel pro sledování na GPU.

```
p ← průsečík z textury
n ← normála povrchu z textury
hn ← směr dopadu paprsku z textury
d ← difúzní vlastnost materiálu z textury
s ← spekulární vlastnost materiálu z textury
i ← index lomu z textury
t ← průhlednost materiálu z textury
r ← poloměr sběrné koule
rad ← spočítej zářivou energii plochy(p, r, f_mapy)
barva ← 0
while dokud není dosažena maximální hloubka do
  inkrementuj hloubku
  barva = barva + rad
  if materiál má pouze difúzní složku then
    break;
  end if
  if materiál je spekulární then
    hn ← spočítej směr odrazu paprsku (hn, n)
    if (3DDDA(paprsek, scena)) then
      // průsečík se scénou existuje
      rad ← spočítej zářivou energii plochy(p, r, f_mapy)
      aktualizuj(n, d, s, i, t)
    end if
  else
    // materiál je průhledný
    hn ← spočítej směr zalomení paprsku (hn, n, i)
    if (3DDDA(paprsek, scena)) then
      // průsečík se scénou existuje
      rad ← spočítej zářivou energii plochy(p, r, f_mapy)
      aktualizuj(n, d, s, i, t)
    end if
  end if
end while
pixel ← barva
```



Obrázek 5.7: Výběr 27 buněk, které protíná sběrná koule



Obrázek 5.8: Na prvním obrázku je ukázka chybně sesbíraných fotonů patřících jinému povrchu, což je patrné na hranách boxu. Na druhém obrázku jsou fotony filtrovány podle normály povrchu

Kapitola 6

Testování a výsledky

V této kapitole provedeme a popíšeme testování vytvořené aplikace. Zaměříme se na otestování rychlosti generování primárních paprsků a fotonů, stavby uniformní mřížky, sledování fotonů a sledování paprsku spolu s odhadem globálního osvětlení pomocí fotonových map. Také se podíváme na to, zda předpoklady uvedené v předchozí kapitole platí.

K testování využijeme známých modelů používajících se k demonstraci globálního osvětlení scény:

- Cornell box - pravděpodobně nejznámější scéna pro testování osvětlení. Jedná se o krabici s plošným zdrojem světla umístěným ve stropě a dvě z jejich stěn mají různou barvu. Krabice může obsahovat různé objekty pro potřeby testování. My do ní budeme postupně vkládat model kvádru, koule, stanfordského králíka nebo stanfordského draka. Jednotlivé modely se liší počtem trojúhelníků a vlastnostmi materiálu.
- Sponza atrium - vymodelované nádvoří paláce v Dubrovníku s podloubím a množstvím podpěrných sloupů. Scéna je obzvlášť vhodná k testování nepřímého osvětlení.
- Conference room - model skutečné konferenční místnosti se stolem a židlemi kolem tohoto stolu a kolem stěn. V originále obsahuje model i několik zrcadlově odrazivých materiálu, ale my využijeme upravenou scénu s pouze difúzním materiálem.

Všechny tyto modely byly staženy z archívu Morgana McGuirea [43]. Informace o počtu trojúhelníků, počtu vrcholů a vlastnostech materiálů jednotlivých testovacích scén jsou obsaženy v tabulce 6.1. Každá testovací scéna bude obsahovat jedno všesměrové bodové světlo. Toto světlo se v Cornell boxu nachází ve středu vrchní stěny, ve scéně s modelem Sponzy je umístěno uprostřed pod horním okrajem celé stavby a konferenční místnost jej obsahuje ve středu stropu, viz obrázek 6.2.

K měření časových údajů využijeme vysoce přesného časovače `glfwGetTime` poskytovaného knihovnou GLFW. Aplikaci otestujeme na stejném desktopovém PC, na kterém byla vytvořena. Specifikace tohoto stroje jsou následující:

- Dvoujádrový procesor G860 Intel, 3GHz
- Operační paměť 4GB DDR3 RAM
- Grafická karta NVIDIA GT 640 (architektura Kepler, 384 CUDA jader, 2GB DDR3 RAM, 128 bitová sběrnice)
- Operační systém Linux Ubuntu 12.04.2 LTS

Scéna	Počet trojúhelníků	Počet vrcholů	Výskyt materiálů ve scéně		
			Difúzně odrazivé	Zrcadlově odrazivé	Průhledné
Cornell + 2 krabice	42	40	ano	ne	ne
Cornell + 2 koule	2188	1116	ano	ano	ano
Cornell + králík	65416	32814	ano	ne	ano
Cornell + drak	100014	50028	ano	ne	ano
Spoza	66447	67588	ano	ne	ne
Conference	331179	193488	ano	ne	ne

Tabulka 6.1: Specifikace jednotlivých testovacích scén

6.1 Rychlost stavby uniformní mřížky pro scénu

Uniformní mřížka pro scénu byla implementována tak, aby reflektovala rozměry jednotlivých modelů, jejichž velikost je převedena do rozmezí -1 až 1. Avšak mřížka má vždy stejný počet buněk v jednotlivých souřadnicových osách a vytváří tak krychli zaobalující celou scénu. Měření provedeme pro všechny scény a mřížky o rozměrech 8, 16, 32, 64 a 128 buněk v jedné ose. Jednotlivé scény uložíme do paměti GPU při inicializaci CUDA a čas potřebný k tomuto uložení měřit nebudeme.

Výsledky měření jsou uvedeny v tabulce 6.2. Z výsledků je patrné, že čas pro stavbu mřížky roste se zvětšujícím se rozlišením uniformní mřížky. Tento růst závisí na zvyšujícím se počtu testů průniků trojúhelníku s buňkami, které musí dané vlákno vykonat, což je především problém u scén obsahujících větší trojúhelníky, jež pak leží ve více buňkách. Vzniká také větší počet ukazatelů trojúhelníků na buňky, které je potřeba seřadit.

Z tabulky 6.2 dále vyplývá, že stavba mřížky pro scénu Cornell + 2 krabice, která obsahuje pouhých 42 trojúhelníků, je od rozlišení $64 \times 64 \times 64$ výrazně pomalejší, než pro ostatní scény. Je to dáno právě tím, že obě krabice jsou složeny z několika velkých trojúhelníků překrývajících značné množství buněk. Navíc samotný Cornell box je složen z pouze deseti velkých trojúhelníků, z čehož pramení značně pomalejší stavba mřížky pro scény Cornell, než pro scény Sponza nebo Conference složených z velkého množství menších trojúhelníků. U menších trojúhelníků tedy jednotlivá vlákna provedou mnohonásobně menší počet testů mezi svými trojúhelníky a buňkami.

6.2 Rychlost photon tracingu na GPU

Jak jsme uvedli v předchozí kapitole, sledování primárních fotonů je prováděno pomocí rasterizace scény z pozice světla. Pro demonstraci rychlosti rasterizace zvolíme šest světel, ze kterých je potřeba vykreslit scénu do textur celkem 36 krát (6 krát pro vytvoření cube mapy jednoho světla). Měření provedeme celkem 20 krát a z výsledných hodnot spočítáme aritmetický průměr.

Naměřené hodnoty pro rasterizaci jsou uvedeny v tabulce 6.3. Jak lze vidět, doba rasterizace se pohybuje v řádech jednotek milisekund i pro poměrně velký počet generovaných fotonů (6000000). Rychlost navíc není příliš ovlivněná počtem trojúhelníků ve scéně.

V dalším testování se již zaměříme na rychlost sledování fotonů v CUDA. Pro urychlení sledování využijeme uniformní mřížky se scénou. Jedním z problémů, který musíme

Scéna	Rozlišení mřížky				
	$8 \times 8 \times 8$	$16 \times 16 \times 16$	$32 \times 32 \times 32$	$64 \times 64 \times 64$	$128 \times 128 \times 128$
Cornell + 2 krabice	1.35	3.09	10.13	45.03	245.76
Cornell + 2 koule	1.06	2.38	5.89	22.63	116.26
Cornell + králík	3.84	4.57	8.61	23.97	113.28
Cornell + drak	5.43	6.81	13.46	37.04	166.39
Spoza	3.89	4.75	6.26	11.87	29.09
Conference	9.75	10.32	13.55	20.53	45.06

Tabulka 6.2: Rychlost stavby uniformní mřížky pro jednotlivé scény. Naměřené hodnoty jsou uvedeny v milisekundách

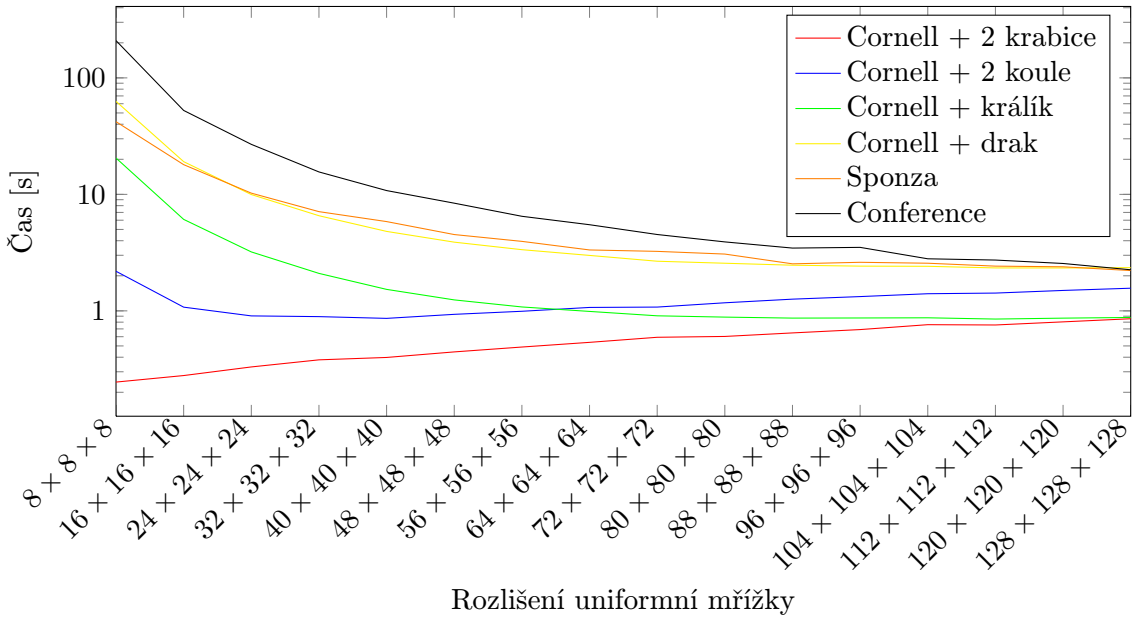
při tomto testování řešit je, jaké rozlišení mřížky pro jednotlivé scény použijeme. Jelikož v aplikaci nebyla implementována žádná heuristická funkce, která by vhodné rozměry vypočítala, provedeme testování pro různá rozlišení uniformní mřížky. Vliv daných rozlišení mřížky na sledování fotonů je patrný z grafu znázorněného na obrázku 6.1. Svislá osa tohoto grafu je v logaritmickém měřítku a na grafu vidíme, že zvětšováním rozlišení mřížky klesá doba výpočtu u scén s velkým počtem trojúhelníků. Nejvýraznějšího zrychlení je dosaženo u scény Conference, kde se doba výpočtu snížila z přibližně 200s na 2s. Na základě těchto výsledků vybereme pro jednotlivé scény nejlepší rozlišení uniformní mřížky a provedeme následující měření.

V tabulce 6.4 můžeme vidět rychlost sledování fotonů v závislosti na počtu vygenerovaných fotonů. Dané hodnoty jsou vypočteny aritmetickým průměrem z dvaceti měření. Do výsledných časů je taky započtena doba potřebná pro vygenerování počátečních stavů náhodných čísel pomocí knihovny Thrust. Je patrné, že se doba výpočtu zvyšuje s lineární závislosti na rostoucím počtu vygenerovaných fotonů. Lineární růst ukazuje, že jednotlivá vlákna warpu nejsou ukončovány současně, čímž by došlo k zvýšení výpočetního výkonu. Je to dáno implementací ruské rulety pro ukončování sledování fotonů. Jednotlivá vlákna reprezentující fotony budou tedy ukončována podle náhodného rozložení pravděpodobnosti.

Z tabulky taky zjistíme, že časové hodnoty narůstají při vyšším počtu trojúhelníků ve scéně, což je způsobeno větším počtem testování průsečíku mezi fotonem a trojúhelníky. Tento růst již není lineárně závislý na počtu trojúhelníků, ale závisí především na geometrii scény. Čím více je geometrie uniformní, tedy trojúhelníky jsou se stejným náhodným rozložením umístěny ve scéně, tím optimálnější bude průchod uniformní mřížkou. Problémy tedy způsobují detailní objekty ve scéně, jejichž trojúhelníky jsou uloženy v malém počtu buněk mřížky, kdy je potřeba toto velké množství trojúhelníku v dané buňce procházet a testovat. Pokud se ale podíváme do tabulky 6.4 podrobněji, mohlo by se zdát, že výše uvedená tvrzení nejsou pravdivá. Pro scénu Cornell + 2 koule obsahující 2188 trojúhelníků trvá výpočet srovnatelnou dobu se scénou Cornell + králík s 65416 trojúhelníky, který je tedy hodně detailní a navíc poměrně malý, viz obrázek 6.2(c). Po důkladném testování však můžeme zjistit, že problém je na straně vlastností materiálu. U scény Cornell + králík je nastavena menší hodnota difúzního odrazu fotonů od stěn, než je tomu u scény Cornell + 2 koule, čímž dochází k rychlejšímu ukončování sledování jednotlivých fotonů. Můžeme tedy říci, že rychlost photon tracingu bude poměrně hodně ovlivněna vlastnostmi materiálu ve scéně.

Scéna	Počet fotonů (6 světel)			
	250000	500000	1000000	6000000
Cornell + 2 krabice	4.01	4.44	5.83	7.24
Cornell + 2 koule	4.22	4.85	6.09	7.62
Cornell + králík	4.21	4.99	6.21	7.61
Cornell + drak	4.85	5.61	7.12	8.01
Spoza	5.01	5.55	6.71	8.25
Conference	5.21	5.86	7.43	7.71

Tabulka 6.3: Rychlost generování primárních fotonů pomocí rasterizace. Naměřené hodnoty jsou uvedeny v milisekundách



Obrázek 6.1: Vliv rozlišení mřížky na rychlost photon tracingu u daných scén. Počet generovaných fotonů je 1000000

6.3 Rychlost stavby uniformní mřížky pro fotony

V rámci testování rychlosti stavby mřížky pro fotony provedeme opět 20 měření a výsledné hodnoty získáme aritmetickým průměrem. K testování nyní použijeme scénu Cornell + 2 koule, ve které se vyskytují kaustiky a scénu Conference, která je naopak plně difúzní, tedy bez kaustik. Vytvoření mřížky pro kaustiky je však nutné pro každou scénu, tedy i pro plně difúzní, protože nemůžeme předem zjistit přítomnost kaustik ve scéně. Jak jsme si uvedli v předchozí kapitole, mřížka je adaptivní k poloměru sběrné koule. Hodnoty poloměrů zvolíme 0.08, 0.04, 0.02 a 0.01, což jsou rozměry vztažené k rozsahu scény od -1 do 1 v kartézské soustavě souřadnic (každá scéna je do tohoto rozsahu převedena). Počet buněk v mřížce může být různý v závislosti na rozměrech scény.

Výsledky měření pro Cornell + 2 koule nalezneme v tabulce 6.5. Z této tabulky lze vyčíst, že doba výstavby mřížky jak pro fotony, tak pro kaustiky narůstá se zvyšujícím se počtem generovaných fotonů. Podobně jako u stavby mřížky pro scénu je to dáno spouštěním více

Scéna	Rozlišení mřížky	Počet fotonů			
		250000	500000	1000000	6000000
Cornell + 2 krabice	$8 \times 8 \times 8$	61.40	122.50	242.81	1444.95
Cornell + 2 koule	$40 \times 40 \times 40$	223.71	445.30	879.41	5246.84
Cornell + králík	$112 \times 112 \times 112$	218.25	435.11	858.46	5082.26
Cornell + drak	$128 \times 128 \times 128$	609.83	1198.48	2354.28	13982.40
Spoza	$128 \times 128 \times 128$	573.40	1128.26	2228.06	13422.83
Conference	$128 \times 128 \times 128$	667.38	1291.95	2130.24	12361.50

Tabulka 6.4: Rychlost photon tracingu na GPU při vybraných rozlišeních uniformní mřížky pro jednotlivé scény. Naměřené hodnoty jsou uvedeny v milisekundách

vláken v kernelu pro nalezení odpovídající buňky k fotonu, do které patří a následným řazením vyššího počtu fotonů. Při větším rozlišení je doba stavby mřížky pro daný počet fotonů téměř stejná. Jeden foton totiž vždy patří pouze do jedné buňky a nevzniká tak větší pole s ukazateli, než je počet fotonů, jak tomu bylo u stavby mřížky pro scénu a trojúhelníky. Z tabulky je taky patrné, že stavba mřížky pro fotony je časově podobně rychlá jako pro kaustiky, což se dá očekávat, jelikož se ve scéně nachází dva poměrně velké objekty, díky kterým mohou kaustiky vzniknout. Počet těchto fotonů v mapě kaustik pravděpodobně bude přibližně stejný jako počet fotonů v globální mapě.

U scény Conference, viz tabulka 6.6, je rychlost stavby mřížky pro obyčejné fotony podobná jako u scény Cornell + 2 koule. Rozdíl je pouze u časů stavby mřížky pro kaustiky, které jsou nyní nižší i přes to, že je potřeba všechny kernely pro stavbu spouštět se stejným počtem vláken jako u obyčejných fotonů. Zrychlení je dosaženo při hledání počátku a konce buněk. Pokud v mapě kaustik nejsou žádné fotony, což je u čistě difúzní scény Conference zajištěno, mají všechny ukazatele na buňky stejnou hodnotu a zrychlení je dosaženo tím, že není potřeba zapisovat počátek a konec do globální paměti GPU, viz detaily implementace uniformní mřížky 5.7.1.

Poloměr sběrné koule	Fotony				Kaustiky			
	250k	500k	1M	6M	250k	500k	1M	6M
0.08 (17^3)	4.82	7.29	11.44	53.55	3.92	6.37	10.27	51.11
0.04 (33^3)	4.38	6.74	11.06	53.22	3.59	6.27	10.57	51.48
0.02 (65^3)	4.46	6.84	11.14	55.79	3.78	6.44	10.49	51.17
0.01 (130^3)	5.22	8.26	13.15	65.83	4.82	7.86	12.21	59.81

Tabulka 6.5: Rychlost stavby uniformní mřížky pro daný počet fotonů a kaustik ve scéně Cornell + 2 koule. Hodnota 17^3 představuje rozlišení mřížky $17 \times 17 \times 17$, počet fotonů je dán hodnotou 250k, což značí 250000 a 1M je 1000000. Naměřené hodnoty jsou uvedeny v milisekundách

Poloměr sběrné koule	Fotony				Kaustiky			
	250k	500k	1M	6M	250k	500k	1M	6M
0.08 (21^3)	6.98	7.01	11.17	55.58	3.09	4.94	7.72	35.88
0.04 (41^3)	4.73	7.07	11.88	53.85	3.59	5.30	7.84	35.89
0.02 (82^3)	4.63	7.31	13.98	53.65	3.32	5.82	8.02	35.71
0.01 (163^3)	6.47	9.43	13.82	63.53	5.18	6.99	9.13	37.42

Tabulka 6.6: Rychlost stavby uniformní mřížky pro daný počet fotonů a kaustik ve scéně Conference. Hodnota 21^3 představuje rozlišení mřížky $21 \times 21 \times 21$, počet fotonů je dán hodnotou 250k, což značí 250000 a 1M je 1000000. Naměřené hodnoty jsou uvedeny v milisekundách

6.4 Rychlost sledování paprsku na GPU

V předposlední části otestujeme rychlost vykreslování jednotlivých scén pomocí metody sledování paprsků s využitím fotonových map. Pro účely tohoto testování byla vytvořena jednoduchá kamera, která se po scéně pohybuje ve směru jedné zvolené osy tam a zpět, podle zadaného minima a maxima. Kameru nastavíme tak, aby vždy snímala scénu. Její výchozí pozice v jednotlivých scénách je ukázána na obrázku 6.2. Na daném obrázku jsou taky patrné pozice světla, které zůstávají konstantní po celou dobu testování. Počet sledování sekundárních paprsků nastavíme na 4, díky čemuž lze sledovat jak odražené, tak zalomené paprsky. V klasické rekurzívní implementaci by toto číslo udávalo úroveň zanoření. Uniformní mřížka pro jednotlivé scény bude mít rozměry získané na základě grafu z obrázku 6.1 z předchozí podkapitoly. Kernel bude spouštěn ve 2D s počtem 8×8 vláken v jednom bloku. Všechna měření provedeme pro 100 snímků ve směru pohybu kamery. Výsledné časové hodnoty získáme aritmetickým průměrem z těchto 100 měření.

Vliv sběru fotonů z 8 nebo 27 buněk na rychlost vykreslování

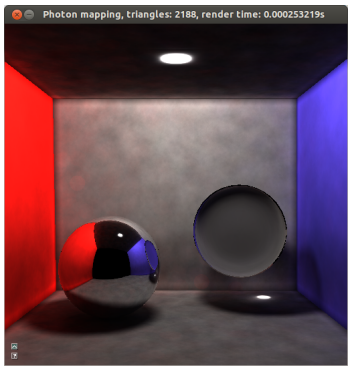
Při implementaci jsme si ukázali dva možné přístupy ke sběru fotonů z uniformní mřížky. Také jsme uvedli předpoklad, že sběr fotonů a následné vykreslení scény bude pomalejší při použití 8 buněk s délkou strany rovnou průměru sběrné koule, než při 27 buňkách s délkou stran rovnou poloměru koule. Ověření provedeme měřením ve všech scénách s 1000000 vygenerovaných fotonů a konstantním poloměrem sběrné koule pro fotony a kaustiky.

Výsledky jsou znázorněny grafem na obrázku 6.3 a vyplývá z nich, že předpoklad byl správný. Použijeme-li pro sběr pouze 8 buněk, má výsledná podmřížka větší objem, než je tomu u 27 buněk a může se v ní tedy nacházet více fotonů. Pro sběr těchto fotonů je pak potřeba více přístupů do paměti, a taky více testů na vzdálenost fotonů od středu sběrné koule. Pro další testování budeme tedy používat sběr pomocí 27 buněk. Tomuto využití nebrání ani fakt, že mřížka pro 27 buněk musí mít větší rozlišení, neboť jsme výše ukázali, že větší rozlišení téměř nemá vliv na rychlost stavby mřížky pro fotony.

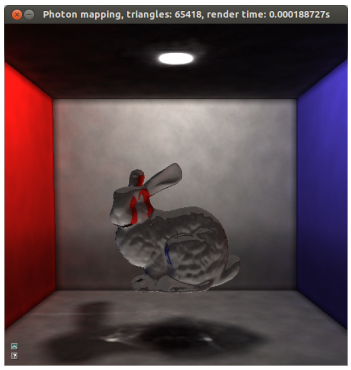
Vliv poloměru sběrné koule na kvalitu a rychlost vykreslování

Součástí implementované aplikace je možnost interaktivně měnit velikost poloměru sběrné koule. Změnou poloměru se tedy mění i rozlišení mřížky. Vliv změny poloměru na rychlost

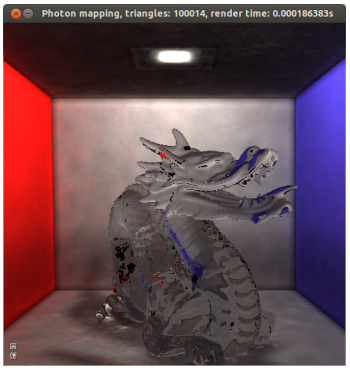
(a) Cornell + 2 krabice



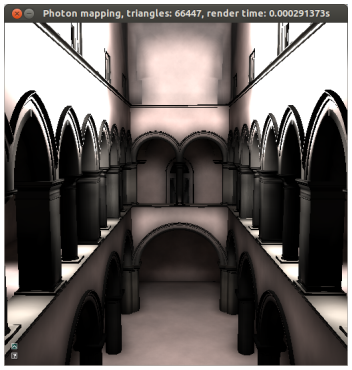
(b) Cornell + 2 koule



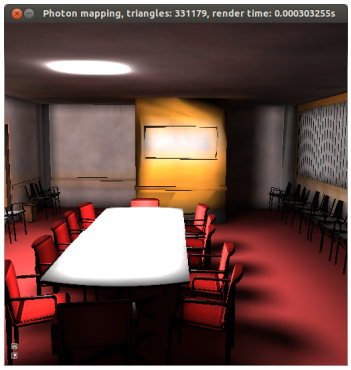
(c) Cornell + králík



(d) Cornell + drak



(e) Sponza



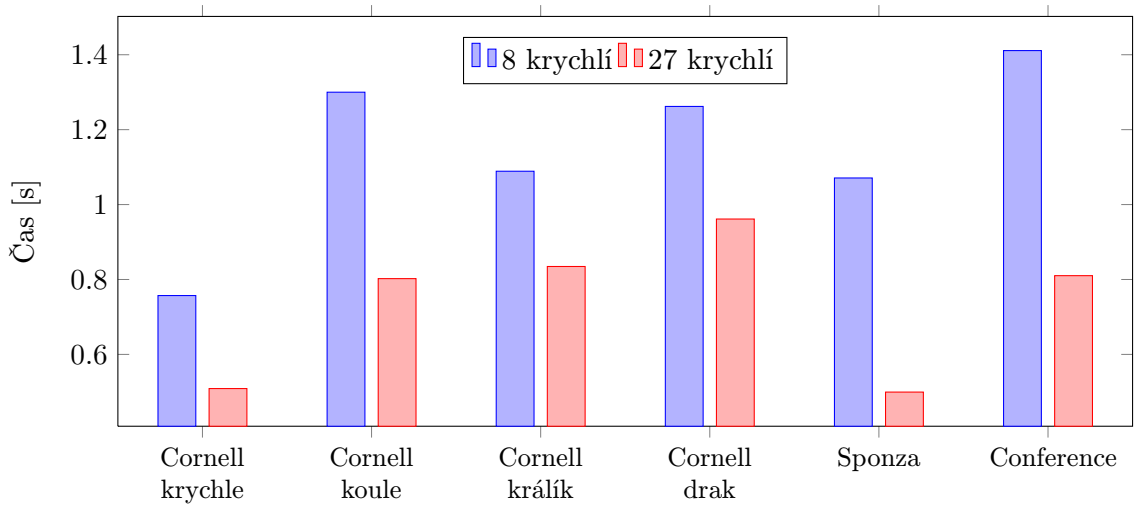
(f) Conference

Obrázek 6.2: Ukázka počátečního nastavení kamery a světla v jednotlivých scénách

vykreslování otestujeme na dvou scénách, a to na Cornell + 2 koule a Conference. Měření provedeme pro poloměry 0.08, 0.04, 0.02 a 0.01 vzhledem ke scénám převedeným do rozsahu -1 až 1. Dále zvolíme rozlišení obrazu 512×512 a 500000 vygenerovaných fotonů. Sběrná koule pro kaustiky má konstantní poloměr nastaven na 0.03.

Naměřené minimální, maximální a průměrné časové hodnoty pro průlet scénou vidíme v tabulce 6.7. Je zřejmé, že průměrná rychlost vykreslování roste při zmenšování poloměru sběrné koule, což je dáno jemnějším rozlišením mřížky pro sběr fotonů, a tedy snížením počtu přístupů do paměti při sběru fotonů pro daný průsečík paprsku se scénou. Dále z tabulky zjistíme, že je u obou scén pro každý poloměr poměrně markantní rozdíl mezi minimálním a maximálním časem potřebným pro vykreslení výsledného obrázku. Je to způsobeno již zmiňovanou neschopností uniformní mřížky přizpůsobit se geometrii scény, v tomto případě geometrii fotonů. Konkrétně je problém v blízkosti světla ke stropu nebo obecně k nějaké stěně, příp. jiného difuzního objektu, kde pak vzniká vysoká koncentrace fotonů. Toto velké množství fotonů pak leží v několika málo buňkách, což způsobuje zpomalování při sběru fotonů, pokud je dané místo v záběru kamery. Příklady koncentrace fotonů na jednom místě v jednotlivých testovacích scénách můžeme vidět na obrázku 6.2, kde se jedná o velmi jasné kolečko v blízkosti pozice světla.

Při zmenšování poloměru sběrné koule nejen klesá čas výpočtu, ale také roste šum v jednotlivých vykreslených scénách, viz obrázky 6.4 a 6.5. Tento šum vzniká kvůli malému počtu vygenerovaných fotonů, které pak schází k odhadu zářivé energie při malém poloměru sběrné koule v místě průsečíku se scénou. Extrémní případ šumu při velmi malém poloměru



Obrázek 6.3: Rychlost sledování paprsku při použití sběru fotonů z 8 a 27 buněk. Počet generovaných fotonů je 1000000 a poloměr sběrné koule $r = 0.04$, resp. $r = 0.03$ pro kaustiky. Rozlišení obrazu je 512×512

je vidět na obrázku 6.6. Na tomto obrázku je již zřetelná pravidelná textura fotonů, která vzniká díky rasterizaci primárních fotonů.

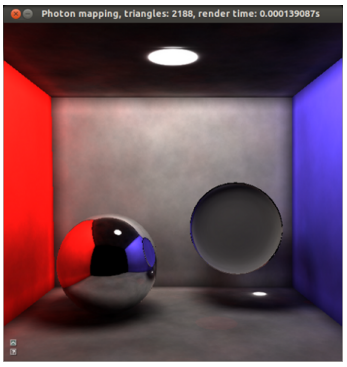
Jednou z možností jak lze šumu ve výsledných obrázcích předcházet, je vygenerování většího počtu fotonů do scény. Výpočet osvětlení pak konverguje k přesnějšímu výsledku při použití menšího poloměru sběrné koule. Při vyšším počtu fotonů však poměrně hodně narůstá doba sledování fotonů, viz tabulka 6.4, stejně jako čas sledování paprsku, což je patrné z grafu na obrázku 6.7.

Scéna	Čas [ms]					
	Min	Avg	Max	Min	Avg	Max
Cornell + 2 koule	$r = 0.08$			$r = 0.04$		
	351.50	735.01	1259.05	94.58	408.29	784.50
	$r = 0.02$			$r = 0.01$		
	31.84	344.90	652.70	30.78	196.38	373.90
Conference	$r = 0.08$			$r = 0.04$		
	691.52	1452.39	2463.59	164.88	476.72	907.95
	$r = 0.02$			$r = 0.01$		
	61.40	230.92	518.71	30.38	189.11	525.29

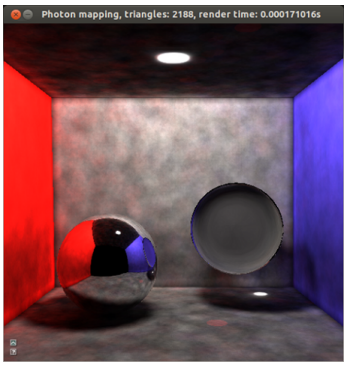
Tabulka 6.7: Rychlost sledování paprsku pro různé poloměry r sběrné koule. Poloměr koule pro kaustiky je konstantní $r = 0.03$. Počet vygenerovaných fotonů je 500000. Naměřené hodnoty jsou uvedeny v milisekundách

Vliv rozlišení obrazu na rychlost vykreslování

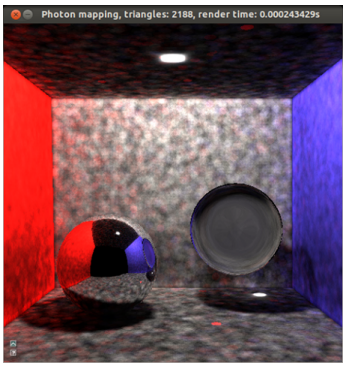
Jedná se o poslední test, který provedeme. Postupně budeme zvyšovat rozlišení scény z 512×512 až na 1024×1024 s přírůstkem 128×128 pixelů. Pokud přepočteme daná rozlišení na



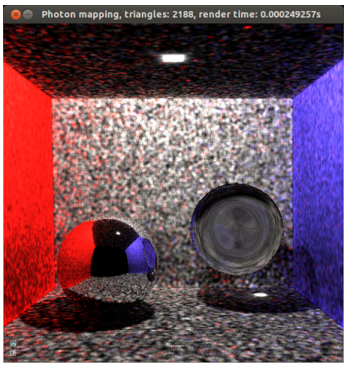
(a) $r = 0.08$



(b) $r = 0.04$



(c) $r = 0.02$

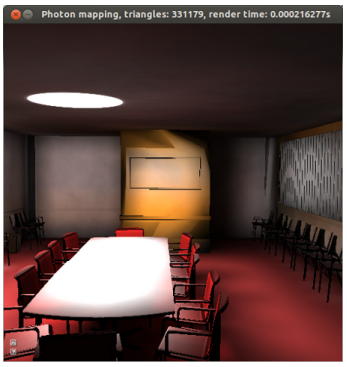


(d) $r = 0.01$

Obrázek 6.4: Vliv velikosti poloměru r sběrné koule na výsledné vykreslení scény Cornell boxu s koulemi. Počet vygenerovaných fotonů je 500000

pixely, které budou zpracovávány jednotlivými vlákny, pak by vykreslení obrazu 1024×1024 mělo být asi 4 krát pomalejší než vykreslení obrazu s rozlišením 512×512, tedy doba výpočtu poroste lineárně. Měření provedeme postupně na všech scénách.

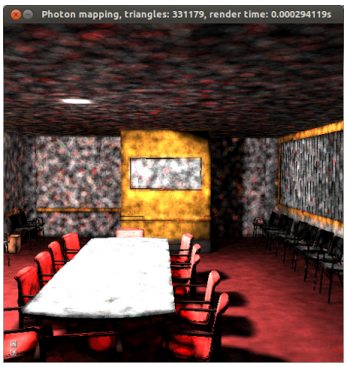
Výsledky jednotlivých měření jsou vneseny do grafu na obrázku 6.8. Z grafu je patrné, že doba výpočtu neroste lineárně se zvyšujícím se rozlišením u žádné scény, čímž se tedy potvrzuje domněnka z implementační části práce 5.7.3. Při větším rozlišení obrazu je větší pravděpodobnost, že všechna vlákna warpu (32 vláken) v jednom bloku (8×8) budou mít průsečík se stejným trojúhelníkem a je také dobrá šance, že průsečíky budou ležet ve stejné buňce uniformní mřížky pro fotony. Vlákna v rámci warpu pak budou ukončovány současně, což ve výsledku způsobí nárůst výpočetního výkonu přepočteného na jednotlivá vlákna.



(a) $r = 0.08$

(b) $r = 0.04$

(c) $r = 0.02$



(d) $r = 0.01$

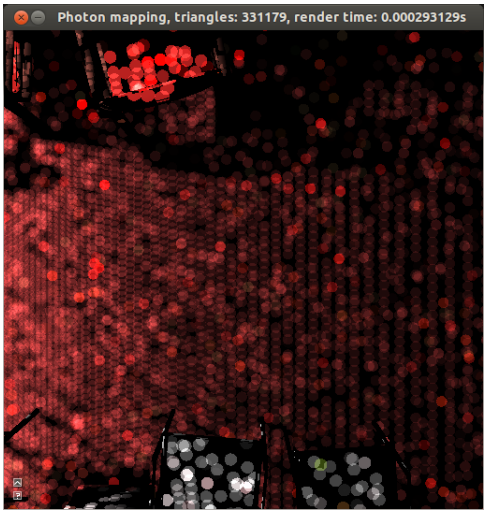
Obrázek 6.5: Vliv velikosti poloměru r sběrné koule na výsledné vykreslení scény konferenční místnosti. Počet vygenerovaných fotonů je 500000

6.5 Paměťová náročnost aplikace

V testování jsme se nezmínili o rychlosti kopírování dat mezi operační pamětí a pamětí GPU. V rámci implementované aplikace probíhá tento přenos při nahrání scény na GPU, které je provedeno při inicializaci CUDA před samotnými výpočty osvětlení. Dalším přenosem je uskutečněn při stavbě uniformní mřížky pro scénu, kdy je potřeba spočítat počet odkazů každého trojúhelníku na buňky. Tento případ jsme zahrnuli do testování stavby mřížky pro scénu. Všechna ostatní data jsou počítána a uchovávána přímo na GPU a ostatním kernelům je předáván pouze odkaz na tato data. Stejně je tomu tak u interoperability mezi OpenGL a CUDA.

Implementovaná aplikace je celkově poměrně náročná na paměť GPU. Musíme uchovávat trojúhelníky scény spolu s uniformní mřížkou, dvě fotonové mapy a jejich mřížky, a také textury s primárními průsečíky paprsků (ty se průběžně aktualizují, ale nemění svoji velikost). Velké texturové atlasy pro světla jsou použity pouze jednou při výpočtu photon tracingu a pak je jejich paměť uvolněna.

Paměťové nároky aplikace při vykreslování scén s rozlišením 512×512, 6000000 vygenerovaných fotonů a konstantním poloměrem sběrné koule $r = 0.04$, resp. $r = 0.03$ jsou ukázány v tabulce 6.8. Z tabulky můžeme vyčíst, že nejvíce paměti zabírají fotonové mapy, a to 275MB každá. Mapa kaustik se vytváří i pro čistě difúzní scény a stálo by za úvahu, zda není lepší tuto mapu z implementace zcela odstranit. Tímto krokem bychom získali

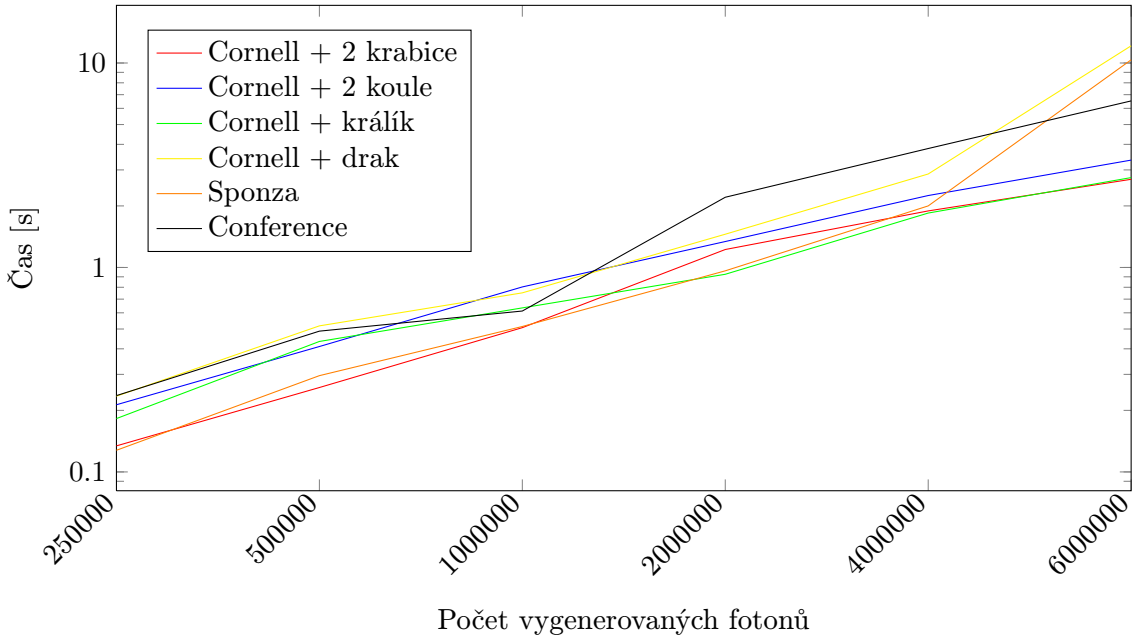


Obrázek 6.6: Ukázka problému s pravidelnou texturou, která vzniká při generování fotonů pomocí rasterizace

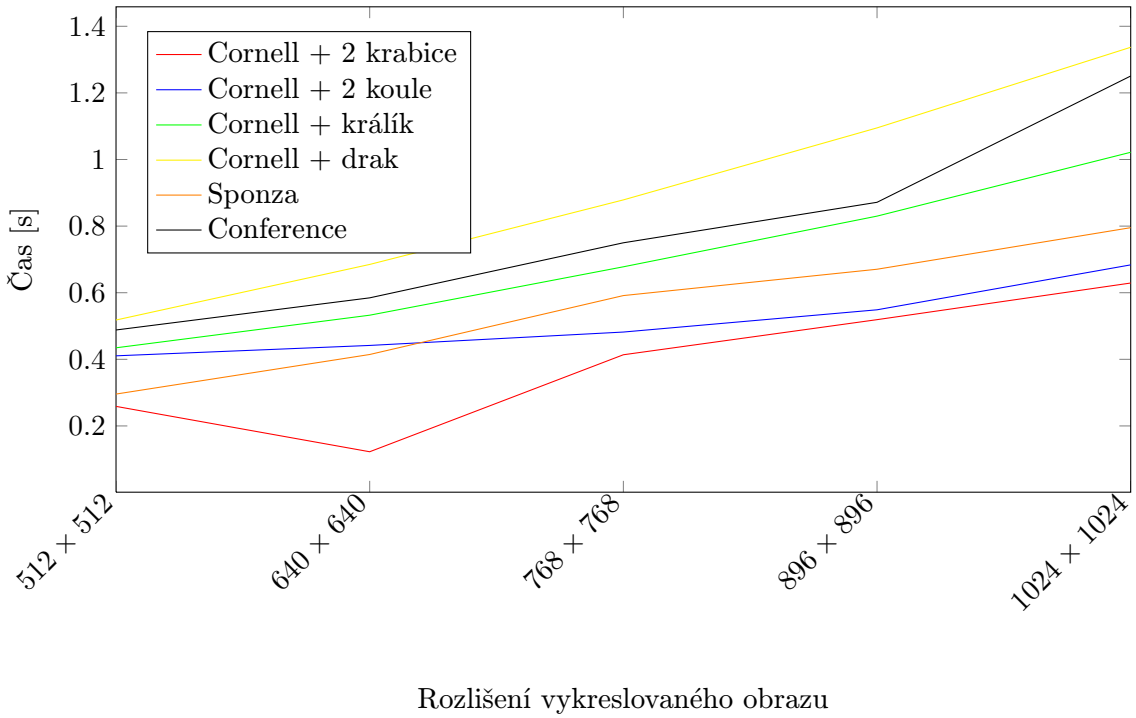
více paměti, která je nyní potřebná pro mapu a její uniformní mřížku, ale zároveň přijdeme o možnost zobrazení detailů kaustik při menším počtu vygenerovaných fotonů. Zásah do implementované aplikace by však byl poměrně drastický.

Scéna	Rozlišení mřížky	Velikost [MB]					
		Textury	Celá scéna	Mřížka scény	Dvě fotonové mapy	Dvě mřížky pro fotony	Celkem
Cornell + 2 krabice	8^3	20	0	0	2×274	$45 + 45$	658
Cornell + 2 koule	40^3	20	0	0	2×274	$45 + 45$	658
Cornell + králík	112^3	20	2	11	2×274	$45 + 45$	671
Cornell + drak	128^3	20	6	18	2×274	$45 + 45$	682
Spoza	128^3	20	4	18	2×274	$46 + 47$	683
Conference	128^3	20	22	20	2×274	$45 + 46$	701

Tabulka 6.8: Paměťové nároky jednotlivých scén při 6000000 vygenerovaných fotonů, rozlišení obrazu 512×512 a poloměru sběrné koule $r = 0.04$, resp. $r = 0.03$ pro kaustiky



Obrázek 6.7: Vliv různého počtu vygenerovaných fotonů na rychlost vykreslování daných scén při rozlišení obrazu 512×512 a poloměru sběrné koule $r = 0.04$, resp. $r = 0.03$ pro kaustiky



Obrázek 6.8: Vliv rozlišení vykreslovaného obrazu na rychlost sledování paprsku u daných scén. Počet generovaných fotonů je 500000 a poloměr sběrné koule $r = 0.04$, resp. $r = 0.03$ pro kaustiky

Kapitola 7

Závěr

Cílem mé práce bylo nastudování metody photon mapping, která řeší výpočet globálního osvětlení ve scéně a vychází z photon tracingu. Dále jsem měl navrhnout a implementovat jednoduchou aplikaci na GPU, která demonstruje výpočet osvětlení pomocí photon mappingu se zaměřením na vykreslování sekvence více snímků stejné scény. Dané cíle jsem úspěšně splnil.

Práce je rozčleněna do několika logických celků. V první části jsem podrobně popsal metodu photon mapping a její součásti, tedy vytvoření fotonových map pomocí photon tracingu a následné vykreslování obrazu metodou sledování paprsku, rozšířenou o odhad osvětlení z fotonových map. V této části jsem pro srovnání uvedl některé další příklady metod určených k výpočtu globálního osvětlení. Dále jsem se v práci zabýval metodami efektivního dělení 3D prostoru, které jsou základem k urychlování photon tracingu a sledování paprsku. Jelikož hlavním požadavkem na implementaci bylo urychlování výpočtů pomocí GPU, popsal jsem některá dostupná API pro programování GPU a metodiku implementace algoritmů na GPU. Na základě těchto získaných teoretických znalostí jsem provedl, zdokumentoval a otestoval implementaci demonstrační aplikace pro výpočet photon mappingu na GPU.

V rámci implementace jsem vytvořil unikátní návrh výpočtu photon mappingu využívající spolupráce knihovny OpenGL a CUDA. Tato spolupráce je postavena na rychlém výpočtu primárních průsečíku fotonů a paprsků se scénou pomocí rasterizace v OpenGL a následného dokončení výpočtu v CUDA. Při testování aplikace jsem ukázal, že mnou vytvořený návrh je poměrně efektivní pro vykreslování několika snímků za sekundu u statické scény, kde je výpočet osvětlení proveden pouze jednou. Výsledná rychlost hodně záleží na zvolení parametrů uživatelem. Je to dáno především implementací uniformní mřížky, jako metody řešící dělení prostoru, která se nedokáže dobře adaptovat ke geometrii jednotlivých scén. Rychlost také závisí na počtu generovaných fotonů a požadované kvalitě výsledného obrazu. Jednou z klíčových vlastností aplikace, na kterou jsem obzvlášť hrdý, je implementace všech důležitých součástí photon mappingu včetně uniformní mřížky na GPU.

V průběhu testování aplikace jsem přišel na několik možných vylepšení, které se budu snažit implementovat. Jedná se především o urychlení výpočtu photon tracingu, který momentálně nedovoluje použití aplikace pro dynamické scény. Jednou z možností tohoto urychlení bude omezení počtu odrazů, které může foton vykonat. V nynější podobě je totiž ukončení sledování fotonu závislé pouze na rozhodnutí ruské rulety a maximální počet odrazů nelze předem určit, což způsobuje značné časové rozdíly při ukončování jednotlivých vláken spuštěných na GPU. Dalším vylepšením bude snížení značných nároků aplikace na paměť. Jednu možnost, a to odstranění fotonové mapy kaustik, jsem již zmínil v kapitole

věnované testování. Poměrně důležitým rozšířením je interaktivní průchod scénou s vypočteným osvětlením. Toto rozšíření se mi již částečně podařilo implementovat v podobě adaptivní změny rozlišení uniformní mřížky při sběru fotonů. Při průchodu scénou se tak zmenšuje poloměr sběru fotonů, čímž se urychluje výpočet, ale výsledný obraz je značně zašuměný. Po zastavení se již provede výpočet obrazu v požadované kvalitě. Pokud by se mi uvedená vylepšení podařilo implementovat, nabízela by se možnost urychlení výpočtu osvětlení tak, aby bylo použitelné pro dynamické scény v reálném čase.

Literatura

[1] Jensen, H. W.: Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*. London, UK, UK: Springer-Verlag, 1996. ISBN 3-211-82883-4, s. 21–30.

[2] Lafortune, E. P.; Willems, Y. D.: Bi-Directional Path Tracing. In *PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL GRAPHICS AND VISUALIZATION TECHNIQUES (COMPUGRAPHICS ?93*. 1993. s. 145–153.
Dostupné z WWW: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.3913&rep=rep1&type=pdf

[3] Jensen, H. W.: *Realistic Image Synthesis Using Photon Mapping*. Natic, Massachusetts, 2001. ISBN 1-56881-147-0.

[4] Cook, R. L.; Porter, T.; Carpenter, L.: Distributed ray tracing. *SIGGRAPH Comput. Graph.* Leden 1984, ročník 18, 3. s. 137–145, ISSN 0097-8930.
Dostupné z WWW: http://doi.acm.org/10.1145/964965.808590

[5] Geometrie/Raytracing, *Wikiknihy* [online]. 2012 [cit. 2012-12-30].
Dostupné z WWW: http://cs.wikibooks.org/wiki/Geometrie/Raytracing

[6] Saucier, R.: *Computer Generation of Statistical Distributions*. Storming Media, 2000. ISBN 978-1-423-54028-1.

[7] Jensen, H. W.: A practical guide to global illumination using ray tracing and photon mapping. In *ACM SIGGRAPH 2004 Course Notes*. SIGGRAPH '04, New York, NY, USA: ACM, 2004.
Dostupné z WWW: http://doi.acm.org/10.1145/1103900.1103920

[8] Arvo, J.; Kirk, D.: Particle transport and image synthesis. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '90, New York, NY, USA: ACM, 1990. ISBN 0-89791-344-2, s. 63–66.
Dostupné z WWW: http://doi.acm.org/10.1145/97879.97886

[9] Nicodemus, F. E.: *Geometrical Considerations and Nomenclature for Reflectance*. National Bureau of Standards monograph, U.S. Government Printing Office, 1977.
Dostupné z WWW: http://graphics.stanford.edu/courses/cs448-05-winter/papers/nicodemus-brdf-nist.pdf

[10] Kajiya, J. T.: The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '86, New York, NY,

USA: ACM, 1986. ISBN 0-89791-196-2, s. 143–150.
Dostupné z WWW: http://doi.acm.org/10.1145/15922.15902

[11] Cook, R. L.; Porter, T.; Carpenter, L.: Distributed ray tracing. *SIGGRAPH Comput. Graph.* Leden 1984, ročník 18, 3. s. 137–145, ISSN 0097-8930.
Dostupné z WWW: http://doi.acm.org/10.1145/964965.808590

[12] Goral, C. M.; Torrance, K. E.; Greenberg, D. P.; aj.: Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '84, New York, NY, USA: ACM, 1984. ISBN 0-89791-138-5, s. 213–222.
Dostupné z WWW: http://doi.acm.org/10.1145/800031.808601

[13] Bentley, J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM*. Září 1975, ročník 18, 9. s. 509–517, ISSN 0001-0782.
Dostupné z WWW: http://doi.acm.org/10.1145/361002.361007

[14] Wald, I.; Havran, V.: On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *Interactive Ray Tracing 2006, IEEE Symposium on*. 2006. s. 61–69.

[15] Jansen, F. W.: Data structures for ray tracing. In *Proceedings of a workshop (Eurographics Seminars on Data structures for raster graphics*. New York, NY, USA: Springer-Verlag New York, Inc., 1986. ISBN 0-387-16310-7, s. 57–73.

[16] Fuchs, H.; Kedem, Z. M.; Naylor, B. F.: On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph.* Červenec 1980, ročník 14, 3. s. 124–133, ISSN 0097-8930, doi:10.1145/965105.807481.
Dostupné z WWW: http://doi.acm.org/10.1145/965105.807481

[17] Binary Space Partition (BSP) Tree, *Princeton University* [online]. 2000 [cit. 2013-05-01].
Dostupné z WWW: http://www.cs.princeton.edu/courses/archive/fall00/cs426/lectures/raycast2/sld018.htm

[18] Meagher, D.: Geometric modeling using octree encoding. *Computer Graphics and Image Processing*. June 1982, ročník 19, 2. s. 129–147.
Dostupné z WWW: http://dx.doi.org/10.1016/0146-664X(82)90104-6

[19] Samet, H.: Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics*. 1989, ročník 13. s. 445–460.

[20] Agate, M.; Grimsdale, R. L.; Lister, P. F.: The HERO Algorithm for Ray-Tracing Octrees. In *Advances in Computer Graphics Hardware IV (Eurographics'89 Workshop)*. London, UK, UK: Springer-Verlag, 1991. ISBN 3-540-53473-3, s. 61–73.

[21] Ray / Octree traversal, *Don't Panic / Jeroen Baert's Blog* [online]. 2012 [cit. 2013-05-01].
Dostupné z WWW:
http://www.forceflow.be/2012/04/20/ray-octree-traversal/

[22] Fujimoto, A.; Tanaka, T.; Iwata, K.: Tutorial: computer graphics; image synthesis. kapitola ARTS: accelerated ray-tracing system, New York, NY, USA: Computer Science Press, Inc., 1988, ISBN 0-8186-8854-4, s. 148–159.

[23] Amanatides, J.; Woo, A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *In Eurographics ?87*. 1987. s. 3–10.

[24] Bresenham, J. E.: Algorithm for computer control of a digital plotter. *IBM Systems Journal*. 1965, ročník 4, 1. s. 25–30, ISSN 0018-8670, doi:10.1147/sj.41.0025.

[25] Gottschalk, S.; Lin, M. C.; Manocha, D.: OBBTree: a hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. SIGGRAPH '96, New York, NY, USA: ACM, 1996. ISBN 0-89791-746-4, s. 171–180.
Dostupné z WWW: http://doi.acm.org/10.1145/237170.237244

[26] Akenine-Möller, T.: Fast 3D triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05, New York, NY, USA: ACM, 2005.
Dostupné z WWW: http://doi.acm.org/10.1145/1198555.1198747

[27] Akenine-Möller, T.; Haines, E.; Hoffman, N.: *Real-Time Rendering, Third Edition*. Taylor & Francis, 2011. ISBN 9781439865293.

[28] Gregerson, A.: Implementing Fast MRI Gridding on GPUs via CUDA. Project report, University of Wisconsin - Madison, 2008.
Dostupné z WWW: http://www.nvidia.co.kr/docs/IO/47905/ECE757_Project_Report_Gregerson.pdf

[29] TeamB3D: Stream Processors, *Beyond3D* [online]. 2007 [cit. 2012-12-30].
Dostupné z WWW: http://www.beyond3d.com/content/articles/29/2

[30] F. Abi-Chahla, F. C.: Scalable Processor Array, *Tom's Hardware* [online]. 2008 [cit. 2012-12-30].
Dostupné z WWW:
http://www.tomshardware.com/reviews/nvidia-gtx-280,1953-7.html

[31] NVIDIA: CUDA C Programming Guide, *NVIDIA* [online]. 2013 [cit. 2013-01-08].
Dostupné z WWW:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[32] NVIDIA: CUDA C Best Practices Guide, *NVIDIA* [online]. 2012 [cit. 2013-01-01].
Dostupné z WWW:
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[33] OpenGL Overview, *OpenGL* [online]. 2012 [cit. 2012-12-30].
Dostupné z WWW: http://www.opengl.org/about/

[34] NVIDIA: CUDA Parallel Computing Platform, *NVIDIA* [online]. 2012 [cit. 2012-12-30].
Dostupné z WWW: http://www.nvidia.com/object/cuda_home_new.html

[35] Gaster, B.; Howes, L.; Kaeli, D. R.; aj.: *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. Newnes, 2012. ISBN 0-124-05520-6.

[36] Farber, R.: CUDA, Supercomputing for the Masses: Part 15, *Dr. Dobb's* [online]. 2010 [cit. 2013-01-01].
Dostupné z WWW: http://www.drdobbs.com/architecture-and-design/cuda-supercomputing-for-the-masses-part/222600097

[37] OptiX, *NVIDIA* [online]. 2013 [cit. 2013-05-01].
Dostupné z WWW: http://www.nvidia.com/object/optix.html

[38] Framebuffer Object, *OpenGL* [online]. 2013 [cit. 2013-01-07].
Dostupné z WWW: http://www.opengl.org/wiki/Framebuffer_Object

[39] Import Formats, *Assimp* [online]. 2013 [cit. 2013-05-01].
Dostupné z WWW:
http://assimp.sourceforge.net/main_features_formats.html

[40] Popov, S.; Günther, J.; Seidel, H.-P.; aj.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*. Září 2007, ročník 26, 3. s. 415–424, (Proceedings of Eurographics).

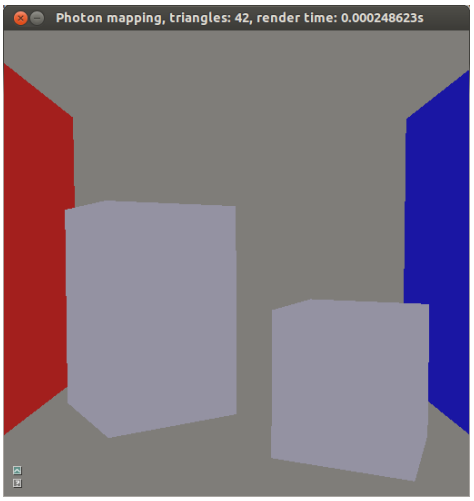
[41] Zhou, K.; Hou, Q.; Wang, R.; aj.: Real-time KD-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers*. SIGGRAPH Asia '08, New York, NY, USA: ACM, 2008. ISBN 978-1-4503-1831-0, s. 126:1–126:11.
Dostupné z WWW: http://doi.acm.org/10.1145/1457515.1409079

[42] Kalojanov, J.; Slusallek, P.: A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09, New York, NY, USA: ACM, 2009. ISBN 978-1-60558-603-8, s. 23–28.
Dostupné z WWW: http://doi.acm.org/10.1145/1572769.1572773

[43] McGuire, M.: McGuire Graphics Data, *Williams College* [online]. 2013 [cit. 2013-05-02].
Dostupné z WWW: http://graphics.cs.williams.edu/data

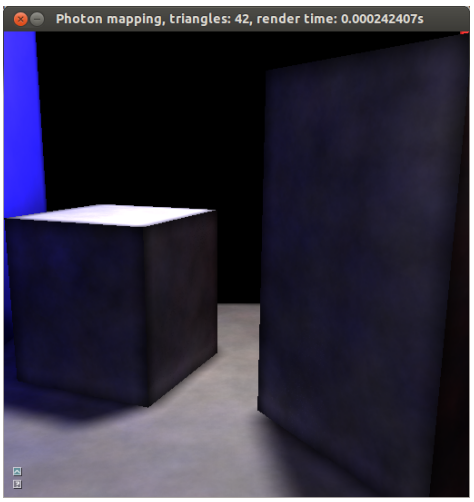
Příloha A

Ukázky osvětlení scén

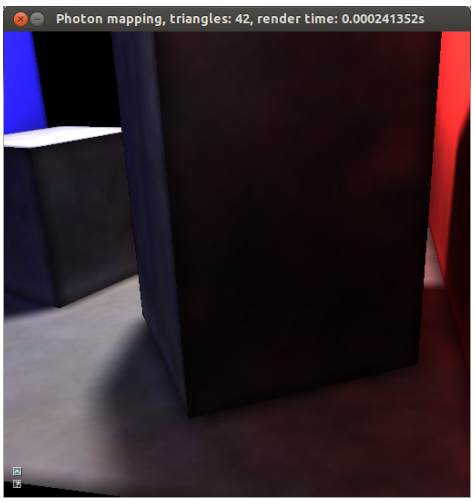


(a) Bez osvětlení

(b) Zapnuté osvětlení

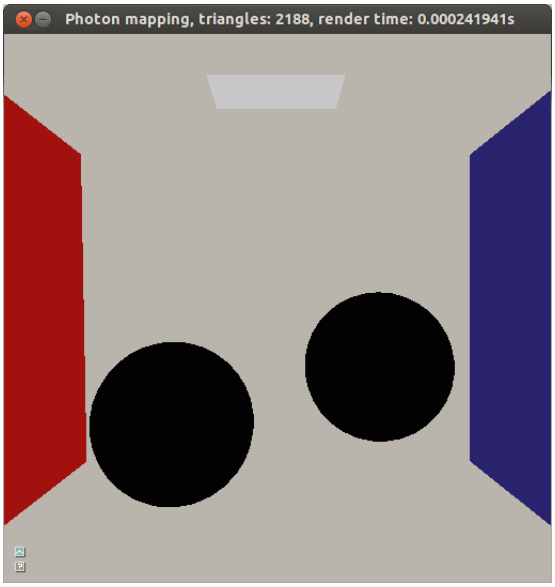


(c) Nepřímé osvětlení patrné na stěnách krabic

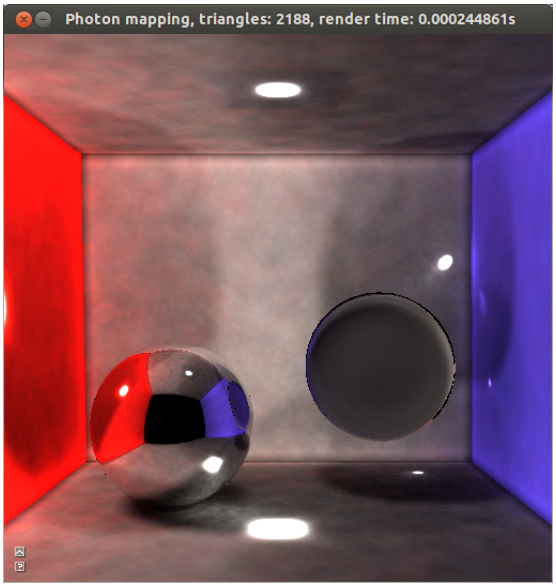


(d) Nepřímé osvětlení patrné na stěnách krabice a podlaze

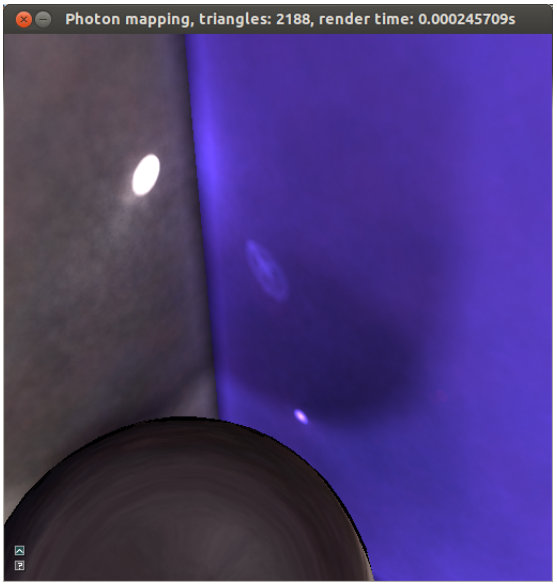
Obrázek A.1: Cornell box s krabicemi



(a) Bez osvětlení



(b) Tři světla ve scéně



(c) Detail kaustik

Obrázek A.2: Cornell box se zrcadlově odrazivou a průhlednou koulí

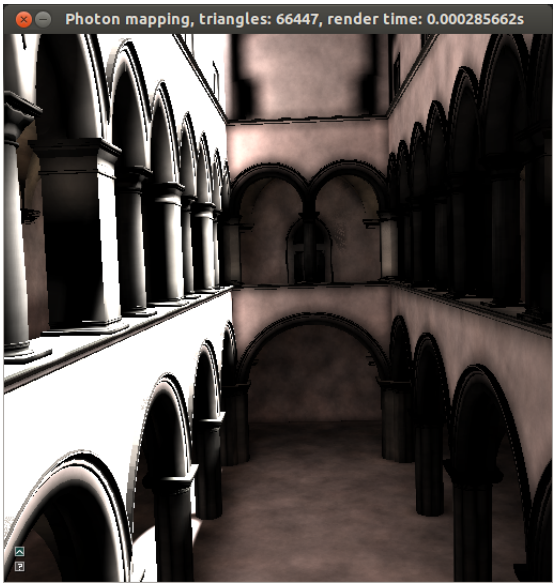
(a) Bez osvětlení

(b) Množství kaustik při zapnutém osvětlení

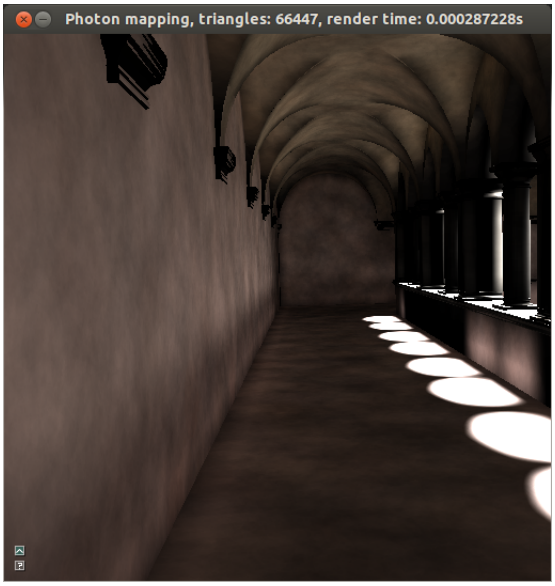
(c) Detail kaustik

Obrázek A.3: Cornell box s drakem. Světlo je umístěno ve spodní části

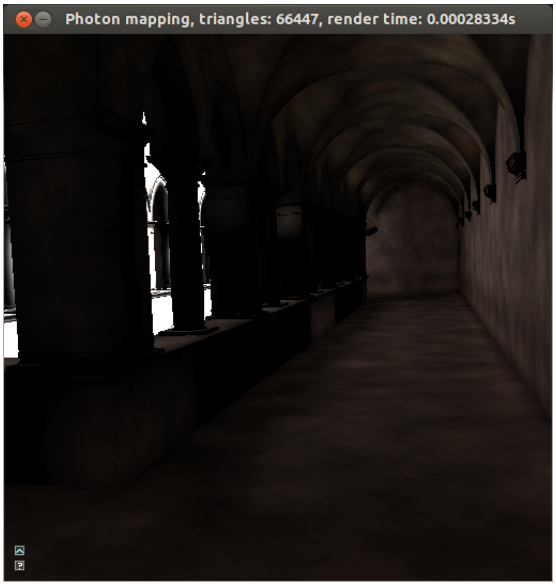
(a) Bez osvětlení



(b) Zapnuté osvětlení

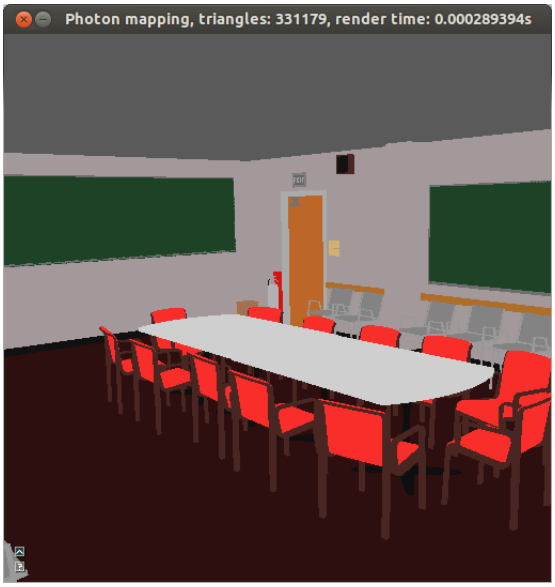


(c) Nepřímé osvětlení v chodbě



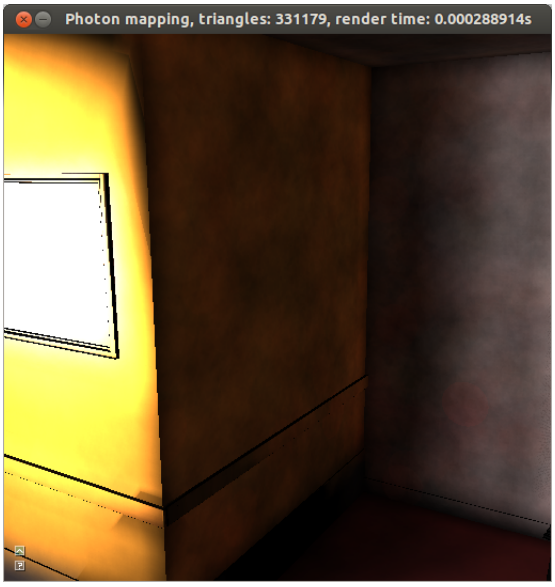
(d) Nepřímé osvětlení v chodbě odvrácené od světla

Obrázek A.4: Test směrového světla simulujícího svit slunce ve scéně Sponza

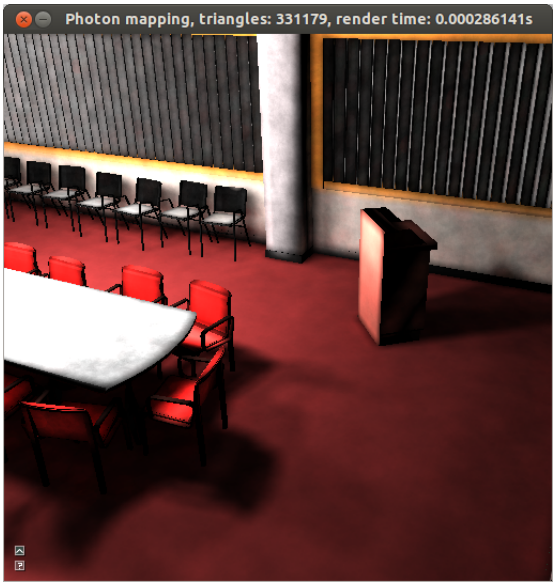


(a) Bez osvětlení

(b) Zapnuté osvětlení



(c) Nepřímé osvětlení v rohu místnosti



(d) Stíny židlí a řečnického pultu

Obrázek A.5: Osvětlení pomocí tří světel ve scéně Conference

Příloha B

Instalace aplikace

Pro správnou instalaci programu je potřeba zachovat adresářovou strukturu aplikace. Aplikace je multiplatformní a lze ji přeložit na systému Windows nebo Linux. V počítači musí být nainstalováno CUDA SDK v aktuální verzi. Dále je nutné nainstalovat program CMake minimální verze 2.8.7. Překlad aplikace ve Windows je podmíněn nainstalovaným prostředím Visual Studio 2009 nebo 2010. Grafická karta musí podporovat OpenGL v minimální verzi 3.3 a tato knihovna musí být v systému přítomna.

Zadáním příkazu `cmake CMakeLists.txt` v kořenové složce aplikace se vytvoří buď soubor **makefile** v prostředí Linux nebo projekt `sln` ve Windows. V rámci provedení tohoto příkazu se také přeloží všechny požadované externí knihovny.

Po přeložení aplikace pomocí **makefile** v Linuxu se vytvoří spustitelný soubor ve složce **bin**, odkud je potřeba jej spouštět. Při překladu aplikace pomocí Visual Studia ve Windows je spustitelný soubor vytvořen v **bin\Release**.

Jedním z problémů, které se mohou vyskytnout při spouštění aplikace, je zamrzání nebo padání programu s následnou nutností restartu počítače. Nejedná se o chybu programu. Zamrzání způsobuje ovladač grafické karty, který se snaží ukončit proces běžící delší dobu, než je povoleno. Ve většině případů je tato doba omezena na 2 sekundy. Zrušení tohoto ukončování lze docílit úpravou registrů ve Windows nebo upravením souboru xorg.conf v Linuxu.

Příloha C

Ovládání aplikace

Ovládání aplikace probíhá pouze přes uživatelské rozhraní. Po spuštění aplikace se objeví dialog patrný na obrázku C.1(a). V tomto dialogu je potřeba zvolit cestu k modelu (relativní nebo absolutní), rozlišení vykreslovaného obrazu, které je poté neměnné, počet generovaných fotonů a velikost jedné strany uniformní mřížky pro scénu. Po potvrzení se objeví hlavní nabídka aplikace, viz obrázek C.1(b), kde jednotlivé položky znamenají:

Camera rotation, x, y, z - rotace kamery a pohyb po scéně.

Radius photons - velikost sběrné koule pro fotony.

Radius caustics - velikost sběrné koule pro kaustiky.

Photons intensity - zvyšování nebo snižování intenzity osvětlení u globálních fotonů.

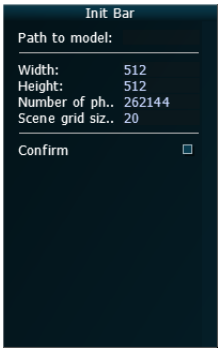
Caustics intensity - zvyšování nebo snižování intenzity osvětlení kaustik.

Scene grid size - změna rozlišení uniformní mřížky scény.

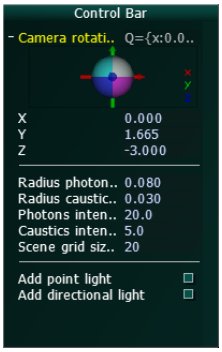
Add point light - přidání všesměrového světla do scény. Po kliknutí se objeví dialog pro nastavení pozice světla a jeho barvy, viz obrázek C.1(c)

Add directional light - přidání směrového světla do scény. Po kliknutí se objeví dialog pro nastavení pozice světla a jeho barvy, viz obrázek C.1(d). Pomocí *Clip left, right, bottom, top* lze vybrat požadovanou část scény, na kterou se má světlo zaměřit.

Pokud vložíme do scény alespoň jedno světlo, objeví se v hlavním dialogu tlačítko ke spuštění photon tracingu, viz obrázek C.1(e). Po provedení photon tracingu je možné spustit sledování paprsku pomocí položky *Ray tracing* a příp. povolit interaktivní mód zaškrtnutím položky *Ray tracing* a *Interactive mode* současně.

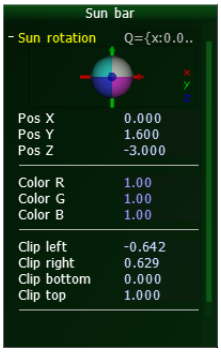


(a) Načtení modelu a počáteční inicializace

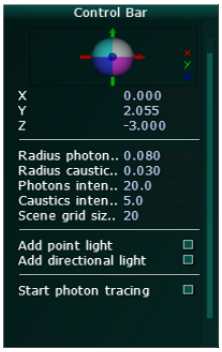


(b) Ovládání pohybu, nastavení parametrů

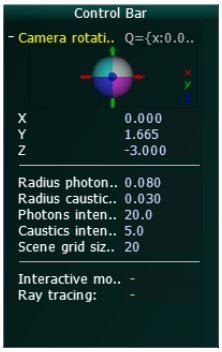
(c) Nastavení všesměrového světla



(d) Nastavení směrového světla



(e) Spuštění photon tracingu



(f) Spuštění sledování paprsku

Obrázek C.1: Uživatelské rozhraní aplikace

Příloha D

Obsah přiloženého CD

POSTER	Vytvořený plakát v elektronické podobě.
PROGRAM	Složka s aplikací.
bin	Zde se vytvoří spustitelný soubor v Linuxu.
Release	Obsahuje potřebné dynamické knihovny pro Windows.
external	Potřebné knihovny třetí strany.
shaders	Shadery pro OpenGL.
src	Zdrojové kódy aplikace.
TEXT	Zdrojový kód této práce vytvořené pomocí LaTeX