České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **David Král**

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: **Srovnání knihoven pro vrhání paprsku**

Pokyny pro vypracování:

Prostudujte pět vybraných knihoven pro vrhání paprsku (viz seznam literatury). Popište vlastnosti a možnosti těchto knihoven a definujte sadu parametrů pro srovnání těchto knihoven. Výsledky srovnání sumarizujte ve formě přehledové tabulky. Proveďte důkladné testy knihoven v rámci jednoduchých demonstračních aplikací vytvořených v jazyce C++, které umožní srovnání jejich efektivity a kvality generovaných výstupů. Pro srovnání použijte nejméně pět různých scén. Výsledky srovnání prezentujte ve formě přehledných tabulek a grafů jak v textu práce, tak ve formě webové prezentace.

Seznam odborné literatury:

[1] NVidia OPTIX. http://www.nvidia.com/object/optix.html
[2] Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In proceedings of High-Performance Graphics, 2009.
[3] Intel Embree. http://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels
[4] PowerVR OpenRL. http://community.imgtec.com/developers/powervr/openrl-sdk/
[5] Mitsuba. http://www.mitsuba-renderer.org/
[6] J. Žára a kol. Moderní počítačová grafika, Computer Press, 2005.
[7] Matt Pharr, Greg Humphreys. Physically Based Rendering, From Theory To Implementation. Morgan Kaufmann, 2010.

Vedoucí: doc. Jiří Bittner Ing., Ph.D.

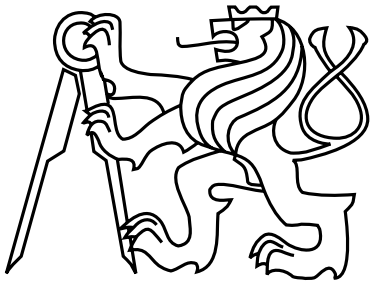
Platnost zadání: do konce letního semestru 2015/2016

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 25. 3. 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Bakalářská práce

Srovnání knihoven pro vrhání paprsku

David Král

Vedoucí práce: doc. Ing. Jiří Bittner, Ph.D.

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Web a multimedia

20. května 2015

Poděkování

Rád bych poděkoval vedoucímu práce doc. Ing. Jiřímu Bittnerovi, Ph.D. za cenné rady, postřehy, názory a pravidelně poskytované konzultace. Dále bych chtěl poděkovat své rodině a přítelkyni za podporu při studiu.

Prohlášení

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

V Teplýšovicích dne 20. 5. 2015 .....

Abstract

Ray tracing is a technique in computer graphics used for rendering images by tracing the path of light through pixels of an image plane. These techniques are trying their best to simulate phenomena that happen, when light interacts with objects in real world. It means that images produced with these techniques attain great realism, but it requires more computational power than techniques generating images much less realistic.

The subject of this bachelor thesis is a comparison of five selected tools that support the creation of applications using casting or tracing ray and this will be achieved through the comparison parameters of these tools and by testing their efficiency and quality.

Abstrakt

Vrhání, či sledování paprsku jsou techniky v počítačové grafice používané ke generování obrázků pomocí sledování cesty světla skrz pixely obrazové roviny. Tyto techniky se snaží co nejvíce simulovat jevy, jenž nastávají při interakci světla s objekty v reálném světě. To znamená, že obrázky produkované těmito technikami dosahují velké realističnosti. Je k tomu ale zapotřebí většího výpočetního výkonu, než je tomu u technik generujících obrázky mnohem méně realistických.

Předmětem této bakalářské práce je porovnat pět vybraných nástrojů, které podporují tvorbu aplikací používajících vrhání, či sledování paprsku. Toho bude dosaženo jak pomocí porovnání parametrů těchto nástrojů, tak pomocí testů jejich efektivity a kvality.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato práce analyzuje a srovnává knihovny, které mají za úkol sloužit k snadné implementaci algoritmů globálních osvětlovacích metod, především těch, které jsou postaveny na vrhání paprsků.

Nejprve jsem rozebral teorii zabývající se problematikou algoritmu vrhání paprsku a pokračoval jsem popisem složitějších algoritmů, které vrhání paprsku rozšiřují, a jedním z nich je sledování cesty (Path Tracing). Poté jsem jednotlivě popsal možnosti a vlastnosti vybraných knihoven. Tato část je zakončena srovnávací tabulkou, která ukazuje, jaké vlastnosti jsou v knihovnách podporovány a jaké ne. Další a poslední částí jsou testy, jenž mají za úkol porovnat rychlost a kvalitu vykreslených obrázků zvolených testovacích scén. Tyto scény jsou až na jednu malou výjimku, kterou popíši dále v textu, vykreslovány algoritmem sledování cesty.

1.1 Teoretická část

V této sekci jsem popsal teorii, která je nutná pro porozumění problematice týkající se vykreslování počítačových scén pomocí některých globálních zobrazovacích metod. Nejprve se věnuji algoritmům postaveným na vrhání paprsku, jako jsou sledování paprsku vyššího řádu, distribuované sledování paprsku nebo sledování cesty. Poté jsem pokračoval částí pojednávající o možnostech zrychlení těchto algoritmů. Mezi těmito možnostmi je použití akceleračních struktur nebo použití výkonnějšího algoritmu pro test průsečíku paprsku s objektem.

1.1.1 Sledování paprsku

V knize [ZBF05] je popsána zobrazovací rovnice, která pro všechny body povrchu objektů ve scéně popisuje vycházející zář. Sledování paprsku je jednou z globálních osvětlovacích metod, které mají za úkol částečně zobrazovací rovnici řešit a vykreslit scénu s větší věrohodností a přesností než je tomu u lokálních metod, kde je osvětlení vypočteno pro každý objekt nezávisle, a kde tedy bez použití speciálních rozšíření není možné vykreslit jevy jako například stíny a vícenásobné a zrcadlové odrazy.

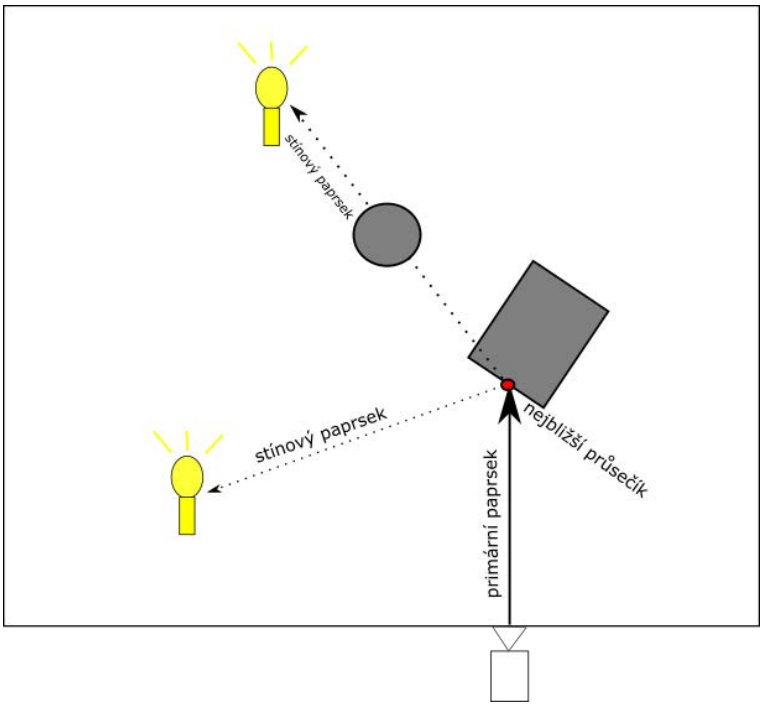
Sledování paprsku je metoda, která vychází od pozorovatele. To znamená, že paprsky jsou vrhány z bodu, kde je umístěna kamera. Je tomu tedy obráceně, než v reálném světě,

kde je světlo šířeno od světelných zdrojů. Základní podoba algoritmu je schopna zobrazit pouze ostré stíny.

Základní algoritmus sledování paprsku, kterému také říká vrhání paprsku, je popsán zde:

```
1. Pro každý pixel vyhledej nejbližší průsečík paprsku s objektem
2. Pokud průsečík nenalezen => přiřaď barvu pozadí
3. Pokud nalezen, vyšli z něj paprsek ke světelnému zdroji (stínový) a vyhodnoť,
je-li průsečík osvětlen.
4. Pokud je osvětlen, tak vyhodnoť výslednou barvu jako funkci vlastností světla
a materiálu objektu
```

Pro každý pixel scény je vyslán primární paprsek, který nalezne nejbližší průsečík s objektem a dále na své cestě scénou nepokračuje. Vrhaní paprsku je naznačeno na obrázku 1.1, kde lze pozorovat, že pro zobrazení stínů je třeba z průsečíku paprsku s objektem vyslat ke každému světelnému zdroji stínový paprsek, který pokud k němu bez překážky dorazí ukáže, zda-li je bod osvětlen.



Obrázek 1.1: Vizualizace algoritmu vrhání paprsku

1.1.2 Sledování paprsku vyššího řádu

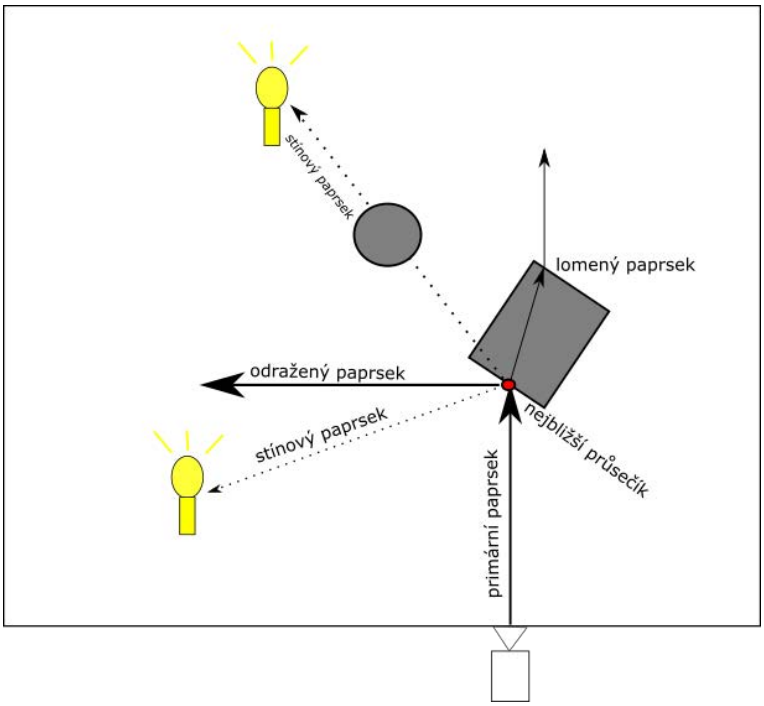
Metoda popsaná v sekci 1.1.1 se dá dále rozšířit na sledování paprsku vyššího řádu (tzv. Whitted-style ray tracing), kde barva na povrchu protnutého tělesa není dána jen materiálem tohoto tělesa, ale i dalšími tělesy, které mohou být protnuty paprsky, jenž jsou z původního průsečíku vrhány v závislosti na koeficientech odrazivosti a průhlednosti. Další

paprsky jsou vrhány do té doby, než dosáhnou zadané maximální hloubky. Nyní je metoda schopna zobrazit i zrcadlové odrazy.

Algoritmus sledování paprsku vyššího řádu je popsán zde:

```
1. Pro každý pixel vyhledej nejbližší průsečík paprsku s objektem
2. Pokud průsečík nenalezen => přiřaď barvu pozadí
3. Pokud nalezen, vyšli z něj paprsek ke světelnému zdroji (stínový) a vyhodnoť,
je-li průsečík osvětlen.
4. Pokud je osvětlen, tak vyhodnoť barvu jako funkci vlastností světla a  materiálu
5. Pokud je hloubka paprsku nižší nebo stejná než maximální, tak vyšli odražený
paprsek, nebo lomený paprsek (záleží na koeficientech odrazivosti a průhlednosti)
a nastav jim hloubku o jedna vyšší
6. Původnímu průsečíku nastav barvu jako funkci barvy z bodu 4 a barev paprsků
z bodu 5
```

Tento algoritmus je také zobrazen na obrázku 1.2, kde lze sledovat, že testování osvětlení bodu funguje stejně jako u vrhání paprsku, jen jsou z průsečíku dále vrhány zrcadlově odražené a lomené paprsky.



Obrázek 1.2: Vizualizace algoritmu sledování paprsku vyššího řádu

Obrázek 1.3: Obrázky vykreslené pomocí vrhání paprsku a sledování paprsku vyššího řádu

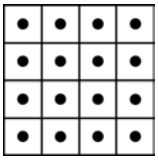
1.1.3 Distribuované sledování paprsku

Někdy je možné poměrně jednoduše algoritmus sledování paprsku rozšířit a dosáhnout tím větší věrohodnosti a kvality výsledného obrázku. Mezi rozšíření patří například zajištění antialiasingu, vzorkování světla tak, aby se chovalo jako plošné, počítání hloubky ostrosti a rozmazání pohybem (Motion Blur). Takto vylepšený algoritmus se nazývá Distribuované sledování paprsku, jak je popsáno např. v [SAM09]. Některá ze zmiňovaných rozšíření jsou popsány v následujících sekcích.

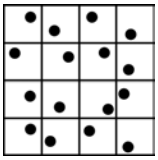
1.1.3.1 Antialiasing

Podle [ZBF05] je aliasing nízkofrekvenční informace, která před vzorkováním v původním signálu nebyla přítomna. Vzniká při nedodržení Shannonova teorému, který říká, že vzorkovací frekvence musí být minimálně dvakrát vyšší, než je maximální frekvence vzorkovaného signálu. Alias se při vykreslování může projevovat například jako zubatá čára na hranách objektů, nebo jako moire při vzorkování hustého pravidelného vzorku.

Při klasickém vzorkování pixelu se pixel rozdělí na subpixely a vzorkuje se vždy uprostřed subpixelu. Výsledná barva je průměrem všech vzorků subpixelů. Tomuto se říká pravidelné vzorkování, které je zobrazeno na obrázku 1.4. Cestou k odstranění aliasingu je tzv. roztřesení (jittering). Zde se postupuje stejně jako u pravidelného vzorkování, ale ze středů subpixelů se vzorky náhodně posunou. Pokud je tedy vzorků naměřeno více, lze za výslednou hodnotu považovat jejich průměr. Jak vypadá roztřesení je znázorněno na obrázku 1.5.



Obrázek 1.4: Pravidelné vzorkování pixelu



Obrázek 1.5: Vzorkování pixelu pomocí roztřesení

1.1.3.2 Měkké stíny

Ve standardním sledování paprsku se počítá pouze s bodovými světly. To znamená, že bod je vždy s jistotou osvětlený, nebo ne. Klíčem k dosažení zobrazení měkkých stínů je udělat ze světla bodového plošné. Jaký je rozdíl mezi měkkými a tvrdými stíny je patrné z obrázku 1.6, který je dostupný na adrese http://i48.tinypic.com/mki2h2.jpg.

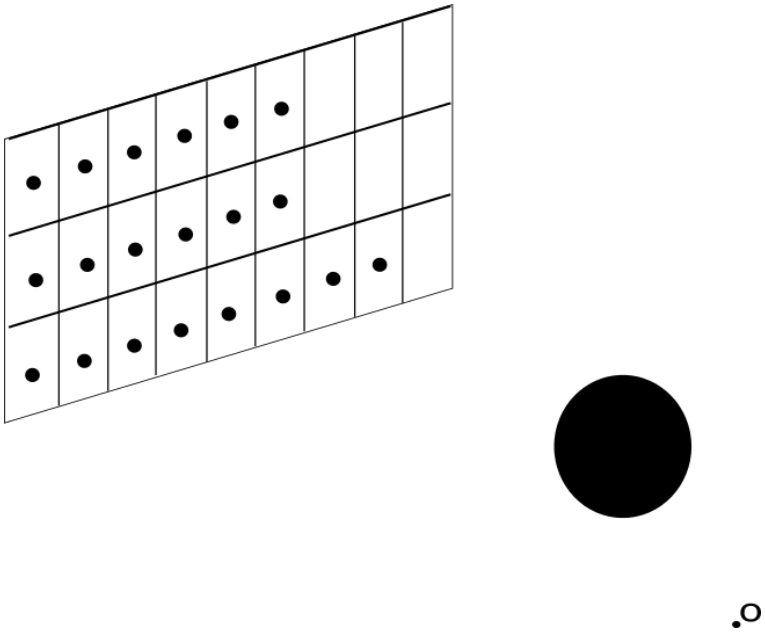
Obrázek 1.6: Rozdíl mezi tvrdým (vlevo) a měkkým (vpravo) stínem

Prvním možným řešením tvorby měkkých stínů je umístit vedle sebe více bodových světel a v bodě průsečíku paprsku s objektem vzorkovat všechna tato světla. To je zobrazeno na obrázku 1.7, kde jsou viditelná bodová světla pro bod O označena černým puntíkem. Problémem této metody je, že pokud je bodových světel málo, jsou přechody mezi „rozdílnými stíny“ ostré a viditelné.

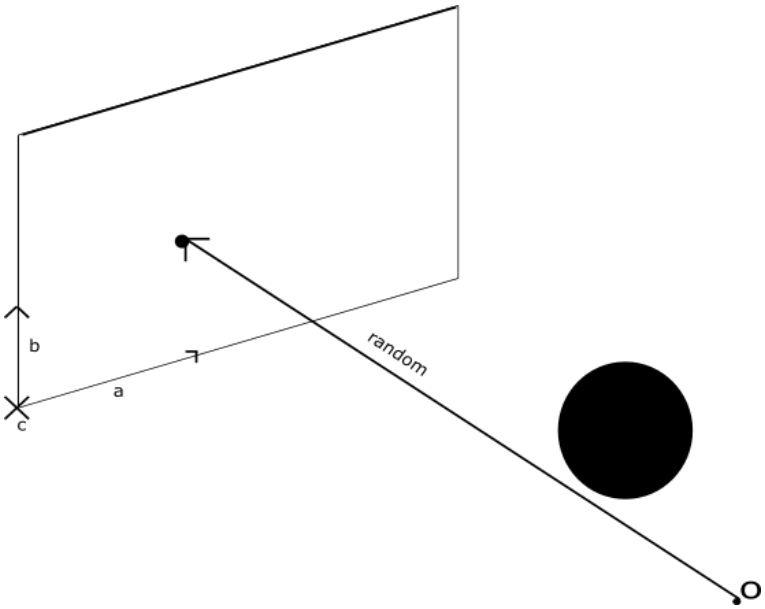
Druhým řešením je představit si, že na určité ploše je bodových světel nekonečné množství a náhodně je vzorkovat. To je zobrazeno na obrázku 1.8. Pokud je tedy plocha určena rohem C a dvěma vektory hran a a b, tak můžeme pomocí dvou náhodných čísel r a s z rozsahu od nuly do jedné vždy dostat bod R, který se nachází na ploše daného světla pomocí vztahu:

$$R = C + ar + bs$$

Jak vypadá obrázek vykreslený vylepšeným sledováním paprsku, popsaným v sekci 1.1.3, je vidět na obrázku 1.9.



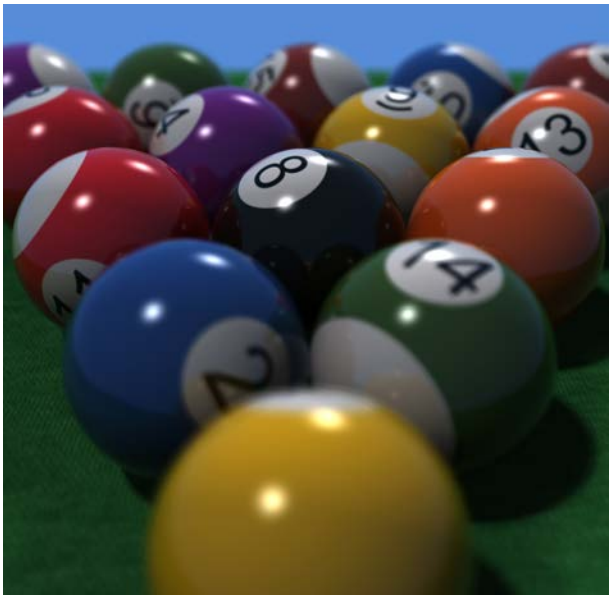
Obrázek 1.7: Vzorkování velkého množství bodových světel



Obrázek 1.8: Náhodné vzorkování plošného světla

1.1.4 Integrace Monte Carlo

Jak jsem již zmínil, globální osvětlovací metody mají za úkol řešit zobrazovací rovnici. Metoda Monte Carlo zjednodušuje toto řešení, protože analytické řešení je velice náročné a někdy



Obrázek 1.9: Vykreslený obrázek pomocí Distribuovaného sledování paprsku

i nemožné. Monte Carlo nám vždy dá jiné řešení, které je závislé na zvolených náhodných číslech. V průměru je ale řešení správné a dostaneme ho průměrováním výsledků několika běhů Monte Carlo metod se stejným vstupem.

Touto metodou chceme většinou vyčíslit složité integrály, které ve vykreslování vznikají. Princip je v [PH10] popsán na příkladě vyhodnocení jedno-dimenzionálního integrálu:

$$\int_a^b f(x)\,\mathrm{d}x$$

kde lze řešení dostat pomocí vzorce:

$$(b-a)/n \sum_{i=1}^{n} f(X_i)$$

pro $X_i \in [a, b]$. Jedná se tedy o aproximaci řešení problému pomocí stochastického vzorkování. V [ZBF05] je uvedeno, že tyto metody pomalu konvergují. Na zdvojnásobení přesnosti řešení je potřeba čtyřikrát více vzorků, protože přesnost roste s odmocninou jejich počtu.

Metoda Monte Carlo se dá dále urychlit. Existuje více možností, jak toho dosáhnout. Jednou z těchto metod je tzv. ruská ruleta, která je většinou implementována v ukázkových aplikacích, které jsou přiloženy k vybraným knihovnám. Tato metoda je dobře popsána v [ZBF05]. Funguje tak, že se snaží přenést výpočty ze vzorků, které jsou na výpočet náročné a mají malý malý vliv na celkový výsledek. K jejímu vykonání potřebujeme znát pravděpodobnost p, že se světlo odrazí do vyšetřovaného směru. S pravděpodobností $1 - p$ se vzorek dále nevyhodnocuje a je nahrazen nějakou konstantní hodnotou (často se používá nula). S pravděpodobností p se vzorek dále vyhodnocuje, ale je vážen vzorcem $1/p$, což vynahrazuje vzorky, které byly přeskočeny. Dalšími metodami urychlení jsou například Importance Sampling nebo Stratified Sampling, které jsou detailně popsány v [PH10]

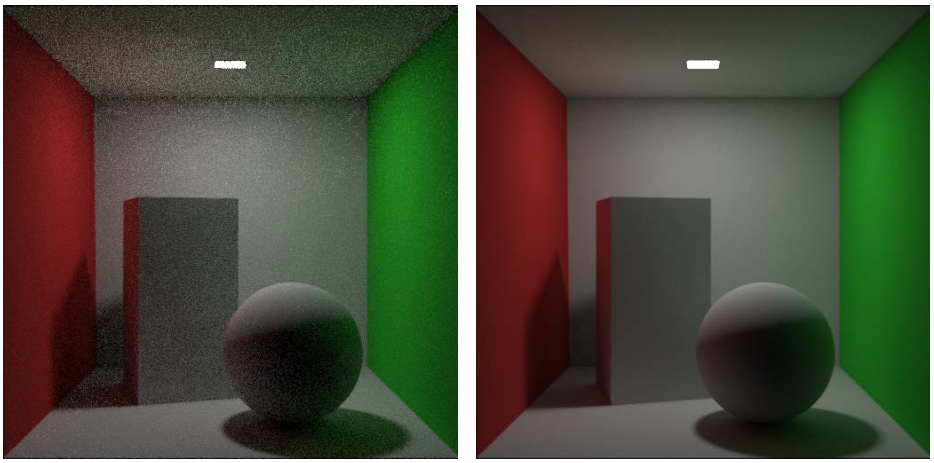
1.1.5 Sledování cesty

Sledování cesty je dalším algoritmem globálních osvětlovacích metod a je popsán například v [ZBF05]. Stejně jako sledování paprsku je to metoda, která vychází od pozorovatele. Největším rozdílem je fakt, že sledování cesty dokáže pracovat s plošnými světelnými zdroji a dokáže počítat příspěvky od nepřímých (difúzních) odrazů a tím dochází k jevu zvanému přenos barvy. Dále dokáže vypočítat kaustiky. Funguje tak, že se každým pixelem vyšle velké množství primárních paprsků a ty se dále sledují. V každém průsečíku se stejně jako u sledování paprsku vyhodnotí lokální osvětlení a vyšlou se stínové paprsky. Zde ale výpočet nekončí. Z průsečíku se vyšle další paprsek směrem, který se určí náhodným vzorkováním funkce BRDF, což je obousměrná odrazová distribuční funkce, která určuje poměr odražené radiance v daném bodě ke vstupní radianci. Výpočet poté pokračuje rekurzivně dále a může skončit například využitím metody ruská ruleta, která byla zmíněna v sekci 1.1.4, nebo nastavením maximální hloubky cesty.

Algoritmus sledování cesty je popsán zde.

```
1. Pro každý vzorek vyhledej nejbližší průsečík paprsku s objektem
2. Pokud průsečík nenalezen => přiřaď barvu pozadí
3. Pokud nalezen, vyšli z něj paprsek ke světelnému zdroji (stínový) a vyhodnoť,
je-li průsečík osvětlen.
4. Pokud je osvětlen, tak vyhodnoť barvu jako funkci vlastností světla a  materiálu
5. Vypočti směr nového paprsku pomocí vzorkování funkce BRDF
6. Vyšli nový paprsek směrem z bodu 5 a jeho příspěvek připočti k výsledné barvě
```

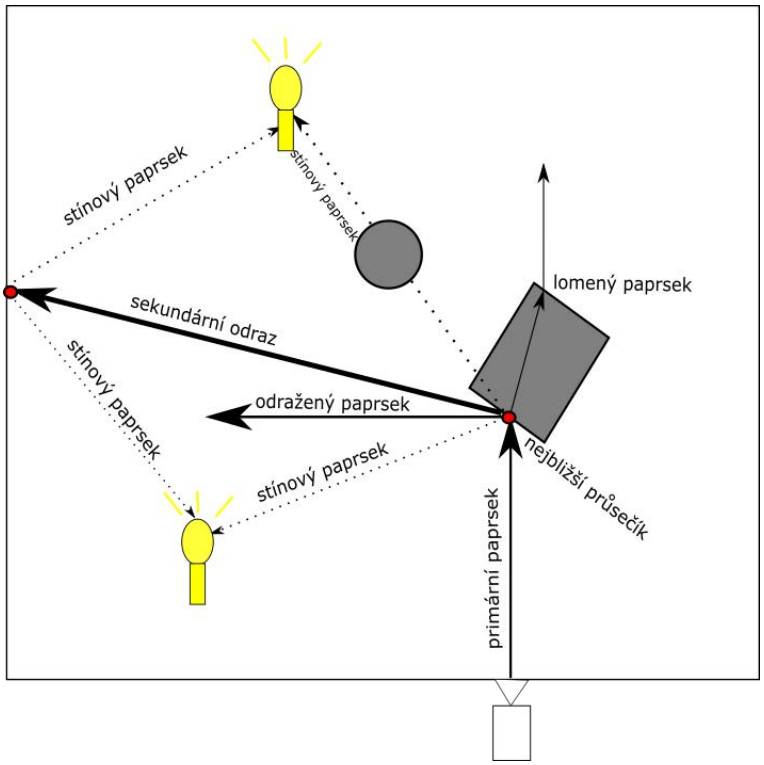
Algoritmus je také zobrazen na obrázku 1.11. Na něm je vidět sekundární odraz z prvního průsečíku paprsku s objektem. Z průsečíku tohoto odraženého paprsku jsou opět vysílány paprsky stínové.



Obrázek 1.10: Obrázky vykreslené pomocí sledování cesty s málo a více vzorky

Sledování cesty používá integraci pomocí dříve zmiňované metody Monte Carlo. Výsledná barva pixelu je totiž průměrem všech vzorků, které byly tímto pixelem vyslány. Na výslednou kvalitu má velký vliv počet vzorků vyslaných pixelem. Bohužel jeho zvyšování má velký vliv

na rychlost sledování cesty. Díky výpočetní náročnosti a realističnosti výsledků jsem tento algoritmus zvolil pro realizaci srovnávacích testů, které jsem popsal v kapitole 3.



Obrázek 1.11: Algoritmus sledování cesty

1.1.6 Urychlování výpočtů nad paprsky a scénou

Je důležité podotknout, že dosud zmiňované vykreslovací algoritmy jsou výpočetně náročné, a proto již bylo představeno několik způsobů jejich urychlení. Mezi metody urychlování výpočtů nad paprsky a scénou patří především:

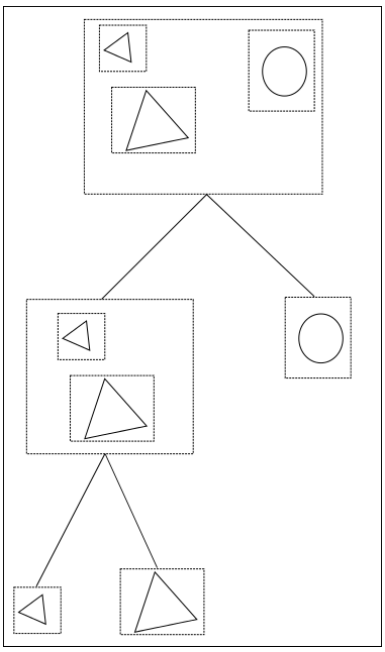
- použití akceleračních struktur
- použití rychlejšího výpočtu průsečíku paprsku s objektem

V následujících sekcích budou tyto metody stručně popsány.

1.1.6.1 Akcelerační struktury

Akcelerační struktury jsou takové struktury, díky kterým je možné výrazně zredukovat počet nutných testů, protne-li paprsek objekt ve scéně. Těchto struktur existuje více typů, ale všechny těží z toho, že je scéna rozdělena na části a tyto části jsou uspořádány hierarchicky. To velmi urychlí jejich procházení (složitost se mění na logaritmickou).

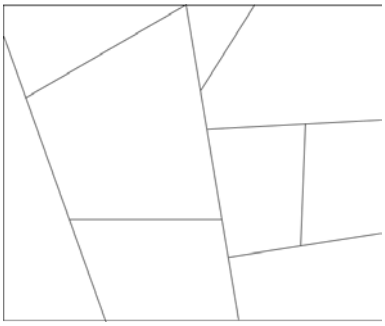
Mezi nejpoužívanější akcelerační struktury se řadí tzv. hierarchie obálek (BVH). Jejich vlastnosti a metody staveb jsou detailněji popsány v [PH10]. V BVH jsou uložena jednotlivá primitiva spolu se svými obálkami ve stromové struktuře. Konkrétní primitiva se svými obálkami jsou uložena v listech a ve vnitřních uzlech jsou uloženy další obálky, které zastřešují menší části scény. Paprsek se nejprve otestuje na průsečík s obálkou vyššího řádu, a pokud není průsečík nalezen, není nutné se zabývat podstromem této obálky. Obálek je několik typů. Mezi nejpoužívanější se řadí koule a osově zarovnaný kvádr (označovaný jako AABB). Jak může vypadat hierarchie obálek pro jednoduchou scénu je naznačeno na obrázku 1.12.



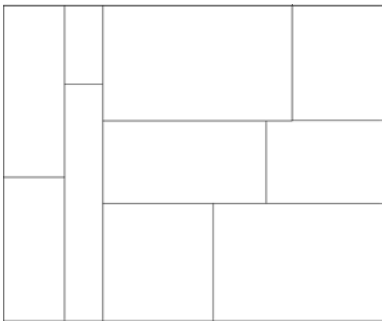
Obrázek 1.12: Jednoduchá hierarchie obálek

Vybrané knihovny pro vrhání paprsku, které jsou v této práci analyzovány, nabízí několik druhů algoritmů stavby a dokonce i průchodu BVH na výběr. Některé metody stavby jsou rychlejší než jiné, ale postaví BVH, která není tak výkonná při průchodu, jako by byla jiná, postavená algoritmem, který je pomalejší než první zmiňovaný. Mezi pomalejší metody stavby patří například SBVH popsána v dokumentaci knihovny NVIDIA Optix [NVIa], která ale postaví velice kvalitní hierarchii, jenž je velice efektivní při procházení. V [PH10] je popsána další používaná metoda: SAH-BVH. Ta poskytuje odpovědi na otázky typu: „Jaké množství rozdělení primitiv povede k větší kvalitě BVH pro testy průsečíků paprsku s objektem?“

Další používanou akcelerační strukturou je tzv. KD-strom. Je to jedna z variant dělení prostoru scény. Těchto variant je také větší množství. Mezi další patří například BSP strom, který adaptivně dělí prostor na nepravidelně velké části. Začíná s obálkou, která zastřešuje celou scénu. Ta je dále rozdělována obecně umístěnými řezy. BSP dělení je možné sledovat na obrázku 1.13. KD-strom oproti BSP stromu určuje, že řezy jsou vždy kolmé na jednu ze souřadnicových os. Toto dělení je možné pozorovat na obrázku 1.14. V [PH10] je opět popsána metoda stavby KD-stromu, která využívá rekurze a staví ho od shora dolů.



Obrázek 1.13: BSP dělení prostoru



Obrázek 1.14: KD dělení prostoru

1.1.6.2 Průsečík paprsku s objektem

Metod, jak vyhodnotit průsečík s objektem ve scéně je velké množství. Pro algoritmy používající akcelerační struktury jsou důležité především testy na průsečík s trojúhelníkem, koulí nebo obecně s polygonem.

V [PH10] i v [SAM09] je popsána metoda testu průsečíku paprsku s trojúhelníkem, která se nazývá Möller, Trumbore metoda a je založena na použití barycentrických souřadnic, které poskytují cestu, jak parametrizovat trojúhelník s vrcholy $p0, p1, p2$ pomocí dvou proměnných b_1 a b_2:

$$p(b_1, b_2) = (1 - b_1 - b_2)p_0 + b_1 p_1 + b_2 p_2$$

Proměnné b_1 a b_2 musí být větší než nula a jejich součet musí být menší nebo roven jedné. Jestliže O je počátek a d je směrový vektor, můžeme paprsek $R(t)$ zapsat jako:

$$R(t) = O + td$$

Rovnice průsečíku je tedy:

$$O + td = (1 - b_1 - b_2)p_0 + b_1 p_1 + b_2 p_2$$

Pomocí techniky, která je zmíněna v článku, kde Möller a Trumbore uvádí tuto metodu, lze

tuto rovnici přepsat na tvar:

$$(-d\, e_1\, e_2) \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = s$$

V této rovnici $e_1 = (p_1 - p_0)$,$e_2 = (p_2 - p_0)$ a $s = (O - p_0)$. Parametry b_1, b_2 a t lze dostat řešením lineární soustavy rovnic pomocí Cramerova pravidla.

Test průsečíku paprsku s koulí je popsán například v [ZBF05]. Jestliže O je počátek a d je směrový vektor, můžeme paprsek $R(t)$ zapsat jako:

$$R(t) = O + td$$

Dále pokud je dána koule se středem v S a poloměrem r, můžeme vytvořit rovnici dosazením souřadnic paprsku do rovnice kulové plochy. Odtud:

$$(o_0 + td_0 - s_0)^2 + (o_1 + td_1 - s_1)^2 + (o_2 + td_2 - s_2)^2 - r^2 = 0$$

Tím získáme kvadratickou rovnici a ta má podle hodnoty diskriminantu 0,1, nebo 2 řešení. Paprsek tedy kouli buď mine, dotkne se jí, nebo jí projde.

Ve srovnávací tabulce, jenž je v sekci 2.6 jsem uvedl některé další metody testů průsečíku paprsku s objekty. Ty jsou používány vybranými knihovnami.

Kapitola 2

Analýza vybraných knihoven

V této kapitole se zabývám jednotlivými knihovnami pro vrhání paprsku, které byly pro tuto práci zvoleny. Je nutné podotknout, že zde je vrháním paprsku myšleno všeobecně vysílání paprsků do scény a řešení jejich případných průsečíků s objekty. Není tedy výhradně implementován jediný zobrazovací algoritmus. Některé knihovny k implementaci poskytují pouze nástroje, některé naopak zvládají vykreslovat scény pomocí spousty globálních osvětlovacích metod.

2.1 Intel Embree

2.1.1 Popis

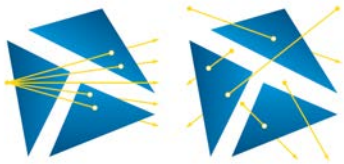
Autoři Embree [WWB+14], [EW11], [Inta] popisují jako open source framework pro vrhání paprsku, který pracuje na x86 centrálních procesorových jednotkách (CPU). Poskytuje výkonná jádra pro úkony nad paprsky a tato jádra lze ovládat pomocí poskytovaného rozhraní, díky kterému lze Embree začlenit do již existujících renderovacích systémů. Embree má za úkol dosahovat velkého výkonu především pro takové renderování, kde jsou používány nekoherentní paprsky. Nekoherentní paprsky jsou takové, které nesdílí počátek a jsou vysílány různými směry bez použití jakéhokoliv předpisu pro jejich šíření. To je rozdíl Monte Carlo metod od metod, kde je pro šíření paprsků využívána jejich koherence. Tyto metody jsou potom omezeny například na vykreslování tvrdých stínů a jednoduchých lomů a odrazů světla. Monte Carlo metody naopak dokáží vykreslit spousty dalších optických jevů. Rozdíl mezi koherentními a nekoherentními paprsky je znázorněn na obrázku 2.2.

Autoři Embree dále poukazují na to, že je velice těžké přizpůsobovat používaná jádra systémů vrhajících paprsky pro hardware, který se velice rychle vyvíjí a mění. Proto velké množství aplikací používajících vrhání paprsku nepoužívá ta nejrychlejší a nejoptimalizovanější jádra pro cílovou architekturu hardware a běží pomaleji, než by mohly.

2.1.2 Architektura

Jak jsem již zmínil, Embree poskytuje výkonná jádra pro vrhání paprsku. Ta podporují úkony jak nad průniky jednotlivých paprsků, tak nad průniky celých paketů. Pakety paprsků jsou

Obrázek 2.1: Ukázkový model, který je použit pro prezentaci ukázkového rendereru, jenž je na Embree postaven. Tento model byl poskytnut Martinem Lubichem (www.loramel.net) a HDR světlo firmou Lightmap Ltd (www.lightmap.co.uk)

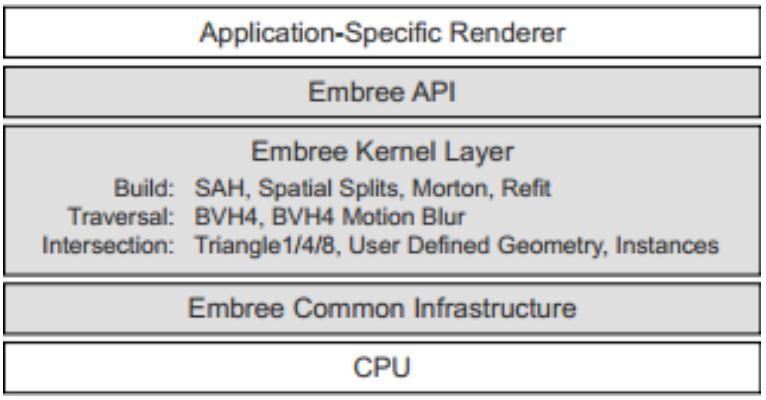


Obrázek 2.2: Koherentní a nekoherentní paprsky [EW11]

paprsky zařazené do takzvaných vektorových pruhů (vector lanes) a scénou poté prochází celý tento pruh. Mezi pakety a jednotlivými paprsky lze dynamicky za běhu programu volit, ale pro nekoherentní paprsky se doporučuje použít paprsky jednotlivé. Pakety se vyplatí použít pro případy, kde se používají paprsky koherentní.

Jaké jsou komponenty Embree a jaká je jejich struktura je dobře vidět na obrázku 2.3.

Použití Embree spočívá především v začlenění jeho jader do existujících aplikací a využití jeho velkého výkonu. Tato jádra implementují hlavně stavby a průchody akceleračních struktur a výpočty nad průsečíky paprsků s objekty. Jsou implementována tak, aby využívala paralelismu, vláken a synchronizace a dokázala to na různých platformách, operačních systémech a s použitím různých kompilátorů.



Obrázek 2.3: Struktura Embree [WWB+14]

Jako akcelerační struktury používá Embree výhradně BVH. Obsahuje také několik druhů staveb těchto struktur. Mezi ně patří SAH-BVH konstrukce popisovaná v sekci 1.1.6.1, nebo Morton-Code BVH konstrukci, která dosahuje největšího výkonu. V listech BVH může být uloženo například primitivum s označením $bvh4.triangle4i$, které je zde uloženo přímo, takže je BVH v paměti uložena kompaktně, nebo mohou obsahovat například $bvh4.triangle4$, což je pouze ukazatel do paměti, kde je primitivum skutečně uloženo. Použitím druhého typu se zrychlí průchod BVH, ale je potřeba více paměti. Jako další možnost se nabízí uložení primitiva typu $bvh4mb$, které je použito při vykreslování efektu Motion Blur (rozmazání pohybem). Tento typ ukládá dvě kolekce obálek primitiva a tyto kolekce jsou při průchodu interpolovány.

Embree umožňuje několik druhů reprezentací primitiv. Jsou to $triangle[1, 4, 8][i, v, n]$. Každá reprezentace obsahuje vrcholy, indexy vrcholů a data o normálách. Dále také obsahuje metodu na test průsečíku. Jednou z těchto metod je i metoda Möller, Trumbore, která byla popisována 1.1.6.2. Test na průsečík může být vykonáván paralelně pro více trojúhelníků (například pro 4, nebo 8), v závislosti na používané instrukční sadě.

2.1.3 Použití

Jak jsem již zmínil, Embree je doporučeno používat pomocí svého rozhraní. Jeho použití je naznačeno na obrázku 2.4. Je možné specifikovat geometrii, přiřadit jí různé vlastnosti (statická, dynamická, Motion Blur, hair), dokonce zadat vlastní geometrii, nebo nastavit parametry pro stavbu BVH.

Struktura paprsku může vypadat následovně:

```
struct RTCray{
**vstupy**
vec3f org;
vec3f dir;
float tnear;
float tfar;
```

```
// create a container for scene geometry
RTCScene scene = rtcNewScene(...);

// add a triangle mesh object to the scene
unsigned id = rtcNewTriangleMesh(scene, ...);

// write vertex pcsitions into vtx[] array
vtx = rtcMapBuffer(scene, id, RTC_VERTEX_BUFFER);
...
rtcUnmapBuffer(scene, id, RTC_VERTEX_BUFFER);

// write vertex indices per face into tri[] array
tri = rtcMapBuffer(scene, id, RTC_INDEX_BUFFER);
...
rtcUnmapBuffer(scene, id, RTC_INDEX_BUFFER);

// indicate that the scene is fully defined
rtcCommit(scene);

// initialize a ray
RTCRay ray;
...

// hit result is returned in ray.{geomID, ...}
rtcIntersect(scene, ray);

// discard the scene and its contents
rtcDeleteScene(scene);
```

Obrázek 2.4: Použití Embree API [WWB+14]

```
float time;
**********
**výstupy**
vec3f Ng;
float u,v;
int geomID;
int primID;
int instID;
**********
}
```

Kde *org* a *dir* jsou počátek a směr paprsku, *tnear* a $tfar$ jsou intervaly paprsku a *time* je používán k Motion Blur. $tfar$ je vzdálenost průsečíku, Ng je nenormalizovaná normála geometrie, u a v jsou lokální souřadnice průsečíku, $geomID$ je ID geometrie, $primID$ je ID primitiva geometrie a $instID$ je ID instance. Nad paprsky jsou metody *rtcIntersect* a *rtcOccluded*, které reportují první průsečík s objektem, nebo jakýkoliv průsečík s objektem. Dále je možné použít metodu *intersectionFilter*, která může například akceptovat, nebo ignorovat průsečík. To může být použito např. pro kolekci všech průsečíků po cestě paprsku.

K Embree je také připojeno několik tutoriálů, které pomohou lépe porozumět používání

této knihovny. Dále byl vytvořen ukázkový renderer, který implementuje algoritmus sledování cesty. Tento renderer existuje ve dvou provedeních. Prvním provedením je skalární renderer. Ten je určen pro analýzu výkonnosti Embree. Druhým provedením je vektorizovaný renderer. Ten demonstruje potencionálně dosažitelný výkon Embree v plně paralelizovaných profesionálních renderovacích aplikacích a používá Intel SPMD Program Compiler (ISPC) [Intb], který pomáhá využívat SSE, AVX, AVX2 a Xeon Phi™ instrukce. Embree ukázkový renderer se řídí stylem PBRT, který je popsán v [PH10]. Obsahuje integrátory, samplery, materiály a obousměrné odrazové distribuční funkce (BRDF). Dále obsahuje ambientní, směrová, bodová a HDRI světla. Integrátor sledování cesty je jednosměrný (od oka) a používá quasi-Monte Carlo vzorování, ruskou ruletu a lokální vyhodnocování přímého osvětlení. Není zde podpora pro další globální osvětlovací metody, jako například Photon Mapping a dvousměrové sledování cesty.

Příklad použití ukázkového rendereru je zde:

```
-i \BP_DVD\Scenes\cornell\cornell.obj
-vp 278 274.4 1383.27
-vi 278 274.4 279.6
-vu 0 1 0
-fov 37
-renderer pathtracer { depth = 5 spp = 8 }
-gamma 2.0
-quadlight 302.375000 546.799988 298.387512 -48.75 0 0 0 0 -39.375 100 100 100
```

Dále je Embree použito v rendereru Corona, nebo pro tvorbu náhledového pluginu do programu Autodesk Maya 2014.

2.2 NVIDIA Optix

2.2.1 Popis

Obrázek 2.5: Scéna vykreslená pomocí aplikace Design Garage, která je vytvořena pomocí Optixu

Systém NVIDIA Optix [PBD+10] je popisován jako knihovna pro vrhání paprsku, vytvořená pro grafické procesorové jednotky (GPU) od firmy NVIDIA, a jiné vysoce paralelní architektury. Jeho jádrem je takzvaný just-in-time kompilátor, který generuje algoritmy založené na vrhání paprsku kombinováním programů zadaných uživatelem. Tyto programy obstarávají generování paprsků a stínování objektů. Dále definují, jak paprsek prochází scénou a řeší jeho průsečíky s objekty. Vytvořené algoritmy mohou být použity pro vykreslování obrázků, detekci kolizí nebo pro zkoumání fyzikálních jevů, jako například šíření akustických vln. Pro jednoduché použití se v Optixu programuje nad jednopaprskovým modelem s plnou podporou rekurze. Neprogramuje se zde pomocí paketů paprsků nebo konstrukty SIMD, jako je tomu u jiných knihoven.

Autoři dále zmiňují, jak je knihovna navržena. Především se poukazuje na to, že nejsou zabudovány speciální koncepty jako světla, stíny nebo odrazivosti a jsou podporovány pouze mechanismy pro vyjádření interakce paprsku s objekty. Dále se poukazuje na to, že algoritmy postavené na vrhání paprsku je možné sestavit kombinováním malého množství programovatelných operací a Optix toho skutečně využívá. Tím se velmi podobá rasterizačnímu rozhraní OpenGL. Při používání knihovny je programátor odstíněn od nutnosti implementovat algoritmy, jakými jsou například stavba či průchod akceleračních struktur nebo řazení paprsků. Optix se dále snaží přizpůsobovat vrhání paprsku dostupnému hardware tak, aby bylo co možná nejefektivnější. Scéna v něm je reprezentována jako objektový model, který využívá dědičnosti pro kompaktní reprezentaci parametrů. Tento objektový model staví graf scény, která je tak organizována co možná nejefektivněji a umožňuje například instancování, nebo

použití vnořených akceleračních struktur.

2.2.2 Architektura

Systém Optix je rozdělen na dvě části. První částí je hostitelské rozhraní, které umožňuje nastavit objekt zvaný **Context**, sestavit graf scény a spustit jádra pro vrhání paprsku. Toho je dosaženo programováním v jazyce C a vykonáváním programu na CPU. Context obsahuje všechny informace o scéně a programovatelných operacích, které jsou k ní přiřazeny. Jak již bylo řečeno, scéna je reprezentována jako graf. Ten může obsahovat následující uzly:

- **Group** má svojí akcelerační strukturu a může být použit jako kořen procházeného grafu.
- **Geometry Group** je list a může obsahovat objekty primitiv a materiálů. Také může mít svou akcelerační strukturu.
- **Transform** má přiřazenu matici 4x3 pro vykonávání afinních transformací a musí mít přiřazeného právě jednoho potomka.
- **Selector** má přiřazen Visit Program, který je použit při procházení potomků.

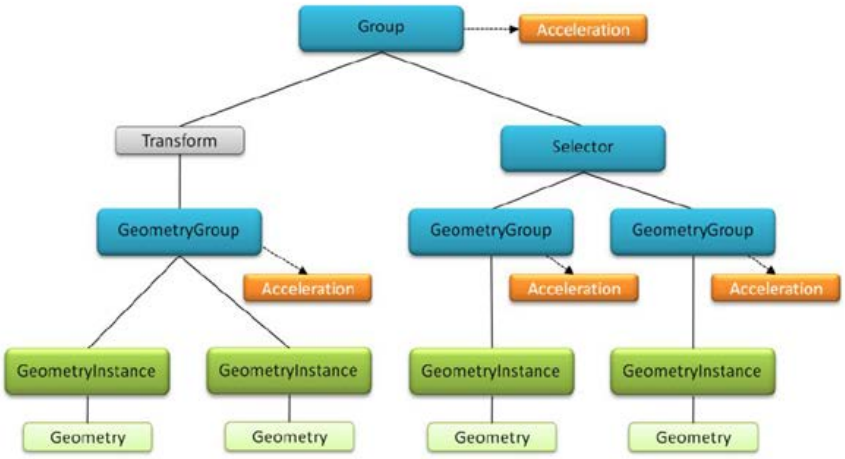
Uzel **Geometry Group** může obsahovat tyto objekty:

- **Geometry Instance** mapuje Geometry Objekt na sadu materiálů.
- **Geometry** obsahuje seznam primitiv. Každý má přiřazen program pro výpočet obálky a pro řešení průsečíku paprsku s ním a tyto programy jsou sdíleny všemi primitivy v tomto objektu.
- **Material** obsahuje informace o výpočtech stínování a programech, které jsou pro tyto výpočty používány.

Ukázkový graf scény je naznačen na obrázku 2.6.

Do objektu **Context** je dále možné nastavit proměnné, které budou přístupné kdykoliv během vykonávání vrhání a procházení paprsků scénou. Příklad takovéto proměnné může být pole s informacemi o všech světlech ve scéně. Analogií těchto proměnných jsou v OpenGL proměnné *uniforms*. Druhou variantou proměnných je takzvaný atribut. Ty mohou být použity pro komunikaci mezi programem, který je vykonán při nalezení průsečíku paprsku s objektem a programy pro nejbližší nebo jakýkoliv průsečík. V OpenGL jsou analogicky zavedeny proměnné *varyings*. Dále je možné nastavit několik druhů paprsků pro rozdílné chování, například stínové a primární.

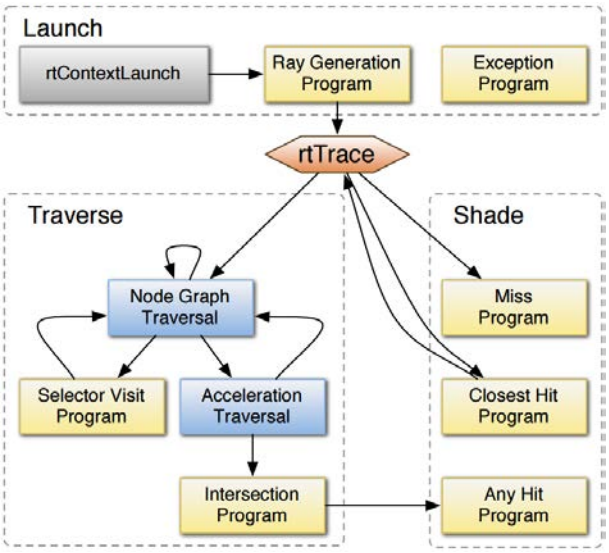
Druhou částí Optixu jsou programovatelné operace nad paprsky. Tak jako se v OpenGL vytvářejí algoritmy založené na rasterizaci pomocí vykonávání shaderů, tak lze v Optixu vytvořit algoritmy založené na vrhání paprsku pomocí sady programů zadaných programátorem vykonávaných na GPU. Těchto programů je několik druhů:



Obrázek 2.6: Graf scény v Optixu. Z dokumentace [NVIa]

- **Ray Generation** Pomocí volání *rtContextLaunch* se vytvoří několik instancí tohoto programu. Tento program může například nastavit model kamery, vysílat paprsky a těm nastavit parametry, nebo ukládat výslednou barvu do výstupního bufferu pod indexem aktuální instance programu.
- **Intersection** implementuje testy průsečíků, při jeho nalezení se může spočítat normála, souřadnice textur a další atributy založené na souřadnicích průsečíku.
- **Bounding Box** umožňuje vypočítat obálku spojenou s primitivem.
- **Closest Hit** je vyvolán při nalezení nejbližšího průsečíku paprsku s geometrií scény. Typicky provádí výpočty jako například stínování, vysílání dalších paprsků, nebo ukládání barvy do atributu paprsku.
- **Any Hit** je volán pro každý průsečík, který je nalezen. Může například ukončit paprsek pomocí *rtTerminateRay*, nebo ho ignorovat pomocí *rtIgnoreIntersection* například v závislosti na materiálu objektu.
- **Miss** nastane, když se paprsek nesetká s žádným objektem. Používá se například pro přiřazení barvy pozadí.
- **Exception** se používá pro případy, že v systému nastane nějaká výjimka, například při vyčerpání paměti.
- **Selector visit** umožňuje programovatelnost průchodu uzlovým grafem. Příkladem je rozhodnutí, jak pokračovat v průchodu v závislosti na délce paprsku.

Jak probíhá vykonávání programů v Optixu je naznačeno na obrázku 2.7.



Obrázek 2.7: Programy v Optixu [PBD+10]

Po vykonání těchto programů lze výsledná data číst z výstupních bufferů a ta mohou být využita například pro zobrazení vykresleného obrázku nebo mohou sloužit jako vstup pro další fázi vykonávání programů spuštěnou pomocí *rtContextLaunch*.

I Optix samozřejmě disponuje akceleračními strukturami. Jak již bylo naznačeno, jednotlivé části grafu scény mohou obsahovat svoji akcelerační strukturu. Toho se může využít například, pokud je část scény dynamická a potřebuje svoji akcelerační strukturu znovu vytvářet při každém průchodu. Další výhodou Optixu je takzvané instancování. Instancování znamená replikaci již existující geometrie vytvořením nového odkazu na ni. S odkazem na existující geometrii se doporučuje vytvářet i odkazy na asociovanou akcelerační strukturu. Takto vytvořená instance může být spojena s různými objekty materiálů nezávisle na původní instanci. Mezi typy akceleračních struktur používaných v systému Optix patří například SBVH zaměřená na kvalitu hierarchie nebo LBVH zaměřená na rychlost konstrukce. Všechny používané struktury lze dohledat v dokumentaci dodávané k instalačnímu balíčku Optixu. Při stavbě akceleračních struktur se nejprve pro každou geometrii zavolají programy **Bounding Box**, které vrátí příslušnou obálku. Dále se pak jádra Optixu postarají o vytvoření zadané struktury, skládající se z těchto obálek.

Jak jsem již zmínil v úvodu, jádrem knihovny je takzvaný just-in-time kompilátor. Ten zkombinuje všechny programovatelné části poskytnuté uživatelem, zanalyzuje graf scény a pokusí se vyhledat možné optimalizace. Nakonec vytvoří jádro, které je vykonáváno na GPU pomocí systému CUDA. Programy od uživatele jsou v hostitelském rozhraní k dispozici ve formě PTX (Parallel Thread Execution), protože CUDA kompilátor se postará o jejich převedení z CUDA C formy, ve které je uživatel programuje.

2.2.3 Použití

Při používaní knihovny Optix je nejdůležitějším krokem nastavit objekt **Context**. Ten zastřešuje všechny další využívané objekty jako textury, geometrie, programy definované uživatelem atd. Je třeba ho vytvořit pomocí funkce *rtContextCreate*. Dále je třeba natavit vstupní bod pro paprsky. Těch může být několik a jejich počet se nastaví např. pomocí volání $rtContextSetEntryPointCount(context, 1)$, které nastaví počet bodů na jeden. Poté je třeba nastavit **Ray Generation** a **Exception** programy. To lze pomocí funkcí *contextSetRayGenerationProgram* a *rtContextSetExceptionProgram*. Každý vstupní bod musí mít přiřazen alespoň **Ray Generation** program.

Dalším krokem je nastavení druhů paprsků. Těch je možné nastavit velké množství. Každý druh může být vysílán za různým účelem a může mít jiné vlastnosti jako například parametry, program pro nejbližší průsečík, program pro jakýkoliv průsečík a program pro žádný průsečík (miss). Počet druhů paprsků se nastaví pomocí funkce *rtContextSetRayTypeCount*. Jak mohou vypadat struktury pro jednotlivé druhy paprsků je naznačeno zde:

```
struct RadiancePL{
float3 color;
int recursion_depth;
};

struct ShadowPL{
float3 attenuation;
};
```

Ray Generation program poté může tyto paprsky vytvořit a vyslat pomocí volání $rtTrace(top\_object, ray, payload)$. V dalších vyvolaných programech může být cesta paprsku například ukončena pomocí funkce *rtTerminateRay*. Nad objektem **Context** může být vytvořeno mnoho dalšího nastavení a ta jsou popsána detailně v dokumentaci knihovny Optix.

Po zavolání funkce *rtTrace* je procházen graf scény od jejího kořene. Jsou tedy procházeny objekty popsané v sekci 2.2.2. Ty lze nastavit stejně jako objekt **Context**. Objekt **Geometry** je vytvořen pomocí funkce *rtGeometryCreate*. Dále je nutné přiřadit mu **Bounding Box** a **Intersection** programy pomocí *rtGeometrySetIntersectionProgram* a *rtGeometrySetBoundingBoxProgram*. Dalším objektem je **Material**, který musí obsahovat především **Any Hit** a **Closest Hit** programy. Ty lze nastavit pomocí volání funkcí *rtMaterialSetClosestHitProgram* a *rtMaterialSetAnyHitProgram*. Dalšími důležitými objekty jsou **GeometryInstance** a **GeometryGroup**. Jak tyto objekty nastavovat a propojovat je detailně popsáno v dokumentaci Optixu. Velice důležitými objekty jsou akcelerační struktury. Ty lze vytvořit pomocí funkce *rtAccelerationCreate*. Součástmi akceleračních struktur jsou *builders* (obstarají tvorbu) a *traversers* (obstarají průchod). Ti se nastavují pomocí *rtAccelerationSetBuilder* a *rtAccelerationSetTraverser*.

Poslední programovatelnou částí jsou programy psané v CUDA C jazyce a předávané do hostitelského rozhraní v podobě PTX stringu. Programy lze **Contextu** přiřadit například pomocí volání $rtProgramCreateFromPTXFile(context, filename, function, program)$. Tyto programy dále komunikují s hostitelským rozhraním pomocí dříve zmiňovaných proměnných

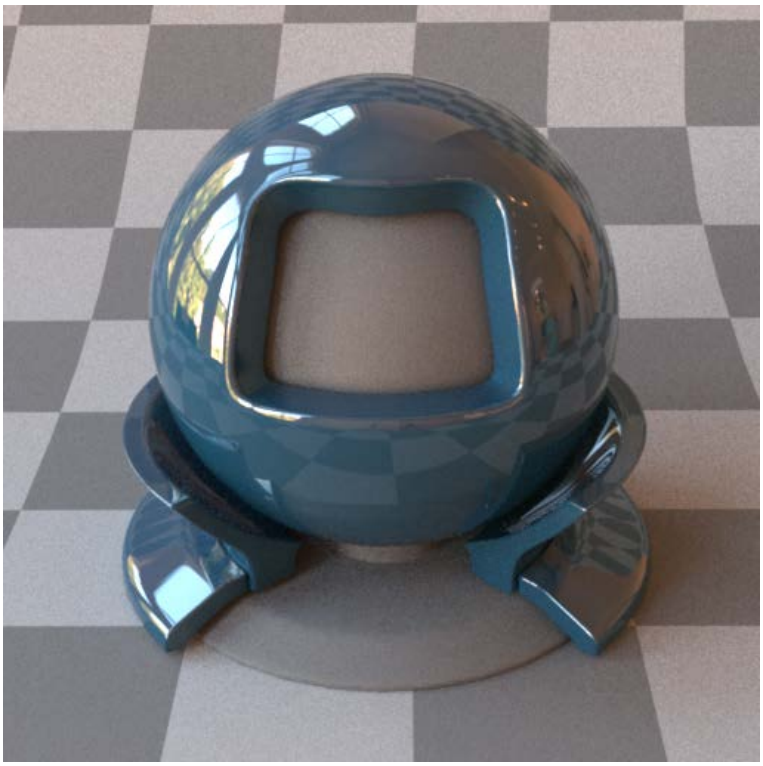
a ty jsou deklarovány pomocí *rtDeclareVariable*. Jak může vypadat ukázkový **Ray Generation** program je znázorněno zde:

```
rtBuffer<uchar4,2> outputBuffer;
rtDeclareVariable(uint2,index,rtLaunchIndex, );
rtDeclareVariable(rtObject,topObject, , );
rtDeclareVariable(float3,eye, , );
rtDeclareVariable(float3,U, , );
rtDeclareVariable(float3,V, , );
rtDeclareVariable(float3,W, , );

struct Payload{
uchar4 result;
};

RT_PROGRAM void pinhole_camera(void){
uint screen = outputBuffer.size();
float2 d = make_float2(index)/make_float2(screen) * 2.f - 1.f;
float3 origin = eye;
float3 direction = normalize(d.x * U + d.y * V + W);
optix::Ray ray = optix::make_Ray(origin,direction,0,0.05f,RT_DEFAULT_MAX);
Payload payload;
rtTrace(topObject,ray,payload);
outputBuffer[index] = payload.result;
}
```

Ke knihovně Optix je přiloženo velké množství ukázkových aplikací, které implementují zobrazovací algoritmy od vrhání a sledování paprsku, přes sledování cesty, až po mapování fotonů. Je také přiložen tutoriál, který je zaměřen především na seznámení nových uživatelů s programováním v CUDA C jazyce. Tutoriál je rozdělen na jedenáct částí, které postupně přidávají funkčnosti od vykreslování objektů, zobrazování stínů a zrcadlových efektů, až po texturování a zobrazování průhlednosti. Dále existuje interaktivní vykreslovací aplikace, která je na Optixu postavena a implementuje Monte Carlo sledování cesty. Její název je Design Garage. Ta navíc používá další knihovnu pro průchod paprsků scénou a tou je Scenix [NVIb]. Scenix navíc zajišťuje interaktivitu celé aplikace a stará se o celý graf scény. Je napsán v jazyce C++ a poskytuje velké množství znovupoužitelných hierarchických struktur. Data jsou redukována a shromažďována tak, aby aplikace využívající tuto knihovnu byly co možná nejefektivnější.

2.3 Mitsuba

2.3.1 Popis



Obrázek 2.8: Typická scéna pro Mitsubu, která se používá pro testování materiálů

Mitsuba [Wen10] není primárně určena k vývoji aplikací založených na vrhání paprsku, ale je to již hotový vykreslovací systém. Ten je napsán ve stylu PBRT [PH10] a byl vyvinut k výzkumným účelům. Mitsuba je napsána v přenosném C++ programovacím jazyce a je zaměřena pro práci na centrálních procesorových jednotkách, pro které obsahuje velké množství optimalizací, jako například optimalizace pro instrukční množinu SSE2 pro x86 a x86-64 platformy.

Tento systém je postaven na seskupení velkého množství samostatných částí a knihoven. Obsahuje velké množství implementovaných vykreslovacích algoritmů a zaměřuje se hlavně na experimentální techniky, které zatím nenašly uplatnění v průmyslových rendererech. Tyto techniky lze používat přes příkazový řádek, nebo přes přehledné grafické rozhraní.

2.3.2 Architektura

Jak jsem již zmínil, Mitsuba se skládá z velkého množství pluginů a částí. Tyto části jsou implementovány pomocí propojených tříd a ve zdrojových kódech i dokumentaci lze snadno vypozorovat, jak fungují, či případně jejich funkčnost upravit. Hlavní rozdělení Mitsuba rendereru lze naznačit takto:

- **Core library** je knihovna, která implementuje základní funkčnosti, jako například přenositelnost mezi platformami, řízení vstupů a výstupů, vytváření datových struktur, nebo spravování pluginů.
- **Rendering library** je knihovna, která obsahuje abstrakce nutné pro načtení a reprezentaci scény.
- **Hardware acceleration library** je knihovna, která implementuje především OpenGL obal, který je potřebný pro finální zobrazení vykresleného obrázku, nebo podporu pro interaktivní zobrazení načtené scény, která je poté připravena k vykreslení pomocí zvoleného algoritmu.
- **Bidirectional library** je knihovna, která obsahuje vrstvu pro podporu obousměrných zobrazovacích algoritmů, jakým je například obousměrné sledování cesty.
- Dále Mitsuba obsahuje podmnožinu těchto knihoven, které jsou napsány pomocí programovacího jazyka Python.

Mitsuba umožňuje použití pomocí rozdělení několika jejích instancí do skupiny spolupracujících výpočetních zařízení, které mohou pracovat na společném úkolu. Podařilo se tak Mitsubu přinutit pracovat na více než tisíci jádrech, která spolupracovala na vykreslování stejného obrázku. Dále se Mitsuba snaží spořit paměť výpočetního zařízení, takže je možné na spotřebním hadware načíst i scény, které obsahují více než třicet milionů trojúhelníků.

Pro větší realističnost vykreslených obrázků Mitsuba podporuje velké množství modelů rozptylu světla na povrchu objektů. Obsahuje Lambertian surfaces, což jsou povrchy, ze kterých je světlo rozptýleno rovnoměrně do všech směrů nezávisle na úhlu dopadu světla. Dále obsahuje průhledné nebo odrazivé materiály, další obousměrné odrazové distribuční funkce (BRDF), stejně jako funkce pro simulaci podpovrchového rozptylu světla (BSSRDF) a další modely, které jsou určeny pro vykreslování objemů. Tyto objemy mohou být reprezentovány voxelovými oktanovými stromy nebo hierarchickými mřížkami. Oba druhy reprezentace lze najít v [ZBF05].

Jádro pro vrhání paprsku používá jako akcelerační struktury výhradně KD stromy postavené pomocí metody SAH (surface area heuristic), které dokáží být postaveny se složitostí O(n log n). Tento přístup stavby KD stromů je popsán v [WH06]. K rychlému průchodu KD stromu používá rychlý traverzační algoritmus, který navrhl Havran a kol. [HKBŽ97]. Na platformách Intel je možné vysílat pakety koherentních paprsků pomocí SSE2 množiny instrukcí.

Samotné vykreslovací algoritmy jsou implementovány v podobě integrátorů. Těch je dostupné poměrně velké množství. Jejich neúplný výčet je zde:

- Zastínění okolím (Ambient occlusion)
- Přímé osvětlování (uvažuje pouze světlo přímo dopadající ze světelného zdroje. Například Ray Casting)
- Monte Carlo sledování cesty (Monte-Carlo Path Tracing)
- Mapování fotonů (Photon Mapping)

- Sledování adjungovaných částic (Adjoint Particle Tracing)
- Obousměrné sledování cesty (Bidirectional Path Tracing)
- Instantní radiosita (Instant Radiosity)
- Progresivní mapování fotonů (Progressive Photon Mapping)
- Metropolis přenos světla (Path Space Metropolis Light Transport)
- a další

2.3.3 Použití

Mitsuba pro specifikaci parametrů scény používá soubory ve formátu XML. Pomocí těchto souborů lze nastavit specifikovat:

- **Tvary** vykreslovaných objektů, které jsou asociovány se svými odrazovými funkcemi, materiály a texturami.
- **BRFDs**, které popisují jak světlo interaguje s povrchem objektu
- **Textury**, které jsou asociovány s odrazovými funkcemi
- **Objemy** pro definici zdrojů dat objemů
- **Zdroje světla** pro nastavení tvarů a typů zdrojů světel
- **Senzory** pro nastavení parametrů a typů kamer
- **Integrátory** pro nastavení parametrů vykreslovacích algoritmů
- **Vzorkovač** pro nastavení typů a parametrů generátorů vzorků pro vykreslovací algoritmy
- **Filmy** pro definici archivace výstupních dat z vykreslovacích algoritmů
- **Filtry** pro nastavení parametrů pro použité filtry pro rekonstrukci obrázku z výstupních dat z vykreslovacích algoritmů

Lze vybírat z velkého množství použitelných tvarů. Mezi nimi jsou například disky, krychle a koule, nebo je možné načíst data z formátu obj a ply. Velké množství druhů na výběr nabízí Mitsuba i v případě odrazových funkcí, které definují materiál objektu. Materiály mohou být například Lambertovský (difuzni), průhledné, zrcadlové, plastické nebo anizotropní. Textury lze generovat například jako šachovnice nebo mřížky. Při nastavování zdrojů světla lze vybrat například bodový, směrový nebo plošný, který musí být asociován s objektem, který je zadán tvarem. I modelů kamer je na výběr mnoho. Mezi nimi jsou perspektivní,ortografické nebo sférické. Všechny druhy popisovaných nastavení lze dohledat v dokumentaci Mitsuby.

Ukázka části XML Mitsuba souboru, kde je mimo jiné vidět, jak musí být plošné světlo asociováno se svým objektem, je zde:

```
<scene version="0.5.0">
<integrator type="path">
<integrator value="5" name="maxDepth">
<integrator value="1" name="rrDepth">
</integrator>
<sensor type="perspective">
<float value="37" name="fov"/>
<string value="x" name="fovAxis"/>
<transform name="toWorld">
<lookat up="0, 1, 0" origin="278, 274.4, 1383.27" target="278,  274.4, 1382.27"/>
</transform>
<sampler type="independent">
<integer value="32" name="sampleCount"/>
</sampler>
<film type="hdrfilm">
<integer value="512" name="height"/>
<integer value="512" name="width"/>
<rfilter type="box"/>
</film>
</sensor>
<shape type="obj">
<string value="meshes\light.obj" name="filename"/>
<boolean value="true" name="faceNormals"/>
<emitter type="area">
<rgb value="100, 100, 100" name="radiance"/>
</emitter>
</shape>
</scene>
```

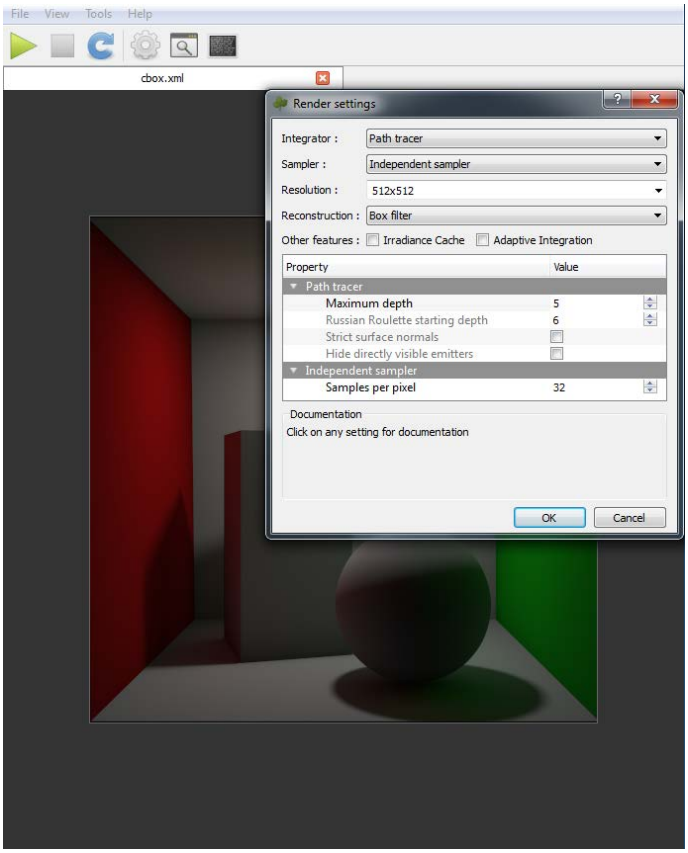
Jak jsem již zmínil v sekci 2.3.1, Mitsuba lze ovládat buď přes příkazový řádek, nebo pomocí grafického rozhraní. Ukázka použití pomocí příkazového řádku je zde:

```
$ mitsuba path-to/my-scene.xml
```

Vykreslování pomocí více výpočetních zařízení lze vyvolat například pomocí:

```
$ mitsuba -c machine1;machine2;... path-to/my-scene.xml
```

Grafické rozhraní včetně dialogu pro nastavení vykreslovacího algoritmu je zobrazeno na obrázku 2.9.

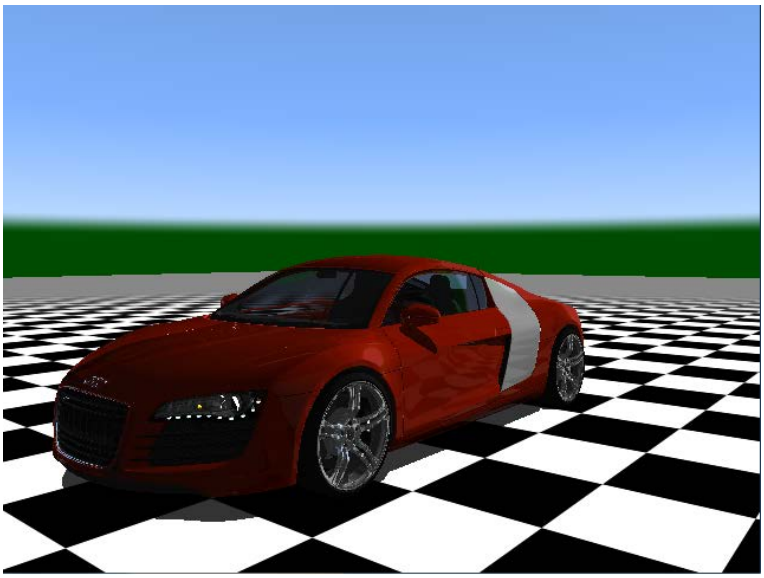


Obrázek 2.9: Grafické rozhraní Mitsuby

Mitsuba je také dostupná jako doplněk programů Blender a Rhinoceros.

2.4 PowerVR OpenRL

2.4.1 Popis



Obrázek 2.10: Scéna, která je součástí ukázkové aplikace dodávané s OpenRL

OpenRL [Ima] je rozhraním, které má za úkol podporovat tvorbu aplikací založených na vrhání paprsku. Velice se podobá rasterizačnímu rozhraní OpenGL, ze kterého přebírá mnoho konceptů a je tak dobře propojitelné s aplikacemi napsanými právě v OpenGL. Během výpočtů dokáže využít jak CPU a GPU, tak i další výpočetní zařízení. OpenRL obsahuje run-time kompilátor programů napsaných uživatelem (shaderů), transparentně staví akcelerační struktury a snaží se co nejlépe mapovat program využívající vrhání paprsků na dostupný hardware.

Je důležité podotknout, že OpenRL neobsahuje implementaci již hotových konceptů, jakým je například globální osvětlování, antialiasing, nebo rozptyl světla pod povrchem objektu. Nabízí ale nástroje a funkce k tomu, aby si uživatel mohl tyto funkce napsat sám. OpenRL také nemusí sloužit jen k implementaci grafických aplikací, ale může být použito k implementaci aplikací, které slouží k simulaci fyzikálních jevů.

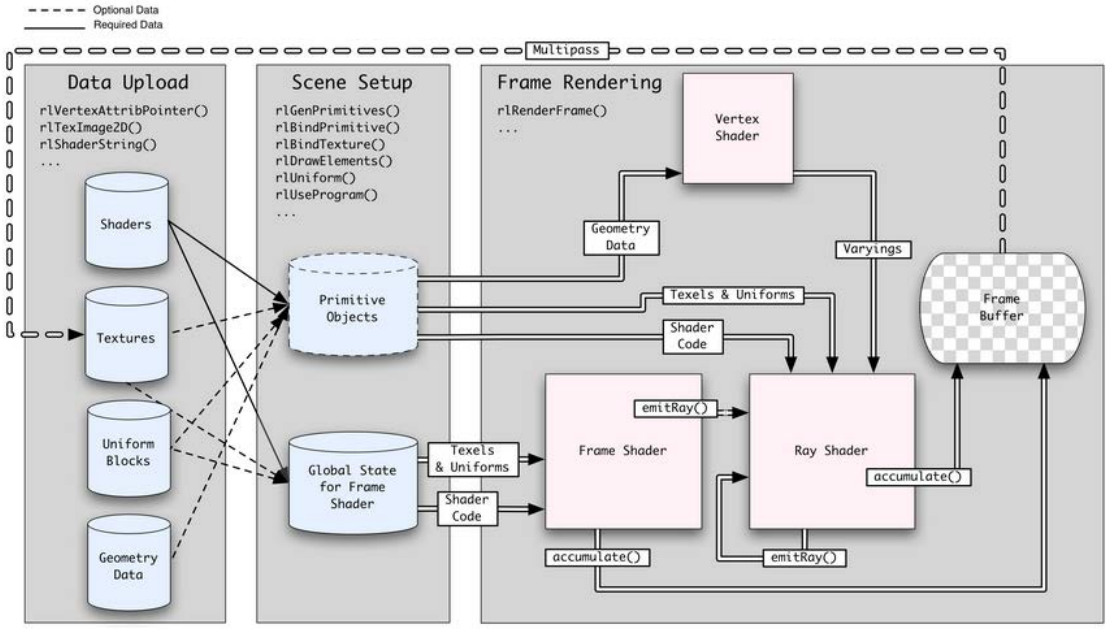
2.4.2 Architektura

Systém OpenRL je zodpovědný za:

- uchování geometrie scény
- uchování textur, uniformních bloků dat, vertex a frame bufferů
- hledání průsečíků paprsků se scénou

- provádění shader programů

V hostitelském rozhraní je nejprve potřeba nastavit kontext, který je poté použit při provádění shader programů. Ty jsou napsány v jazyce RLSL, což je jazyk velice podobný jazyku GLSL, který je používán pro psaní shader programů pro OpenGL a RLSL se snaží dodržovat konvence v GLSL stanovené. Aplikaci napsanou v OpenRL lze rozdělit na tři části: nahrání dat, sestavení scény a vykreslování snímků. Co tyto fáze obnášejí je znázorněno na obrázku 2.11.



Obrázek 2.11: Fáze aplikace napsané v OpenRL. Z dokumentace [Ima]

Nahrání dat a sestavení scény je vykonáváno pomocí hostitelského rozhraní, které je napsané v jazyce C. Vykreslení snímku je iniciováno také tímto rozhraním, ale zahrnuje především interakci paprsků se scénou, která probíhá v shader programech napsaných pomocí jazyka RLSL.

Než se v OpenRL vyšlou paprsky do scény, všechna data o scéně musí být již zpřístupněna. Data mohou být jednoho z těchto typů:

- data o geometrii scény
- uniformní data
- data o texturách
- shader programy

Data o geometrii scény většinou obsahují souřadnice vrcholů objektů, normály vrcholů,souřadnice pro mapování textur nebo indexy vrcholů. OpenRL akceptuje data v podobě trojúhelníků a

nebo trojúhelníkových pásů (triangle strip) a tato data jsou stejně jako v OpenGL uložena do vertex bufferů. Uniformní bloky dat definují data, která mohou být dostupná pro shader programy, kde mohou být použita k výpočtům. Příkladem těchto dat mohou být souřadnice zdrojů světla. Formát těchto dat poskytovaných hostitelským rozhraním musí odpovídat formátu, v jakém tato data shader programy požadují. Uniformním blokům dat se podobají data o texturách, která jsou ale přístupná pouze pomocí sampleru. Textury v OpenRL podporují také techniku zvanou MIP mapping, která umožňuje texturu načíst ve více rozlišeních a ta mohou být použita při vykreslování k anti-aliasngu. Textury se dále od uniformních dat liší tím, že musí být předány v jednom ze zadaných formátů. Posledním typem dat jsou shader programy, které mohou být třech typů. Kromě frame a vertex shaderů jsou to i ray shadery. Vertex shadery manipulují s geometrií scény zatímco ray shadery mohou být použity k definici materiálů objektů scény. Frame shadery operují s výstupními daty a mohou sloužit k definici kamery ve scéně.

Fáze sestavení scény vytváří asociace mezi načtenými daty z předchozí fáze a tím definuje objekty, které do scény patří. Příkladem sestavení scény může být přiřazení železného materiálu nějakému objektu, příprava uniformního bloku dat, který bude specifikovat zdroje světla a příprava textury, která bude definovat jak bude povrch objektu drsný. Všechny objekty ve scéně jsou reprezentovány pomocí objektů zvaných primitive objects. Tyto objekty dohromady tvoří graf scény a mohou zastřešovat tato data:

- asociace s daty o geometrii
- asociace se shader programy
- asociace s uniformními daty
- hodnoty uniformních dat

Tato data se nastavují primitivům, jen když jsou konkrétní primitiva zvolena. Jediným shaderem, který není asociován s žádným primitivem objektem je frame shader. Uniformní data nastavovaná, když není zvoleno žádné primitivum jsou dostupná právě z frame shaderu. Výstupní data z frame shaderu se ukládají do frame bufferu, který musí být se scénou asociován také. Po zapsání výstupních dat může být tento buffer použit hostitelským rozhraním například k samotnému zobrazení vykreslovaného obrázku.

Poslední fáze, která se nazývá vykreslování snímků nastává, když je scéna sestavena a probíhá v ní provádění shaderů zadaných uživatelem. Pro každý vrchol všech primitiv scény nejprve proběhne vertex shader, který musí stanovit pozici vrcholu. Dalším shaderem je frame shader, který je zdrojem všech primárních paprsků, které jsou do scény vyslány a je vykonán pro každý pixel obrazové roviny. Posledním shaderem je ray shader, který je zodpovědný za implementaci efektů materiálů a osvětlení na povrchu objektu, který byl protnut některým paprskem. Před ray shaderem může být vykonán ještě ray prefix shader, který může upravovat atributy paprsku, ještě předtím, než paprsek protne nějaký objekt.

2.4.3 Použití

Jak jsem již zmínil v sekci 2.4.2, nejprve je potřeba nahrát a sestavit data o celé scéně. Data o geometrii jsou uložena v poli vrcholů a pro přesun do něj používá `RL_ARRAY_BUFFER`.

Tato data jsou poté předávána vertex shaderům, kde probíhá jejich zpracování. Jak lze data nahrát a propojit je s vertex shaderem je naznačeno na obrázku 2.12.

```
//2D position vertex attribute data in host memory
RLfloat positions[3 * 2] = {  0.0f, 0.0f,    0.9f, 0.0f,    0.0f, 0.9f };

//Create a buffer object for the position vertex array
RLbuffer positions_buffer;
rlGenBuffers(1, &positions_buffer);

//Transfer positions into the vertex array
//using the binding point RL_ARRAY_BUFFER
rlBindBuffer(RL_ARRAY_BUFFER, positions_buffer);
rlBufferData(RL_ARRAY_BUFFER, 3 * 2 * sizeof(RLfloat),
             positions, RL_STATIC_DRAW);
```

```
//Get the index for the attribute variable positionAttribute in the vertex shader for the program triangle_program
int positionAttributeIndex = rlGetAttribLocation(triangle_program,
                                                 "positionAttribute");

//Bind a primitive that triangle_program is associated with
rlBindPrimitive(RL_PRIMITIVE, primitive_triangle);

//Bind the buffer containing the vertex array
rlBindBuffer(RL_ARRAY_BUFFER, positions_buffer);

//Assign the vertex array to the position attribute of the vertex shader
rlVertexAttribBuffer(positionAttributeIndex, 2, RL_FLOAT,
                     RL_FALSE, sizeof(RLint)*2, 0);
```

Obrázek 2.12: Definování dat o geometrii v OpenRL. Z dokumentace [Ima]

Aby se zabránilo duplikaci souřadnic vrcholů, které jsou použity vícekrát, umožňuje OpenRL vykreslovat vrcholy v pořadí zadaném pomocí pole indexů vrcholů. Příklad použití je naznačen na obrázku 2.13.

```
//32-bit vertex indices used to reference vertex arrays
static RLuint indices[11] = { 0, 3, 1, 4, 2, 5, 8, 4, 7, 3, 6 };

//Place indices in vertex index array contained in buffer
RLbuffer indices_buffer;
rlGenBuffers(1, &indices_buffer);
rlBindBuffer(RL_ELEMENT_ARRAY_BUFFER, indices_buffer);
rlBufferData(RL_ELEMENT_ARRAY_BUFFER, 11 * sizeof(RLuint), indices, RL_STATIC_DRAW);
```

Obrázek 2.13: Použití indexů pro vrcholy v OpenRL. Z dokumentace [Ima]

Po načtení dat o geometrii jé třeba zavolat funkci *rlDrawArrays* pro neindexovanou geometrii, nebo *rlDrawElements* pro geometrii indexovanou.

Pro přiřazení shaderu k primitive objektu je třeba shader nejprve vytvořit a zkompilovat pomocí funkcí *rlCreateShader*, *rlShaderSource* a *rlCompileShader*. Dále je třeba ho přiřadit k objektu program použitím funkcí *rlAttachShader* a *rlLinkProgram*. Ke zvolenému primitivu se poté přiřadí program objekt pomocí *rlUseProgram*.

Textury v OpenRL mohou mít nastaveny několik druhů atributů. Mezi nimi jsou i takzvané wrap módy a ty mohou nabývat hodnoty `RL_REPEAT`, nebo `RL_CLAMP_TO_EDGE`. Dalším atributem jsou filter módy, které mohou nabývat některé z těchto hodnot:

- `RL_NEAREST`
- `RL_LINEAR`

- `RL_NEAREST_MIPMAP_NEAREST`
- `RL_LINEAR_MIPMAP_LINEAR`

Textury se vytváří pomocí funkcí *rlGenTextures* a *rlBindTexture* a data do nich jsou načtena pomocí *rlTexImage2D*, nebo *rlTexImage2D*.

Uniformní bloky dat, které jsou přístupné z příslušných shaderů jsou v OpenRL buď předdefinovaných typů, jako například floatů, integerů, matic, nebo vektorů, nebo mohou být ve formátu definovaném shaderem. Nejprve je nutné nalézt index uniformního atributu v shaderu pomocí funkce *rlGetUniformLocation* a poté nastavit atribut pomocí funkcí, které začínají na *rlUniform*. Pro uniformní data, která mají formát zadaný shaderem je nutné nalézt index uniformního atributu v shaderu pomocí funkce *rlGetUniformBlockIndex*, dále je nastavit pomocí *rlUniformBlockBuffer* a nakonec namapovat pomocí *rlMapBuffer*.

Samotná primitiva, která jsem popsal v sekci 2.4.2, a která obsahují stav jednotlivých objektů ve scéně, se vytváří pomocí *rlGenPrimitives* a zvolí se pro nastavování pomocí *rlBindPrimitive*.

Po sestavení scény se provádí shader programy zadané a uživatelem a propojené s konkrétními primitivy, která scéna obsahuje. Prvním shaderem je frame shader. Ten má zabudované některé proměnné, jako jsou například `rl_FrameSize`, `rl_ViewportSize`, `rl_ViewportPosition`, `rl_FrameCoord` nebo `rl_PixelCoord`. Může v něm dojít k vyslání paprsku pomocí funkce *emitRay*. Nejrpve je ho ale třeba vytvořit pomocí *createRay*. Paprsek má také zabudované proměnné jako jsou například maxT, což je největší vzdálenost, jakou může paprsek urazit, origin, což je souřadnice bodu, ze kterého byl paprek vyslán, nebo direction, což je směrový vektor paprsku. Další atributy jako například barva mohou být nastaveny pomocí *rayAttribute*. Dalším shaderem je ray shader, který stejně jako frame shader může obsahovat funkci *setup*, kde může dojít například k nastavení počtu vyslaných paprsků. Výstupní data lze ukládat do příslušného frame bufferu pomocí funkce *accumulate*. Posledním shaderem je vertex shader, kde se musí nastavit parametr `rl_Position`. Příklady shaderů jsou na obrázcích 2.14, 2.15 a 2.16.

```
// The positionAttribute vertex attribute uploaded to a vertex buffer object
attribute vec4  positionAttribute;

// A variable representing the animation time value
uniform float   time;

void main()
{
        // Set the position, as a function of the positionAttribute and time
        rl_Position = positionAttribute + vec4(0.0, sin(time), 0.0, 0.0);
}
```

Obrázek 2.14: Vertex shader v OpenRL. Z dokumentace [Ima]

```
uniform vec3 cameraPosition;

void main()
{
        createRay();

        // Set the ray's origin to our camera position
        rl_OutRay.origin = cameraPosition;

        // Set the ray's direction dependent on the position of the pixel in
        // the frame buffer
        rl_OutRay.direction = vec3(rl_FrameCoord.xy / rl_FrameSize.xy
                                   - 0.5, -1.0);

        emitRay();
}
```

Obrázek 2.15: Frame shader v OpenRL. Z dokumentace [Ima]

```
// Define the brightness uniform, that is constant across the surface of the
// entire object
uniform vec3 brightness;

// Define the normal varying that smoothly varies across the surface
varying vec3 normal;

// The entry point where the code execution begins, when the ray intersects
// this object
void main()
{
        // Accumulate the color contribution into the frame buffer
        accumulate(brightness * (0.5 + 0.5 * normal));
}
```

Obrázek 2.16: Ray shader v OpenRL. Z dokumentace [Ima]

Všechny funkce, které OpenRL nabízí jsou podrobně popsány v dokumentaci, která je přiložena k instalačnímu balíčku knihovny. Tato knihovna je použita v profesionálním rendereru Brazil, který slouží jako doplněk pro programy Rhinoceros, 3D Studio Max a Autodesk VIZ. Dále je použita pro projekt Heat Ray 3.0 [Whi], který implementuje vykreslování pomocí algoritmu sledování cesty.

2.5 Understanding the Efficiency of Ray Traversal on GPUs (CUDA RT)

2.5.1 Popis

Obrázek 2.17: Scéna Conference, která je vykreslena při prvním spuštění aplikace [ALK] pomocí difúzních paprsků

Tento název nese výzkum [ALK],[AL09], který se zabývá nasazením operací spojených s vrháním paprsku, především průchodu akceleračních struktur a testování paprsku na průsečík s objektem scény, na SIMD/SIMT (architektury) stroje, zejména grafické procesory od firmy NVIDIA. Autoři upozorňují, že implementací pro pro grafické procesory je mnoho, ale pro málo z nich je známo, jestli se blíží teoretické hranici možného dosažitelného výkonu, a pokud ne, jaká je příčina.

Autoři výzkumu tedy vytvořili aplikaci, která má za úkol ukázat horní hranici dosažitelného výkonu a existující implementace s ní porovnávají. Výzkumem zjistili, že v existujících implementacích má velký vliv na nedosahování optimálního výkonu špatná distribuce práce na hardware. Aplikace, která bylo pro výzkum vytvořena je vlastně nejrychlejším GPU řešením pro vrhání paprsků této doby, který dokáže řešit jak přímé osvětlení, tak zastínění okolím nebo dokáže vrhat difúzní paprsky pro osvětlení nepřímé.

2.5.2 Architektura

Operace popsané v sekci 2.5.1 jsou souhrnně nazývány *trace*(). Příčinou špatného výkonu *trace*() může být používání neoptimalizovaných výpočtů, malá propustnost paměti nebo něco úplně jiného. Autoři k problému přistoupili tak, že naimplementovali optimalizované varianty dosud známých nejrychlejších jader pro vrhání paprsku napsaných pomocí systému CUDA a porovnali je se svou aplikací, kterou jsem zmínil v sekci 2.5.1. Ukázalo se, že tato jádra poměrně zaostávají za teoretickým výkonnostním optimem a že je to způsobeno primárně špatnou distribucí práce na hardware. Používaná aplikace ale neuvažuje stínování, které na výkon může mít obrovský vliv.

Celá aplikace používá jako akcelerační strukturu hierarchii obálek a využívá jednotkový test na průsečík paprsku s trojúhelníkem, jehož autorem je Sven Woop. Hierarchie obálek jsou stavěny jako SAH-BVH, které jsem popisoval v sekci 1.1.6.1 a používají číslo 8 jako maximální velikost listových uzlů. Pro zvýšení kvality BVH jsou všechny velké trojúhelníky děleny na menší části. Paprsky jsou řazeny do osnov pomocí Mortonových kódů.

Aplikace dále využívá jader, která implementují samotnou *trace*(). Ta jsou optimalizována pro různé architektury grafických karet a při používání aplikace lze zvolit, které jádro bude použito. Těchto jader jsou dva druhy. Prvním druhem jsou taková jádra, která využívají průchod celých paketů paprsků, což znamená, že skupina paprsků vždy putuje stejnou cestou. Toho je dosaženo sdíleným zásobníkem mezi pakety, což způsobí, že je paprsky navštíveno mnoho uzlů BVH, které ale nejsou protnuty. Na druhou stranu se ale do každého uzlu přistupuje pouze jednou pro každý paket paprsků.

Druhým druhem jader jsou taková, která používají průchod jednotlivých paprsků skrz uzly BVH, které jsou jimi opravdu protnuty. Pro každý paprsek tak musí být udržován vlastní zásobník. Ukázalo se, že tato jádra jsou vždy výkonnostně lepší, než jádra která využívají průchodu celých paketů, protože paprsky nenavštěvují uzly, které neprotínají. Tato jádra jsou popisována jako „while-while“, kvůli časté organizaci smyčky v těchto jádrech prováděné. Jak taková organizace může vypadat je naznačeno zde:

```
while-while trace():
    while ray not terminated
        while node does not contain primitives
            traverse to the next node
        while node contains untested primitives
        perform a ray-primitive intersection test
```

Některá jádra jsou dále implementována jako takzvaná persistentní. To znamená, že je vytvořeno pouze tolik vláken, aby došlo k naplnění používaného hardwaru. Těmto vláknům poté může být distribuována práce použitím atomického čítače. Takto jsou implementována jak jádra, která využívají průchodu paketů paprsků, tak jádra „while-while“.

Další možností, jak jádra implementovat, je využití takzvaného spekulativního průchodu. Ten využívá toho, že když už má být průchod spuštěn, tak je dobré nechat se ho účastnit všechny paprsky. Je sice možné, že paprsky již nalezly takové uzly, které by mohly být otestovány na průsečík, ale pokud ne, tak jsou tyto paprsky nečinné. Z tohoto důvodu se nabízí průchod provést pro všechny paprsky.

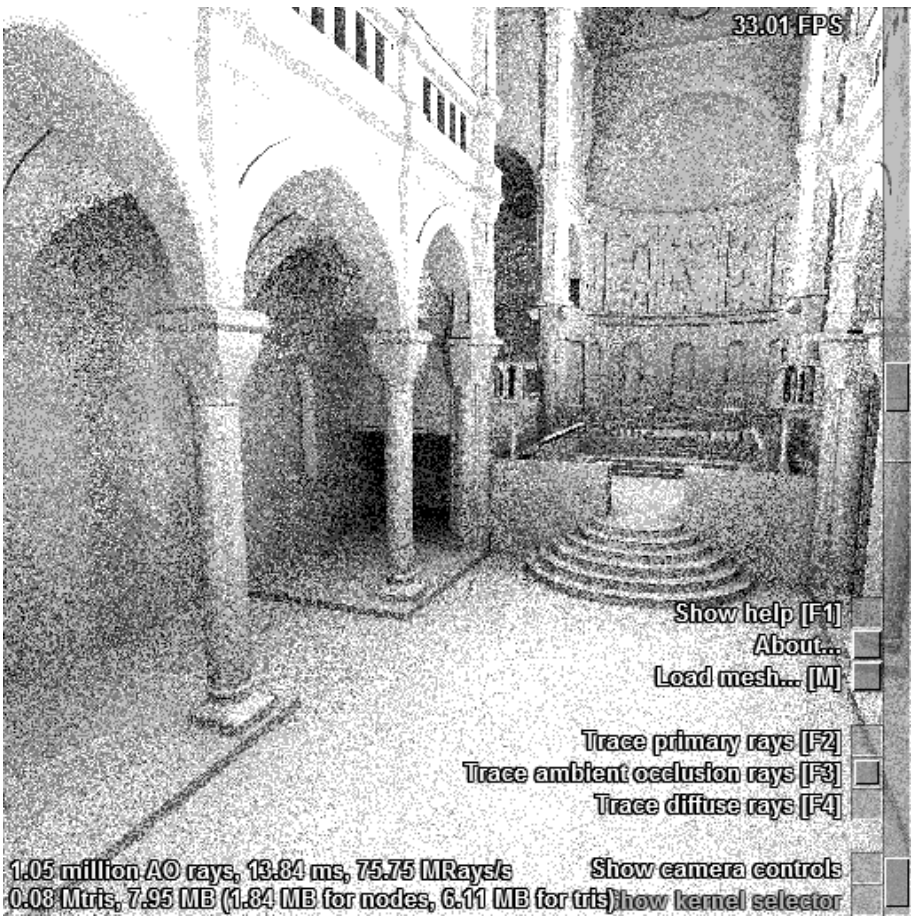
Dále byla přidána jádra pro distribuci práce na grafické karty, které mají novější architekturu typu Kepler, nebo Fermi. Jak tato jádra fungují je popsáno v přídavném článku [ALK12] k článku [AL09].

2.5.3 Použití

Aplikace lze použít ve dvou módech. Prvním je takzvaný interaktivní mód, ve kterém lze načíst a procházet zadanou scénu ve formátu obj. Dále lze zvolit mezi třemi druhy vykreslování. Tyto tři druhy jsou:

- primární osvětlení
- zastínění okolím
- vykreslování pomocí difúzních paprsků

Dále je možné nastavit parametry kamery a zvolit jádro, které bude použito pro *trace*(). Jak vypadá interaktivní mód je vidět na obrázku 2.18.



Obrázek 2.18: Interaktivní mód

Druhým módem je takzvaný benchmark, který je ovládán pomocí příkazového řádku, ve kterém je možné nastavit důležité parametry, stejně jako u módu interaktivního. Aplikace dále měří výkon pro zadané scény a *trace*() jádra. Výsledky zapíše do textového souboru. Ty lze poté porovnat s výsledky dodanými autory v archivu s aplikací. Příklad použití interaktivního módu je zde:

```
benchmark --size=512x512
--mesh=sibenik.obj
--camera="TWkZw1df9nu1a6sYv//16wmy/Vo47y19:Qxcv/ulTW109Qx7w////m100"
--samples=256 --kernel=kepler_dynamic_fetch
```

2.6 Srovnávací tabulka parametrů

V této sekci uvádím tabulku 2.1, která slouží k porovnání vybraných parametrů jednotlivých knihoven. Tyto parametry jsem rozdělil do kategorií a těmi jsou:

- Obecné
- Vrhání paprsků
- Akcelerační struktury
- Testy průsečíků
- Materiály
- Světla
- Vykreslovací algoritmy

Je důležité říci, že jako dostupné parametry jsem označil pouze ty, o kterých je zmínka ve zdrojích, nebo které jsem dohledal ve zdrojových kódech aplikací, které jsou ke knihovnám dodávány. Do těchto aplikací jsem nezapočítal ukázkový renderer ke knihovně Embree, popisovaný v sekci 2.1.3, ani aplikaci Design Garage, postavenou na knihovně Optix, popisovanou v sekci 2.2.3. Zmíněné aplikace většinou materiály a světla, která jsou u knihoven označeny za nedostupné, obsahují.

Dalším problémem je, že ne všechny parametry byly dohledatelné. Týká se to především knihovny OpenRL, která ve své dokumentaci popisuje používání všech funkcí, ale nikde není zmínka o tom, jak jsou tyto funkce naimplementovány. V OpenRL se také netvoří akcelerační struktury pomocí psaní kódu a knihovna vše dělá a nastavuje sama. Není tedy jasné, jaké akcelerační struktury jsou použity a jak jsou stavěny.

Z tabulky 2.1 je možné sledovat, že ne všechny zvolené knihovny slouží jen jako nástroj pro tvorbu vlastních aplikací. Některé z nich jsou již hotové aplikace, které mají za úkol demonstrovat svoji funkčnost a poskytovat rozšiřitelnost. To je nejvíce patrné na renderu Mitsuba, který nelouží jako framework, ale tvoří kompletní vykreslovací systém, který obsahuje spoustu struktur a funkčností při vykreslování používaných.

Další hotovou aplikací je CUDA RT, která má za úkol demonstrovat především výkon a nezaměřuje se na to, aby obsahovala velké množství znovupoužitelných struktur, ale obsahuje jen podstatné minimum, které je třeba k tomu, aby fungovala tak, jak jsem popsal v sekci 2.5.3. To se ve srovnávací tabulce projevilo tak, že CUDA RT obsahuje nejméně políček se znaménkem + ze všech knihoven.

Zbývající knihovny: Embree, Optix a OpenRL, jsou již typickými frameworky, které disponují programovou dokumentací a slouží k tvorbě aplikací psaných uživatelem. Embree, které je řešením pro CPU, nabízí velké množství možností svého použití. Zajímavé je například možnost použití „Hair“ geometrie, instancování geometrie nebo tvorby geometrie vlastní. Dále Embree nabízí velice efektivní řešení týkající se akceleračních struktur.

U Optixu je velkou výhodou rozsáhlá programová dokumentace a také velké množství ukázkových aplikací, které implementují samotné vykreslovací algoritmy. Ty jsou implementovány pomocí uživatelských CUDA programů. Optix také nabízí největší množství možností použití akceleračních struktur.

Poslední knihovnou je OpenRL, které se velice podobá rasterizačnímu rozhraní OpenGL. K OpenRL je také dodáváno velké množství ukázkových aplikací, avšak ty nejsou zaměřeny na implementaci samotných vykreslovacích algoritmů, ale spíše na jiné zajímavé použití shaderů psaných v RLSL. OpenRL disponuje největší programovou dokumentací, takže je poměrně dobře použitelné. Navíc jsou zde zachovány koncepty z OpenGL, takže například nahrávání geometrie zde probíhá totožně. Bohužel není moc zmínek o tom, jak funguje samotná vnitřní struktura knihovny a jak jsou naimplementovány funkce, které uživatel používá. To je viditelné ve sloupci s akceleračními strukturami, pro který jsem nemohl dohledat, jaké struktury jsou použity a jak jsou vytvářeny.

	Embree	Optix	OpenRL	Mitsuba	CUDA RT
Obecné					
Licence	Open Source	Pro nekomerční	Open Source	Pro komerční souhlas	BSD Licence
Optimalizace	CPU x86	CUDA GPU	CPU i GPU	CPU	CUDA GPU
Jazyk	ISPC,C++	CUDA,C++	OpenRL,RLSL	C++	CUDA,C++
Kompilátor	ISPC,VS	CUDA,VS	Shader compiler,VS	VS,Scons,Python	CUDA,VS
Ukázkové aplikace	+	+	+	+	+
Zdrojové kódy ukázek	+	CMake	+	+	+
GUI	-	-	-	+	+
Interaktivní scéna	+	+	+	+	+
Plnohodnotný renderer	-	-	-	+	-
Pipeline	-	+	+	-	-
Shadery	-	-	+	-	-
Scény ze souborů	+	+	-	+	+
Vrhání paprsku					
Jendotlivé paprsky	+	+	+	+	+
Pakety paprsků	+	-	-	+	+
Autom. pakety vs. single	+	-	-	-	-
Dědičnost param. scény	-	+	-	-	-
Instancování gemetrie	+	+	-	+	-
Uživatelova geometrie	+	+	-	-	-
Hair geometrie	+	-	-	+	-
Akcelerační struktury					
BVH	+	+	?	-	+
SAH BVH	+	-	?	-	+
BVH Morton-Code	+	-	?	-	-
Split BVH	+	+	?	-	-
Karras BVH	-	+	?	-	-
Median BVH	-	+	?	-	-
KD Strom	-	+	?	+	-
Nutnost volání	+	+	-	+	+
Testy průsečíků					
Moller Trumbore	+	?	?	+	-
Plücker Variant	+	?	?	-	-
Woop's unit	-	?	?	-	+
Bezier Curve	+	?	?	-	-
Materiály					
Difúzní	+	+	+	+	+
Odrazivý	+	+	+	+	+
Průhledný	+	+	+	+	+
Metal	-	-	-	+	-
Plastic	-	-	-	+	-
Bump map	-	-	-	+	-
Cloth	-	-	-	+	-
Světla					
Ambientní	+	+	+	+	+
Směrové	+	+	+	+	-
Bodové	+	+	+	+	-
Plošné	+	+	-	+	-
Environment	-	-	-	+	-
Vykreslovací algoritmy					
Vrhání paprsku	+	+	+	+	+
Sledování paprsku	-	+	+	+	-
Sledování cesty	+	+	-	+	-
Obousměr. sledování c.	-	-	-	+	-
Ruská ruleta	+	+	-	+	-
Mapování fotonů	-	+	-	+	-
MLT	-	-	-	+	-

Tabulka 2.1: Srovnávací tabulka parametrů knihoven

Kapitola 3

Srovnávací testy

V této části práce jsou prezentovány výsledky srovnávacích testů knihoven, které byly zhotoveny pomocí aplikací přiložených ke knihovnám, nebo takových, které vznikly úpravou těchto aplikací. Aplikace umí až na jednu výjimku vykreslovat scény pomocí algoritmu sledování cesty. U Embree jsem použil ukázkový renderer popisovaný v sekci 2.1.3. Pro testování Mitsuby jsem zvolil integrátor Path Tracer. Zmiňovanou výjimkou byla aplikace přiložená k 2.5.1, která neuvažuje stínování a neimplementuje tedy sledování cesty, a proto jsem namísto toho použil možnost vysílání difúzních paprsků. Ty jsou vyslány vždy z nejbližšího průsečíku nalezeného paprskem primárním a jejich cesta po nalezení dalšího průsečíku končí.

U Optixu jsem upravil aplikaci „path-tracer“, která neuměla načítat scény ze souborů. Musel jsem tedy nalézt nějaký objektový loader, kterým ale naštěstí Optix disponuje a jeho použití je vidět v další ukázkové aplikaci. Samotná aplikace „path-tracer“ obsahovala tvorbu scény pomocí paralelogramů, kterým byly souřadnice a barvy materiálu zadávány ručně. Pozměnil jsem tedy kód psaný v C++, který umožní načtení geometrie a její nastavené do objektu **Context**. Dále jsem musel propojit difúzní materiál loaderem načtený s příslušnou proměnnou v samotném CUDA uživatelském programu, který sledování cesty vykonává. Světlo je tvořeno pomocí paralelogramu stejně, jako byla tvořena geometrie v původní aplikaci, protože pro testovací scény bylo použito vždy jen jedno. Dále jsem přidal rozhodnutí, jaké světlo a jaká kamera bude použita v závislosti na parametru zadaném z příkazové řádky, který obsahuje název jedné z pěti testovacích scén. V uživatelském CUDA programu stačilo dopsat omezení maximální hloubky rekurze paprsků. Celá úprava čítala asi 180 řádků kódu psaného v C++.

Pro OpenRL jsem upravil aplikaci „CornellBox“, která také neuměla načítat scény ze souborů, pracovat s plošnými světly a vysílat difúzní paprsky. K načítání scén ze souborů jsem využil knihovnu Assimp [Tea], pro kterou existuje dostatečné množství tutoriálů týkajících se jejího začlenění do OpenGL, a vzhledem k podobnosti OpenRL k OpenGl bylo její začlenění velmi podobné. Další úpravou v hostitelském rozhraní psaném v OpenRL bylo nastavení uniformních proměnných týkajících se především světla, a také nastavení textur, které sloužily k distribuci náhodných čísel, jenž jsou používaná v shaderech, psaných v RLSL, implementujících samotné sledování cesty. Poslední úpravou v hostitelském rozhraní bylo přidání rozhodnutí, jaké světlo a jaká kamera bude použita v závislosti na parametru zadaném z příkazové řádky, který obsahuje název jedné z pěti testovacích scén. Úprava v jazyce OpenRL čítala celkově asi 430 řádků. Další výraznou změnu jsem musel provést v shaderech, protože

ty neuměly správně nastavit kameru, pracovat s více vzorky na pixel, vzorkovat plošné světlo a vysílat náhodné sekundární odrazy. Shadery jsem upravil s inspirací v projektu [Whi]. Celkově úpravy v RLSL kódu obsahují asi 165 řádků.

Testy jsem provedl ve dvou variantách a každou z těchto variant jsem naměřil na dvou počítačových sestavách, aby bylo viditelné, jak se výkon zvyšuje s lepším hardwarem.

V každém testu bylo vykreslováno pět scén. Tyto scény jsou:

- Cornell box - 4,8k trojúhelníků
- Conference - 331k trojúhelníků
- Sibenik cathedral - 80k trojúhelníků
- Fairy forest - 174k trojúhelníků
- Dragon - 871k trojúhelníků

Testy jsou zaměřeny jak na výkon, který je udáván v milionech vzorcích za sekundu, tak na kvalitu vykresleného obrázku se zadanými parametry.

3.1 Test bez ruské rulety

Prvním testem bylo vykreslování scén zmíněných v sekci 3 s největším možným počtem vzorků na pixel, aby knihovny obrázek stihly vykreslit do zadaného času, který je v tabulce vždy zachycen v řádku nad výsledky. Čas potřebný k vykreslení je vždy uveden ve sloupci s označením t a je uveden v sekundách. Počet vzorků na pixel je uveden ve sloupci označeném jako Spp a výkon je uveden ve sloupci s označením $Speed$ a vypočítal jsem ho jako $Spp * rozliseni/t/1000000$, a uvedl tedy v milionech vzorcích za sekundu, jak jsem zmínil v sekci 3.

Test jsem nastavil takto:

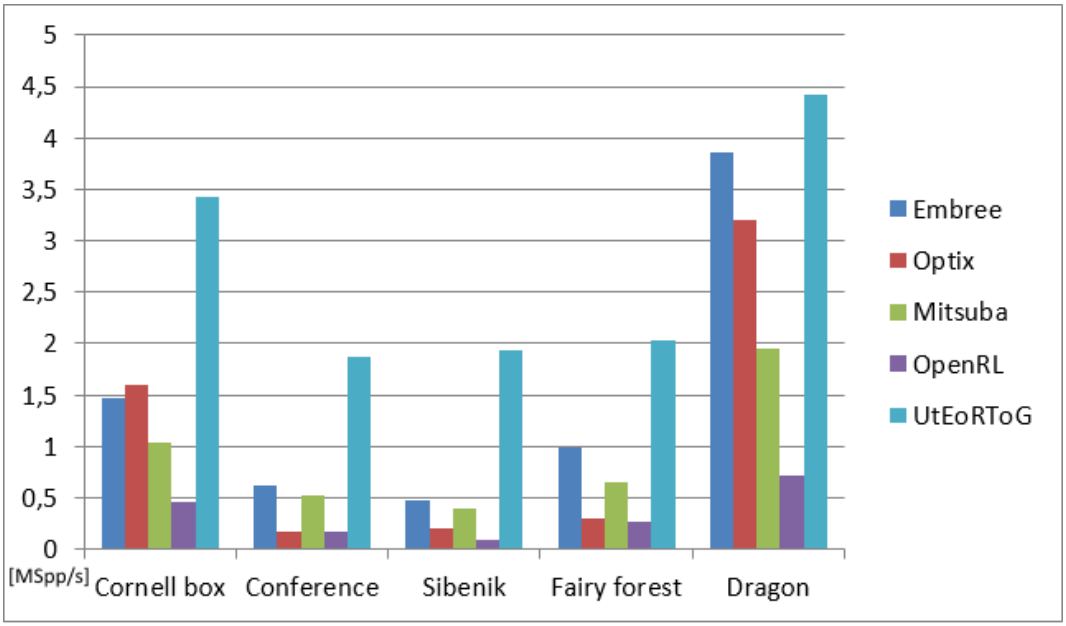
- Hloubka rekurze = 5
- Rozlišení = $512 * 512$ pixelů
- Ruská ruleta aktivní = ne
- Akcelerační struktura = BVH, u Mitsuby lze jen $KDstrom$

3.1.1 První PC sestava

Nejprve byly scény vykreslovány na PC sestavě, dále označované jako **PC1**, s těmito parametry:

- Grafická karta = Nvidia GeForce GTX 650, 128 bit šířka sběrnice, 1GB GDDR5 paměť
- Procesor = Intel i5-750, 4 jádra, 2,66 Ghz frekvence procesoru, 64 bit instrukční set, 8 MB cache

- Operační paměť = Kingston, 2x 2048 MB DDR3 ,666,7 MHz frekvence
- Operační systém = Microsoft Windows 7, Service pack 1, 64 bitová verze



Obrázek 3.1: Výsledky testu bez ruské rulety na PC1

Z výsledků prezentovaných v tabulkách A.1 - A.5 a na obrázku 3.1 se dá vypozorovat především to, jak se mění výkon knihoven se změnou složitosti scény. Složitostí scény se nemyslí pouze počet trojúhelníků, které tvoří zobrazované objekty, ale také poloha kamery vzhledem k objektům ve scéně.

Při vykreslování scény Cornell box pomocí PC1 se jako nejrychlejší prokázala knihovna CUDA RT, která druhou nejrychlejší knihovnu Optix převyšuje zhruba 2,1 krát. Těsně za Optixem se pohybuje knihovna Embree a za ní figuruje Mitsuba. Nejpomalejší knihovnou na této scéně je OpenRL.

Na scéně Conference se pořadí výrazně mění, což je způsobeno především zmiňovanou složitostí scény. Nejrychlejší je tak, jako na všech scénách vykreslovaných pomocí PC1, opět CUDA RT. Druhá v pořadí je už ale Embree, za kterou se těsně drží Mitsuba. Velký propad ve výkonu byl zaznamenán u knihovny Optix, která je téměř vyrovnaná s OpenRL a za nejrychlejší knihovnou tyto dvě knihovny zaostávají přibližně 11,8 krát. Je tedy patrné, že především výkonnost Optixu je velmi závislá na složitosti scény.

Při vykreslování scény Sibenik cathedral se pořadí nijak nemění. Důvodem je, že scéna je také poměrně složitá, stejně jako byla poměrně složitá scéna předchozí. Zvláštností je, že u všech knihoven výkon oproti scéně Conference klesl, zatímco u Optixu a CUDA RT, které slouží pro práci na NVIDIA grafických procesorech, mírně vzrostl.

Na scéně Fairy forest nedošlo ke změnám v pořadí a výkon všech knihoven díky méně složité scéně mírně vzrostl. Na scéně Dragon podle tabulky se naproti tomu výrazně zvedl

výkon Optixu, který byl na této scéně dokonce rychlejší než Mitsuba. U ostatních knihoven oproti scéně Fairy forest došlo k vzrůstu výkonu méně než čtyřnásobnému, zatímco u Optixu byl vzrůst více než jedenáctinásobný.

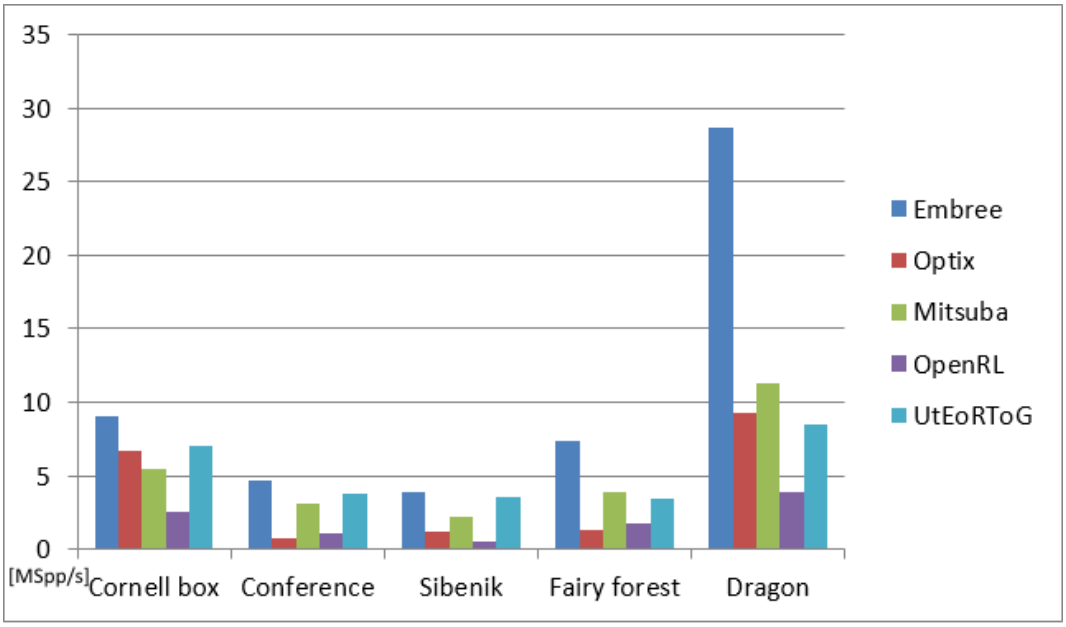
Zvláštností také je, že na všech scénách kromě Dragon scény si drží knihovny kromě CUDA RT se zvyšujícími se vzorky za sekundu téměř konstantní výkonnost, zatímco v tabulce A.5 je vidět, že pro všechny knihovny kromě Optixu se se zvyšujícími se vzorky za sekundu výkon zvedá. U OpenRL je to dokonce skoro dvojnásobně.

3.1.2 Druhá PC sestava

Druhá počítačová sestava využitá k testování, dále označována jako **PC2**, měla tyto parametry:

- Grafická karta = Nvidia Tesla K20C, 2496 jader, 5GB GDDR5 paměť
- Procesor = Intel Xeon E5-2630 v3, 8 jader, 2,4 Ghz frekvence procesoru, v turbo režimu 3,2 GHz, 64 bit instrukční set, 20 MB cache
- Operační systém = Microsoft Windows Server 2012 R2

Pro tuto sestavu jsem zachoval stejné parametry, jako pro měření na sestavě první. To znamená, že jsem scény vykresloval se stejným počtem vzorků na pixel. Díky tomu je dobře viditelné, jak se výkon knihoven změnil s výrazně lepším hardwarovým vybavením PC.



Obrázek 3.2: Výsledky testu bez ruské rulety na PC2

Z výsledků prezentovaných v tabulkách A.1 - A.10 a na obrázku 3.1 a 3.2 se dá pozorovat především to, jak se mění výkon knihoven se změnou používané počítačové sestavy, a tedy, jak knihovny dokáží využít výrazně lepšího hardwaru než takového, který je dostupný v PC1.

Na scéně Cornell box se u Embree zvýšil výkon více než 6 krát, u Optixu tomu bylo více než 4 krát, u Mitsuby více než 5 krát, u OpenRL více než 5,5 krát, ale u CUDA RT jen zhruba 2 krát. To může být způsobeno tím, že jádra, která aplikace používá k průchodu paprsků scénou, nejsou pro kartu, jakou PC2 disponuje, dostatečně optimalizována. Tím je způsobeno, že na PC2 je pro všechny scény nejrychlejší knihovnou Embree. Další zvláštností z tabulky A.6 je, že všechny knihovny svůj výkon s přibývajícími vzorky na pixel poměrně výrazně zvyšují, zatímco u knihovny Optix výkon zvláštně kolísá.

Na scéně Conference je vidět, že nárůst výkonu knihoven se ještě zvýšil, a stejně jako pro minulou scénu některé knihovny zvyšují svůj výkon s přibývajícími vzorky na pixel. Je ale důležité povšimnout si, že pro PC2 je OpenRL rychlejší než Optix, zatímco pro PC1 na obrázku 3.1 byly tyto knihovny ještě vyrovnané.

Pro scénu Sibenik cathedral se výkony oproti předcházející scéně zvýšily u některých knihoven ještě více než u předcházející scény. Na této scéně je Optix oproti předcházející scéně opět rychlejší než OpenRL. Na scéně Fairy forest se vše vyvíjí podobně jako u předcházející scény, jen je zde OpenRL opět rychlejší než Optix, stejně jako tomu bylo u scény Conference.

Pro scénu Dragon se na PC2 na obrázku 3.2, stejně jako na PC1 na obrázku 3.1, výrazně zvedá výkon u Optixu, takže ten je opět rychlejší než OpenRL. Dále tentokrát již všechny knihovny výrazně zvyšují svůj výkon s rostoucím počtem vzorků na pixel.

Z dosud prezentovaných výsledků lze usoudit, že knihovna Embree je velice rychlá pro obě PC sestavy. Zatímco na PC1 byla ještě nejrychlejší knihovna CUDA RT, na PC2 už Embree všechny knihovny výrazně převyšuje. U knihovny Optix výkon velmi záleží na složitosti scény. Pro jednodušší scény dokázala dosahovat na PC1 výkonu srovnatelného s Embree, ale pro scény složitější jde výkon rapidně dolů. Mitsuba od Embree drží poměrně konstantní odstup, ale na PC1 se pro složitější scény výkonnostně dost přibližovala. OpenRL je na PC1 celkově nejpomalejší knihovnou. Na PC2 už ale na některých scénách dokázala být rychlejší než Optix. Zvláštností této knihovny je, že především na PC2 velmi výrazně zvyšovala svůj výkon s rostoucím počtem vzorků na pixel. Jak již bylo řečeno, knihovna CUDA RT byla výrazně nejrychlejší na PC1, ale na PC2 již nedokázala tolik těžit z lepšího hardwaru tak, jak to dokázaly ostatní knihovny. Na PC2 tedy byla rychlejší knihovna Embree a pro scénu Dragon dokonce i Mitsuba.

3.2 Test s ruskou ruletou

Druhý test se velice podobá testu prvnímu. Je tedy opět vykreslováno pět scén a jsou zadány co největší možné počty vzorků na pixel tak, aby knihovny stihly vykreslit obrázky do zadaného času. Rozdílem mezi prvním a druhým testem ale je, že je v aplikacích využit algoritmus ruská ruleta popisovaný v sekci 1.1.4. Dalším rozdílem je, že je vypuštěna knihovna CUDA RT, protože ta neimplementuje algoritmus sledování cesty a cesta paprsků končí vždy v hloubce rekurze rovné dvěma. Algoritmus ruské rulety zde tedy nemá takový význam, jako u knihoven ostatních, kde paprsky končí v hloubce rekurze vyšší.

Test jsem tedy nastavil takto:

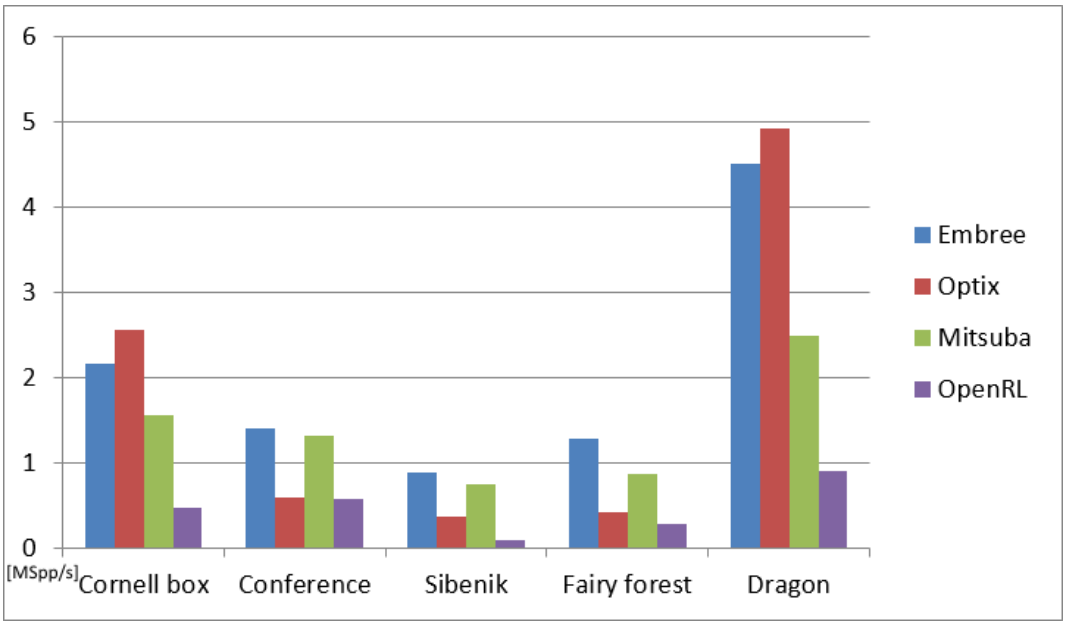
- Hloubka rekurze = 5
- Rozlišení = $512 * 512$ pixelů

- Ruská ruleta aktivní = *ano*
- Akcelerační struktura = BVH, u Mitsuby lze jen $KDstrom$

3.2.1 První PC sestava

První PC sestavou použitou pro tento test byla sestava stejná, jako pro test první. Tato sestava byla zmíněna v sekci 3.1.1.

- Grafická karta = Nvidia GeForce GTX 650, 128 bit šířka sběrnice, 1GB GDDR5 paměť
- Procesor = Intel i5-750, 4 jádra, 2,66 Ghz frekvence procesoru, 64 bit instrukční set, 8 MB cache
- Operační paměť = Kingston, 2x 2048 MB DDR3 ,666,7 MHz frekvence
- Operační systém = Microsoft Windows 7, Service pack 1, 64 bitová verze



Obrázek 3.3: Výsledky testu s ruskou ruletou na PC1

Při vyhodnocení výsledků druhého testu na PC1 bude zajímavé sledovat, jak se výkon knihoven změnil s použitím algoritmu ruská ruleta, který může ukončit cestu některých paprsků ještě dříve než dosáhnou maximální nastavené hloubky rekurze. Tyto výsledky je možné porovnat mezi tabulkami A.1 - A.5 a A.11 - A.15 a na obrázcích 3.1 a 3.3.

Při vykreslování scény Cornell box se u všech knihoven zvýšil výkon přibližně 1,5 krát, jen u OpenRL se výkon na této scéně zvýšil téměř zanedbatelně. Pořadí knihoven tedy zůstává stejné.

U scény Conference se výkon u všech knihoven zvýšil s použitím ruské rulety ještě více. U Embree se zvýšil přibližně 2,3 krát. U Optixu tomu bylo až 3,75 krát, u Mitsuby to bylo více než 2,5 krát a u OpenRL více než 3,3 krát.

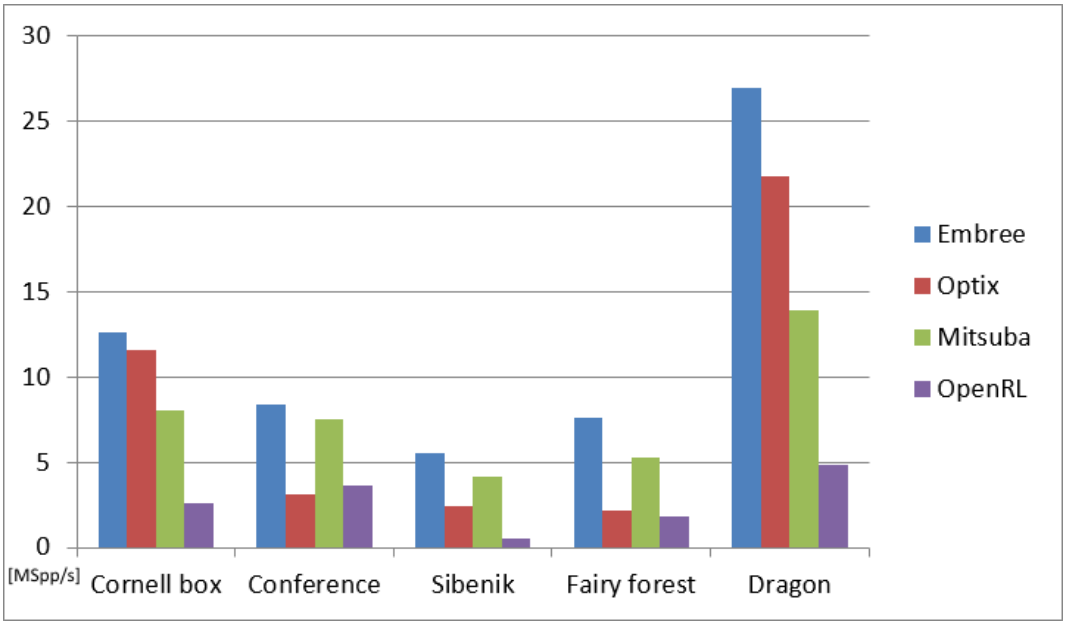
Při vykreslování scény Sibenik cathedral se u Embree, Optixu a Mitsuby zvýšil výkon méně než 2 krát, a to je méně než u předcházející scény. U OpenRL se výkon opět zvýšil pouze nepodstatně. Pořadí knihoven se tedy nijak neměnilo. Na scéně Fairy forest a Dragon je tomu podobně, jen se kromě Optixu výkony zmenšily ještě o dost méně než u scény Sibenik cathedral. U Optixu tomu bylo ale bylo více než 1,5 krát, a tak byl na scéně Dragon nejrychlejší.

Z výsledků testů vytvořených pomocí algoritmu ruská ruleta je oproti výsledkům testů vytvořených bez něj vidět, že algoritmus skutečně dokáže rychlost vykreslování zvýšit. Velmi ale záleží, jaké kritérium je zvoleno pro určení pravděpodobnosti, se kterou je paprsek dále vyšetřován. To se nejvíce projevilo na výsledcích knihovny OpenRL, kde je jako toto kritérium zvoleno maximum z barevných složek difúzní složky materiálu, na který paprsek narazil.

3.2.2 Druhá PC sestava

I druhá PC sestava zůstala stejná, jako ta, která byla použita pro test první. Tato sestava byla popsaná v sekci 3.1.2.

- Grafická karta = Nvidia GeForce GTX 650, 128 bit šířka sběrnice, 1GB GDDR5 paměť
- Procesor = Intel i5-750, 4 jádra, 2,66 Ghz frekvence procesoru, 64 bit instrukční set, 8 MB cache
- Operační paměť = Kingston, 2x 2048 MB DDR3 ,666,7 MHz frekvence
- Operační systém = Microsoft Windows 7, Service pack 1, 64 bitová verze



Obrázek 3.4: Výsledky testu s ruskou ruletou na PC2

Při vyhodnocování výsledků, které přibyly po provedení testu s ruskou ruletou na PC2 je zajímavé pozorovat, jaký je rozdíl mezi zvýšením výkonů při použití a nepoužití tohoto algoritmu na PC1 a zvýšením výkonů při jeho použití a nepoužití na PC2. Změny na PC2 je možné pozorovat z tabulek A.6 - A.10 a A.16 - A.20 a na obrázcích 3.2 a 3.4.

Na scéně Cornell box se výkon u Embree zvedl přibližně 1,4 krát, u Optixu tomu bylo více než 1,7 krát, u Mitsuby přibližně 1,5 krát, ale u OpenRL k nárůstu stejně jako na PC1 téměř nedošlo. Nárůsty výkonu všech knihoven kromě Optixu jsou tedy menší než tomu bylo na PC1. Stejně jako při prvním testu na PC2 se zde výkon knihoven poměrně výrazně zvyšuje se zvyšujícím se počtem vzorků na pixel.

U scény Conference, Sibenik cathedral, Fairy forest a Dragon se také pro všechny knihovny kromě Optixu nárůst s použitím ruské rulety na PC2 oproti jejímu použití na PC1 zmenšil. To způsobilo, že na scéně Dragon byl na PC2 Optix ještě rychlejší než Mitsuba.

Z prezentovaných výsledků lze vyčíst, že algoritmus ruské rulety na PC2 nebyl pro většinu knihoven kromě Optixu takovým urychlením, jako tomu bylo na PC1.

Kapitola 4

Závěr

Tato práce měla za úkol podrobně porovnat pět vybraných knihoven, které slouží k implementaci globálních zobrazovacích metod postavených na vrhání paprsku. Nejprve měly být popsány možnosti a výhody jednotlivých knihoven, poté mělo dojít ke shrnutí vybraných parametrů pomocí srovnávací tabulky a nakonec měly být pomocí aplikací, které budou implementovány za použití vybraných knihoven, vytvořeny testy, jenž budou porovnávat výkonnost a kvalitu jednotlivých knihoven.

V úvodní části jsem nejprve objasnil podstatné pojmy, které se vyskytovaly ve zbývajících částech práce. Poté bylo možné sepsat podstatné informace a popsat možnosti vybraných knihoven. Po prostudování zdrojů, které jsou uvedeny v seznamu literatury, jsem vytvořil tabulku, která má za úkol ukázat, která knihovna disponuje jakými vlastnostmi. Na základě této tabulky se případní uživatelé knihoven mohou rozhodnout, která nejvíce vyhovuje jejich požadavkům.

Poslední samostatnou částí jsou srovnávací testy, které ukazují kvalitu a efektivitu vybraných knihoven. Testy jsem zhotovil pomocí aplikací, které vznikly úpravou takových, které jsou ke knihovnám dodávány, nebo aplikací, které jsem nemusel upravovat, ale musel jsem prostudovat jejich chování a způsob použití. Tyto testy spočívají ve vykreslování pěti různých scén pomocí algoritmu sledování cesty a byly provedeny na dvou různých počítačových sestavách.

Ze všech výsledků obou testů se dá vyčíst, jak je která knihovna výkonná především v závislosti na dostupném hardwaru. Celkově se dá za nejrychlejší knihovnu označit Embree, které sice bylo na první počítačové sestavě na druhém místě, ale na sestavě druhé již bylo nejrychlejší. Výkon knihovny CUDA RT, která dokázala být na první sestavě nejrychlejší, závisí na architektuře použité grafické karty, která navíc musí být kompatibilní se systémem CUDA. Na sestavě první byla grafická karta podobného typu, jako ty, které byly při tvorbě této knihovny testovány. Na druhé sestavě byla karta výkonnější, ale zdá se, že jádra knihovnou používaná ji nedokázala dostatečně využít, a Embree, které dokázalo lépe těžit z velmi výkonného procesoru, tak bylo rychlejší. Dále byl výkon knihovny ovlivněn tím, že zde neprobíhal výpočet tak, jako tomu bylo u knihoven ostatních, protože se zde nevysílaly stínové paprsky. Další v pořadí byla Mitsuba, která na obou sestavách dokázala podávat stabilní výkony. Velkým rozdílem oproti ostatním knihovnám bylo, že Mitsuba jako akcelerační struktury používala KD stromy, a i s jejich použitím výrazně nezaostávala za knihovnami, používajícími BVH. Poslední dvě knihovny sice disponují největším množstvím

programové dokumentace a zřejmě jsou tak uživatelsky nejpřívětivější, ale výkonnostně za předchozími třemi výrazně zaostávaly. Optix sice ještě na méně složitých scénách dokázal dosahovat výsledků podobných knihovnám nejrychlejším, ale u složitých scén za nimi již výrazně zaostával. Úplně nejpomalejší knihovnou se ukázalo být OpenRL, u kterého je těžké určit příčinu, jelikož není jasné, jaké algoritmy spojené s akceleračními strukturami jsou použity, ani jakých typů tyto struktury jsou.

Vzhledem k tomu, že celý projekt byl poměrně rozsáhlý, dá se najít několik možných vylepšení či rozšíření. Mezi ně by mohla patřit implementace zcela vlastních aplikací, které by byly použity pro srovnávací testy. V této práci jsem využil aplikace, které vznikly spíše úpravou již existujících aplikací, dodávaných ke knihovnám jako ukázky jejich funkčnosti a způsobu použití.

Dalším zajímavým rozšířením by mohla být tvorba webového portálu, kam by různí uživatelé mohly vkládat své testy prováděné na knihovnách, které by stejně, jako ty vybrané, sloužily k vrhání paprsků. Takovýto portál by mohl být velmi navštěvovaný a užitečný pro uživatele, kteří se rozhodují, jakou knihovnu pro své vznikající aplikace využít.

Jelikož jsem v této práci pro testy použil pouze scény statické, nabízí se další rozšíření, které by spočívalo v provedení testů pro dynamické scény, protože jejich vykreslování by mohlo ukázat zajímavá čísla, týkající se především výkonu knihoven.

Literatura

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the 1st ACM conference on High Performance Graphics - HPG 09*. Association for Computing Machinery (ACM), 2009.

[ALK] Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on gpus. https://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/. stav z 29.4.2015.

[ALK12] Timo Aila, Samuli Laine, and Tero Karras. Understanding the efficiency of ray traversal on gpus–kepler and fermi addendum. *NVIDIA Corporation, NVIDIA Technical Report NVR-2012-02*, 2012.

[EW11] Manfred Ernst and Sven Woop. Embree: Photo-realistic ray tracing kernels. *White paper, Intel*, 2011.

[HKBŽ97] Vlastimil Havran, Tomas Kopal, Jiří Bittner, and Jiří Žára. Fast robust bsp tree traversal algorithm for ray tracing. *Journal of graphics tools*, 2(4):15–23, 1997.

[Ima] Imagination. PowerVR OpenRL SDK. http://community.imgtec.com/developers/powervr/openrl-sdk/. stav z 29.4.2015.

[Inta] Intel. Embree: High performance ray tracing kernels. http://embree.github.io/. stav z 29.4.2015.

[Intb] Intel. Intel SPMD program compiler. https://ispc.github.io/. stav z 29.4.2015.

[NVIa] NVIDIA. OptiX. https://developer.nvidia.com/optix. stav z 29.4.2015.

[NVIb] NVIDIA. SceniX. https://developer.nvidia.com/scenix. stav z 29.4.2015.

[PBD+10] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)*, 29(4):66, 2010.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann, 2010.

[SAM09] Peter Shirley, Michael Ashikhmin, and Steve Marschner. *Fundamentals of Computer Graphics*. A K Peters/CRC Press, 2009.

[Tea] Assimp Team. Assimp: Open asset import library. http://assimp.sourceforge.net/index.html. stav z 29.4.2015.

[Wen10] Jakob Wenzel. Mitsuba: Physically based renderer. http://www.mitsuba-renderer.org/index.html, 2010. stav z 29.4.2015.

[WH06] Ingo Wald and Vlastimil Havran. 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*, pages 61–69. IEEE, 2006.

[Whi] Cody White. Heatray: OpenRL-based realtime path tracer. https://code.google.com/p/heatray/. stav z 29.4.2015.

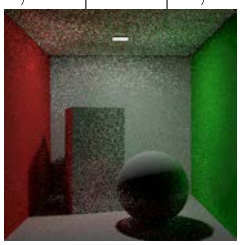
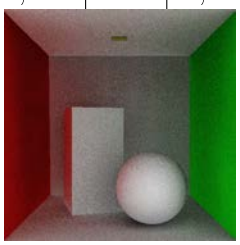
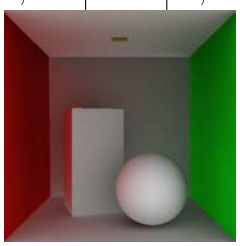
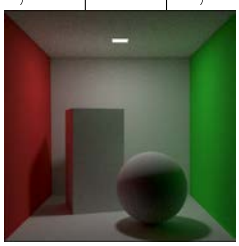
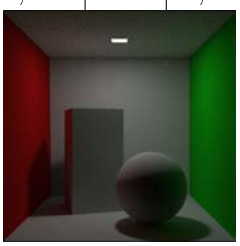
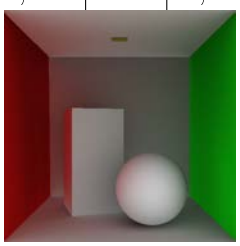
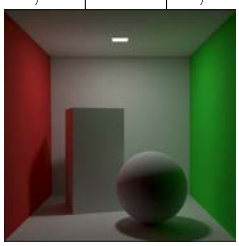
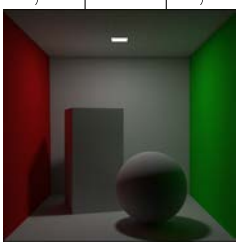
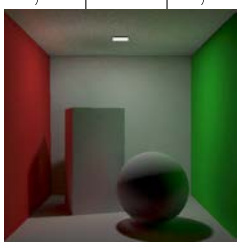
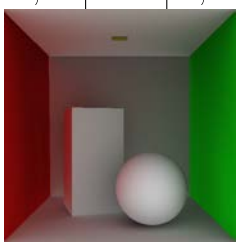
[WWB+14] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG)*, 33(4):143, 2014.

[ZBF05] J. Zára, B. Beneš, and P. Felkel. *Moderní počítačová grafika*, volume 2., přeprac. a rozš. vyd. Computer Press s.r.o, 2005.

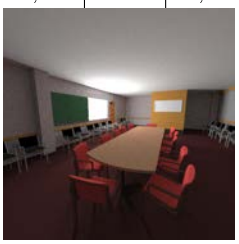
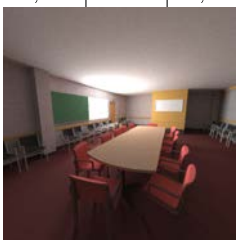
Příloha A

Tabulky s výsledky testů

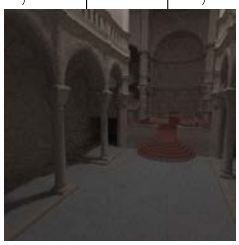
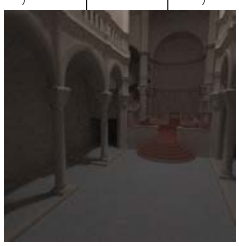
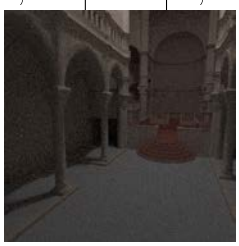
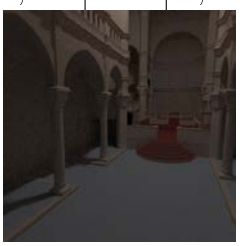
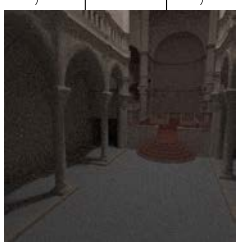
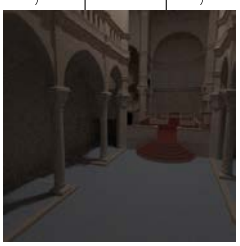
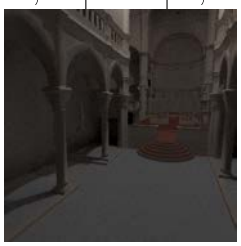
V této příloze se nachází tabulky s výsledky obou testů, které jsou popsány v sekcích 3.1 a 3.2.

Cornell box															
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<1s														
0,73	4	1,44	0,66	4	1,59	0,53	2	0,99	0,73	1	0,36	0,68	8	3,08
														
t<5s														
2,86	16	1,47	4,00	25	1,64	4,04	16	1,04	4,48	8	0,47	4,92	64	3,74
														
t<10s														
5,69	32	1,47	7,89	49	1,63	8,08	32	1,04	8,61	16	0,49	9,46	120	3,33
														
t<40s														
22,75	128	1,48	32,93	196	1,56	32,01	128	1,05	34,32	64	0,49	37,59	512	3,57
														

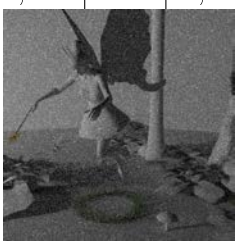
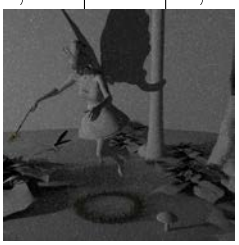
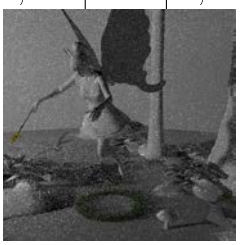
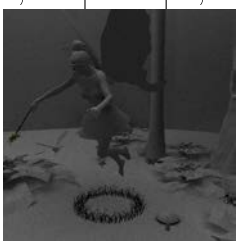
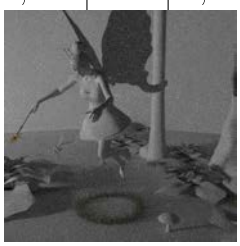
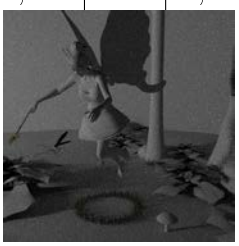
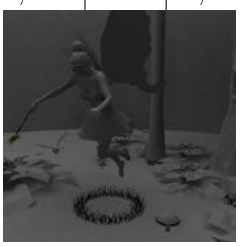
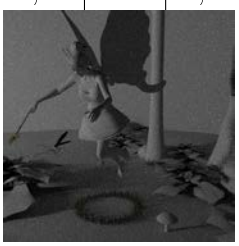
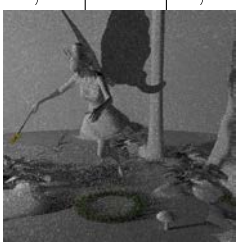
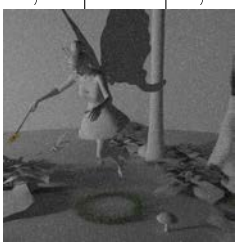
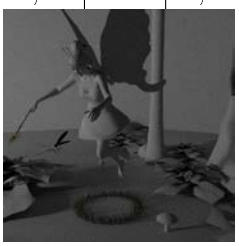
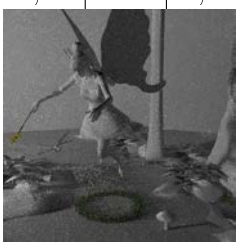
Tabulka A.1: Výsledky testu bez ruské rulety pro Cornell box na PC sestavě 1

Conference															
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**			
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	
t<5s															
3,39	8	0,62	1,60	1	0,16	4,05	8	0,52	3,14	2	0,17	2,56	16	1,64	
															
t<10s															
6,78	16	0,62	6,36	4	0,17	8,08	16	0,52	6,18	4	0,17	9,03	64	1,86	
															
t<20s															
13,67	32	0,61	14,53	9	0,16	16,09	32	0,52	11,70	8	0,18	17,34	128	1,94	
															
t<40s															
27,6	64	0,61	26,57	16	0,16	32,13	64	0,52	23,59	16	0,18	32,58	256	2,06	
															

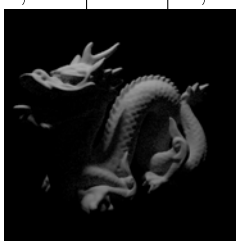
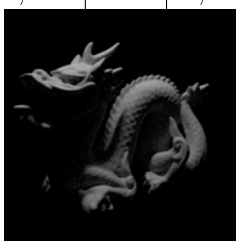
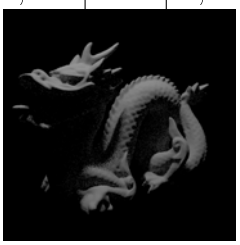
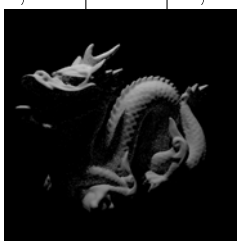
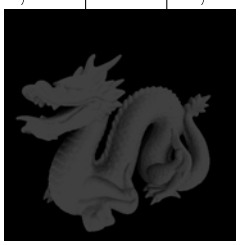
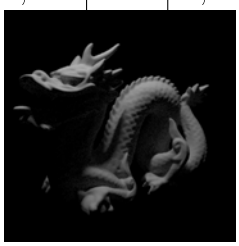
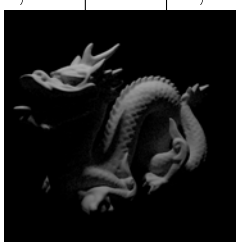
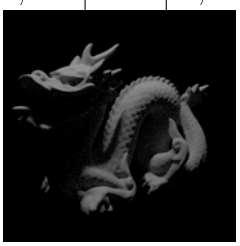
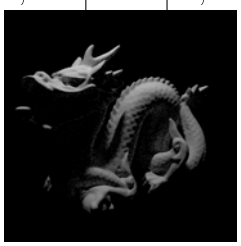
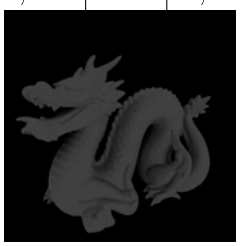
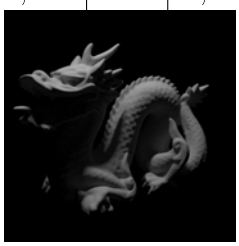
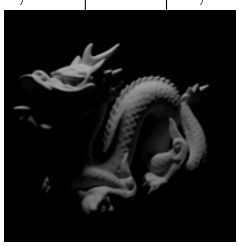
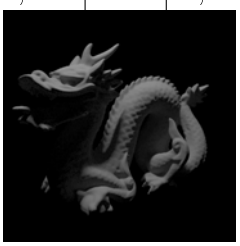
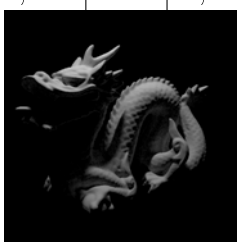
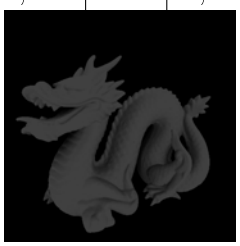
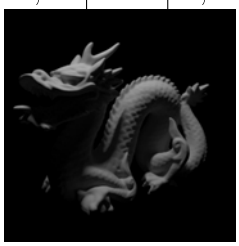
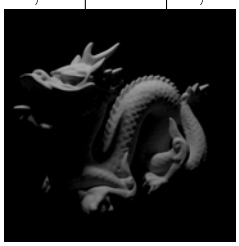
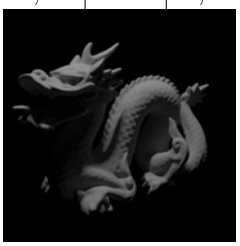
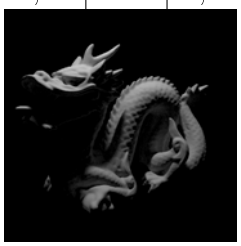
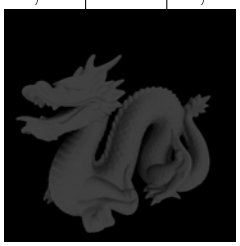
Tabulka A.2: Výsledky testu bez ruské rulety pro Conference na PC sestavě 1

Sibenik															
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s														
4,42	8	0,47	1,30	1	0,20	2,77	4	0,38	3,20	1	0,08	4,60	32	1,82
														
t<10s														
8,92	16	0,47	5,12	4	0,21	5,34	8	0,39	5,85	2	0,09	8,86	64	1,89
														
t<20s														
17,82	32	0,47	11,58	9	0,20	19,98	32	0,42	11,34	4	0,09	17,07	128	1,97
														
t<50s														
35,87	64	0,47	47,78	36	0,20	41,29	64	0,41	43,06	16	0,10	32,67	256	2,05
														

Tabulka A.3: Výsledky testu bez ruské rulety pro Sibenik na PC sestavě 1

Fairy forest														
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s														
4,28	16	0,98	3,60	4	0,29	3,24	8	0,65	4,13	4	0,25	4,39	32	1,91
														
t<10s														
8,64	32	0,97	8,10	9	0,29	6,36	16	0,66	8,08	8	0,26	8,50	64	1,97
														
t<20s														
17,39	64	0,97	14,37	16	0,29	12,66	32	0,66	15,15	16	0,28	16,02	128	2,10
														
t<40s														
33,6	128	1,00	22,92	25	0,29	25,28	64	0,66	29,98	32	0,28	29,36	240	2,14
														

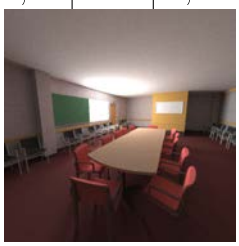
Tabulka A.4: Výsledky testu bez ruské rulety pro Fairy forest na PC sestavě 1

Dragon														
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<1s														
0,57	8	3,70	0,71	9	3,32	0,60	4	1,75	0,98	2	0,54	0,95	16	4,42
														
t<2s														
1,10	16	3,81	1,25	16	3,36	1,09	8	1,92	1,95	4	0,54	1,97	32	4,26
														
t<10s														
8,51	128	3,94	8,20	100	3,20	8,09	64	2,07	9,92	32	0,85	7,57	128	4,43
														
t<20s														
16,90	256	3,97	17,45	196	2,94	16,04	128	2,09	18,49	64	0,91	14,60	256	4,60
														

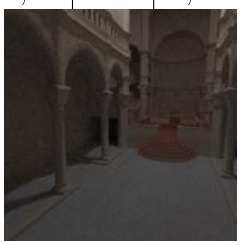
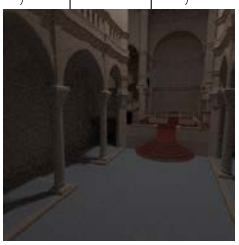
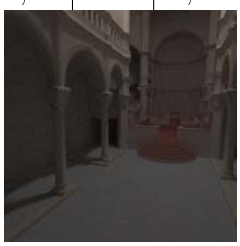
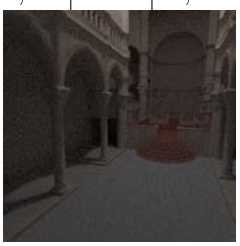
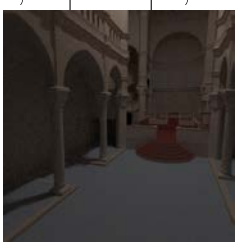
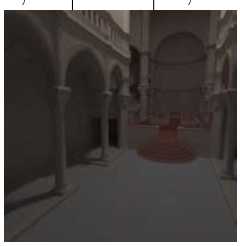
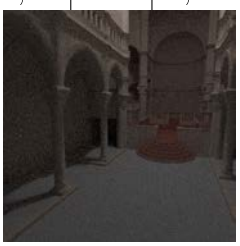
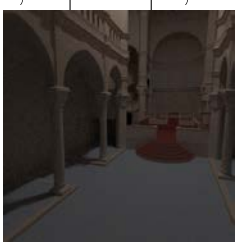
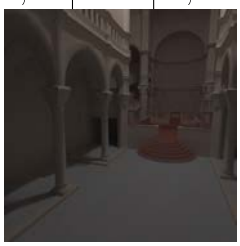
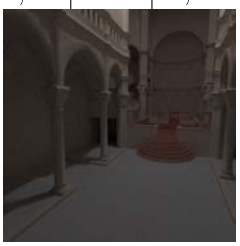
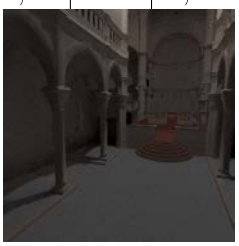
Tabulka A.5: Výsledky testu bez ruské rulety pro Dragon na PC sestavě 1

Cornell box															
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**			
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	
t<1s															
0,12	4	8,74	0,17	4	6,17	0,15	2	3,50	0,25	1	1,04	0,33	8	6,36	
t<5s															
0,47	16	8,92	0,91	25	7,20	0,70	16	5,99	0,77	8	2,72	2,41	64	6,96	
t<10s															
0,90	32	9,32	1,79	49	7,18	1,39	32	6,04	1,38	16	3,04	4,33	120	7,27	
t<40s															
3,62	128	9,27	8,29	196	6,21	5,40	128	6,21	4,94	64	3,40	17,52	512	7,67	

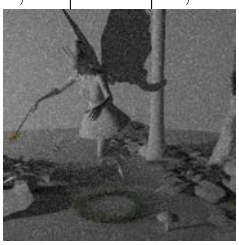
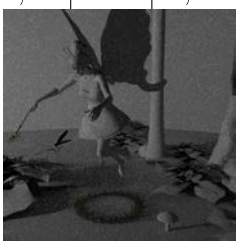
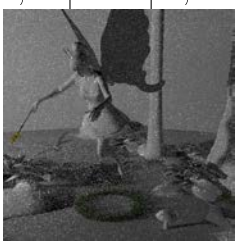
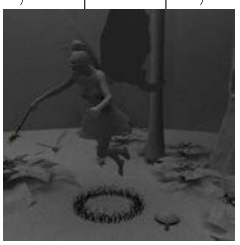
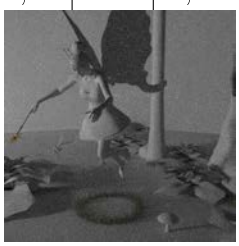
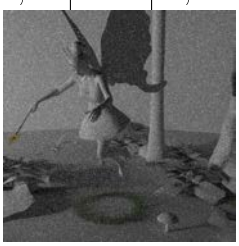
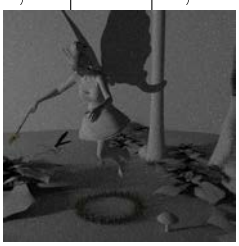
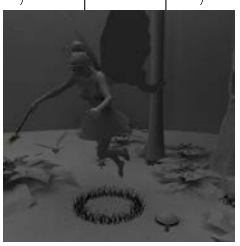
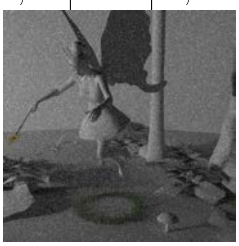
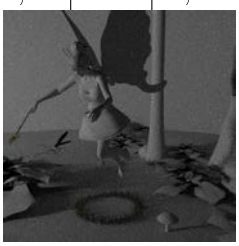
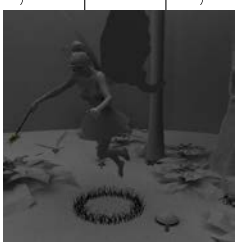
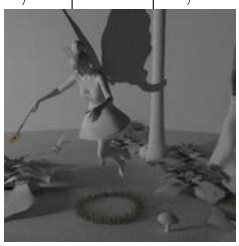
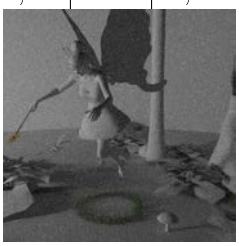
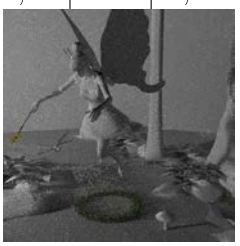
Tabulka A.6: Výsledky testu bez ruské rulety pro Cornell box na PC sestavě 2

Conference															
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**			
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	
t<5s															
0,46	8	4,56	0,40	1	0,66	0,71	8	2,95	0,61	2	0,86	1,31	16	3,20	
															
t<10s															
0,89	16	4,71	1,37	4	0,77	1,35	16	3,11	0,99	4	1,06	4,52	64	3,71	
															
t<20s															
1,78	32	4,71	3,07	9	0,77	2,72	32	3,08	1,75	8	1,20	8,63	128	3,89	
															
t<40s															
3,62	64	4,64	5,70	16	0,74	5,31	64	3,16	3,22	16	1,30	16,17	256	4,15	
															

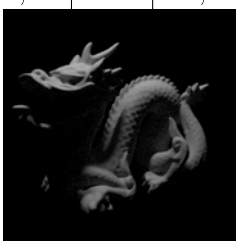
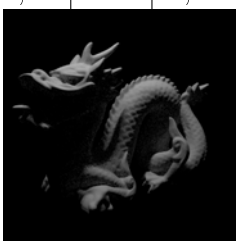
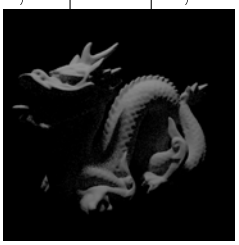
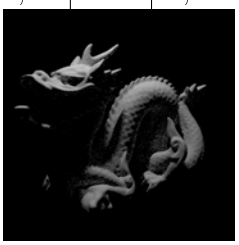
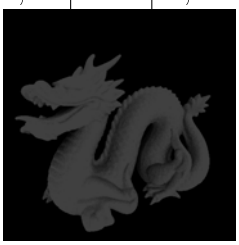
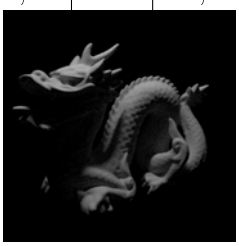
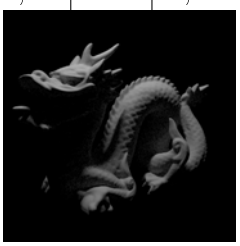
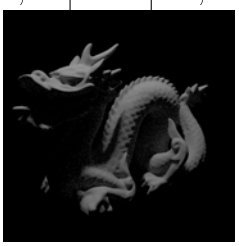
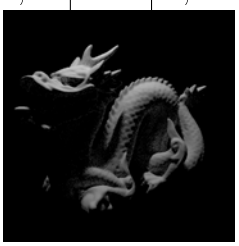
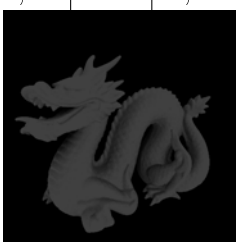
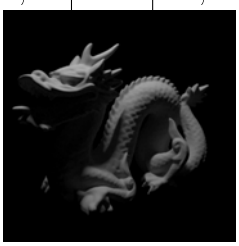
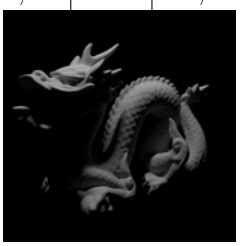
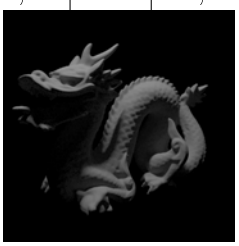
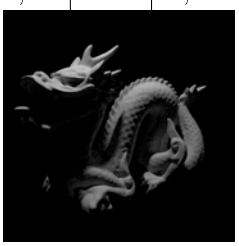
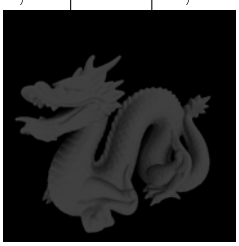
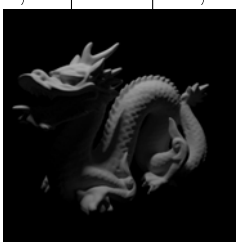
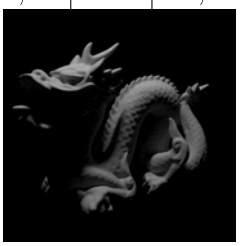
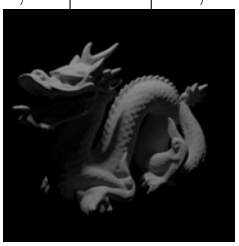
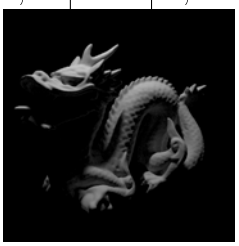
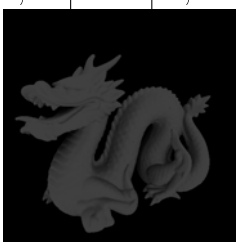
Tabulka A.7: Výsledky testu bez ruské rulety pro Conference na PC sestavě 2

Sibenik															
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s														
0,54	8	3,89	0,23	1	1,14	0,56	4	1,87	0,86	1	0,31	2,64	32	3,18
														
t<10s														
1,11	16	3,78	0,85	4	1,23	1,00	8	2,10	1,22	2	0,43	5,06	64	3,32
														
t<20s														
2,17	32	3,87	1,92	9	1,23	3,48	32	2,41	1,91	4	0,55	9,27	128	3,62
														
t<50s														
4,34	64	3,87	8,22	36	1,15	6,75	64	2,49	6,03	16	0,70	17,45	256	3,85
														

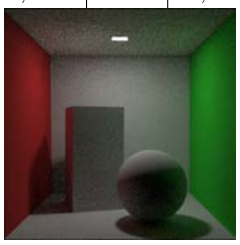
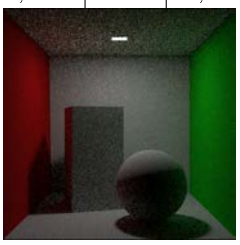
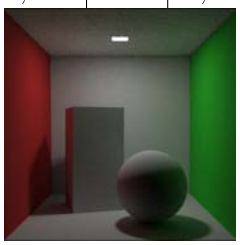
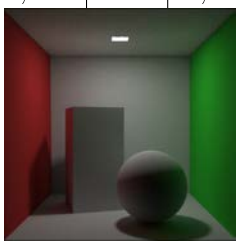
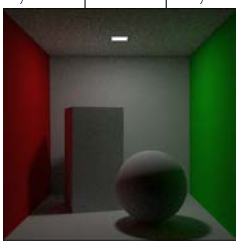
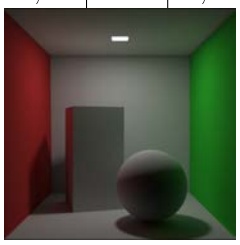
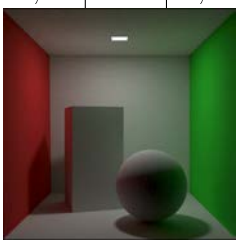
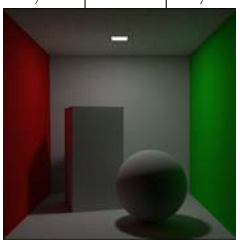
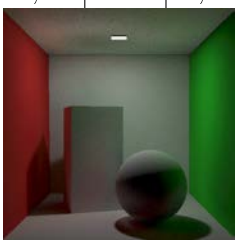
Tabulka A.8: Výsledky testu bez ruské rulety pro Sibenik na PC sestavě 2

Fairy forest														
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s														
0,57	16	7,36	0,81	4	1,30	0,68	8	3,08	0,77	4	1,36	2,44	32	3,44
														
t<10s														
1,14	32	7,36	1,78	9	1,33	1,02	16	4,11	1,27	8	1,65	4,63	64	3,62
														
t<20s														
2,29	64	7,33	3,21	16	1,31	1,99	32	4,22	2,23	16	1,88	8,86	128	3,79
														
t<40s														
4,41	128	7,61	5,31	25	1,23	3,98	64	4,22	4,23	32	1,98	16,02	240	3,93
														

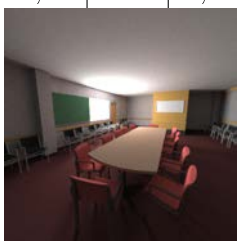
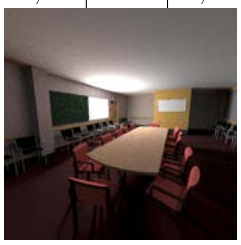
Tabulka A.9: Výsledky testu bez ruské rulety pro Fairy forest na PC sestavě 2

Dragon															
Embree			**Optix**			**Mitsuba**			**OpenRL**			**CUDA RT**			
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	
t<1s															
0,08	8	26,21	0,35	9	6,74	0,11	4	9,53	0,44	2	1,19	0,53	16	7,91	
															
t<2s															
0,15	16	27,96	0,48	16	8,74	0,20	8	10,49	0,52	4	2,02	1,03	32	8,14	
															
t<10s															
1,12	128	29,96	2,27	100	11,55	1,34	64	12,52	1,52	32	5,51	3,77	128	8,90	
															
t<20s															
2,20	256	30,50	5,02	196	10,24	2,69	128	12,47	2,47	64	6,79	7,53	256	8,91	
															

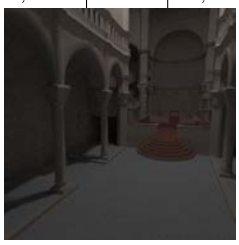
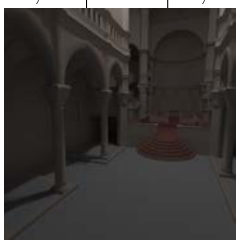
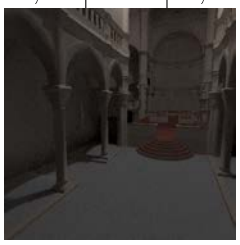
Tabulka A.10: Výsledky testu bez ruské rulety pro Dragon na PC sestavě 2

Cornell box												
Embree			**Optix**			**Mitsuba**			**OpenRL**			
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	
t<1s												
0,98	8	2,14	0,96	9	2,46	0,74	4	1,42	0,73	1	0,36	
												
t<5s												
3,89	32	2,16	3,57	36	2,64	2,66	16	1,58	4,23	8	0,50	
												
t<10s												
7,74	64	2,17	8,01	81	2,65	5,21	32	1,61	8,28	16	0,51	
												
t<40s												
30,86	256	2,18	26,49	256	2,53	20,72	128	1,62	31,92	64	0,53	
												

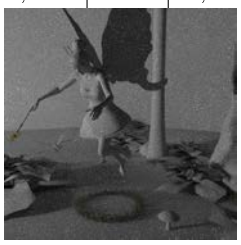
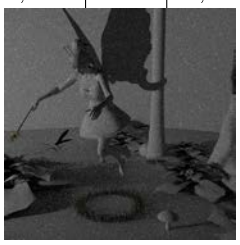
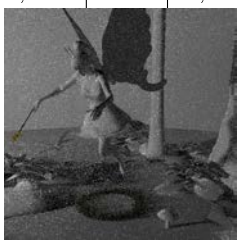
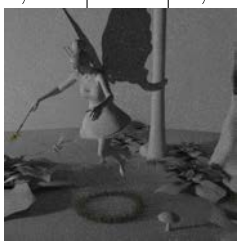
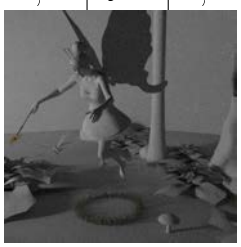
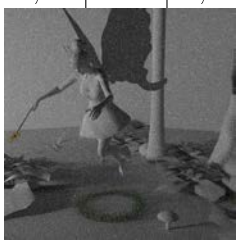
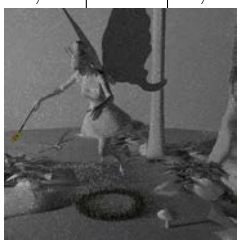
Tabulka A.11: Výsledky testu s ruskou ruletou pro Cornell box na PC sestavě 1 s RR

Conference											
Embree			**Optix**			**Mitsuba**			**OpenRL**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s											
3,02	16	1,39	3,97	9	0,59	3,20	16	1,31	3,85	8	0,55
											
t<10s											
5,98	32	1,40	6,92	16	0,61	6,37	32	1,32	7,25	16	0,58
											
t<20s											
11,93	64	1,41	15,56	36	0,61	12,66	64	1,33	14,54	32	0,58
											
t<40s											
23,5	128	1,43	27,96	64	0,60	25,39	128	1,32	28,58	64	0,59
											

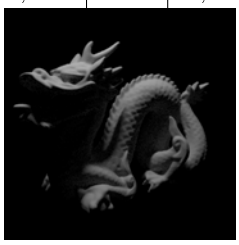
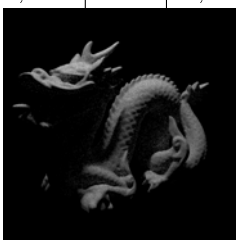
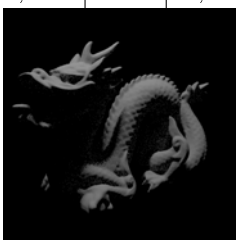
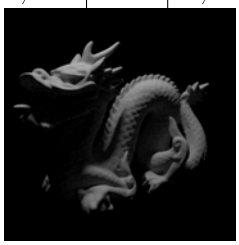
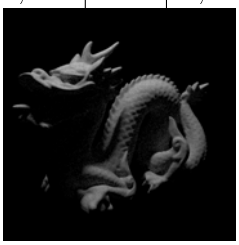
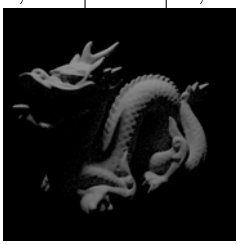
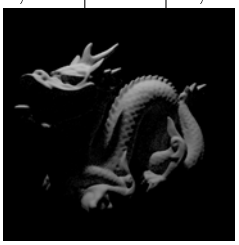
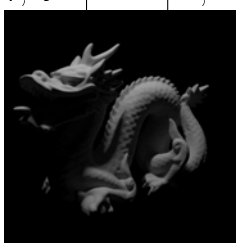
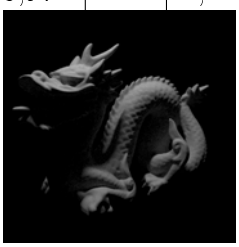
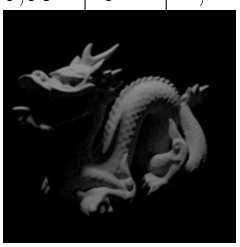
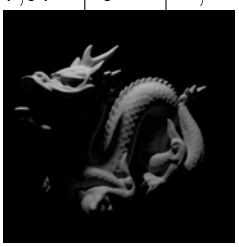
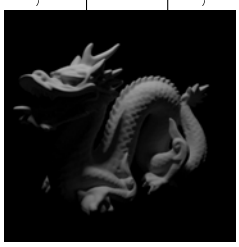
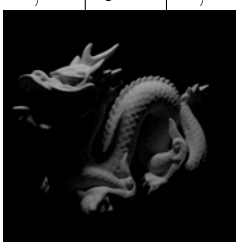
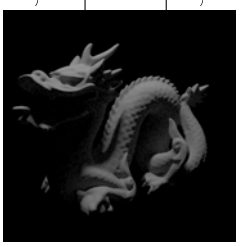
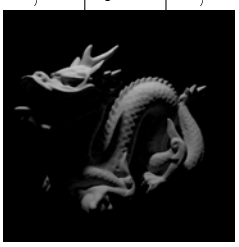
Tabulka A.12: Výsledky testu s ruskou ruletou pro Conference na PC sestavě 1 s RR

Sibenik											
Embree			**Optix**			**Mitsuba**			**OpenRL**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s											
4,68	16	0,90	2,86	4	0,37	2,84	8	0,74	2,94	1	0,09
											
t<10s											
9,46	32	0,89	6,33	9	0,37	5,64	16	0,74	5,36	2	0,10
											
t<20s											
19,04	64	0,88	11,18	16	0,38	11,14	32	0,75	10,34	4	0,10
											
t<50s											
37,72	128	0,89	35,12	49	0,37	22,16	64	0,76	38,85	16	0,11
											

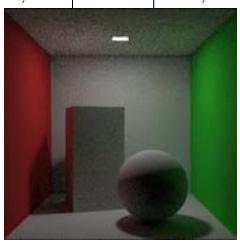
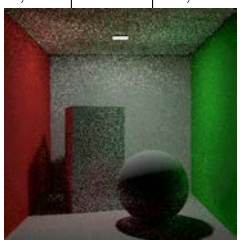
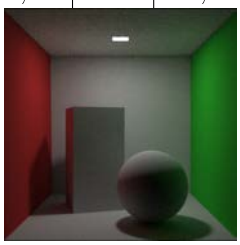
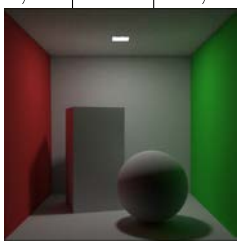
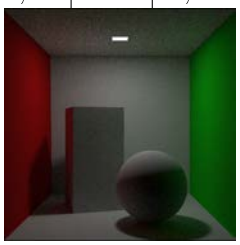
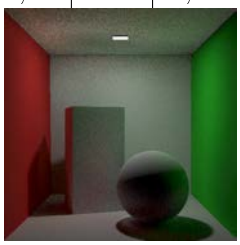
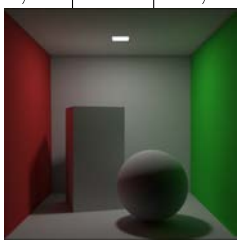
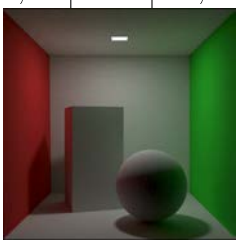
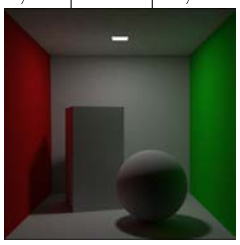
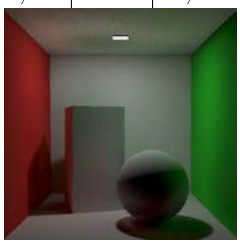
Tabulka A.13: Výsledky testu s ruskou ruletou pro Sibenik na PC sestavě 1 s RR

Fairy forest											
Embree			**Optix**			**Mitsuba**			**OpenRL**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s											
3,30	16	1,27	2,51	4	0,42	4,83	16	0,87	3,84	4	0,27
											
t<10s											
6,61	32	1,27	9,66	16	0,43	9,55	32	0,88	7,74	8	0,27
											
t<20s											
13,24	64	1,27	15,21	25	0,43	19,11	64	0,88	14,06	16	0,30
											
t<40s											
25,69	128	1,31	30,90	49	0,42	38,00	128	0,88	28,11	32	0,30
											

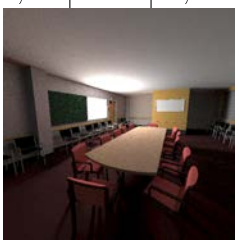
Tabulka A.14: Výsledky testu s ruskou ruletou pro Fairy forest na PC sestavě 1 s RR

Dragon											
Embree			**Optix**			**Mitsuba**			**OpenRL**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<1s											
0,96	16	4,37	0,84	16	5,00	0,89	8	2,36	0,82	2	0,64
											
t<2s											
1,87	32	4,49	1,82	36	5,19	1,70	16	2,47	1,67	4	0,63
											
t<10s											
7,29	128	4,60	8,97	169	4,94	6,53	64	2,57	7,57	32	1,11
											
t<20s											
14,54	256	4,62	18,62	324	4,56	12,92	128	2,60	13,78	64	1,22
											

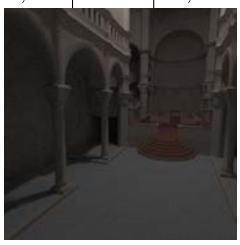
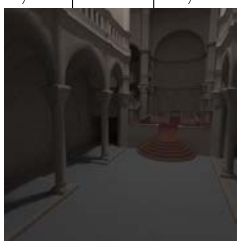
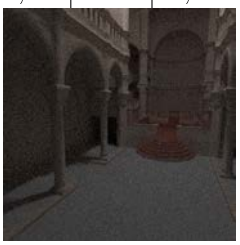
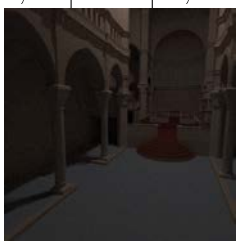
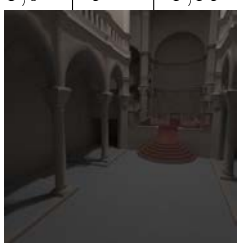
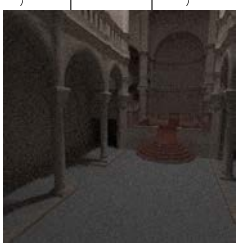
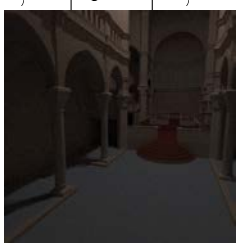
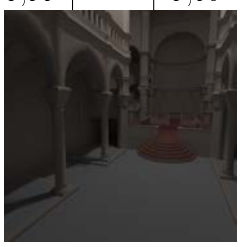
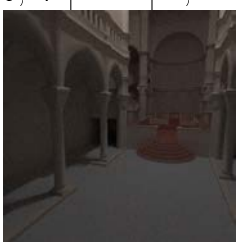
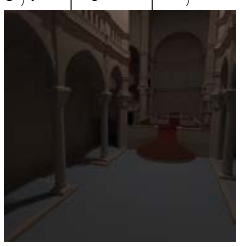
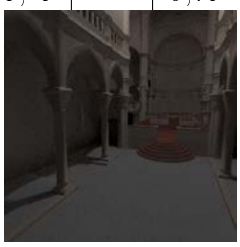
Tabulka A.15: Výsledky testu s ruskou ruletou pro Dragon na PC sestavě 1 s RR

Cornell box											
Embree			**Optix**			**Mitsuba**			**OpenRL**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<1s											
0,17	8	12,34	0,22	9	10,72	0,19	4	5,52	0,24	1	1,09
											
t<5s											
0,66	32	12,71	0,77	36	12,26	0,49	16	8,56	0,76	8	2,76
											
t<10s											
1,32	64	12,71	1,72	81	12,35	0,94	32	8,92	1,33	16	3,15
											
t<40s											
5,26	256	12,76	6,01	256	11,17	3,66	128	9,17	4,73	64	3,55
											

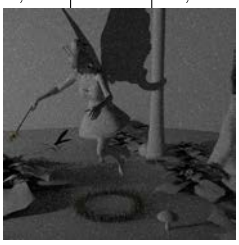
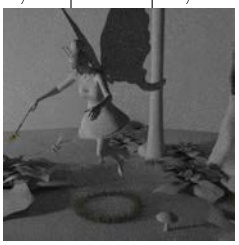
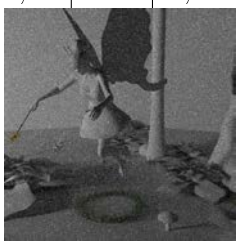
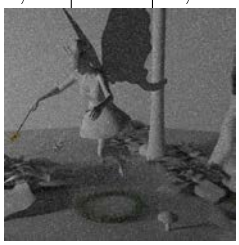
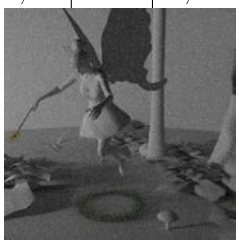
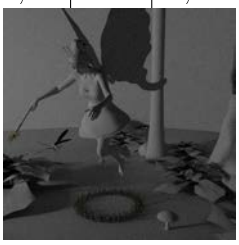
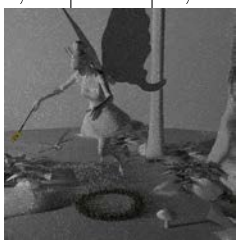
Tabulka A.16: Výsledky testu s ruskou ruletou pro Cornell box na PC sestavě 2 s RR

Conference											
Embree			**Optix**			**Mitsuba**			**OpenRL**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s											
0,50	16	8,39	0,77	9	3,06	0,60	16	6,99	0,72	8	2,91
											
t<10s											
1,00	32	8,39	1,33	16	3,15	1,09	32	7,70	1,18	16	3,55
											
t<20s											
2,01	64	8,35	3,04	36	3,10	2,19	64	7,66	2,12	32	3,96
											
t<40s											
3,95	128	8,50	5,44	64	3,08	4,34	128	7,73	3,91	64	4,29
											

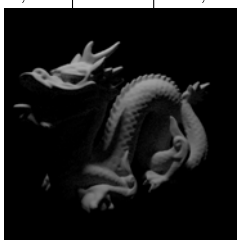
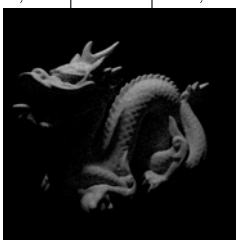
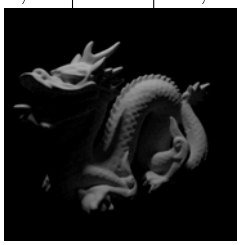
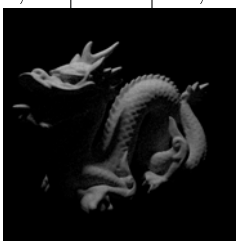
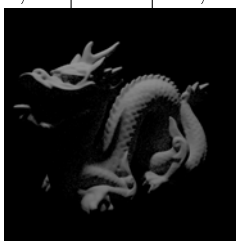
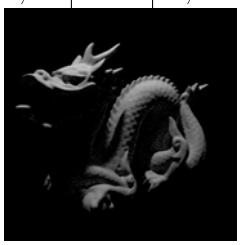
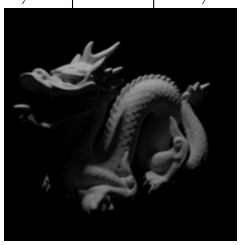
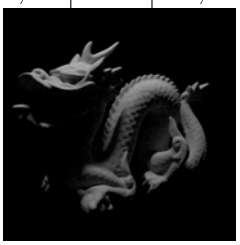
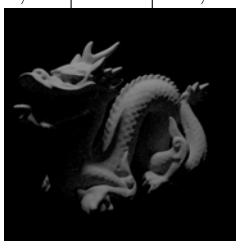
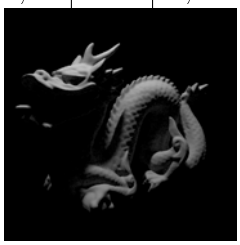
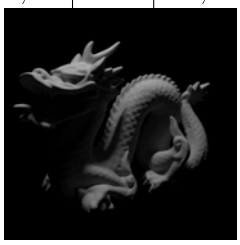
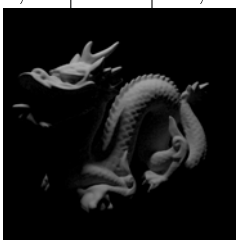
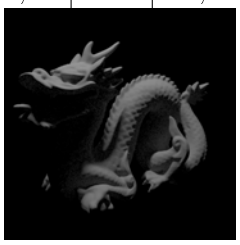
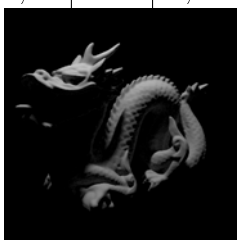
Tabulka A.17: Výsledky testu s ruskou ruletou pro Conference na PC sestavě 2 s RR

Sibenik											
Embree			**Optix**			**Mitsuba**			**OpenRL**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<5s											
0,75	16	5,59	0,46	4	2,28	0,58	8	3,62	0,86	1	0,31
											
t<10s											
1,51	32	5,56	0,97	9	2,43	1,03	16	4,07	1,17	2	0,45
											
t<20s											
3,02	64	5,56	1,71	16	2,45	1,93	32	4,35	1,78	4	0,59
											
t<50s											
5,99	128	5,60	5,27	49	2,44	3,72	64	4,51	5,49	16	0,76
											

Tabulka A.18: Výsledky testu s ruskou ruletou pro Sibenik na PC sestavě 2 s RR

Fairy forest												
Embree			**Optix**			**Mitsuba**			**OpenRL**			
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	
t<5s												
0,55	16	7,63	0,49	4	2,14	0,92	16	4,56	0,72	4	1,46	
												
t<10s												
1,11	32	7,56	1,78	16	2,36	1,55	32	5,41	1,19	8	1,76	
												
t<20s												
2,21	64	7,60	2,91	25	2,25	3,06	64	5,48	2,07	16	2,03	
												
t<40s												
4,32	128	7,77	6,22	49	2,07	6,03	128	5,57	3,90	32	2,15	
												

Tabulka A.19: Výsledky testu s ruskou ruletou pro Fairy forest na PC sestavě 2 s RR

Dragon											
Embree			**Optix**			**Mitsuba**			**OpenRL**		
t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed	t [s]	Spp	Speed
t<1s											
0,16	16	26,21	0,20	16	20,97	0,17	8	12,34	0,47	2	1,12
											
t<2s											
0,31	32	27,06	0,39	36	24,20	0,31	16	13,53	0,52	4	2,02
											
t<10s											
1,24	128	27,06	1,86	169	23,82	1,14	64	14,72	1,16	32	7,23
											
t<20s											
2,44	256	27,50	4,68	324	18,15	2,20	128	15,25	1,86	64	9,02
											

Tabulka A.20: Výsledky testu s ruskou ruletou pro Dragon na PC sestavě 2 s RR

Příloha B

Instalační a uživatelská příručka

Všechny popisované instalace a kompilace knihoven pro vrhání paprsku jsem prováděl na operačním systému Windows, a pro tento systém budu tyto úkony v následujících sekcích také popisovat.

B.1 Intel Embree

Intel Embree je možné stáhnout na adrese http://embree.github.io/downloads.html a to jak v předkompilované, tak nekompilované verzi. Při tvorbě práce byla používána verze 2.3.3.

B.1.1 Předkompilovaná verze

Po stažení předkompilované verze je třeba archiv rozbalit a do proměnných systému je třeba přidat cestu k $lib-x64$ nebo k $lib-win32$ v závislosti na tom, jaká verze operačního systému je na PC nainstalována.

B.1.2 Nezkompilovaná verze

Po stažení nezkompilované verze je nejprve třeba archiv rozbalit. Pro zkompilování je nutné stáhnout a nainstalovat SPMD Compiler z odkazu https://ispc.github.io/downloads.html. Po rozbalení archivu je třeba do proměnných systému přidat cestu k *ispc.exe*. Pro kompilaci Embree je třeba otevřít soubor *embree.sln*.

Při prvním pokusu o kompilaci se může objevit tato chybová hláška:

```
Project file contains ToolsVersion="12.0", which is not supported by this version
of MSBuild. Treating the project as if it had ToolsVersion="4.0".
```

V tomto případě je nutné změnit ToolVersion na verzi 10.

B.1.3 Ukázkový renderer

Ukázkový renderer ke knihovně Embree lze stáhnout na adrese http://embree.github.io/renderer.html a to jak v předkompilované, tak nezkompilované verzi.

Předkompilovaná verze lze rovnou po extrakci archivu používat, pokud je nainstalována samotná knihovna. Způsob použití je naznačen v sekci 2.1.3.

Pro kompilaci nezkompilované verze, je nejprve nutné nastavit do proměnných systému proměnnou *EMBREE_INSTALL_DIR*, která bude obsahovat cestu ke složce s instalací knihovny Embree a ujistit se, že je nainstalován SPMD Compiler. Pro kompilaci je nutné otevřít soubor *embree – renderer_vs*2010.*sln*

B.2 NVIDIA Optix

Knihovnu NVIDIA Optix je možné stáhnout až po podání žádosti na adrese https://developer.nvidia.com/optix. Poté budou zájemci na e-mail zaslány přihlašovací údaje na server https://ftpservices.nvidia.com/, odkud je možné stáhnout instalační balíček knihovny. Při tvorbě práce byla používána verze 3.5.3. Po instalaci je možné používat všechny ukázkové aplikace, ale je nutné mít grafický adaptér kompatibilní se systémem CUDA.

B.2.1 Zdrojové kódy aplikací

Pro získání zdrojových kódů ukázkových aplikací je potřeba nainstalovat:

- CUDA Toolkit starší než 6.5
- CMake http://www.cmake.org/DOWNLOAD/

Dále je nutné postupovat podle souboru *INSTALL – WIN*, který je umístěn ve složce *SDK*.

B.3 PowerVR OpenRL

Stažení této knihovny je možné na adrese http://community.imgtec.com/developers/powervr/openrl-sdk/. Po instalaci jsou ukázkové aplikace dostupné jak v předkompilované, tak nezkompilované verzi.

B.3.1 Nezkompilovaná verze

Při kompilaci ukázkových aplikací lze narazit na chybovou hlášku

```
xcopy neni nazvem vnitrniho ani vnejsiho prikazu, spustitelneho programu nebo davkoveho souboru.
```

Pro její odstranění je nutné mít v proměnných systému nastavenu cestu do složky *win*32.

B.4 Mitsuba

Stažení předkompilované a nekompilované verze Mitsuby je možné na odkazu http://www.mitsuba-renderer.org/download.html. Při tvorbě práce byla použita verze 0.5.0.

B.4.1 Předkompilovaná verze

Po rozbalení archivu lze Mitsuba bez problému používat. Způsob použití je naznačen v sekci 2.3.3.

B.4.2 Nezkompilovaná verze

Pro kompilaci je nejdříve nutné nainstalovat:

- Scons http://www.scons.org/
- Python https://www.python.org

Dále je nutné stáhnout dependence z odkazu https://www.mitsuba-renderer.org/repos/dependencies_windows, nakopírovat je do složky s Mitsubou a přejmenovat je na *dependencies*

Poté je třeba ze složky *build* zvolit správný *py* soubor pro konfiguraci, zkopírovat ho do složky s Mitsubou a přejmenovat na $config.py$. Pro 64-bitový Windows byl správným konfiguračním souborem soubor $config - win64 - msvc2010$.

Zbývá už jen otevřít soubor $mitsuba - msvc2010.sln$ ve složce *build*.

B.5 Understanding the Efficiency of Ray Traversal on GPUs (CUDA RT)

Stažení archivu je možné na adrese https://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/downloads/detail?name=gpu-ray-traversal-1.4.zip&can=2&q=. V archivu je jak předkompilovaná, tak nekompilovaná verze. Pro použití obou verzí je třeba nastavit $CUDA\_BIN\_PATH$ a $CUDA\_INC\_PATH$ do proměnných systému. Použití je naznačeno v sekci 2.5.3.

Příloha C

Obsah přiloženého DVD

Zde je uveden obsah přiloženého DVD.

- **src** - Zdrojové kódy aplikací použitých k testování: ukázkového rendereru psaného pomocí Embree, aplikace, která je přiložena k článku Understanding the Efficiency of Ray Traversal on GPUs, Mitsuba rendereru, aplikace psané v OpenRL, a aplikací napsaných pomocí Optixu, pro které je soubor *sln* ve složce **\Optix\build**. Pro testování byla použita aplikace *path_tracer*
- **bin** - Obsahuje spustitelné odkazy na používané aplikace
- **scenes** - Obsahuje všech pět scén použitých pro testování, a to jak ve formátu *obj*, tak ve formátu pro Mitsubu
- **scripts** - Obsahuje scripty používané pro měření
- **figures** - Obrázky získané z testů
- **doc** - Obsahuje jak zdrojové soubory celého textu této práce, tak práci ve formátu *PDF*
- **web** - Obsahuje zdrojové soubory webové stránky, která slouží k prezentaci výsledků srovnání parametrů knihoven a výsledků srovnávacích testů.
- *readme.txt* - Popis používání aplikací používaných pro testování