Návrh a implementace open-source rendereru využívajícího metodu Path tracing

Design and implementation of an open-source renderer using path tracing

Bc. Filip Sadloň

Diplomová práce
2010

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2009/2010

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Filip SADLOŇ**
Osobní číslo:	**A08471**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Návrh a implementace open-source rendereru využívajícího metodu Path tracing**

Zásady pro vypracování:

1. **Vytvořte rešerši na téma SW rendering komplexních scén a srovnání dostupných open-source řešení.**
2. **Srovnejte vybrané metody globálního osvětlení scén.**
3. **Vytvořte dokumentaci a popis algoritmů Path tracing, Bidirectional path tracing, Metropolis light transport.**
4. **Na základě vhodně zvolených metod a algoritmů implementujte softwarový renderer využívající metodu path tracing.**
5. **Srovnejte vlastnosti Vaší implementace a výkon rendereru s konkurenčními produkty.**
6. **Vytvořte programovou a uživatelskou dokumentaci rendereru.**
7. **Vytvořte skript pro export scény z programu Blender do formátu podporovaného nově vytvořeným rendererem.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. SOBOTA, B. Počítačová grafika a jazyk C: Kopp, 1995, 272s, ISBN 80-85828-52-9.
2. HARMS, D., MCDONALD, K. Začínáme programovat v jazyce Python. 1. vyd. Praha: Computer Press, 2003. 456 s. ISBN 80-7226-799-X.
3. Blender website, URL: http://www.blender.org/.
4. PHARR, Matt, HUMPHREYS, Greg. Physically based rendering: from theory to implementation. [s.l.] : Morgan Kaufmann, 2004. 1056 s. ISBN 978-0-12-553180-1.
5. HAVRAN, Vlastimil. Heuristic Ray Shooting Algorithms. [s.l.], 2001. 220 s. Dizertační práce.
6. DUTRE, Philip. Global Illumination Compendium. [online]. 2003.
7. WALD, Ingo, HAVRAN, Vlastimil. On building fast kd-trees for ray tracing, and on doing that in O(N log N). Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing. 2006, s. 61-69.
8. WALD, Ingo. Realtime Ray Tracing and Interactive Global Illumination. [s.l.], 2004. 311 s. Dizertační práce.
9. VEACH, Eric. ROBUST MONTE CARLO METHODS FOR LIGHT TRANSPORT SIMULATION. [s.l.], 1997. 432 s. Dizertační práce.
10. FOG, Agner. Optimizing software in C++. [s.l.] : [s.n.], 2008. 144 s.

Vedoucí diplomové práce: Ing. Michal Bližňák, Ph.D.
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **19. února 2010**

Termín odevzdání diplomové práce: **8. června 2010**

Ve Zlíně dne 19. února 2010

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

prof. Ing. Vladimír Vašek, CSc.
děkan

prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

ABSTRAKT

Diplomová práca sa zaoberá renderovaním komplexných 3D scén a výpočtom globálneho osvetlenia. Postupne sú vysvetlené jednotlivé komponenty, ktoré tvoria softwarový renderer ako materiály, kamery, urýchľovacie štruktúry a zdroje svetla. Práca ďalej popisuje algoritmy, ktoré je možné použiť na riešenie zobrazovacej rovnice a na výpočet globálneho osvetlenia. Spracované sú postupne algoritmy path tracing, bidirectional path tracing a metropolis light transport.

Cieľom praktickej časti práce je navrhnúť a implementovať softwarový renderer, ktorý využíva algoritmus path tracing. Ďalej sa práca zaoberá tvorbou exportného skriptu pre program Blender, pomocou ktorého je možné 3D scény z Blenderu exportovať do formátu podporovaného týmto rendererom.

Kľúčové slová: sledovanie ciest, globálne osvetlenie, zobrazovacia rovnica, softwarový rendering, path tracing, obojsmerný path tracing, metropolis light transport, BRDF

ABSTRACT

The thesis deals with rendering of complex 3D scenes and global illumination calculation. Step by step, basic components of software renderer like materials, cameras, acceleration structures and light sources are explained. Then algorithms for solving the rendering equation and global illumination computation like path tracing, bidirectional path tracing and metropolis light transport algorighms are discussed.

The aim of the practical part of thesis is to design and implement software renderer that employs path tracing algorithm. Then creation of export script for Blender is described. Export script can be used to export 3D scene from Blender into file format supported by the created renderer.

Keywords: path tracing, global illumination, rendering equation, software rendering, bidirectional path tracing, metropolis light transport, BRDF

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně …….

podpis diplomanta

OBSAH

ÚVOD

Diplomová práca sa zaoberá renderovaním 3D scén a výpočtom globálneho osvetlenia. V práci sú popísané najznámejšie algorimy pre tento výpočet a vysvetlené základné komponenty rendereru.

S postupným zrýchľovaním výkonu počítačov a nástupom viacjadrových procesorov sa čoraz viac využívajú presné metódy simulácie svetla. Pri tvorbe kvalitných renderov sa už takmer výlučne využívajú métódy založené ná výpočte globálneho osvetlenia. Kvalitným algoritmom pre výpočet glogálneho osvetlenia je možné s pomerne malým úsilím dosahovať fotorealistických výsledkov. S pôvodným raytracingom je možné na súčasných počítačoch dosiahnuť rýchlosť protrebnú pre interaktívne zobrazenie, realtime raytracing je predmetom aktívneho výzkumu. Ďalší priestor pre zrýchlenie priniesol pomerne nový štandard OpenCL. OpenCL umožňuje aby sa na výpočtoch podielali aj grafické karty, ktoré dosahujú nemalých výkonov a umožňujú paralelné spracovanie veľkého množstva dát.

Fyzikálne založené metódy majú okrem kvality výstupu aj iné výhody ako napríklad jednoduché a intuitívne nastavenie materiálov a osvetlenia. Nevýhody sú hlavne dlhšia doba potrebná na vytvorenie renderu a v niektorých prípadoch tiež menšia flexibilita. Hlavne pri umeleckom využití fotorealizmus a fyzikálne presné osvetlenie nie sú vždy žiadané.

Cieľom práce je popísať teoretické základy nutné na vytvorenie rendereru a zároveň ich implementovať do programu. Výstupom práce je renderer s názvom fiiRay, ktorý implementuje algoritmy path tracing, bidirectional path tracing a Metropolis Light Transport. Hlavným cieľom bolo vytvoriť renderer, ktorý bude podporovať kvalitné algoritmy pre výpočet osvetlenia, bude rýchly a ľahko rozšíriteľný o nové funkcie. Ďalším cieľom bolo vytvoriť renderer, ktorý bude ľahko a intuitívne ovládateľný bez zbytočne veľkého množstva parametrov.

I. TEORETICKÁ ČASŤ

1 GLOBÁLNE OSVETLENIE

Pojmom globálne osvetlenie sa v počítačovej grafike označujú spôsoby výpočtu osvetlenia, ktoré okrem priameho svetla pri výpočte zohľadňujú aj nepriame osvetlenie. Nepriamym osvetlením sa rozumie svetlo, ktoré sa zo svetelného zdroja k pozorovateľovi šíri viacerými difúznymi, zrkadlovými alebo inými odrazmi. Výpočet globálneho osvetlenia je založený na zobrazovacej rovnici (1.1), ktorú formuloval v roku 1986 James Kajiya.

Na to aby bolo možné vypočítať globálne osvetlenie v scéne a vytvoriť z 3D scény 2D obrázok je nutné vyriešiť viacero problémov:

1. Odkiaľ a ako sa bude scéna zobrazovať- tento problém rieši kamera
2. Odkiaľ sa do scény šíri svetlo- je nutné vytvoriť zdroje svetla
3. Problém viditeľnosti- je nutné definovať geometrický tvar objektov scény aby bolo možné zistiť priesečníky lúčov s telesami
4. Materiály- je nutné určiť akým spôsobom sa lúče svetla odrážajú od objektov

1.1 Označovanie ciest

Aby bolo možné rozlíšiť medzi jednotlivými typmi ciest, ktoré môžu vzniknúť zaviedol Paul Heckbert značenie ciest. Cesty sa značia podľa typu vrcholov, ktoré obsahujú. Môžu nastať prípady:

L – svetlo

E – pozorovateľ

D – difúzny odraz

S – zrkadlový odraz

Na popis kombinácií rôznych vrcholov v cestách sa využívajú regulárne výrazy:

(e)* – nula alebo viac výskytov

(e)+ – jeden alebo viac výskytov

(e1|e2) – výskyt udalosti e1 alebo e2

Jednotlivé algoritmy na riešenie globálneho osvetlenia je možné popísať pomocou typov ciest, ktoré sú schopné riešiť. Kvalita výpočtu globálneho osvetlenia závisí na tom, aké typy ciest algoritmus uvažuje. Kompletné globálne osvetlenie obsahuje cesty L(D|S)*E. Algoritmus, ktorý uvažuje všetky tieto cesty je napríklad path tracing. Naproti tomu ray tracing neuvažuje odrazy medzi viacerými difúznymi materiálmi a zanedbáva napr. cesty typu LDDD*E. Ak algoritmus zanedbáva určité typy ciest, potom vo výslednom obrázku chýba príspevok z týchto ciest a získané riešenie nie je kompletným riešením zobrazovacej rovnice. [2]

Algoritmy, ktoré riešia problém globálneho osvetlenia teda možno deliť na presné metódy (unbiased) a metódy ktoré vypočítajú len aproximované globálne osvetlenie (biased). Medzi presné metódy patrí napríklad path tracing a obojsmerný path tracing, typické aproximačné metódy sú photon mapping a irradiance caching. Inou možnosťou delenia algoritmov je smer v ktorom sledujú šírenie svetla. Path tracing sleduje lúče z kamery, light tracing zo svetiel, obojsmerný path tracing kombinuje tieto dva prístupy.

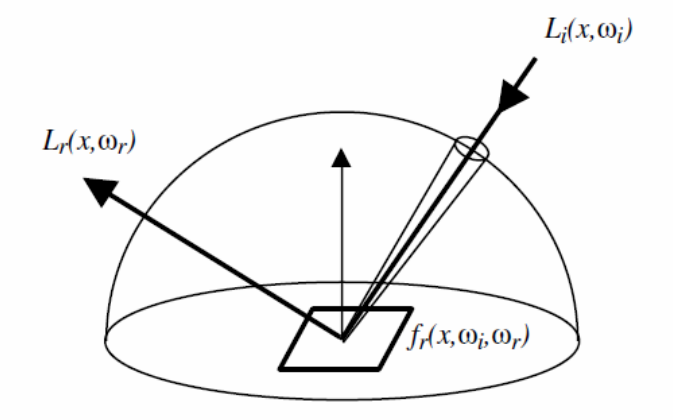
1.2 Zobrazovacia rovnica

Zobrazovacia rovnica je integrálna rovnica, ktorá určuje žiarenie bodu ako súčet emitovaného a odrazeného žiarenia. Zobrazovaciu rovnicu predstavil v roku 1986 James Kajiya.[15]

$$L_o(x,\omega_o) = L_e(x,\omega_o) + L_r(x,\omega_o) \tag{1.1}$$

Odrazené žiarenie L_r je možné vypočítať ako integrál prichádzajúceho žiarenia L_i nad pologuľou umiestnenou v bode na povrchu telesa a orientovanou tak, že jej severný pól je v smere normály povrchu. [15]

$$L_r(x,\omega_o) = \int_{\Omega^+} f_r(x,\omega_i \rightarrow \omega_o) L_i(x,\omega_i) \cos\theta_i d\omega_i \tag{1.2}$$



Obr. 1: Odrazené žiarenie[1]

Odrazené žiarenie L_r je tiež možné vypočítať ako integrál prichádzajúceho žiarenia L_i cez plochu. V niektorých prípadoch je vhodné použiť na výpočet drazeného žiarenia integrál nad plochov (napr. výpočet priameho osvetlenia),

$$L_r(x,\omega_o) = \int_A L_o(y,\overrightarrow{xy}) f_r(x,\overrightarrow{xy},\omega) V(x,y) G(x,y) dA \tag{1.3}$$

Kde *V(x,y)* je funkcia viditeľnosti definovaná ako:

$$V(x,y) = \begin{cases} 1 & visible \\ 0 & else \end{cases} \tag{1.4}$$

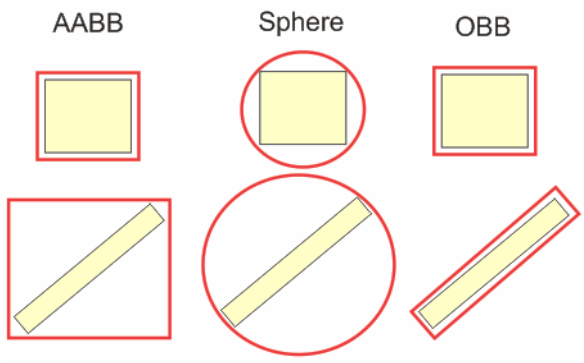
G(x,y) je geometrický faktor definovaný ako:

$$G(x,y) = \frac{\cos(N_x,\overrightarrow{xy})\cos(N_y,\overrightarrow{yx})}{\left\|\overrightarrow{xy}\right\|^2} \tag{1.5}$$

2 URÝCHĽOVANIE VYHĽADÁVANIA PRIESEČNÍKOV

2.1 Obalové telesá

Vyhodnocovanie priesečníkov so zložitými objektami je výpočetne náročné, je preto zvyčajne vhodné použiť pre takýto objekt alebo skupinu objektov nejaké obalové teleso. Testovanie priesečníku s obalovým telesom je oveľa rýchlejšie a ak sledovaný lúč nepretína obalové teleso, nie je nutné testovať ani objekty, ktoré sa v ňom nachádzajú. Prínos takéhoto prístupu záleží hlavne na dvoch faktoroch: na zložitosti výpočtu priesečníku s obalovým telesom a na tesnosti, s akou teleso obaluje objekt. Ak je obalové teleso nepresné a objekt v ňom vypĺňa len jeho časť, potom môžu často nastávať prípady, kedy test obalového telesa rozhodol, že je nutné testovať objekt v ňom, aj keď lúč tento objekt v skutočnosti nepretína. Naopak ak je obalové teleso zložitejšie môže obalovať objekt tesnejšie, ale zároveň narastá zložitosť vyhodnotenia priesečníku [2].



Obr. 2: Rôzne typy obalových telies

Existuje mnoho typov obalových telies, napr. osovo zarovnaný kváder (AABB- axis-aligned bounding box), orientovaný kváder (OBB- oriented bounding box), obalová guľa, elipsoid. Rôzne typy obalových telies sa hodia na rôzne úlohy, pre potreby ray tracingu sa najviac hodí AABB kvôli svojej jednoduchosti, ktorou kompenzuje menšiu tesnosť.

2.2 Mailboxing

Mailboxing (poštové schránky) je metóda, ktorou sa urýchľuje vyhľadávanie priesečníkov pri viacnásobnom testovaní lúča voči objektu. Viacnásobné testy nastávajú ak objekt v nejakej urýchľovacej štruktúre presahuje viac buniek. Aby nebolo nutné test vykonávať opakovane, výsledky prvého testu medzi daným lúčom a objektom sa uložia. Pred každým výpočtom priesečníku je nutné skontrolovať schránku, ak bol test už

vykonaný použije sa výsledok uložený v schránke, ak nie vypočíta sa priesečník a uloží sa do schránky pre ďalšie použitie. [2]

2.3 Urýchľovacie štruktúry

Vyhodnocovanie priesečníkov lúčov a objektov v scéne je časovo náročná operácia. Hľadanie najbližšieho priesečníku hrubou silou má zložitosť *O(n)*, pretože je nutné lúč testovať voči každému objektu. Takýto prístup je možné použiť len vo veľmi jednoduchých scénach s malým počtom objektov. Preto je pri renderovaní bežných scén nutné požiť nejakú urýchľovaciu štruktúru, ktorá bude pri hľadaní najbližšieho priesečníku zohľadňovať rozloženie objektov v priestore. Cieľom takýchto štruktúr je zamietnuť testovanie tých objektov, ktoré sa nenachádzajú v blízkosti lúča. Objekty, ktoré sú v blízkosti lúča by mali byť testované tak, aby bol najbližší priesečník nájdený po minimálnom počte testov. [2]

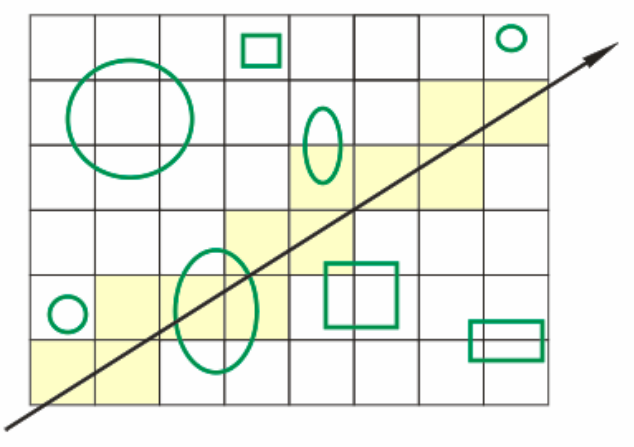
Existuje množstvo typov urýchľovacích štruktúr, každá je vhodná pre iný typ problému. Hlavnými parametrami, ktoré ovplyvňujú voľbu konkrétneho algoritmu sú: pamäťová náročnosť, rýchlosť prechodu, doba vytvorenia, priemerný počet testovaných objektov na jeden lúč, platforma. Štruktúry sa tiež rozlišujú tým, či rozdeľujú priestor(BSP) alebo rozdeľujú objekty(BVH).

BSP (Binary Space Partitioning) stromy sú hierarchické štruktúry, ktoré adaptívne delia priestor na rôzne veľké časti. Vďaka tomu dosahujú dobré výsledky aj pre nerovnomerné rozloženie objektov v scéne. Tvorba stromu začína získaním obalového kvádra scény, ak je počet objektov vačší ako povolené minimum priestor sa rozdelí deliacou rovinou. Objekty scény sa následne rozdelia podľa ich polohy voči deliacej rovine, ak túto rovinu pretínajú su priradené na obidve strany. Delenie priestoru sa ukončí ak počet objektov v uzle stromu klesne pod stanovenú hranicu alebo ak sa dosiahne maximálna hĺbka stromu. Podľa počtu deliacich rovín alebo ich polohy požno rozlíšiť viacero typov BSP stromov. Najznámejšie sú oktalový strom (octree), ktorý delí priestor tromi osovo zarovnanými rovinami a kd-strom. Kd-strom delí priestor jednou osovo zarovnanou rovinou.

2.3.1 Pravidelná mriežka

Pravidelná mriežka je jedna z najjednoduchších štruktúr. Mriežka delí priestor na menšie, rovnako veľké časti- voxely. Každý voxel nesie informáciu o objektoch, ktoré sa v ňom nachádzajú. Pri vyhľadávaní priesečníku s lúčom sa postupne prechádzajú voxely, ktoré lúč pretína a testujú sa len objekty uložené v týchto voxeloch. Vďaka tomu mriežka znižuje počet testov nutných k nájdeniu najbližšieho priesečníku. Objekty, ktoré neležia blízko lúča nie sú vôbec testované a keďže sa voxely prechádzajú od najbližšieho po najvzdialenejší, po nájdení priesečníku nie je nutné testovať ďalšie voxely, lebo bližší priesečník nemôže existovať. Prechod mriežkou sa realizuje pomocou algoritmu podobnému 3D DDA. [2]

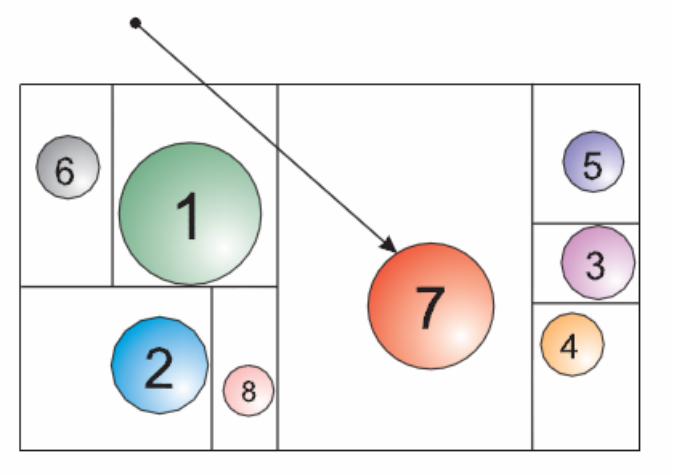
Jednoduchosť tejto štruktúry ale prináša viacero problémov. Jedným problémom je nerovnomerné rozmiestnenie objektov v priestore, môže nastať situácia, keď je v jednom voxeli veľký počet objektov. Možným riešením je zmenšenie voxelov alebo hierarchická mriežka. Zmenšením voxelov je síce možné zrýchliť prehľadávanie priestoru s veľkým počtom objektov, ale spomalí sa prechod prázdnym priestorom. Hierarchická mriežka je mriežka, ktorej voxely môžu byť ďalej delené ak obsahujú veľa objektov. Jednotlivé voxely v tomto prípade môžu obsahovať celú ďalšiu mriežku. Ďalším problémom je viacnásobné testovanie priesečníku voči jednému objektu- môže nastať situácia, keď objekt presahuje niekoľko voxelov, pri prechode každým z týchto voxelov je nútné spočítať prieseník s objektom. Problém presahujúcich objektov je možné riešiť mailboxingom.



Obr. 3: Prechod pravidelnou mriežkou

2.3.2 KD-strom

KD-strom je špecifickým prípadom BSP stromu. KD-strom delí priestor na dve časti osovo zarovnanou rovinou. Delenie priestoru prebieha rekurzívne a ukončí sa po splnení niektorej s ukončovacích podmienok. Jednou ukončovacou podmienkou je počet objektov v uzle stromu, ak počet klesne pod stanovenú hranicu delenie sa ukončí. Ďalšou typickou ukončovacou podmienkou je maximálna hĺbka stromu. Táto podmienka ošetruje situáciu, keď sa niekoľko objektov prekrýva a nie je možné rozdeliť priestor tak, aby počet objektov klesol pod minimálnu hranicu. KD-strom je vhodný hlavne na renderovanie statických scén pretože jeho vytvorenie môže byť pri zložitých scénach časovo náročné. Pri statických scénach, kde je strom tvorený len raz na začiatku sa doba tvorby kompenzuje rýchlosťou renderovania. [5]



Obr. 4: Rozdelenie priestoru pri použití KD-stromu [12]

2.3.2.1 Delenie priestoru

Rovinu, ktorou sa rozdelí priestor na dve časti je možné zvoliť viacerými metódami. Delenie priestoru prebieha vždy v jednej z troch súradnicových osí X, Y, Z. Každá z týchto metód vedie k vytvoreniu iného stromu a má zásadný vplyv na rýchlosť stromu, teda aj na výkon celého programu. Okrem rýchlosti vyhľadávania priesečníkov závisí od zvolenej metódy aj doba vytvorenia stromu. [7]

- Medián: deliacu rovinu volí tak, aby bol na obidvoch jej stranách približne rovnaký počet objektov, objekty musia byť zoradené v smere osi delenia.
- Priestorový medián: najjednoduchšia metóda, os delenia sa zvolí v smere najväčšieho rozmeru a priestor sa rozdelí na polovicu.

- SAH (Surface Area Heuristic): táto metóda je založená na predpoklade, že počet lúčov pretínajúcich objekt je úmerný veľkosti povrchu objektu. Pri určovaní deliacej roviny sa teda nezohľadňuje len počet objektov, ale aj veľkosť priestoru v ktorom sa nachádzajú. SAH uvažuje deliace roviny len na miestach, kde sa mení počet objektov, teda na začiatku a na konci každého objektu vzhľadom k jednej zo súradnicových osí. Pre každú potenciálnu rovinu sa vyhodnotí cena *C* ako

$$C = C_t + C_i \cdot (A_L \cdot N_L + A_R \cdot N_R) \tag{2.1}$$

kde:

C_t – cena prechodu do ďalšieho uzla (cost of traversal)

C_i – cena výpočtu priesečníku (cost of intersection)

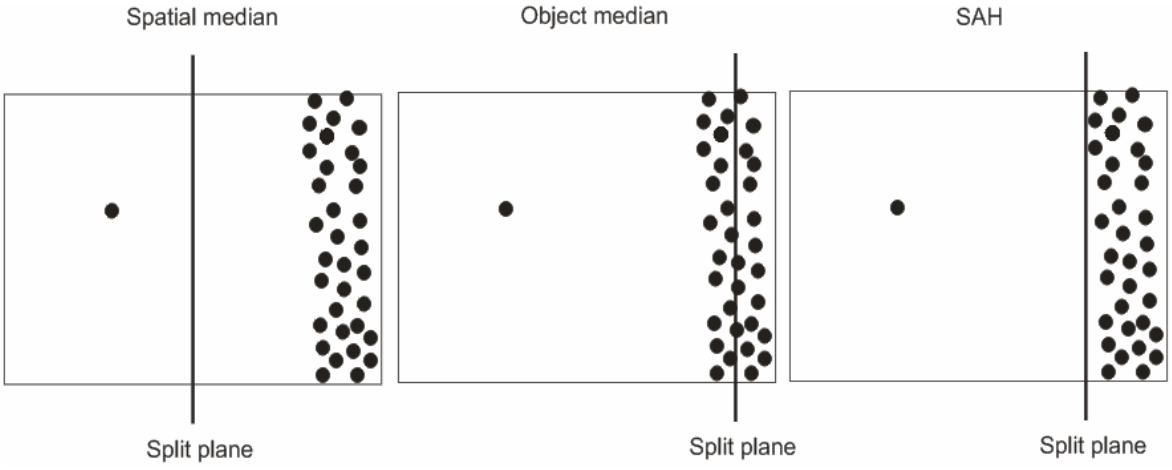
A_L – plocha naľavo od deliacej roviny

A_R – plocha napravo od deliacej roviny

N_L – počet objektov naľavo od deliacej roviny

N_R – počet objektov napravo od deliacej roviny

Takto sa postupuje pre všetky tri osi a nakoniec sa vyberie deliaca rovina s najnižšou cenou. Minimálna cena sa určí ako cena za vytvorenie listu stromu, vypočíta sa ako $C_{\min} = C_i \cdot N$ (N je celkový počet objektov). Ak neexistuje rovina s menšou cenou ako C_{min} tak je delenie zastavené a z uzla sa vytvorí list stromu. Vďaka tomu sa delenie priestoru zastaví ak sa nedá nájsť vhodná deliaca rovina [5].



Obr. 5: Umiestnenie deliacej roviny v závislosti na použitej metóde

2.3.2.2 Prechod KD-stromu

KD-strom sa pri vyhľadávaní priesečníku prechádza rekurzívne. Uzly ktoré je nutné testovať sa volia podľa polohy priesečníku lúča a deliacej roviny, podľa smeru lúča a podľa jeho počiatočného bodu. V závislosti na vzájomnej polohe lúča a deliacej roviny môžu nastať tri prípady: lúč pretína len ľavý uzol, lúč pretína len pravý uzol, lúč pretína obidva uzly. V prípade, že lúč pretína obidva uzly a v prvom sa nenachádza žiadny priesečník je nutné otestovať aj druhý [5].

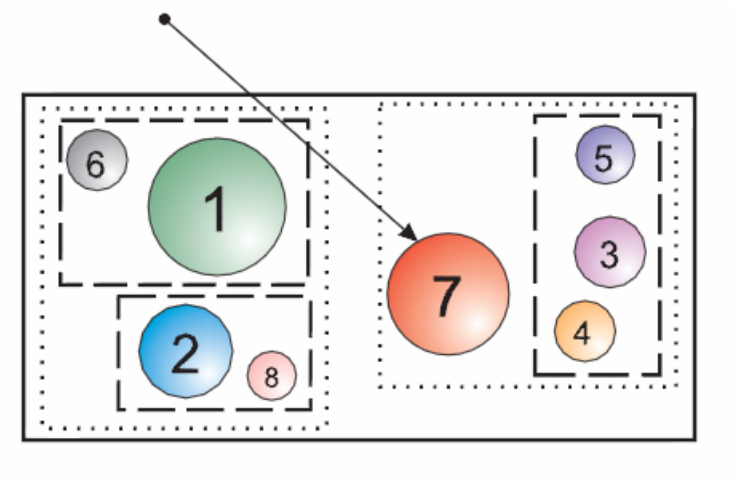
2.3.3 BVH

BVH je hierarchia obalových telies. Ak je pre každý objekt v scéne dané obalové teleso, je možné jednotlivé telesá hierarchicky spájať do skupín a vytvoriť z nich strom. Pri prechode stromu potom platí, že ak lúč nepretína obalové teleso, potom nepretína ani obalové telesá a objekty v ňom. Na rozdiel od kd-stromu, ktorý rekurzívne delí priestor rovinou na 2 časti, BVH rekurzívne rozdeľuje množinu objektov na menšie časti.

2.3.4 Delenie objektov

Pri tvorbe BVH existujú tri hlavné postupy- zhora nadol, zdola nahora a vkladaním. Pri postupe zhora nadol sa na začiatku vypočíta obalové teleso všetkých objektov a následne sa objekty rekurzívne delia na menšie skupiny. V delení sa pokračuje až kým obalové teleso neobsahuje len jeden objekt- list stromu. Naopak metóda zdola nahor začína tvorbu stromu od listov tak, že v každom obalovom telese je jeden objekt. Obalové telesá sa následne spájajú do skupín až kým sa nespoja všetky do jedného. Vkladacia metóda je online metóda- nevyžaduje aby boli známe všetky objekty pred tvorbou stromu, umožňuje ich neskôr pridávať. [17]

Tvorba stromu zhora nadol je najpoužívanejšia metóda kvôli svojej rýchlosti a jednoduchosti implementácie. Existuje viac spôsobov ako rozdeľovať objekty do skupín, od zvolenej metódy závisí kvalita vytvoreného stromu a teda aj rýchlosť vyhľadávania priesečníkov. Podobne ako u Kd-stromu je možné objekty rozdeľovať podľa priestorového alebo objektového mediánu a pomocou heuristiky SAH. [12]



Obr. 6: Rozdelenie objektov pri použití BVH [12]

2.3.5 Prechod BVH stromu

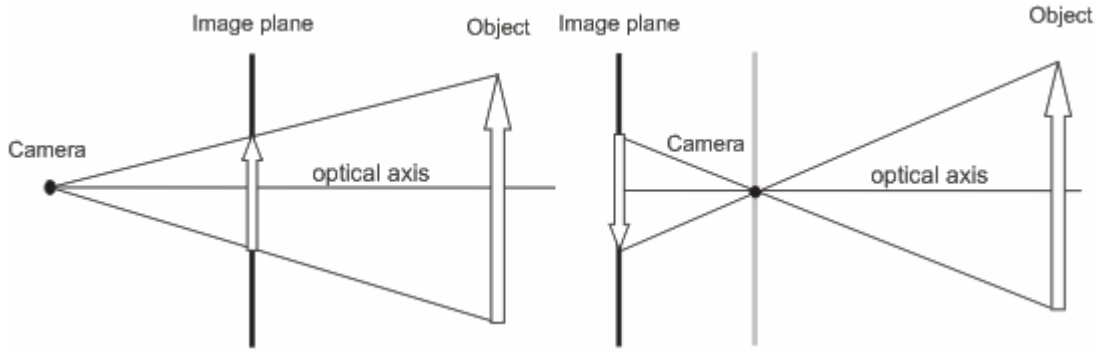
Prechod BVH stromu pri hľadaní priesečníku je veľmi jednoduchý, platí že ak lúč nepretína obalové teleso tak nepretína ani objekty v obalovom telese. Na priesečník sa teda testujú len tí potomkovia uzlu, ktorých AABB lúč pretína a postupuje sa rekurzívne.

3 MODEL KAMERY

Modely kamery popisujú, akým spôsobom sú objekty v 3D priestore vykreslené na výslednom 2D obrázku. Tento spôsob mapovania sa nazýva projekcia, podľa druhu projekcie rozlišujeme niekoľko typov kamier. Výhodou algoritmov, ktoré sú založené na sledovaní lúčov je, že k zmene projekcie stačí zmeniť spôsob, akým sa generujú primárne lúče.

3.1 Pinhole kamera

Pinhole kamera (camera obscura) je najjednoduchší model kamery, lúče svetla prechádzajú nekonečne malou štrbinou. Tento typ kamery modeluje perspektívnu projekciu, ale nie je s ním možné simulovať hĺbku ostrosti (DOF). Pinhole kamera je definovaná len dvoma parametrami- veľkosťou uhla projekcie a stredom projekcie. Pre každý bod v priestore prechádza stredom projekcie len jeden lúč a zobrazí sa práve v jednom bode na obrazovej rovine. To spôsobuje, že všetky objekty v scéne sú zaostrené. Perspektívna projekcia spôsobuje, že objekty rovnakej veľkosti umiestnené bližšie ku kamere sú zobrazené väčšie ako tie vo väčšej vzdialenosti od kamery. V reálnom modeli svetelné lúče prechádzajú otvorom a dopadajú na obrazovú rovinu, obraz je zrkadlovo obrátený. V matematickom modeli lúče vychádzajú z jedného bodu a následne prechádzajú obrazovou rovinou. [19]

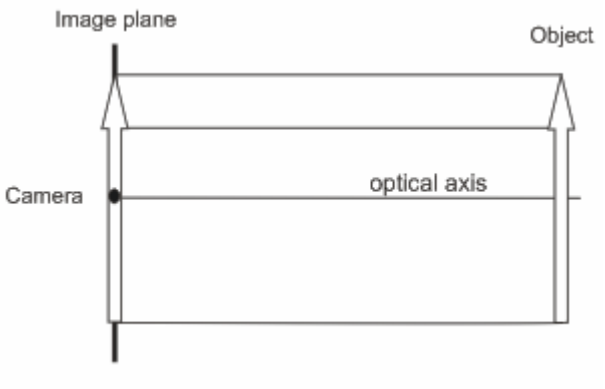


Obr. 7: Porovnanie matematického(vľavo) a reálneho modelu(vpravo) pinhole kamery

3.2 Orthografická kamera

Ortografická kamera využíva paralelnú projekciu, všetky lúče generované týmto typom kamery sú kolmé na projekčnú rovinu. Ortografická kamera neskresľuje veľkosť objektov so vzdialenosťou od kamery, zachováva rovnobežnosť priamok a relatívnu

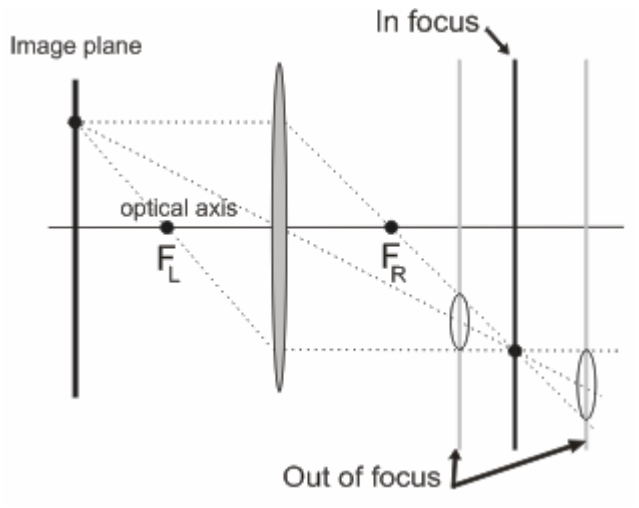
vzdialenosť objektov. Tento typ kamery je chodný najmä na vytváranie rôznych technických renderov (nárys, bokorys atď.).



Obr. 8: Ortografická projekcia

3.3 Thin lens kamera

Skutočné kamery zobrazujú 3D priestor cez sústavu šošoviek a majú mnoho parametrov, preto sa v počítačovej grafike využívajú zjednodušené matematické modely. Tento typ kamery modeluje kameru s tenkou šošovkou, napriek tomu, že je to jednoduchý model umožňuje dosiahnuť pomerne zaujímavé efekty. Thin lens kamera má navyše niekoľko parametrov: zaostrená vzdialenosť, ohnisková vzdialenosť.



Obr. 9: Do jedného bodu obrazovej roviny sa zobrazuje viac bodov priestoru

Na rozdiel od kamery pinhole má v skutočnosti štrbina určitú plochu a preto sa bod v priestore môže zobraziť na obrazovej rovine ako určitá plocha a naopak jeden bod obrazovej roviny môže prijímať svetlo z rôznych bodov v priestore. Tento fakt sa prejavuje ako rozostrenie objektov, ktoré neležia v rovine zaostrenia. Plocha, do ktorej sa body

v priestore zobrazujú na obrazovej rovine je zvyčajne kruhová a nazýva sa rozptylový kruh (Circle of confusion, CoC). Veľkosť CoC závisí napríklad na ohniskovej vzdialenosti alebo na vzdialenosti objektov od zaostrenej roviny. [10]

Lúč z kamery sa vytvára získaním lúča rovnako ako u pinhole kamery, pre tento lúč sa vypočíta priesečník s rovinou zaostrenia. Následne sa počiatočný bod lúča umiestni náhodne v CoC. Výsledný lúč prechádza získaným priesečníkom s rovinou zaostrenia. Aby bolo možné vypočítať veľkosť CoC je potrebné zistiť vzdialenosť obrazovej roviny od šošovky *dImage*. [10]

$$d\,\mathrm{Im}age = \frac{f \cdot t}{t - f} \tag{3.1}$$

kde:

f- ohnisková vzdialenosť šošovky

t- vzdialenosť priesečníku s rovinou zaostrenia

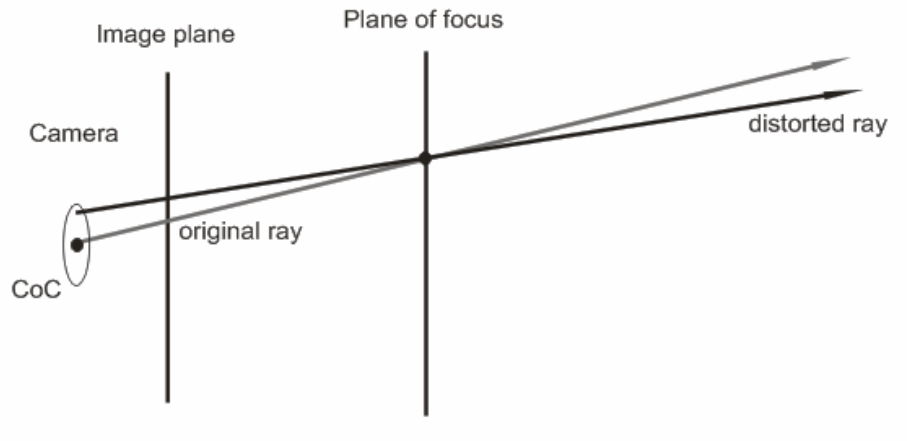
Veľkosť rozptylového kruhu *cocRad* sa následne určí ďalším vzťahom.

$$cocRad = apertureSize \cdot \frac{d\,\mathrm{Im}age - focalDist}{d\,\mathrm{Im}age} \tag{3.2}$$

kde:

apertureSize- clona kamery

focalDist- vzdialenosť roviny zaostrenia



Obr. 10: Generovanie lúča u thin lens kamery

4 MATERIÁLY

4.1 Sférické súradnice

Pri výpočtoch interakcií lúčov s objektami je nutné vyhodnocovať pri každom odraze od materiálu odchádzajúci lúč. V niektorých prípadoch je vhodnejšie na tento výpočet použiť namiesto kartézskych súradníc sférické.

4.1.1 Rovnomerné vzorkovanie pologule

Ak *r1* a *r2* sú náhodné čísla potom sférické súradnice sú dané [6]:

$$\varphi = 2 \cdot \pi \cdot r1 \tag{4.1}$$

$$\theta = a\cos(r2) \tag{4.2}$$

Kartézske súradnice:

$$x = \cos\varphi \cdot \sqrt{1 - r2^2} \tag{4.3}$$

$$y = \sin\varphi \cdot \sqrt{1 - r2^2} \tag{4.4}$$

$$z = r2 \tag{4.5}$$

PDF:
$$p = \frac{1}{2\pi} \tag{4.6}$$

4.1.2 Vzorkovanie vážené podľa cosínusu úhla

Ak *r1* a *r2* sú náhodné čísla potom sférické súradnice sú dané [6]:

$$\varphi = 2 \cdot \pi \cdot r1 \tag{4.7}$$

$$\theta = a\cos(\sqrt{r2}) \tag{4.8}$$

Kartézske súradnice:

$$x = \cos\varphi \cdot \sqrt{1 - r2} \tag{4.9}$$

$$y = \sin\varphi \cdot \sqrt{1 - r2} \tag{4.10}$$

$$z = \sqrt{r2} \tag{4.11}$$

PDF:
$$p = \frac{\cos\theta_o}{\pi} \tag{4.12}$$

4.2 BSDF

BSDF (bidirectional scattering distribution function) je matematická funkcia, ktorá popisuje spôsob, akým povrch rozptyluje svetlo. BSDF je zovšeobecnená verzia distribučnej funkcie, v praxi sa používajú BRDF(obojsmerná odrazová funkcia), BTDF(obojsmerná funkcia lomu) a BSSRDF(obojsmerná funkcia podpovrchového rozptylu). Všetky BxDF funkcie fungujú ako čierna skrinka, ako vstup preberajú dvojicu vektorov ω_i a ω_o a výstupom je hodnota, ktorá popisuje aká veľká časť svetla dopadajúceho zo smeru ω_i sa od povrchu odrazí do smeru ω_o. [2]

$$BRDF = \frac{L_0}{E_i} \tag{4.13}$$

Fyzikálne prijatelná BRDF musí mať nasledovné vlastnosti:

Vzájomnosť (reciprocity)- hodnota BRDF sa nemení ak sú vymenené smery ω_i a ω_o

$$f_r(x, \omega_i, \omega_o) = f_r(x, \omega_o, \omega_i) \tag{4.14}$$

Zachovanie energie (energy conservation)- celková energia odrazeného svetla je menšia alebo rovná energii prichádzajúceho svetla pre všetky smery ω.

$$\int_{s^2} f_r(x, \omega_o, \omega) \cos\theta d\omega \leq 1 \tag{4.15}$$

BTDF je možné definovať podobne ako BRDF s tým rozdielom, že u BTDF ω_i a ω_o nie sú v rovnakej pologuli nad povrchom a BTDF nemusí spĺňať podmienku vzájomnosti.

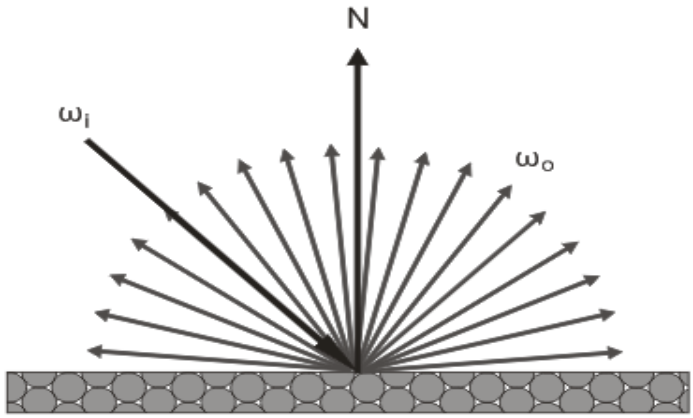
4.2.1 Difúzna BRDF

Difúzny odraz je odraz, ktorý nezávisí na smere pozorvania, smer odrazeného lúča je teda nezávislý od smeru prichádzajúceho lúča. Takáto odrazová funkcia je vhodná na modelovanie matných povrchov ako napr. papier, matné nátery atď.

Odrazové lúče je možné generovať rovnomerným vzorkovaním pologule nad povrchom. Problém je, že čím je väčší uhol medzi lúčom a normálou, tým menší príspevok bude lúč mať. Vhodnejšie je ak sú častejšie generované lúče okolo normály pretože ich príspevok je väčší. Takéto rozloženie lúčov je možné dosiahnuť vzorkovaním váženým podľa cosínusu úhla medzi normálov a lúčom. [2]

BRDF: $$f = \frac{1}{\pi} \tag{4.16}$$

PDF: podľa použitej metódy vzorkovania pologule



Obr. 11: Schématické znázornenie difúznej BRDF

4.2.2 Oren-Nayar

Oren-Nayar je model, ktorý slúži na simulovanie difúzneho odrazu u materiálov s určitou drsnosťou. Drsnosť materiálu určuje parameter *sigma.* Oren-Nayar je vhodný na modelovanie materiálov ako je napríklad betón alebo hlina. [20]

BRDF: $$f = \frac{1}{\pi} \cdot \left(A + B \cdot \max(0, \cos(\phi_i - \phi_r)) \cdot \sin\alpha \cdot \tan\beta\right) \tag{4.17}$$

PDF: podľa použitej metódy vzorkovania pologule

$$A = 1 - 0.5 \cdot \frac{\sigma^2}{\sigma^2 + 0.33} \tag{4.18}$$

$$B = 0.45 \cdot \frac{\sigma^2}{\sigma^2 + 0.09} \tag{4.19}$$

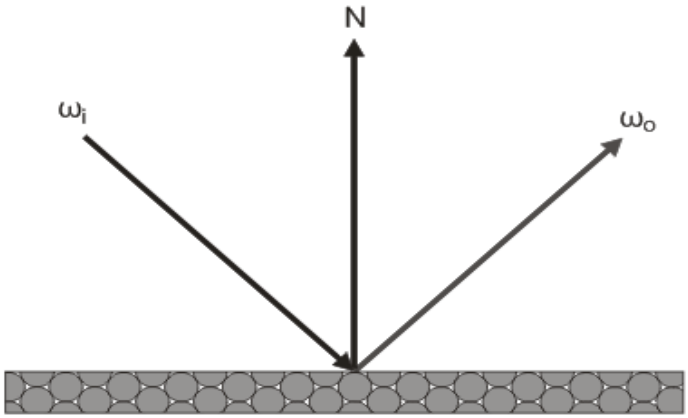
4.2.3 Specular

Pojmom specular sa označuje dokonalý odraz, pri tomto type odrazu platí, že uhol ktorý zviera odrazený lúč s normálou povrchu je rovný uhlu, ktorý zviera s normálou prichádzajúci lúč a pre každý prichádzajúci lúč existuje práve jeden odrazený lúč. Pre odrazený lúč platí vzťah 4.20.

$$\omega_o = \omega_i + (-2 \cdot n.\omega_i) \cdot n \tag{4.20}$$

BRDF: $$f = 1 \tag{4.21}$$

PDF: $$f = \cos\theta_o \tag{4.22}$$



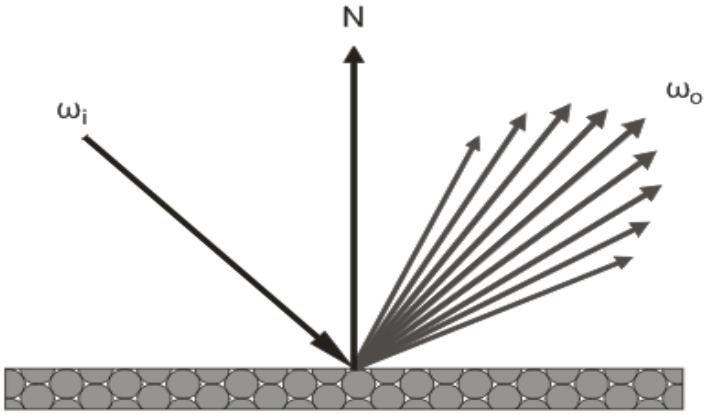
Obr. 12: Schématické znázornenie zrkadlovej BRDF

4.2.4 Blinn-Phong

V roku 1973 Phong predstavil nový empirický model osvetlenia, Blinn ho neskôr upravil použitím tzv. *halfway* vektora. Halfway vektor je vektor, ktorý vznikne sčítaním prichádzajúceho a odchádzajúceho lúča. Tento model napodobuje tzv. glossy materiál-materiál, ktorý má určitú drsnosť a odrazené lúče sa šíria okolo vektoru perfektného odrazu. Drsnosť materiálu je určená hodnotou exponentu. Pre hodnotu exponentu 0 je typ odrazu podobný difúznemu odrazu, ak sa hodnota exponentu blíži k nekonečnu odraz sa stáva zrkadlovým.[16]

BRDF:
$$f = \cos^{\exp} \theta_h \tag{4.23}$$

PDF:
$$p = \cos^{\exp} \theta_h \tag{4.24}$$



Obr. 13: Schématické znázornenie Phongovej BRDF

4.3 Textúrovanie

Textúrovanie je spôsob pridávania detajlov na geometriu. Podľa spôsobu ako sa textúra získa rozlišujeme procedurálne textúry a rastrové textúry. Procedurálne textúry sú vyjadrené matematickou funkciou, ktorej funkčná hodnota nejakým spôsobom závisí na polohe v 3D priestore. Výhodou procedurálnych textúr je, že pri zväčšovaní/zmenšovaní nestrácajú kvalitu. Nevýhodou je, že matematicky je možné vyjadriť len veľmi malé množstvo materiálov, preto v praxi tento druh textúr nemá veľké uplatnenie. Princíp rastrových textúr spočíva v aplikácii 2D obrázkov na 3D modely. Výhoda rastrových textúr spočíva v tom, že je možné použiť akýkoľvek obrázok/fotografiu.[9] Textúrovanie umožňuje pridať na modely detajly bez toho aby musela byť upravená geometria modelu a umožňuje jednoducho zvýšiť ich realističnosť. Obrázok namapovaný na objekt ako textúra môže ovplyvňovať okrem farby (diffuse mapa) objektu aj iné vlastnosti ako napr. priehľadnosť(alpha mapa), kombináciu materiálov, vlastnosti BRDF materiálu(napr. specular mapa), normály porchu(bump mapa) atď.[2]

4.3.1 Bump mapping

Bump mapping je technika, ktorá využíva textúru- výškovú mapu na úpravu normál povrchu. Vďaka tomu je možné na povrch pridávať rôzne detajly vo forme výstupkov alebo jamiek bez úpravy modelu.

4.3.2 Mapovanie textúr

Spôsob, akým sa textúry nanášajú na 3D objekty sa nazýva mapovanie. Existuje veľa spôsobov mapovania textúr, napr. UV-mapovanie, sférické, cylindrické, ploché atď.

4.3.2.1 UV-mapovanie

UV mapovanie je najpoužívanejší spôsob nanášania textúr na 3D objekty. Na výpočet súradníc pixelu sa využívajú parametre *u* a *v* povrchu, ktoré sa len prevedú do súradníc textúry. Pre každý vrchol geometrie je nutné vytvoriť vopred UV súradnice namapovaním textúry na model.

4.3.2.2 Sférické mapovanie

Sférické mapovanie vytvára okolo objektu guľu, na ktorú sa premietaju body na objekte. Body sa na guľu premietajú pozdĺž vektora zo stredu gule do daného bodu. Takéto mapovanie sa často využíva pre sférické textúry pozadia.

4.4 Kombinované materiály

Skutočné materiály môžu pozostávať z viacerých komponentov, materiály môžu zároveň kombinovať viaceré druhy odrazov. Inou možnosťou kombinácie je textúra, textúra môže meniť nielen farbu ale aj typ odrazu materiálu. Možností ako skombinovať dva alebo viacero materiálov je samozrejme mnoho.

5 ZDROJE SVETLA

Aby bolo možné na renderovanom obrázku vôbec niečo vidieť musí byť v scéne umiestnený nejaký zdroj svetla. Nasvietenie scén je možné vytvoriť rôznymi typmi svetiel a ich kombináciami. Pre potreby počítačovej grafiky vzniklo množstvo svetelných modelov, ale len niektoré je možné použiť vo fyzikálnom rendereri.

5.1 Bodové svetlá

Bodové svetlo (point light) je najjednoduchší typ svetla. Bodové svetlo je definované bodom v priestore, z ktorého sa svetlo rovnomerne šíri do všetkých smerov. Jedinými parametrami bodového svetla sú umiestnenie v priestore, farba a intenzita. Typickým rysom bodového svetla sú ostré tiene. Bodové svetlo nemá žiadny rozmer a plochu, preto jeho použitie vo fyzikálnom rendereri vyžaduje určité úpravy. Bodové svetlá je možné použiť len ak algoritmus pre výpočet osvetlenia počíta priame osvetlenie vzorkovaním svetiel, inak je pravdepodobnosť, že lúč zasiahne bodové svetlo 0. [2]

5.2 Plošné svetlá

Plošné svetlá (area light) sú definované nejakým geometrickým útvarom (väčšinou štvorec alebo kruh), ktorý vyžaruje svetlo. Na rozdiel od bodových svetiel majú určitú plochu a môžu vytvárať mäkké tiene, mäkkosť tieňov závisí od veľkosti plochy svetla. Keďže sa svetlo z povrchu plošného svetla šíri v rôznych smeroch je pri vzorkovaní takýchto svetiel vhodné použiť výpočet integrálu nad plochou. [2]

5.3 Mesh light

Mesh light je špeciálny typ plošného svetla, jeho tvar je definovaný ľubovoľným 3D modelom, umožňuje preto vytvoriť zložitejšie a zaujímavejšie nasvietenie scény. Keďže nie je problém vymodelovať štvorec alebo kruh a použiť tento model ako svetlo, môže tento typ svetla úplne nahradiť plošné svetlá.

5.4 Smerové svetlá

Smerové svetlo je typom svetla pre ktoré platí, že do každého bodu priestoru prichádza osvetlenie z rovnakého smeru. Smerové svetlo sa dá chápať ako bodové svetlo umiestnené v nekonečne. Čím viac sa bodové svetlo vzdaľuje, tým viac sa chová ako

smerové svetlo. Takýto typ svetla je vhodný napríklad na vytvorenie osvetlenia podobného slnku. [2]

5.5 Nekonečne veľké plošné svetlá

Často používaným typom svetla je nekonečné plošné svetlo, využíva sa najmä na nasvecovanie scény pozadím (enviroment lighting). Takéto svetlo sa dá chápať ako obrovská guľa, ktorá obaľuje celú scénu a osvetľuje ju zo všetkých smerov. Okrem svetla jednej farby je tiež možné použiť textúru pozadia (enviroment map) a simulovať tak reálne nasvietenie 3D objektov. Na mapovanie textúry pozadia sa využívajú sférické súradnice.[2]

6 RIEŠENIE ZOBRAZOVACEJ ROVNICE

6.1 Riešenie pomocou metódy Monte Carlo

Metóda Monte Carlo je vhodná hlavne na riešenie viacrozmerných integrálov, pre ktoré neexistujú vhodné numerické metódy. Monte Carlo konverguje rýchlosťou $O(N^{-1/2})$ bez ohľadu na rozmer a hladkosť integrandu. Vďaka tomu je táto metóda vhodná na použitie v grafike, kde sa často vyskytujú mnohorozmerné integrály nespojitých funkcií ako je napríklad zobrazovacia rovnica. Výhodou integrácie metódou Monte Carlo je jej jednoduchosť, na výpočet stačí vytvoriť vzorky a spriemerovať hodnoty funkcie pre jednotlivé vzorky. [1]

Princíp metódy Monte Carlo je výpočet hodnoty integrálu funkcie f náhodným vzorkovaním. Pre N náhodných vzorkov $X_1...X_N$ s hustotou pravdepodobnosti rozloženia $p(X_i)$ je potom odhad integrálu F_N daný vzťahom:

$$F_N = \frac{1}{N} \cdot \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{6.1}$$

6.2 Ruská ruleta

Ruská ruleta sa v path tracingu využíva na ukončovanie ciest. Prinincíp spočíva v tom, že ak zvolíme pravdepodobnosť ukončenia cesty p potom pravdepodobnosť toho, že cesta nebude ukončená je $(1-p)$. Ak bude cesta pokračovať bude mať vačšiu váhu, napr. $L_r' = L_r / (1-p)$. Výhoda ruskej rulety je, že aj keď sa pri výpočte použije menšie množstvo odrazov/ciest, výpočet povedie k správnemu výsledku. Pravidlo, podľa ktorého sa cesty budú ukončovať je možno zvoliť ľubovoľné. Cesty možno náhodne ukončovať napríklad s pevne stanovenou pravdepodobnosťou alebo v závislosti na množstve odrazeného svetla. Na použitom pravidle záleží kvalita výpočtu, nevhodne zvolené pravidlo môže príliž zvyšovať rozptyl. [2]

$$v' = \begin{cases} v / p & \xi < p \\ 0 & else \end{cases} \tag{6.2}$$

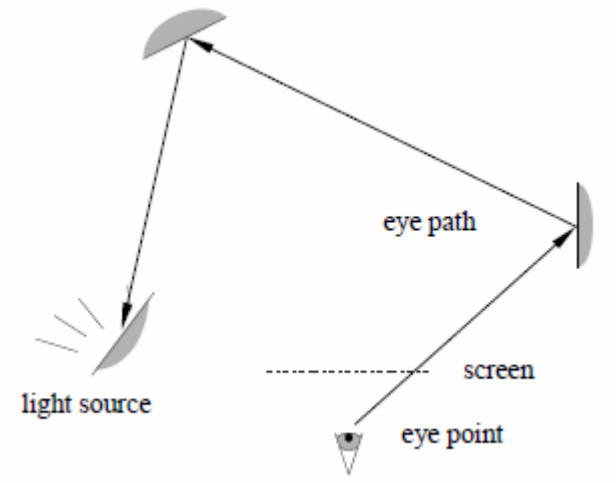
Odhadovaná hodnota:

$$(1-p) \cdot 0 + p \cdot v / p = v \tag{6.3}$$

6.3 Path tracing

Path tracing je algoritmus na výpočet globálneho osvetlenia, ktorý predstavil v roku 1986 James Kajiya. Path tracing sa využíva na vyhodnotenie zobrazovacej rovnice pomocou metódy Monte-Carlo. Kajiya v podstate rozšíril algoritmus distributed ray tracing o výpočet osvetlenia pomocou viacnásobných difúznych odrazov, ktoré môžu značne prispieť k realistickosti výsledného obrazu. Hlavnými výhodami path tracingu je, že pomocou neho možno vytvoriť s pomerne malým úsilím fotorealistické obrázky a tiež jednoduchosť implementácie tohto algoritmu.. Hlavné nevýhody sú pomalá konvergencia a nutnosť fyzikálne správnej reprezentácie geometrie, svetiel a materiálov. Inak by riešenie zobrazovacej rovnice viedlo k správnemu riešeniu nespávneho problému. Vysokofrekvenčný šum, ktorý vzniká pri výpočte je viditeľný aj po niekoľko tisíc vzorkoch na pixel.

Pri path tracingu sú lúče generované z kamery a „vystrelované“ smerom do scény. Lúče sú teda sledované v opačnom smere ako sa v skutočnosti šíria- od pozorovateľa smerom k svetlu. Problém tohto prístupu sú malé svetlá. Ak sa v scéne nachádzajú malé svetlá s veľkou intenzitou, pravdepodobnosť že náhodný lúč svetlo zasiahne je nízka a preto sa zvyšuje rozptyl výsledku. [3]



Obr. 14: Schématické znázornenie path tracingu [3]

Lúče svetla sa v priestore nekonečne veľa krát odrážajú, keďže s počtom odrazov klesá energia, ktorú jednotlivé lúče nesú, klesá aj ich príspevok do výsledného riešenia. Navyše so zvyšujúcov sa hĺbkou, do ktorej lúče sledujeme rastie tiež náročnosť výpočtu. V praxi sa preto využíva parameter maximálna dĺžka cesty. Keď lúč dosiahne určitej

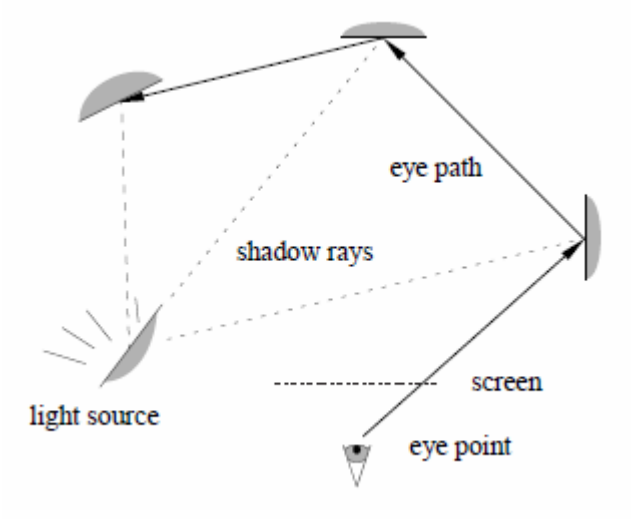
dĺžky, predpokladáme, že jeho príspevok je zanedbateľný. Výpočet sa ukončí a generuje sa nový lúč. Takýto prístup ale narúša správnosť výsledku a zavádza do odhadu systematickú chybu. Tento problém pomáha riešiť ruská ruleta. Ruská ruleta zavádza do ukončovania ciest náhodnosť, cesty ukončuje s určitou pravdepodobnosťou a tým zaručí, že odhad zostáva nestranný.

```
1. Vygeneruj lúč z kamery. pathThroughput = 1.
2. Nájdi najbližší priesečník lúča so scénov.
3. if( materiál je emitter )
       return pathThroughput*material->Le
   else
       pathThroughput*=material->reflectance
       vygeneruj nový lúč podľa BRDF materiálu
       GOTO 2.
```

Ukážka kódu 1: Jednoduchý algoritmus path tracingu

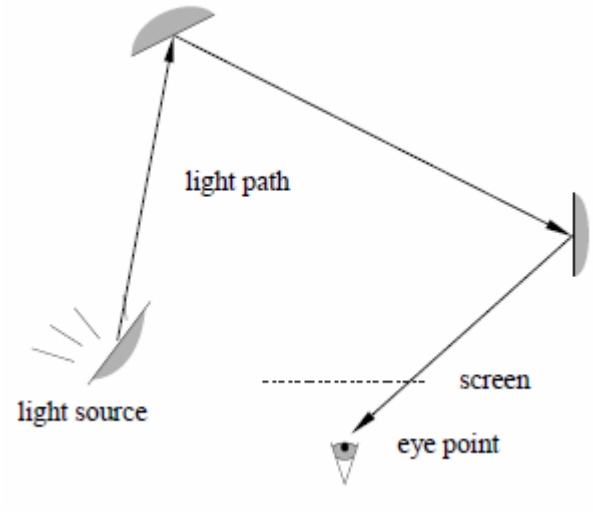
Základnú verziu path tracingu je možné rozšíriť o výpočet priameho osvetlenia, tejto metóde sa tiež hovorí odhad nasledujúcej udalosti (next event estimation). Na rozdiel od základného algoritmu sa v tomto prípade pri sledovaní ciest nečaká na to, že lúč náhodne zasiahne svetlo. V každom kroku sa vzorkuje jeden alebo viac náhodných bodov na svetlách a so súčasným uzlom cesty sa prepojí pomocou tieňových lúčov. Táto metóda rozkladá výpočet odrazeného svetla na výpočet priameho a výpočet nepriameho osvetlenia (vzťah 6.4). Vzorkovanie svetiel umožňuje použiť na osvetlenie aj malé aj bodové svetlá bez toho aby sa zvýšil rozptyl výpčtu. [3]

$$L_r(x,\omega_o) = L_d(x,\omega_o) + L_i(x,\omega_o) \tag{6.4}$$



Obr. 15: Schématické znázornenie path tracingu s výpočtom priameho osvetlenia [3]

6.4 Light tracing

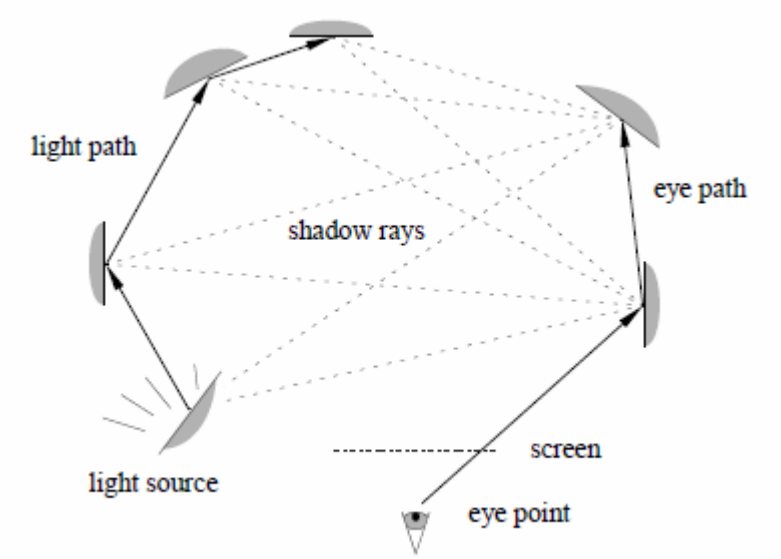


Obr. 16: Schématické znázornenie light tracingu [3]

Light tracing je obdobou path tracingu s tým rozdielom, že lúče nie sú generované na kamere ale na zdrojoch svetla. Zo zdrojov svetla sa lúče šíria scénou a ak zasiahnu kameru, ich príspevok sa zaznamená na príslušný pixel. Rovnako ako je path tracing nevhodný pre scény s malými svetelnými zdrojmi je light tracing nevhodný pre scény s malou kamerou. Len minimum z vygenerovaných lúčov zasiahne nejaký pixel na obrazovej rovine. Ak je použitá kamera pinhole s nekonečne malým otvorom tak ju nezasiahne dokonca žiadny lúč. Rovnako ako pri path tracingu je možné použiť metódu odhadu nasledujúcej udalosti a prepájať vrcholy cesty s kamerou priebežne. Light tracing je vhodný hlavne na

renderovanie scén s malými svetlami a efektívne vzorkuje cesty typu LSS*DE ktoré vytvárajú tzv. kaustiky. [3]

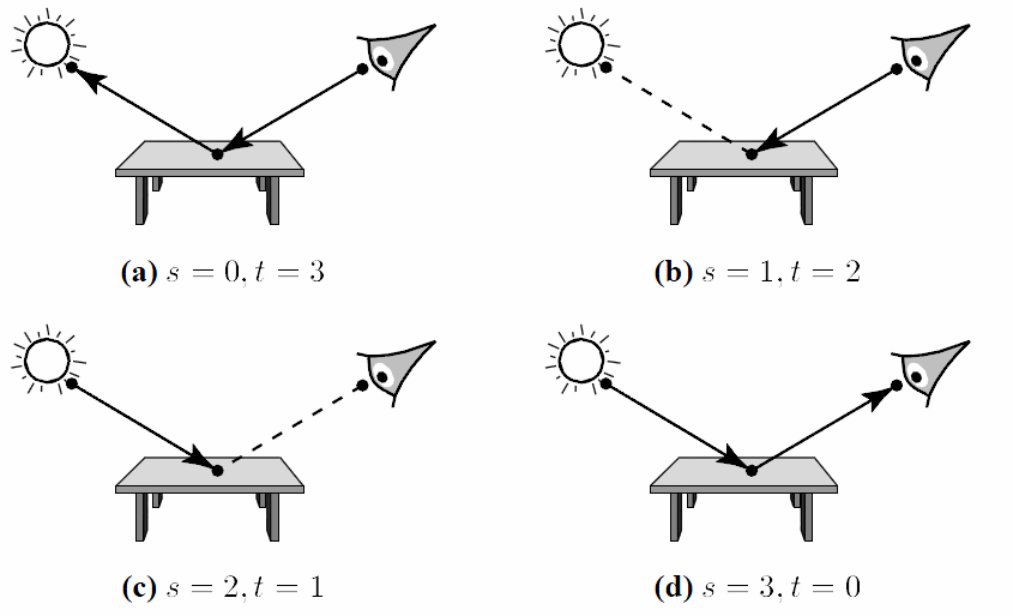
6.5 Obojsmerný path tracing



Obr. 17: Schématické znázornenie obojsmerného path tracingu [3]

Obojsmerný path tracing vzniká spojením path tracingu a light tracingu a spája tak výhody oboch algoritmov. Obojsmerný path tracing dokáže riešiť problémy nepriameho osvetlenia efektívnejšie a robustnejšie ako obyčajný path tracing. Na výpočet vzorku sa generujú dve cesty- jedna cesta z kamery a druhá cesta zo svetla. Pre každú cestu sa ukladajú jej vrcholy a následne sa ich kombináciou vytvárajú nové cesty. Na vytvorenie novej cesty tak stačí test viditeľnosti medzi jej dvoma koncovými bodmi. Takýmto spájaním vznikajú rôzne typy ciest, ktoré sú vhodné pre efektívne vzorkovanie rôznych druhov osvetlenia. [1]

Kombinovaním svetelných ciest a ciest z kamery rôznych dĺžok vzniká celá rodina vzorkovacích techník. Každá z týchto techník je vhodná na efektívne vzorkovanie určitej množiny svetelných efektov. Každá technika vytvára cestu určitej dĺžky k. Celkovú dĺžku cesty k určíme ako $k = s+t-1$, kde s je počet vrcholov svetelnej cesty a t je počet vrcholov cesty z kamery. Pre každú dĺžku cesty k existuje $k+2$ vzorkovacích techník (Obr. 18). [1]



Obr. 18: Možné spôsoby vytvorenia cesty s dĺžkou k=2 [1]

Každej vytvorenej ceste je nutné priradiť určitú váhu tak aby súčet váh pre každú dĺžku cesty *k* bol rovný 1. Najjednoduchším spôsobom výpočtu váh je uniformný výpočet, ktorý priradí každej vzorkovacej technike s dĺžkou *k* rovnakú váhu. Pre *k=2* (Obr. 18) dostávame teda váhu *w = 1/(k+2) = 0.25*. Výpočet váh je nutné upraviť podľa toho, aké vzorkovacie techniky pre určitú dĺžku *k* uvažujeme. Napríklad ak pri výpočte zanedbávame techniku *s=3, t=0* (Obr. 18 (d)) budú výsledné váhy *w = 1/(k+1) = 0.33.*

```
1. eyePath = Sleduj cestu z kamery
2. lightPath = Sleduj cestu zo svetla
3. for each( eV in eyePath )
      for each( lV in lightPath )
          if( visible(eV, lV) )
               w = weight(eV, lV)
               contribution += (eyePathThroughput + lightPathThroughput)*w
```

Ukážka kódu 2: Algoritmus BDPT

6.6 Metropolis light transport

Algoritmus MetropolisLight Transport je inšpirovaný metódou vzorkovania Metropolis vyvinutou v 50-tych rokoch. MLT je aplikáciou vzorkovania Metropolis na problém šírenia svetla, vo svojej dizertačnej práci ho predstavil Eric Veach. Tento algoritmus pri renderovaní obrázku k vytvoreniu nových svetelných ciest využíva mutácie predchádzajúcich svetelných ciest. Každá mutácia môže byť s určitou pravdepodobnosťou prijatá alebo zamietnutá, aby sa zaručilo že cesty budú vzorkované podľa toho aký príspevok majú do obrázku. Vďaka tomu je algoritmus schopný účinne riešiť aj problémy, ktoré sú pre predchádzajúce algoritmy ťažko riešiteľné. Ak jé nájdená dôležitá svetelná cesta tak je možné formou mutácií dôkladne preskúmať jej okolie v priestore.. MLT tak dokáže efektívne riešiť napr. silné nepriame osveletlenie, svetlo prechádzajúce malým otvorom v geometrii, kaustiky atď. [1]

Pre každú mutáciu je nutné vypočítať pravdepodobnosť prijatia *a* podľa vzťahu (6.5), kde *T* je prechodová funkcia pokusu. *T(x->y)* určuje pravdepodobnosť prechodu zo vzorku *x* do pokusného vzorku *y*. Pokusný vzorok *y* je následne prijatý alebo zamietnutý podľa pravdepodobnosti *a*. Podrobný popis výpočtu pravdepodobnosti prijatia pre rôzne typy mutácií je možné nájsť v [1].

$$a(x \to y) = \min\left\{1, \frac{f(y) \cdot T(y \to x)}{f(x) \cdot T(x \to y)}\right\} \tag{6.5}$$

Obojsmerná mutácia- Mení cestu vymazaním jej časti a následným vygenerovaním novej časti. Počet aj pozícia vymazaných a pridaných vrcholov sa určuje náhodne. Mutácie, ktoré spôsobujú menšiu zmenu svetelnej cesty majú väčšiu pravdepodobnosť prijatia. Preto je vhodné prideliť pravdepodobnosti na zmazanie a doplnenie vrcholov tak, aby dochádzalo častejšie k menším zmenám. [1]

Perturbácia kaustiky- Používa sa ak cesta pozostáva z jedného alebo viac zrkadlových odrazov od svetla a difúzneho odrazu pred kamerou (napr. LSDE). Mutácia spočíva v miernom pozmenení smeru prvého lúča zo svetla. [1]

Perturbácia šošovky- Podobne ako perturbácia kaustiky ale pre cesty so zrkadlovými odrazmi od kamery (napr. LDSE). V tomto prípade sa mení smer lúča vychádzajúceho z kamery. [1]

```
Vygeneruj náhodnú vzorku x
for(počet mutácií)
    y = mutate(x)
    acceptance = f(y)/f(x) * T(y->x)/T(x->y)
    if(random()<acceptance)
        x = y
    record(x)
```

Ukážka kódu 3: Algoritmus vzorkovania metropolis

6.6.1 Jednoduchá a robustná stratégia mutácií pre MLT

V roku 2001 predstavili Kelemen a Szirmay-Kalos novú stratégiu mutácií pre algoritmus Metropolis Light Transport Stratégia pracuje v priestore náhodných čísel, podľa ktorých boli vytvorené svetelné cesty. Namiesto mutácií ciest v priestore ako v pôvodnom algoritme, ktorý predstavil Veach, je možné aplikovať mutácie na náhodné čísla, podľa ktorých sú cesty vytvorené. Zmutované hodnoty náhodných čísel sa následne prevedú späť do priestoru ciest sledovaním cesty zobrazovacím algoritmom.[21]

Veľkou výhodou tohoto prístupu je jednoduchosť a ľahká implementácia. Stačí upraviť generátor náhodných čísel tak aby podporoval mutácie. Výhodou je tiež, že algoritmus vzorkovania je možné použiť na už existujúci zobrazovací algoritmus. Pravdepodobnosť prijatia *a* sa počíta jednoduchšie ako u pôvodnej metódy (vzťah 6.6). Kde *f(x)* a *f(y)* je intenzita svetla prichádzajúceho cestou.

$$a(x \to y) = \min\left\{1, \frac{f(y)}{f(x)}\right\} \tag{6.6}$$

Nevýhodou tohoto prístupu oproti pôvodnému algoritmu je, že po mutácii je vždy nutné celú cestu generovať odznova. Mutácie ktoré predstavil Veach menia len časť cesty a sú teda menej časovo náročné.

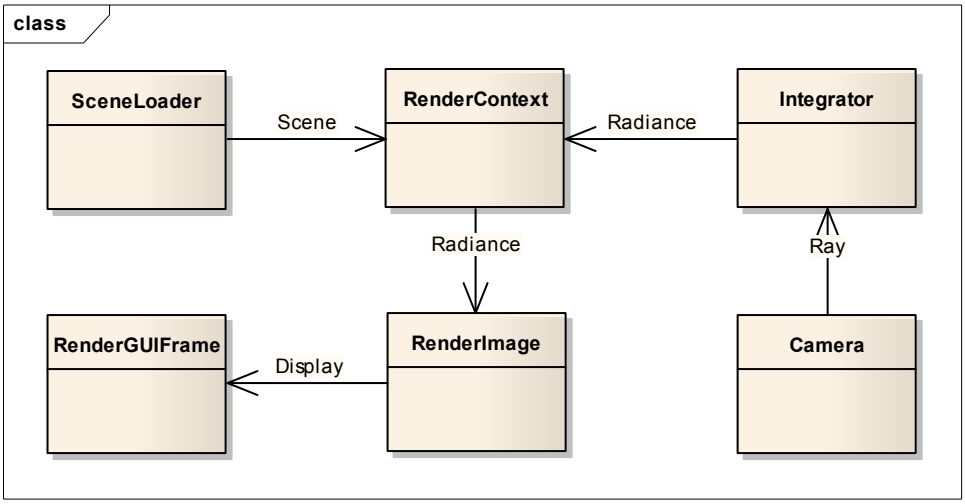
II. PRAKTICKÁ ČASŤ

7 DESIGN A POŽIADAVKY

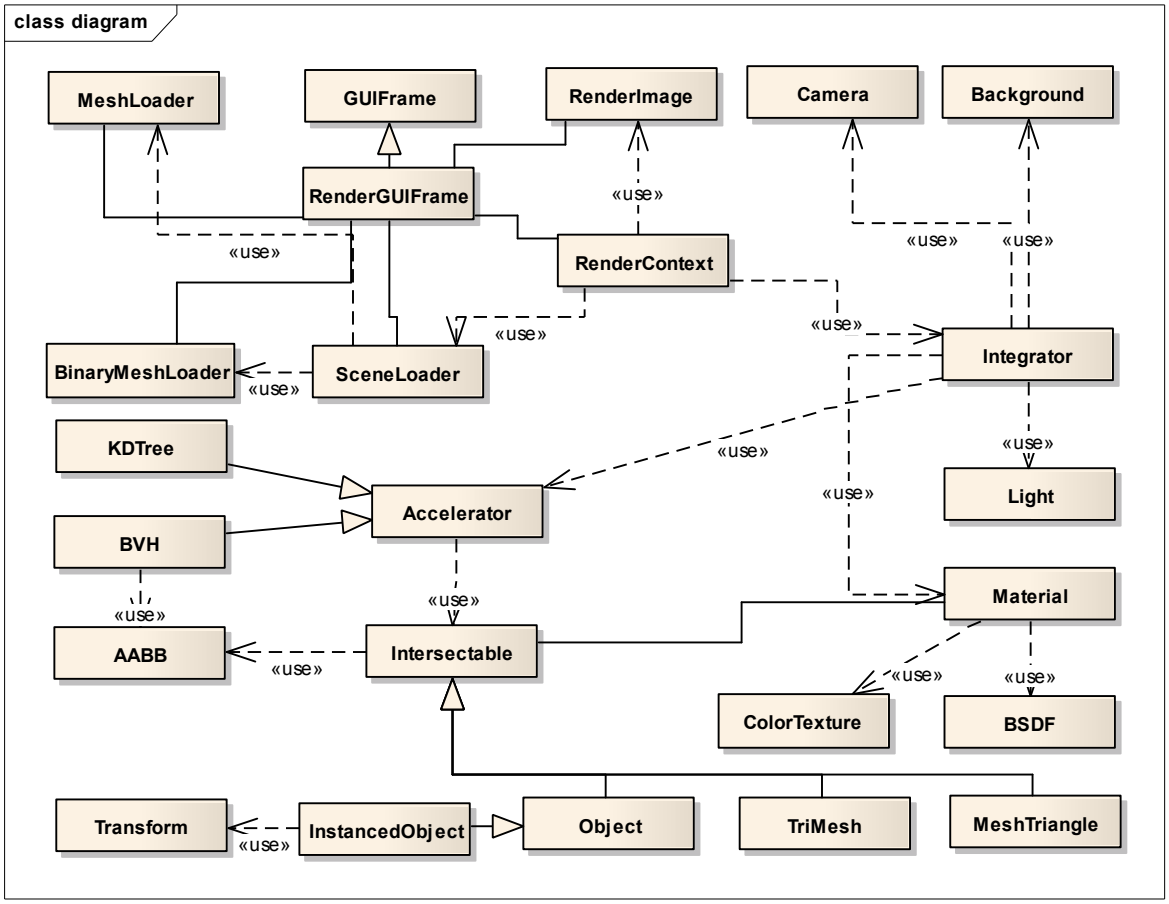
Hlavnými požiadavkami pre program boli rýchlosť a rozšíriteľnosť, preto bol ako programovací jazyk pre realizáciu použitý jazyk C++. Program bol vytvorený systémom abstraktných tried, ktorý umožňuje jednoducho pridávať novú funkcionalitu. Štruktúra programu bola čiastočne ovplyvnená návrhom PBRT [2]. Hlavnou triedou rendereru je trieda *RenderContext*. Na načítanie scény slúži pomocná trieda *SceneLoader* a na zobrazenie výsledkov renderu slúži trieda *RenderGUIFrame*.

Renderovanie obrázku prebieha v renderovacej smyčke. Renderovacia smyčka cyklicky generuje súradnice pixelu na obrazovej rovine. Pre súradnice pixelu následne kamera vygenreuje lúč a tento lúč je vystrelený do scény. Integrátor pre každý lúč spočíta niekoľko odrazov a do renderovacej smyčky vráti množstvo svetla, ktoré prichádza vygenerovanou cestou. Vypočítaný príspevok sa následne zaznamená na pixel buffer.

Podľa toho akým spôsobom sa generujú vzorky na obrazovej rovine je možné rozlíšiť renderovanie po blokoch a progresívne renderovanie. Pri renderovaní po blokoch sa postupne renderujú časti obrázku s nastaveným počtom vzorkov. Pri progresívnom renderovaní sa naraz generujú vzorky pre celý obrázok a výpočet sa postupne spresňuje.



Obr. 19: Diagram vzťahov medzi vybranými triedami



Obr. 20: Diagram tried

8 GEOMETRIA A TRANSFORMÁCIE

Aby bolo možné vkladať do scény rôzne objekty a tvary je nutné vytvoriť pre ne vhodnú reprezentáciu. Pre tento účel bola vytvorená abstraktná trieda *Intersectable*, táto trieda určuje jednotné rozhranie pre všetky objekty scény. Základné funkcie triedy *Intersectable* sú *intersect()* a *getBounds()*, ktoré vyhodnotia priesečník objektu s lúčom resp. vypočítajú jeho AABB. Takéto jednotné rozhranie umožňuje jednoducho pridať nové geometrické tvary a typy objektov, prípadne umožňuje spájať viacero tvarov do jedného.

Trieda *Object* je potomkom *Intersectable*. Slúži na vytváranie objektov, ktoré využívajú mailboxing. Geometrickú informáciu uchováva vo forme ukazateľa na nejaký objekt typu *Intersectable*. Rozšírením triedy *Object* je trieda *InstancedObject*, táto trieda uchováva navyše transformáciu. Transformácia slúži na prevedenie lúča z globálnych súradníc do súradníc objektu a umožňuje tak viacnásobné využitie jednej geometrickej informácie na vytvorenie viacerých objektov. Inštancované objekty majú využitie hlavne u scén, v ktorých sa viackrát vyskytujú rovnaké objekty na rôznych miestach v priestore, resp. s rôznymi transformáciami. Vytvorením inštancií je možné znížiť nároky na pamäť.

8.1 Vektory

Vektor reprezentuje smer a dĺžku v priestore, počet zložiek vektoru závisí na počte dimenzií priestoru. Preto definujeme tri typy vektorov:

Vec2f- vektor s dvoma zložkami *x* a *y*, ktorý slúži hlavne na prácu s UV súradnicami pri mapovaní textúr

Vec3f- „klasický“ vektor s tromi zložkami *x*, *y* a *z*

Vec4f- vektor definovaný ako union float[4] a __m128, datového typu, ktorý využíva SSE

Matematické operácie s vektormi sa vykonávajú s jednotlivými zložkami vektorov, tieto operácie sú definované ako operátory. Operátory umožňujú pracovať s vektormi rovnako ako so štandardnými datovými typmi v C++ a udržujú kód lepšie čitateľný.

8.2 Body, normály, farby

Body na rozdiel od vektorov nereprezentujú smer ale umiestnenie v priestore. Z toho vyplýva, že niektoré operácie, ktoré boli definované pre vektor nemajú pre body zmysel.

Taktiež výpočet transformácie bodov a vektorov je rozdielny. Rovnako ako vektory sú ale body definované tromi zložkami *x, y, z*.

Normála je špeciálnym typom vektoru, je to vektor kolmý na povrch v určitom jeho bode. Všetky normály musia byť normalizované, viaceré časti systému pracujú s týmto predpokladom. Hlavný rozdiel medzi vektormi a normálami je v spôsobe transformácie- pri zmene merítka modelu.

Farby sú ďalším dátovým typom, ktorý môže byť reprezentovaný pomocou trojprvkového vektoru(RGB) alebo štvorprvkového vektoru(RGBA) .

Tieto rozdiely je možné riešiť dvoma spôsobmi. Jeden spôsob je zavedenie nových datových typov pre body, normály a farby, čo zaručí ich striktné rozlíšenie. Na druhú stranu toto riešenie prinesie určité množstvo redundantného kódu. Druhou možnosťou je použitie triedy pre vektor za predpokladu, že programátor dokáže tieto typy rozlíšiť. Keďže druhý spôsob čiastočne zjednoduší implementáciu a rozlíšenie medzi spomínanými typmi je triviálne bude v našom programe použitá jediná trieda VecXf pre vektory, body, normály aj farby.

8.3 Lúče

Lúč je polpriamka, je definovaný počiatočným bodom *o* a smerovým vektorom *d*. Okrem toho má každý lúč svoje jedinečné *ID* kvôli mailboxingu a prevrátenú hodnotu smerového vektora *dInv*, ktorá sa využíva pri prechode KD-stromom a pri testovaní priesečníku s AABB.

8.4 Matice

Matica je implementovaná štruktúrou *Matrix4x4* a obsahuje funkcie pre násobenie, transponovanie matíc a na vytvorenie matíc merítka, rotácie a posunutia. Matice slúžia na vytvorenie transformácií objektov a využívajú sa pri prevode z globálnych súradníc do súradníc objektu a naopak.

8.5 Trojuholníky

Základným geometrickým objektom, ktorý sa v 3D grafike využíva je trojuholník. Vyhodnocovanie priesečníku s trojuholníkmi je jednoduché a rýchle a zároveň pre tento účel existuje viacero kvalitných algoritmov. V našej implementácii bol použitý algoritmus

Moller-Trumbore. Trojuholník je zároveň jediným implementovaným geometrickým útvarom v našom rendereri.

```
bool intersect(const Ray &ray, float &t) const {
    const Vec3f d = ray.d;
    const Vec3f pvec(  AC.cross(d) );
    const float det = AB.dot(pvec);
    if ((det < EPSILON) & (det>-EPSILON)) NOHIT
    const float invDet = 1.f/det;
    // prepare to compute u
    const Vec3f tvec(  ray.o - A  );
    const float u = (tvec.dot(pvec)) * invDet;
    if(u < 0.f || u > 1.f) NOHIT
    // prepare to compute v
    const Vec3f qvec(  AB.cross(tvec)  );
    const float v = (d.dot(qvec)) * invDet;
    if((u + v)> 1.f || v < 0.f ) NOHIT
    t = AC.dot(qvec)*invDet;
}
```

Ukážka kódu 4: Výpočet priesečníku lúča s trojuholníkom pomocou algoritmu Moller-Trumbore

8.6 Sieť trojuholníkov

Sieť trojuholníkov je implementovaná triedou *TriMesh*. Sieť pozostáva z určitého množstva trojuholníkov, ale má tiež rozhranie *Intersectable*. Na urýchlenie vyhľadávania priesečníku siete s lúčmi slúži akcelerátor (KD-strom). Keďže je celá sieť reprezentovaná ako jediný objekt s rozhraním *Intersectable*, je možné geometrickú informáciu ktorú obsahuje využiť viackrát. Pomocou transformácií je možné vytvárať instancované objekty, ktoré majú prakticky nulové nároky na pamäť aj keď sú zložené z veľmi zložitých sietí.

9 URÝCHĽOVANIE VYHĽADÁVANIA PRIESEČNÍKOV

9.1 Obalové telesá

Najvhodnejším obalovým telesom pre použitie v path traceri je osovo zarovnaný kváder. AABB majú svoje využitie nie len ako obalové telesá objektov, ale aj pri tvorbe KD-stromu a v BVH. Malú tesnosť vyvažuje jednoduchosťou a rýchlosťou výpočtu priesečníku. AABB je určený dvoma bodmi *min* a *max*. Na výpočet priesečníku lúča s AABB bol implementovaný tzv. slab test, ktorý predstavil Brian Smits. Tento test uvažuje obalový kváder ako priestor ohraničený šiestimi rovinami (2 roviny pre každú os).[18]

```
// smits algorithm
bool intersect(const Ray &r, float &tmin, float &tmax) const{
    const Vec3f l1 = (min-r.o)*r.dInv;
    const Vec3f l2 = (max-r.o)*r.dInv;
    minmaxf(l1.x,l2.x,tmin,tmax);
    tmin = fmaxf(fminf(l1.y,l2.y), tmin);
    tmax = fminf(fmaxf(l1.y,l2.y), tmax);
    tmin = fmaxf(fminf(l1.z,l2.z), tmin);
    tmax = fminf(fmaxf(l1.z,l2.z), tmax);
    return ((tmax >= 0.f) & (tmax >= tmin));
}
```

Ukážka kódu 5: Výpočet priesečníku lúča s AABB pomocou slab testu

9.2 Akcelerátory

Pre použitie urýchľovacích štruktúr bola vytvorená abstraktná trieda *Accelerator*. Toto rozhranie umožňuje použiť urýchľovaciu štruktúru podľa potreby a jednoduché rozšírenie programu o nové typy štruktúr v budúcnosti. Akcelerátory sa využívajú ako hierarchie objektov v scéne a objekty z triedy TriMesh používajú vlastný akcelerátor na vyhľadanie priesečníku medzi obsiahnutými trojuholníkmi.

9.2.1 KD-strom

KD-strom bol vybratý pre vysokú rýchlosť, kvalitu stromu zaručuje metóda tvorby SAH. KD-strom je ideálny pre použitie vo fyzikálnom offline rendereri pre svoj pomer rýchlosti vyhľadávania priesečníkov a času potrebného na tvorbu stromu. KD-strom je implementovaný triedou *KDTree*, uzly stromu sú implementované triedou *KDNode*. Veľkosť uzlu stromu je 8 bytov, každý uzol nesie informáciu o osi, ktorou je delený (2 bity), index do poľa uzlov, ktorý ukazuje na ľavého potomka (30 bitov) a pozíciu deliacej roviny(32 bitov). List stromu má len ukazateľ na objekt typu *ObjectHolder*. Trieda *ObjectHolder* uchováva objekty, ktoré sú v liste stromu. Okrem tejto triedy bola tiež vytvorená trieda ObjectHolderSSE, ktorá uchováva 4 trojuholníky a je schopná spočítať priesečník pre všetky 4 trojuholníky naraz pomocou inštrukcií SSE. Na tvorbu stromu boli implementované metódy popísané v [7]. Na vytvorenie KD-stromu je teda možné použiť builder so zložitosťou $n.log^2(n)$ alebo $n.log(n)$.

9.2.2 BVH

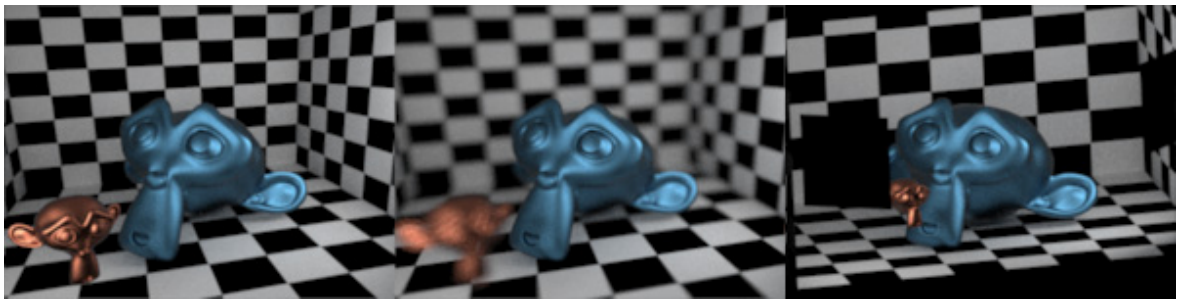
Štruktúra BVH bola implementovaná ako AABB strom triedou *BVH*. Uzly stromy sú definované triedou *BVHNode*. Každý uzol stromu má ukazatele na pravého a ľavého potomka, ukazateľ na *ObjectHolder* a *AABB*, ktorý obaľuje všetkých potomkov uzlu.

Tvorba stromu prebieha rekurzívne zhora nadol pomocu jednoduchého delenia objektov mediánom. Takisto prechod stromu pri vyhľadávaní priesečníku bol implementovaný rekurzívne. Tvorba stromu mediánom ani rekurzívny prechod stromu nie sú príliž efektívne metódy a preto je výkon implementovaného BVH pomerne nízky. Trieda *BVH* preto slúži skôr ako jednoduchý príklad použitia štruktúry s možnosťou vylepšenia v budúcnosti.

10 KAMERA

Kamera je definovaná ako abstraktná trieda, ktorá obsahuje základné vlastnosti a určuje rozhranie kamery. Jednotlivé implementácie modelov kamier sú potomkami tejto triedy. Najdôležitejšou funkciou kamery, ktorú musia potomkovia implementovať je funkcia *getRay()*. Táto funkcia pre zadané súradnice pixelu vytvorí lúč na základe typu projekcie konkrétnej kamery.

Základné vlastnosti, ktoré kameru definujú sú vektory *pos, dir, up*, teda poloha kamery, smer pohľadu kamery a vektor určujúci rotáciu kamery okolo jej osi. V premenných *imageWidth, imageHeight, widthInv* a *heightInv* sú uložené rozmery obrázku a ich prevrátené hodnoty. Premenné *clipStart* a *clipEnd* určujú minimálnu a maximálnu vzdialenosť objektov od kamery. Vektory *dirX* a *dirY* slúžia na prevod z NDC súradníc do globálnych súradníc. Hodnota týchto vektorov závisí na konkrétnom modeli kamery, *dirX* a *dirY* určujú rozmery obrazovej roviny, na ktorú sa výsledný obraz premieta.



Obr. 21: Obrázok vyrenderovaný rôznymi kamerami (zľava pinhole, thin lens, orthographic)

10.1 Pinhole kamera

Pinhole kamera je jednoduchý model kamery, jedinými parametrami tohto modelu sú veľkosť zorného uhla *angle* a orientácia kamery. Veľkosťou zorného uhla a rozmermi obrázku sú určené rozmery obrazovej roviny.

Generovanie lúčov je pomerne jednoduché. Počiatočná pozícia lúča je daná polohou kamery. Bod, v ktorom lúč pretína obrazovú rovinu sa určí ako *(x, y)+(r1, r2)*, pričom *x* a *y* sú súradnice pixelu a *r1* a *r2* sú náhodné čísla z intervalu *<0,1>*. Takto získaný bod sa následne prevedie do NDC súradníc a potom pomocou vektorov *dirX* a *dirY* do globálnych súradníc.

10.2 Orthografická kamera

Ortografická kamera premieta scénu na obrazovú rovinu rovnobežnými lúčmi, okrem orientácie kamery je jej jediným parametrom *scale*, teda veľkosť obrazovej roviny. Obrazovú rovinu ortografickej kamery si možno predstaviť ako veľké plátno (s rozmerom *scale*), na ktoré sa premietajú objekty scény.

Na rozdiel od pinhole kamery sa u ortografickej kamery nepočíta smer lúčov ale ich počiatočný bod. Keďže ide o ortografickú projekciu, smer lúčov je rovnaký pre všetky lúče pre všetky body obrazovej roviny. Počiatočný bod lúča je vlastne bod obrazovej roviny takže sa určí podobne ako v predchádzajúcom prípade- pre pixel so súradnicami x, y je výsledný bod *(x, y)+(r1, r2)*.

10.3 Thin lens kamera

Podobne ako u kamery pinhole aj projekcia kamery s tenkou šošovkou je daná veľkosťou zorného uhla *angle*. Parametre definujúce DOF kamery sú *focalDistance*, teda vzdialenosť od kamery na ktorú je zaostrené a parameter *fstop* pomocou ktorého je možné kontrolovať množstvo rozostrenia. Všetky body, ktoré ležia vo vzdialenosti *focalDistance* od kamery budú na výslednom obrázku zaostrené. Všetky takéto body ležia na pomyselnej rovine kolmej na os kamery- rovine zaostrenia (plane of focus).

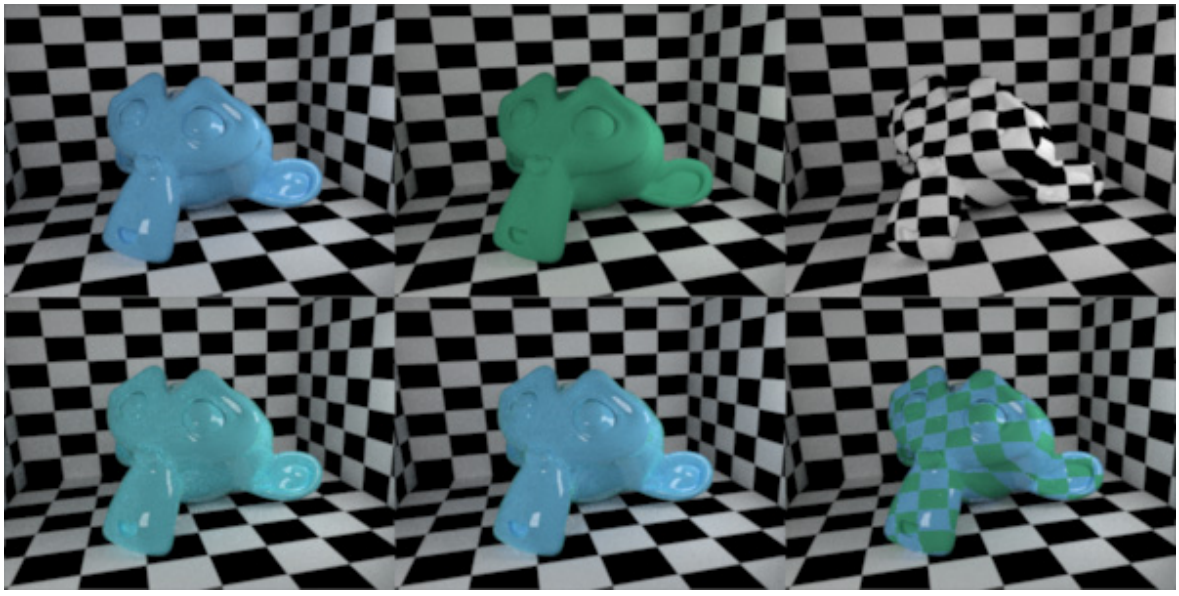
Generovanie lúčov u tohto typu kamery prebieha v dvoch fázach. Najprv sa vytvorí lúč rovnako ako u kamery typu pinhole. Následne sa vypočíta bod *P*, v ktorom tento lúč pretína rovinu zaostrenia. Na základe vzdialenosti bodu *P* od kamery, ohniskovej vzdialenosti šošovky a vzdialenosti roviny zaostrenia sa následne vypočíta veľkosť CoC. Počiatočný bod lúča *o* je ľubovoľný bod v kruhovom okolí pozície kamery s veľkosťou CoC. Smer lúča sa vypočíta jednoducho ako *P-o*.

11 MATERIÁLY

Materiálové vlastnosti objektov sú spojené do jednej triedy *Material*, každý materiál môže mať farebnú textúru, bump mapu, alpha mapu a BSDF definujúcu odrazové vlastnosti materiálu. Trieda materiálu teda tvorí rozhranie pre výpočet všetkých materiálových vlastností povrchu v mieste priesečníku.

11.1 Kombinované materiály

Spojením viacerých materiálov je možné vytvoriť realistickejšie vyzerajúce kombinované materiály. Možností ako skombinovať dva materiály je viacero. Najjednoduchší spôsob je percentuálny pomer, obidvom materiálom je priradená pravdepodobnosť vzorkovania podľa ich pomeru. Materiál, ktorý sa bude vzorkovať je pri každej interakcii lúča s kombinovaným materiálom vybraný náhodne. Takýto spôsob kombinovania materiálov definuje trieda *CombinedMaterial*. Ďalší spôsob kombinácie je implementovaný triedou *IorCombinedMaterial*. V tomto prípade nie sú pravdepodobnosti vzorkovania materiálov určené dopredu, ale menia sa podľa toho aký uhol zviera lúč s normálou povrchu. Pravdepodobnosti vzorkovania materiálov sa určujú pomocou Fresnelovho zákona a indexu lomu kombinovaného materiálu.



Obr. 22: Kombinovanie materiálov- v 1. riadku materiál 1, materiál 2, textúra na kombináciu, v 2. riadku Combined, IorCombined, TextureCombined

Poslednou implementovanou technikou je kombinácia pomocou textúry *TextureCombinedMaterial*. Ako textúra je použitá *ValueMap*- mapa ktorá pozostáva

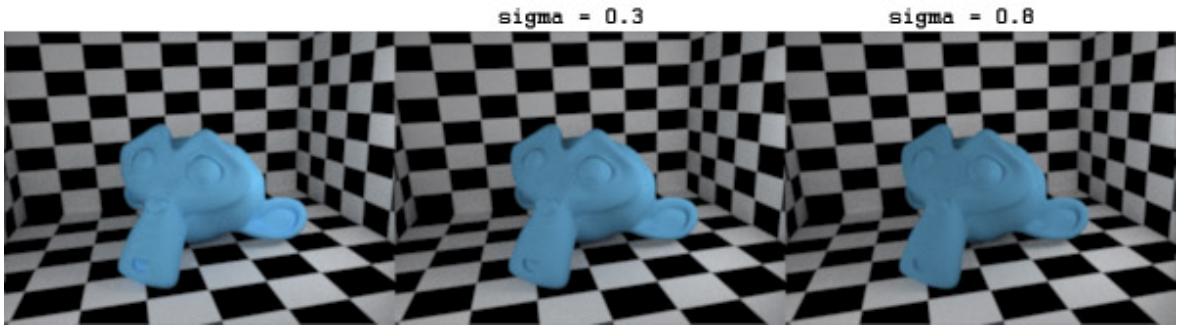
z hodnôt typu *float* v rozsahu *<0,1>*. Tieto hodnoty sú využité ako pravdepodosti pre výber jednotlivých materiálov.

11.2 BSDF

Univerzálne rozhranie pre BSDF je definované abstraktnou triedou *BSDF*. Táto trieda obsahuje okrem funkcie na vyhodnotenie množstva odrazeného svetla *f()*, funckciu ktorá vytvorí z prichádzajúceho lúča lúč odrazený *sample()* a funkcie *isEmitter()* a *getEmissivity()*, ktoré umožňujú zistiť či je materiál svetelný zdroj resp. jeho silu. Aby bolo možné zistiť typ odrazu, ktorým bol vytvorený vygenerovaný lúč, každá BSDF musí nastaviť parameter *ScatteringType*. Jedným z dôvodov pre zaznamenávanie typu odrazu sú odrazy typu specular. Distribučná funkcia takýchto odrazov je Diracova delta funkcia a preto vyžadujú špeciálne zaobchádzanie.

11.2.1 Difúzna a Oren-Nayar

Difúzna BRDF je definovaná triedou *DiffuseBRDF*, lúče sú generované pomocou funkcie *sampleHemisphereCosine()*, teda vzorkovaním pologule nad povrchom v mieste priesečníku. Trieda *OrenNayarBRDF* slúži na modelovanie drsných materiálov s retroodrazom. Jediným parametrom je *sigma*- drsnosť povrchu.

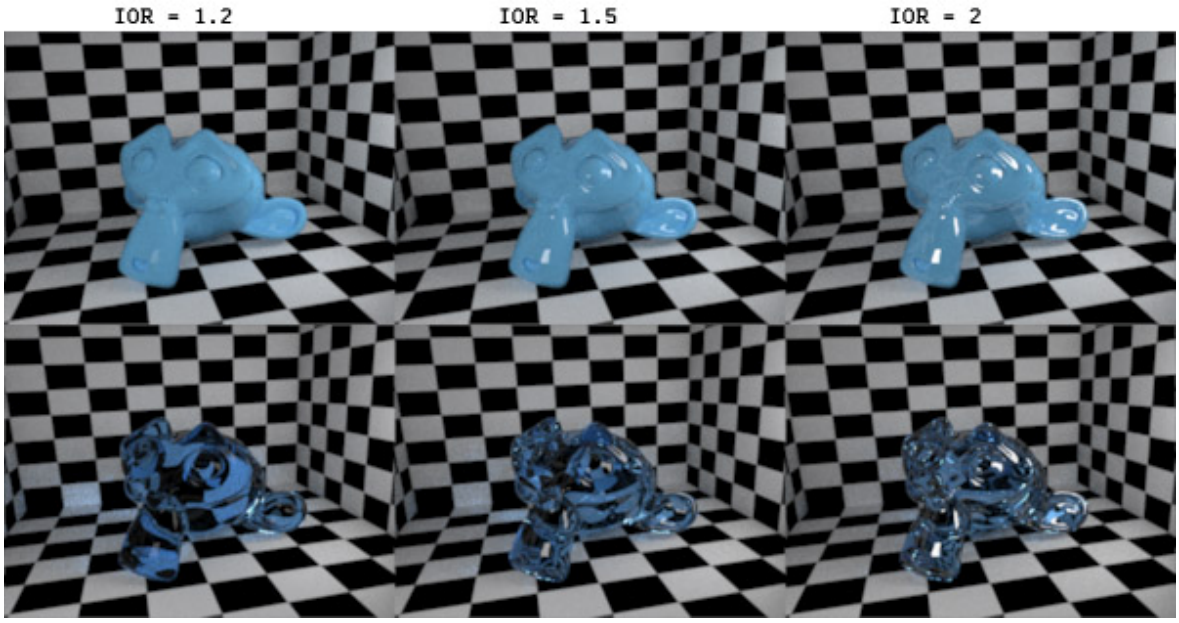


Obr. 23: Difúzny materiál a Oren-Nayar s rôznymi hodnotami sigma

11.2.2 Specular

Specular BRDF slúži na modelovanie materiálov s dokonalým odrazom, definovaná je triedou *PerfectSpecularBRDF*. Okrem tejto triedy bola vytvorená tiež trieda *SpecularBRDF*, táto trieda kombinuje dokonalý a difúzny odraz pomocou parametru *ior*- index lomu.

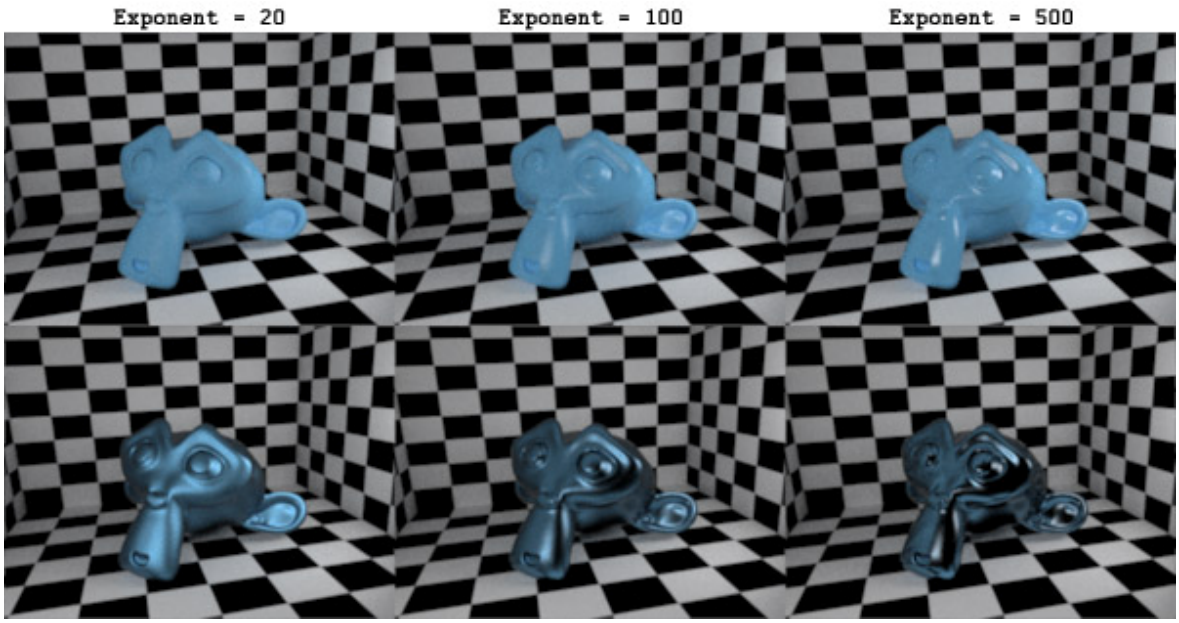
Na modelovanie materiálov podobných sklu bola vytvorená trieda *TransparentSpecularBTDF*. Na rozdiel od predchádzajúcich typov tento umožňuje okrem odrazenia lúča aj jeho prechod cez objekt. Pomer medzi odrazenými a prechádzajúcimi lúčmi opäť určuje index lomu.



Obr. 24: BRDF Specular s difúznym komponentom (1. riadok) a TransparentSpecularBTDF (2.riadok) pre rôzne hodnoty IOR

11.2.3 Phongova

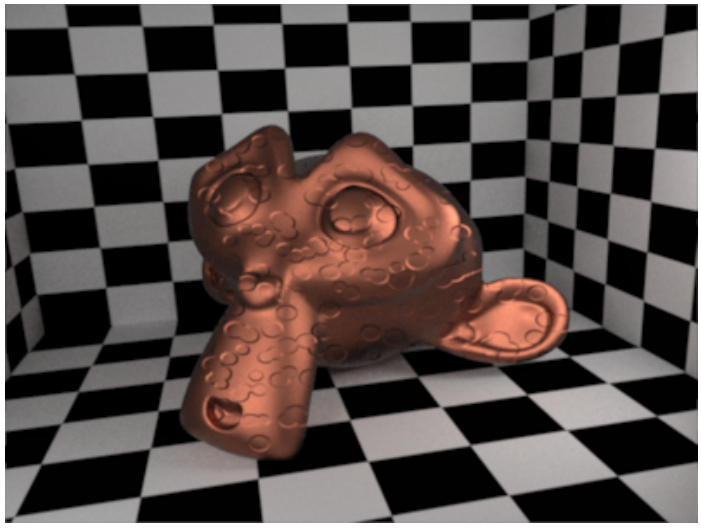
Phongova BRDF je definovaná triedou *PhongBRDF*, na generovanie lúčov slúži funkcia *samplePhongLobe*(), ktorá vypočíta odrazený lúč podľa smeru prichádzajúceho lúča a exponentu. Implementovaná Phongova BRDF má navyše niekoľko parametrov, ktoré umožňujú vytvárať zaujímavejšie materiály. Parameter *specularReflection* určuje, či budú odrazy generované len podľa Phongovho odrazu alebo budú kombinované s difúznym odrazom. Materiály vytvorené so *specularReflection=true* sú vhodné na modelovanie kovov, *specularReflection=false* sa hodí na modelovanie materiálov ako sú plasty s rôznou drsnosťou povrchu. Ďalším parametrom je *ior*, ktorý určuje pomer medzi difúznou a phongovou zložkou. Pomer týchto zložiek sa určuje podľa úhla, ktorý zviera normála a prichádzajúci lúč.



Obr. 25: Phongova BRDF kombinovaná s difúznym odrazom (1.riadok) a Phongova BRDF so specular odrazom (2.riadok) pre rôzne hodnoty exponentu

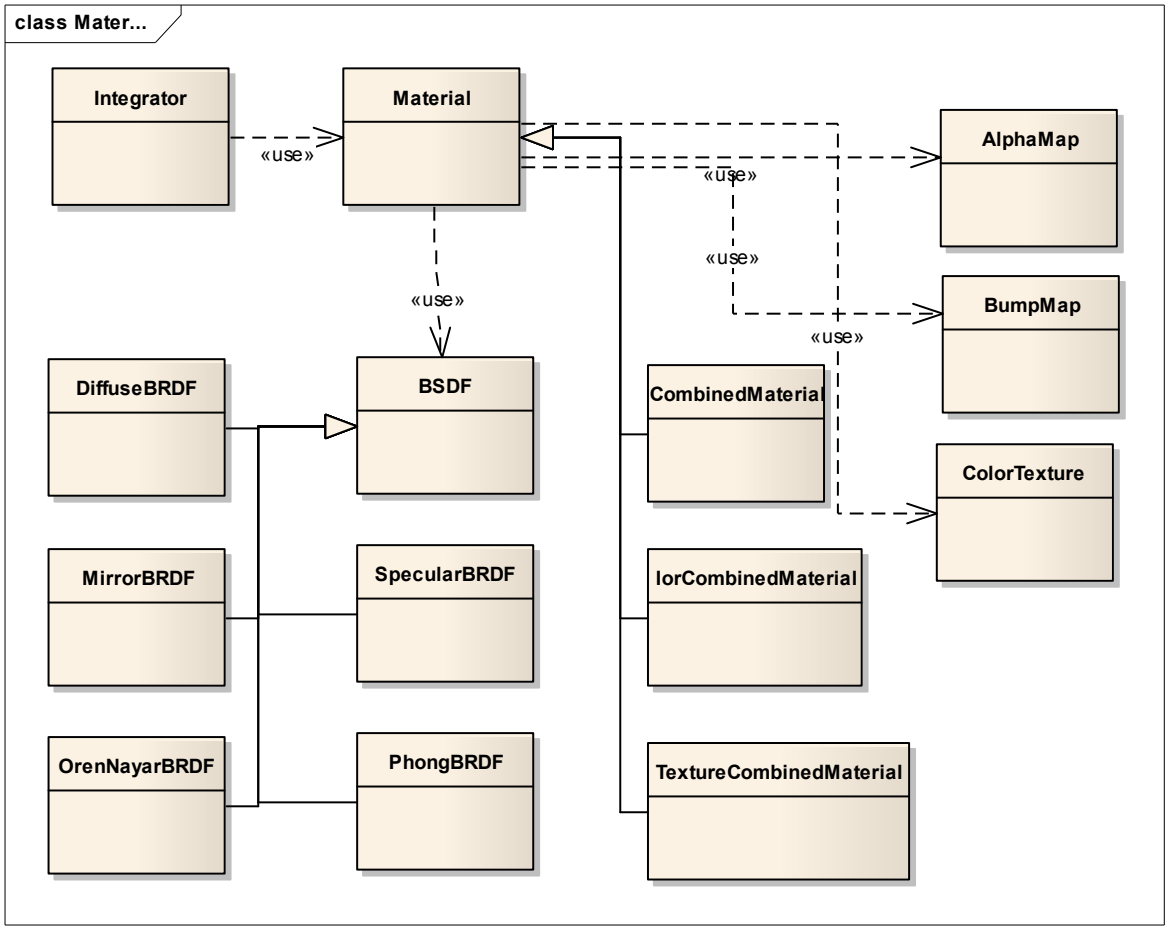
11.3 Textúrovanie

Textúry je možné priradiť materiálom pomocou tried *ColorMap* (farebná textúra) a *BumpMap* (bump mapa). Okrem týchto tried bola vytvorená ešte trieda *ValueMap*, v ktorej sú uložené dáta obrázku v rozsahu od 0 do 1. *ValueMap* môže byť využitá na mapovanie hodnôt z obrázku do rôznych vlastností- napr. výška u bump mapingu alebo pomer materiálov v kombinovanom materiáli.



Obr. 26: Bump mapping

Na načítavanie súborov textúr bola použitá trieda wxImage z knižnice wxWidgets, ktorá umožňuje načítať obrázky vo všetkých najpoužívanejších formátoch ako BMP, JPEG, PNG, GIF, TIFF, TGA atď. Môže nastať situácia, keď viacero materiálov využíva ten istý obrázok textúry. Držať v pamäti viackrát ten istý obrázok by bolo zbytočné, preto bola vytvorená trieda *ImageLoader*. Trieda *ImageLoader* slúži na načítanie obrázkov pre textúry a zároveň každú vytvorenú textúru uloží do STL mapy *colorMaps* alebo *valueMaps* podľa názvu obrázku. V prípade požiadavku na textúru vždy najprv skontroluje, či požadovaná textúra už nie je vytvorená. Ak bola vytvorená použije sa znova existujúca textúra, ak nie nie vytvorí sa z obrázku nová.



Obr. 27: UML diagram pre materiály

12 ZDROJE SVETLA

Pre svetlá bolo vytvorené rozhranie abstraktnou triedou *Light*. Základnými funkciami tohto rozhrania je funkcia *samplePoint()*, ktorá náhodne vygeneruje na povrchu svetla bod a funkcia *evalPDF()*, ktorá vyhodnotí pravdepodobnosť zásahu svetla. Tento bod je následne využitý pri výpočte priameho osvetlenia alebo pri tvorbe svetelnej cesty u obojsmerného path tracingu. Z rôznych typov svetiel popísaných v kapitole 5 bol vybratý typ jediný- *MeshLight.* Doplnenie ďalších typov svetiel by do pripraveného rozhrania nemal byť problém.

Svetlo typu MeshLight sa vytvára na základe 3D modelu s materiálom typu emitter. Je tak možné vytvoriť svetlo ľubovoľnej veľkosti a tvaru. Generovanie náhodných bodov na povrchu svetla prebieha v dvoch krokoch. Najprv sa náhodne vyberie trojuholník, na ktorom sa bude bod generovať a následne sa vygenerujú UV súradnice bodu. Pravdepodobnosť vygenerovaného bodu je daná vzťahom 12.1, kde *pLight* je pravdepodobnosť vybratia svetla, *A* je plocha trojuholníku a *nTris* je počet trojuholníkov.

$$P = pLight \cdot \frac{1}{A} \cdot \frac{1}{nTris} \tag{12.1}$$

13 ALGORITMY PRE VÝPOČET GI

Pre algoritmy na výpočet GI bolo vytvorené jednotné rozhranie *Integrator* s funkciou *radiance()*. Táto funkcia slúži na výpočet žiarenia prechádzajúceho zadaným pixelom. Pomocou tohoto rozhrania je výpočet osvetlenia oddelený od zvyšku systému a umožňuje program jednoducho rozšíriť o nový typ algoritmu.

Z algoritmov pre výpočet globálneho osvetlenia popísaných v kapitole 6 boli pre implementáciu vybrané algoritmy path tracing, path tracing s odhadom nasledujúcej udalosti a obojsmerný path tracing. Okrem toho bol implementovaný tiež algoritmus Metropolis Light Transport vo verzii, ktorú predstavil Kelemen a Kalos. Implementované MLT je teda možné použiť ako vzorkovací algoritmus pre hociktorý z vyššie uvedených integrátorov. Pre integrátory boli vytvorené triedy *SimplePathIntegrator*, *PathIntegrator*, *BidirIntegrator* a *MLTIntegrator*.

Pre integrátory bolo vytvorených niekoľko univerzálnych funkcií. Na výpočet priameho osvetlenia slúži funkcia *traceToLight()*, ktorá vygeneruje náhodný bod na náhodne vybratom svetle a vykoná test viditeľnosti. Funkcia *lightSourcePDF()* vyhodnotí pravdepodobnosť priesečníku lúča so svetlom. Pre potreby obojsmerného path tracingu boli vytvorené funkcie *tracePath()* a *sampleLight()*. Fukcia *tracePath()* slúži na vytvorenie cesty scénou podobne ako u jednoduchého path tracingu. Rozdiel je len v tom, že ukladá všetky uzly cesty do predaného poľa na neskoršie výpočty. Funkcia *sampleLight()* slúži na generovanie náhodného lúča zo svetla, tento lúč následne slúži ako počiatok svetelnej cesty(light tracing).

13.1 Path tracing

Jednotuchý path tracing je implementovaný triedou *SimplePathIntegrator*, tento integrátor je najjednoduchším spôsobom výpočtu globálneho osvetlenia. V porovnaní s ďalšími implementovanými metódami konverguje najpomalšie, obzvlášť ak je použité zložité osvetlenie.

```
while(1){
   isectData.n_s=isectData.n_g;
   isectData.mat->sampleBSDF(ray,wi,isectData,pdf,brdf);
   mat = isectData.mat;
   mat->getColorAt(isectData);
   // reflectance
   Vec3f f = isectData.f;
   // emitter hit
   if( mat->isEmitter()) {
        col += pathThroughput * f * mat->getBSDF()->getEmissivity();
        break;
     }
    // max depth termination
    if(depth==mDepth) break;
    // zero probability termination
    if(pdf<=0) break;
    // rusian roulette
    if (depth>3) {
        survP = f.luminance();
        if (context->random()>survP)
            break;
    }
    // cosine therm
    const float dot= wi.absDot(isectData.n_s);
    // update path throuhput
    pathThroughput *= f*(brdf / (pdf*survP))*dot;
    // set up ray for next iteration
    ray.set(isectData.x,wi);
}
```

Ukážka kódu 6: Jednoduchý path tracing

Path tracing s výpočtom priameho osvetlenia implementuje trieda *PathIntegrator*. Tento integrátor sa hodí hlavne na použitie v scénach so silným priamym osvetlením. V takýchto scénach konverguje oveľa rýchlejšie ako predchádzajúca metóda, je však o niečo pomalší kvôli väčšiemu počtu testovaných lúčov.

13.2 Obojsmerný path tracing

Obojsmerný path tracing bol implementovaný triedou *BidirIntegrator*. Tento integrátor pracuje tak, že najprv vygeneruje cestu z kamery a zo svetla a následne počíta príspevky pre jednotlivé kombinácie ciest. Vyhodnocujú sa rôzne typy ciest, pre každý vrchol cesty z kamery sa počíta priame osvetlenie a následne sa vrchol prepojí na všetky vrcholy cesty zo svetla a vypočítajú sa jednotlivé príspevky. V poslednej fázi sa prepojujú vrcholy cesty zo svetla na kameru (light tracing).

Na váženie ciest u tohto algoritmu bola použitá jednoduchá metóda uniformných váh. Táto metóda podáva dostačujúce výsledky, ale mohla by byť nahradená lepšou. integrátor by mohol byť doplnený o váženie ciest pomocou metódy Multiple Importance Sampling, ktorú predstavil vo svojej dizertačnej práci E. Veach.

13.3 Metropolis Light Transport

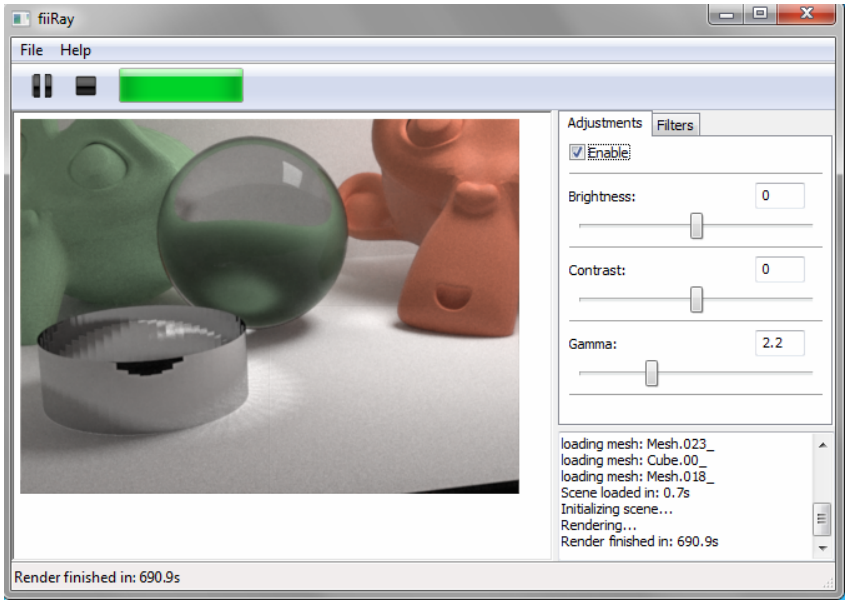
Algoritmus MLT bol implementovaný triedou *MLTIntegrator*. Implementácia bola založená na metóde, ktorú predstavil Kelemen a Szirmay-Kalos. *MLTIntegrator* nie je rovnaký integrátor ako predchádzajúce triedy, jedná sa skôr o vzorkovací mechanizmus. Ako premennú uchováva ukazateľ na niektorý z integrátorov predstavených vyššie a osvetlenie počíta vzorkovaním tohto integrátoru metódou Metropolis. Táto trieda tak umožňuje aplikovať vzorkovanie Metropolis na ktorýkoľvek z implementovaných integrátorov.

14 UŽÍVATEĽSKÉ ROZHRANIE

Pre tvorbu užívateľského rozhrania programu bola zvolená knihovňa wxWidgets. wxWidgets je multiplatformná open-source C++ knihovňa na tvorbu GUI, je dostupná pre množstvo operačných systémov a existujú aj jej porty v iných jazykoch ako C++. wxWidgets funguje ako nádstavba nad API platformy, takže vytvorené programy majú natívny vzhľad. Okrem práce s užívateľským rozhraním umožňuje aj rôzne iné funkcie ako napríklad prácu so súbormi, socketmi, obrázkami, spracovanie XML alebo vykresľovanie pomocou OpenGL. Na tvorbu GUI vo wxWidgets je možné využiť niektorý z dostupných builderov, napr. wxFormBuilder, wxSmith, wxGlade atď.

Návrh užívateľského rozhrania našej aplikácie bol vytvorený v programe wxFormBuilder. FormBuilder umožňuje jednoducho a rýchlo navrhnúť GUI aplikácie bez nutnosti ručne písať kód pre jednotlivé komponenty. Takto bola vytvorená trieda *GUIFrame*, z ktorej bola pomocou dedičnosti vytvorená trieda hlavného okna aplikácie.

Grafické rozhranie rendereru bolo rozdelené na 3 hlavné časti- menu, „plátno“ na ktorom sa zobrazuje renderovaný obrázok a panel nástrojov. V menu je možné nájsť základné funkcie ako otvorenie scény, uloženie obrázku alebo pozastavenie renderu. Panel s nástrojmi je umiestnený v pravej časti okna a sú v ňom umiestnené prvky na jednoduchú úpravu obrázku. V spodnej časti panelu je umiestnený text box, ktorý slúži ako konzola na výpis stavu načítania scény a renderu.



Obr. 28: Vzhľad GUI aplikácie

15 FORMÁT SCÉNY

Ako formát pre export scény bolo zvolené XML kvôli svojej jednoduchosti. Formát založený na XML je flexibilný, prehľadný a umožňuje jednoduchý export scény aj jej následné načítanie. Nevýhodou takéhoto formátu je veľká doba spracovania rozsiahlych súborov, ktoré vznikajú pri použití detajlných modelov v 3D scénach pomerne často.

Informácie o scéne sú rozdelené do dvoch súborov, jeden súbor je pre nastavenia a jeden pre geometriu. Takéto rozdelenie ma niekoľko výhod. Vďaka oddeleniu nastavení nie je pri dolaďovaní parametrov renderu a materiálov nutné exportovať vždy aj geometriu. Keďže súbor nastavení scény je pomerne malý oproti súboru geometrie (ktorý môže mať aj stovky MB) uľahčuje ich oddelenie tiež ručné úpravy a hľadanie chýb. Poslednou výhodou je otvorená možnosť použiť pre geometrické informácie iný formát súboru ako XML, čo môže viesť k menšej dobe exportu aj načítavania scény.

Na načítanie scény je nutné použiť nejaký XML parser. Základné požiadavky, ktoré renderer kladie na takýto parser sú vysoká rýchlosť a malá pamäťová náročnosť pri spracovávaní veľkých súborov. Tieto požiadavky vylučujú použitie parserov, ktoré vytvárajú DOM dokument. Keďže scéna je načítavaná len jedenkrát pred renderovaním a objekty sú vytvárané postupne pri načítavaní je pre tento účel ideálny parser s rozhraním podobným SAX.

Ako XML parser bol použitý irrXML. IrrXML je veľmi jednodúchý XML parser určený hlavne pre použitie v hrách. Spĺňa všetky dané požiadavky, umožňuje len dopredné čítanie súborov, je rýchly a má malé pamäťové nároky. Pre potreby načítania scény bola vytvorená pomocná trieda *SceneLoader*, ktorá zo súboru načíta všetky nastavenia a vytvorí objekty scény, materiály, kameru a pozadie. Okrem triedy *sceneLoader* bola vytvorená ešte trieda *meshLoader*, ktorá zo samostatného súboru načíta geometriu jednotlivých objektov scény. Keďže XML môže byť pre rozsiahle scény pomalé bol vytvorený veľmi jednoduchý binárny formát. Do binárneho formátu su uložené len informácie o geometrii objektov a na jeho načítanie slúži trieda *BinaryMeshLoader*.

16 EXPORTNÝ SKRIPT PRE PROGRAM BLENDER

16.1 Blender

Blender je open source program určený pre tvorbu 3D obsahu, je dostupný na mnohých platformách pod GNU GPL licenciou. Blender obsahuje množstvo nástrojov pre tvorbu 3D scén od modelovania až po strih videa. Blender obsahuje tiež Pythonovské rozhranie, ktoré umožňuje pracovať s internými dátami a funkciami Blenderu. Python API prešlo počas vývoja Blenderu rôznymi zmenami, zásadné zmeny rozhrania prináša verzia 2.50. Keďže skripty vytvorené v skorších verziách nebudú v nových verziách podporované, exportný skript bol vytvorený pre súčasnú verziu Blenderu 2.5 (Alpha2) [8].

16.2 Tvorba užívateľského rozhrania skriptu

Pythonovské rozhranie Blenderu 2.50 umožňuje veľmi jednoduchú tvorbu vlastných GUI prvkov. Nové prvky je možné začleniť do rôznych okien a panelov podľa účelu. Takto je možné dosiahnuť natívny vzhľad a intuitívne ovládanie.

16.2.1 Tlačítka

Tlačítka je možné vytvárať zdedením triedy *bpy.types.Operator* a preťažením funkcie *invoke()*. Každé vytvorené tlačítko ma vlastné ID, text, ktorý sa na ňom zobrazuje a stručný popis.

```
class TestButton(bpy.types.Operator):
    bl_idname = "testButton"                // button ID
    bl_label = "test"                       // label
    bl_description = "this is test button"  // description

    def invoke(self, context, event):
        print("hello!")                     // do something
        return{'FINISHED'}

bpy.types.register( TestButton)
```

Ukážka kódu 7: Vytvorenie vlastného tlačítka

16.2.2 Panely

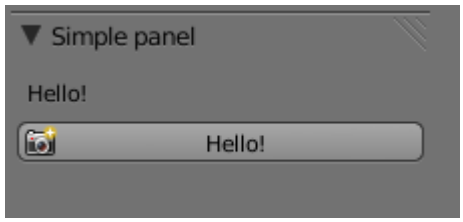
Panely je možné vytvoriť zdedením triedy *bpy.types.Panel* a preťažením funkcie *draw()*. Panely je možné vytvárať v rôznych častiach GUI blenderu, pre naše potreby je najvhodnejší panel vlastností (Properties panel), ktorý obsahuje nastavenia renderu, materiálov, kamery aj pozadia.

```
class TestPanel(bpy.types.Panel):
    bl_label = "Simple panel"      // displayed label
    bl_space_type = "PROPERTIES"   // display in properties panel
    bl_region_type = "WINDOW"      // display in Blender's window
    bl_context = "material"        // display in material context

    def draw(self, context):
        layout = self.layout       // get panel's layout
        row = layout.row()         // create row
        row.label(text="Hello!")   // display label
        // draw button
        row.operator("testButton",text="Hello!",icon='RENDER_STILL')

bpy.types.register( TestPanel)     // register panel to GUI
```

Ukážka kódu 8: Vytvorenie vlastného panelu



Obr. 29: Vytvorený panel

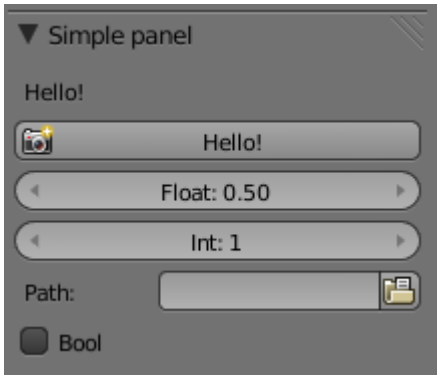
16.2.3 Ostatné prvky

Ďalšie prvky GUI ako napríklad checkboxy, file dialogy, dropdown listy je možné vytvárať podľa premenných, ktoré definujú ich obsah. Blender umožňuje vytvoriť premenné základných datových typov (float, int, bool), string, enum, ukazatele. Pre každú premennú je možné definovať okrem jej názvu aj defaultnú hodnotu, prípadne jej minimum a maximum. Pri vykresľovaní panelu je možné jednoducho zobraziť prvky pomocou funkcie *prop()*. Blender automaticky vykreslí ten prvok rozhrania, ktorý zodpovedá typu premennej a prepojí prvok rozhrania s premennou. Po zadaní novej

hodnoty do GUI sa zmení aj hodnota premennej. Napríklad pre premennú typu bool je automaticky zobrazený checkbox.

```
// register properties into Scene
scn = bpy.types.Scene
scn.FloatProperty(attr="prop1",name="Float",default=0.5,min=0,max=1)
scn.IntProperty(attr="prop2",name="Int",default=1,min=0,max=5)
scn.StringProperty(attr="prop3",name="Path",default="",subtype='FILE_PATH')
scn.BoolProperty(attr="prop4",name="Bool",default=False)
...
    def draw(self, context):
        ...
        scn = scene = context.scene     // get scene
        row = layout.row()
        row.prop(scn,'prop1')           // add property to row
        row = layout.row()
        row.prop(scn,'prop2')
        row = layout.row()
        row.prop(scn,'prop3')
        row = layout.row()
        row.prop(scn,'prop4')
        ...
```

Ukážka kódu 9: Vykresľovanie komponentov



Obr. 30: Rôzne prvky užívateľského rozhrania

17 POROVNANIE S KONKRENČNÝMI PRODUKTAMI

Jednými z najznámejších open-source rendererov, ktoré sú stále vyvíjané a zároveň podporujú export z Blenderu sú Yafaray a Luxrender. Obidva programy sú multiplatformné. Yafaray je určený len pre použitie s Blenderom, pre Luxrender existuje viacero exportérov do rôznych 3D programov (Blender, Maya, 3Ds Max ...).

Luxrender je založený na rendereri PBRT, ktorý vytvoril Matt Pharr a Greg Humphreys pre akademické účely. Luxrender je neustále vyvíjaný od roku 2007, podporuje mnoho funkcií a „state of art" algoritmov.

Yafaray vznikol zo staršieho Yafrayu, účel tohoto programu je rozšíriť možnosti renderovania obrázkov v Blenderi. Na rozdiel od interného rendereru Blenderu umožňuje výpočet globálneho osvetlenia pomocou path tracingu alebo photon mappingu.

Porovnanie týchto dvoch vybraných rendererov s programom, ktorý je produktom tejto práce (fiiRay) je rozdelené na porovnanie dostupných funkcií a porovnanie výkonu. Porovnanie dostupných funkcií je možné vidieť v Tabuľke č. 1, v množstve podporovaných funkcií jednoznačne vedie Luxrender, Yafarayu chýbajú len niektoré pokročilejšie funkcie.

	fiiRay	Luxrender	Yafaray
Path tracing	v	v	v
Bidir. Path tracing	v	v	v*
Metropolis light transport	v	v	x
Volumetrické renderovanie	x	v	v
Spektrálne renderovanie	x	v	x
Rôzne modely kamier	v	v	v
Renderovanie po sieti	x	v	x
Nasvietenie pomocou HDRI	x	v	v
Model oblohy	x	v	v
Možnosť úpravy obrázkov	v	v	x
Multiplatformný	v	v	v

*BDPT v Yafarayi je momentálne len nestabilná vývojová verzia

Tabuľka 1: Porovnanie dostupnosti vybraných funkcií

Druhou časťou porovnania je porovnanie výkonu jednotlivých rendererov. Keďže každý z programov používa rôzne algoritmy pre výpočet osvetlenia nie je možné ich rýchlosť porovnávať podľa počtu vzorkov za sekundu. Ako kritérium porovnania bola zvolená kvalita obrázku za rovnaký čas renderovania. Pre tento účel bola použitá scéna modelu obývačky (príloha č.1). Táto scéna obsahuje komplexné nasvietenie (10 svetiel),

množstvo objektov(>300) a geometriu tvorí približne 600 tisíc vrcholov. Scéna teda umožňuje porovnať programy v náročnejších podmienkach. Každý program bol otestovaný tak, že renderoval scénu v rozlíšení 400x300 pixelov približne 10 minút. Okrem kvality vyrenderovaného obrázku je tiež možné porovnať dobu potrebnú pre export scény z Blenderu, dobu potrebnú pre načítanie a spracovanie scény programom a približné pamäťové nároky programov (Tabuľka č.2). Výsledné obrázky je možné vidieť v prílohe č.2. Najhorší výsledok dosiahol podľa očakávania Yafaray, ktorý podporuje len jednoduchý path tracing. Na obrázku je vidno množstvo vysokofrevenčného šumu typického pre tento algoritus. Obrázok vyrenderovaný v Luxrenderi obsahuje menej vysokofrekvenčného šumu, ale je na ňom vidieť typické flaky spôsobené vzorkovaním Metropolis. Spomedzi troch rendererov dosiahol najlepší výsledok fiiRay.

	fiiRay	Luxrender	Yafaray
Doba exportu	~15s	~20s	~10s
Doba načítania scény	~22s	~28s	~14s
Nároky na pamäť	~360MB	~740MB	~200MB
Čas renderu	~10 min.	~10 min.	~10 min.

Tabuľka 2: Porovnanie výkonu jednotlivých programov

V porovnaní je vidieť, že fiiRay je schopný ostatným produktom konkurovať kvalitou a rýchlosťou, ale zaostáva v množstve implementovaných funkcií. Ako Luxrender tak aj Yafaray však majú za sebou pomerne dlhú dobu vývoja takže tento výsledok nie je príliž prekvapivý. Zaujímavosťou je, že Luxrender počas krátkeho testovania spadol dvakrát a Yafaray raz, pričom spôsobil pád Blenderu a stratu všetkých nastavení.

ZÁVER

Cieľom diplomovej práce bolo vytvoriť rešerš o renderovaní komplexných scén a algoritmoch určených na výpčet globálneho osvetlenia. V teoretickej časti bol popísaný samotný koncept globálneho osvetlenia a zobrazovacej rovnice, ktorú formuloval J. Kajiya. Následne boli popísané základné časti softwarových rendererov ako sú kamery, materiály, svetlá a urýchľovacie štruktúry. Posledná kapitola teoretickej časti sa zaoberá riešením zobrazovacej rovnice a algoritmami, ktoré sú vhodné pre účel výpočtu globálneho osvetlenia. Postupne boli teoreticky popísané algoritmy path tracing, obojsmerný path tracing a Metropolis light transport.

Praktická časť práce sa zaoberá implementáciou softwareového rendereru, ktorý umožňuje výpočet globálneho osvetlenia pomocou metódy sledovania ciest. Táto časť rieši návrh rendereru a implementáciu jednotlivých prvkov popísaných v teoretickej časti. Posledné kapitoly praktickej časti sa zaoberajú tvorbou užívateľského rozhrania programu a tvorbou exportného skriptu pre program Blender.

Produktom praktickej práce je program s názvom fiiRay, tento renderer bol napísaný v programovacom jazyku C++. fiiRay bol napísaný multiplatformne pomocou knižnice wxWidgets a implementuje funkcie a metódy predstavené v teoretickej časti práce. Do programu bola zahrnutá podpora viacerých materiálov, kamier a zobrazovacích algoritmov. Z popísaných zobrazovacích algoritmov boli implementované path tracing, path tracing s odhadom nasledujúcej udalosti, obojsmerný path tracing a Metropolis light transport. Okrem rendereru bol vytvorený tiež exportný sript pre program Blender v jazyku Python. Tento skript umožňuje pre modely vytvorené v Blenderi definovať materiály a nastavenia renderu a následne ich vyexportovať do fiiRayu.

ZÁVER V ANGLIČTINE

The aim of the thesis was to create a literature retrieval about rendering of complex scenes and global illumination algorithms. In the theoretical part of the thesis was described the concept of global illumination computation and rendering equation defined by J. Kajiya. Then basic parts of software renderer like cameras, materials and acceleration structures were explained. Last chapter of the theoretical part deals with methods for solving the rendering equation and algorithms for global illumination computation. Path tracing, bidirectional path tracing and Metropolis light transport were explained.

The practical part of the thesis deals with implementation of software renderer that can compute gloabal illumination using path tracing. This part describes design of renderer and implementation of its parts. Last chapters describes creation of user interface and creation of export script for Blender.

Product of the practical part is renderer called fiiRay. fiiRay was written in cross-platform C++ programming language using wxWidgets library and implements functions and methods described in the theoretical part. fiiRay supports different types of materials, cameras and rendering algorithms. From the rendering algorithms explained in the theoretical part it supports path tracing, path tracing with next event estimation, bidirectional path tracing and Metropolis light transport. Also export script for Blender written in Python programming language was created. This script allows to define materials and render settings for models created in Blender and to export them into fiiRay.

SEZNAM POUŽITÉ LITERATURY

[1] VEACH, Eric. *Robust Monte Carlo methods for light transport simulation*. [s.l.], 1997. 432 s. Dizertační práce.

[2] PHARR, Matt, HUMPHREYS, Greg. *Physically based rendering: from theory to implementation*. [s.l.] : Morgan Kaufmann, 2004. 1056 s. ISBN 978-0-12-553180-1.

[3] LAFORTUNE, Eric. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. [s.l.], 1996. 146 s. Dizertační práce.

[4] WALD, Ingo. *Realtime Ray Tracing and Interactive Global Illumination*. [s.l.], 2004. 311 s. Dizertační práce.

[5] HAVRAN, Vlastimil. *Heuristic Ray Shooting Algorithms*. [s.l.], 2001. 220 s. Dizertační práce.

[6] DUTRE, Philip. *Global Illumination Compendium*. [online]. 2003.

[7] WALD, Ingo, HAVRAN, Vlastimil. *On building fast kd-trees for ray tracing, and on doing that in O(N log N)*. Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing. 2006, s. 61-69.

[8] Blender website, URL: http://www.blender.org/.

[9] Texture mapping. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-05-24]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Texture_mapping>.

[10] *Computational Science and Its Applications*. [s.l.] : [s.n.], 2003. Camera Models and Optical Systems Used in Computer Graphics, s. 983. ISBN 978-3-540-40156-8.

[11] Veach, E. and Guibas, L. J. 1995. Optimally combining sampling techniques for Monte Carlo rendering. In *Proceedings of the 22nd Annual Conference on Computer Graphics and interactive Techniques* S. G. Mair and R. Cook, Eds. SIGGRAPH '95.

[12] MÜLLER, Gordon. *Object Hierarchies for Efficient Rendering*. [s.l.], 2004. 83 s. Diplomová práce. Technischen Universität Braunschweig.

[13] *Blenderartists.org* [online]. 2010 [cit. 2010-05-6]. Scripting examples for 2.5. Dostupné z WWW: <http://blenderartists.org/forum/showthread.php?t=164765>.

[14] P. Hanrahan, *Monte Carlo path tracing*, SIGGRAPH 2001 Course 29: Monte Carlo Ray Tracing.

[15] Kajiya, J. T. 1986. *The rendering equation*. *SIGGRAPH Comput. Graph.* 20, 4 (Aug. 1986), 143-150.

[16] Blinn, J. F. 1977. *Models of light reflection for computer synthesized pictures*. *SIGGRAPH Comput. Graph.* 11, 2 (Aug. 1977), 192-198.

[17] *Bounding volume hierarchy* [online]. 2005 [cit. 2010-05-15]. BVH construction. Dostupné z WWW: <http://www.spiritus-temporis.com/bounding-volume-hierarchy/bvh-construction.html>.

[18] Smits, B. 1998. *Efficiency issues for ray tracing*. J. Graph. Tools 3,2(Feb. 1998), 1-14.

[19] Pinhole camera. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2002, last modified on 2010 [cit. 2010-06-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Pinhole_camera>.

[20] Oren Nayar Reflectance Model. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2007, last modified on 2009 [cit. 2010-06-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Oren%E2%80%93Nayar_Reflectance_Model>.

[21] KELEMEN , Csaba , et al A Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm. [online]. Dostupné z WWW: <http://www.fsz.bme.hu/~szirmay/paper50_electronic.pdf>.

[22] FOG, Agner. *Optimizing software in C++*. [s.l.] : [s.n.], 2008. 144 s.

[23] *Real-Time rendering* [online]. 2009 [cit. 2010-06-01]. Object/Object intersection. Dostupné z WWW: <http://www.realtimerendering.com/intersections.html>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AABB Axis Aligned Bounding Box.

API Application programming interface.

BRDF Bidirectional reflectance distribution function.

BSDF Bidirectional scattering distribution function.

BSP Binary space partitioning.

BVH Bounding volume hierarchy.

CoC Circle of confusion.

DOF Depth of field.

FOV Field of view.

GUI Grafické užívateľské rozhranie.

IOR Index of refraction. Index lomu.

MLT Metropolis Light Transport.

OBB Oriented bounding box.

SAH Surface area heuristic.

STL Standard template library.

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM PRÍLOH

PRÍLOHA P I: OBRÁZKY VYRENDEROVANÉ VO FIIRAYI

Cadillac ’49:

Design obývačky:

Design kúpeľne:

Instancované objekty + DOF (scéna obsahuje 22 miliónov vrcholov):

PRÍLOHA P II: POROVNANIE FIIRAYU S LUXRENDEROM A YAFARAYOM

Scéna vyrenderovaná v programe Yafaray:

Scéna vyrenderovaná v programe Luxrender:

Scéna vyrenderovaná vo vytvorenom programe fiiRay:

PRÍLOHA P III: FORMÁT SCÉNY

```
<?xml version="1.0" encoding="utf-8"?>
<scene>
<geometryFile path="test.msh"/>
<binaryFile path="test.bmsh"/>
<renderSettings>
 <imageWidth>800</imageWidth>
 <imageHeight>600</imageHeight>
 <resolutionPercentage>50</resolutionPercentage>
 <sspp>2</sspp>
 <threads>4</threads>
 <maxDepth>6</maxDepth>
 <renderingType>progressive</renderingType>
 <integrator>bidir</integrator>
 <filter type="gauss"/>
</renderSettings>
<environment>
 <type>black</type>
</environment>
<camera>
 <type>pinhole</type>
 <pos  x="1.709401" y="-2.004909" z="3.153121"/>
 <lookAt  x="-0.128831" y="0.025654" z="-0.032647"/>
 <up  x="0.031278" y="-0.010032" z="-0.131311"/>
 <angle>56.14497</angle>
</camera>
<materials>
 <mat>
  <type>perfectSpecular</type>
  <name>mat5</name>
  <color r="0.84444" g="1.00000" b="1.00000"/>
 </mat>
</materials>
<objects>
 <obj name="Mesh.004#.001">
  <prim name="Mesh.002$#.001_" type="mesh" instance="0"/>
 </obj>
</objects>
</scene>
```

PRÍLOHA P IV: OBSAH PRILOŽENÉHO CD

/praca – text DP vo formáte .doc a .pdf

/zdrojove kody – zdrojové kódy projektu(v zložke src) a projekty v CodeBlocks pre Windows a pre Linux (na preloženie je potrebná knihovňa wxWidgets a Boost). Pri prekladaní na 64-bitovej platformme je nutné odkomentovať riadok č. 42 v súbore src/includes.h (#define POINTER_SIZE_64).

Aktu8lne zdrojové kódy budú dostupné na: http://sourceforge.net/projects/fiiray/

/exportny skript – exportný skript pre Blender

/programova dokumentacia – programová dokumentácia projektu vygenerovaná programom Doxygen

/uzivatelska dokumentacia – užívateľská dokumentácia projektu (inštalácia a rozhranie skriptu)

/program_win32 – program preložený na Windows

/program_linux64 – program preložený na Ubuntu 64

/testovacie sceny – jednoduché scény na otestovanie

/Blender 2.50A2 – verzia Blenderu pre ktorú bol vytvorený skript na Windows aj na Linux