VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

OPTIMALIZOVANÉ SLEDOVÁNÍ PAPRSKU

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE MICHAL HRUBÝ
AUTHOR

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZOVANÉ SLEDOVÁNÍ PAPRSKU

OPTIMIZED RAY TRACING

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE MICHAL HRUBÝ
AUTHOR

VEDOUCÍ PRÁCE DOC. DR. ING. PAVEL ZEMČÍK
SUPERVISOR

BRNO 2008

Zadání bakalářské práce/5816/2007/xhruby16

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Hrubý Michal**
Obor: Informační technologie
Téma: **Optimalizované sledování paprsku**
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte dostupnou literaturu na téma ray tracing (sledování paprsku).
2. Ověřte možnosti dělení scény na podprostory a urychlení výpočtu "ray/object" intersection.
3. Navrhněte datové struktury programu tak, aby umožňoval implementaci různých technik sledování paprsku a mapování textur.
4. Implementujte software podle výše uvedeného popisu.
5. Vyhodnoťte dsažené výsledky a možnosti pokračování práce.

Literatura:

- Dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1-3 zadání

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zemčík Pavel, doc. Dr. Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Michal Hrubý**
Id studenta: 78981
Bytem: Lieskovská 162/7, 018 41 Dubnica nad Váhom
Narozen: 15. 02. 1986, Trenčín
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Optimalizované sledování paprsku
Vedoucí/školitel VŠKP: Zemčík Pavel, doc. Dr. Ing.
Ústav: Ústav počítačové grafiky a multimédií
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:

tištěné formě	počet exemplářů: 1
elektronické formě	počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....

....	....
Nabyvatel	Autor

Abstrakt

Táto práca sa zaoberá problematikou sledovania lúča a jej rôznymi optimalizáciami. Rozoberá matematický princíp sledovania lúča a hľadania priesečníka objektov scény s lúčom. Tiež analyzuje rôzne osvetľovacie modely a efektívne budovanie kd-stromu pre rozdelenie objektov scény. Práca sa zameriava na návrh a implementáciu multiplatformnej a jednoducho rozšíriteľnej aplikácie vykonávajúcej ray-tracing virtuálnej scény v jazyku C++.

Kľúčové slová

Sledovanie lúča, kd-stromy, mapovanie textúr, osvetľovacie modely.

Abstract

This bachelor’s thesis analyzes the problem of ray tracing and its optimizations. It describes mathematical principles of ray tracing and intersection searching of objects in scene and ray. It also analyzes different shading models and efficient building of kd-tree for dividing scene objects. The thesis focuses on design and implementation of multiplatform and easily extensible ray tracing application in C++.

Keywords

Ray-tracing, kd-trees, texture mapping, reflectance models.

Citace

Hrubý Michal: Optimalizované sledování paprsku. Brno, 2008, bakalářská práce, FIT VUT v Brně.

Optimalizované sledování paprsku

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Doc. Dr. Ing. Pavla Zemčíka. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……………………
Michal Hrubý
5. 5. 2008

Poďakovanie

Chcel by som poďakovať všetkým, ktorí mi akýmkoľvek spôsobom pomáhali pri vzniku tejto práce a najmä pánovi Doc. Dr. Ing. Zemčíkovi za konštruktívnu kritiku pri písaní tejto práce.

Obsah

Úvod

Počítače sú dnes súčasťou každodenného života a neoddeliteľnou súčasťou využívania počítačov je vytváranie a zobrazovanie umelých snímok, čím sa zaoberá počítačová grafika. Problémom v počítačovej grafike je generovanie fotorealistických obrázkov, v súčasných grafických kartách sa efekty ako sú tiene a zrkadlové odrazy riešia rôznymi trikmi aby bolo možné rasterizovať scénu v reálnom čase. Naproti tomu ray-tracing predstavuje jednoduchý algoritmus, ktorý prirodzene rieši efekty ako tiene, odrazy a lom svetla. Jeho nevýhodou je nízka rýchlosť, avšak so súčasnými počítačmi sa táto nevýhoda vytráca a čoskoro bude možné prevádzať ray-tracing interaktívne. Preto sa táto téma stále viac dostáva do pozornosti mnohých odborníkov a inžinierov.

Cieľom tejto práce je nahliadnuť do problematiky sledovania lúča a predstaviť rozšíriteľnú aplikáciu umožňujúcu renderovanie obrázkov pomocou ray-tracingu. Vo svojej práci uvádzam základné problémy tohto prístupu – spôsoby ako hľadať priesečníky lúča s objektmi, mapovanie textúr na základné geometrické primitívy. Tiež sa venujem problému osvetlenia a najmä dnes bežne používanému Phongovmu osvetľovaciemu modelu. Aby bola rýchlosť ray-tracingu dostatočná aj v komplexných scénach analyzujem metódy na delenie objektov do podpriestorov.

V druhej kapitole sa zaoberám návrhom aplikácie, použitím algoritmov na hľadanie priesečníkov lúčov a objektov v scéne a hlbšej analýze najpoužívanejších metód na delenie scény do podpriestorov.

Tretia kapitola popisuje samotný návrh tried a implementáciu aplikácie, kde je možné nájsť všetky potrebné informácie pre úspešné implementovanie ray-traceru.

V kapitole testovanie uvádzam najmä časy potrebné na renderovanie niekoľkých testovacích scén a hodnotím tieto výsledky.

Záverečná kapitola zhŕňuje získané poznatky, vrátane mojich prínosov a možných budúcich rozšírení projektu.

1 Úvod do problematiky

V tejto kapitole sú popísané základné princípy generovania fotorealistických obrázkov využitím metódy ray-tracing, táto problematika je však veľmi obšírna a preto sú tu popísané iba niektoré princípy relevantné pre vlastnú bakalársku prácu.

1.1 Ray-tracing

Ray-tracing (alebo tiež sledovanie lúča) je technika používaná na generovanie 2D obrázku z 3D scény [1]. Oproti iným technikám ako sú objektové vizualizačné metódy dokáže ray-tracing generovať obrázky s vysokou mierou fotorealizmu, najmä preto, že simuluje skutočné fyzikálne procesy, ktoré v prírode prebiehajú. Efekty, ktoré sa inými technikami dosahujú značne náročne, ako sú tiene a odrazy sú prirodzeným výsledkom ray-tracingu.

Jedným z problémov pri syntéze obrázkov je určenie správnej farby – zistenie tejto farby možno dosiahnuť spriemerovaním farieb svetiel ktoré dopadajú na daný pixel výsledného obrázku. Preto sa naskytuje otázka ako nájsť lúče, ktoré dopadajú na daný pixel. Jedným zo spôsobov môže byť sledovanie lúčov vysielaných zo všetkých zdrojov svetla v scéne a ich následné trasovanie v scéne – bohužiaľ pri použití tohto prístupu zistíme, že len veľmi malé percento lúčov má vplyv na výsledný obrázok, táto technika sa nazýva „photon mapping“ [2], a aj keď dokáže produkovať efekty s ktorými má spätný ray-tracing problémy (napríklad vzájomné difúzne odrazy), kvôli jeho výpočtovej náročnosti sa nepoužíva až tak často. Naproti tomu, spätný ray-tracing (anglicky označovaný tiež ako eye-based ray-tracing) sleduje lúče vysielané od pozorovateľa cez rovinu výsledného obrázku k jednotlivým objektom v scéne, ktoré môžu byť osvetlené zdrojmi svetla. Touto metódou je zaručené, že každý lúč bude mať vplyv na výsledný obrázok.

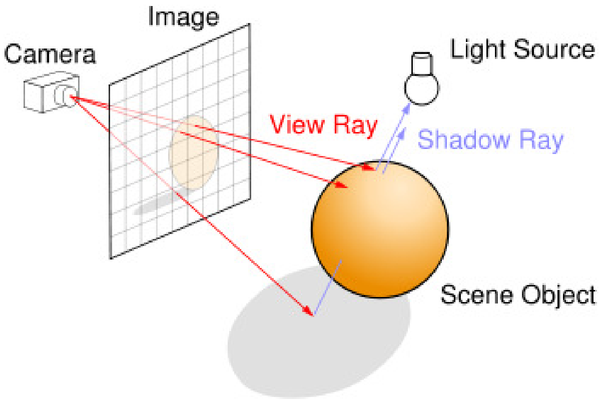
Spätný ray-tracing

Pre zistenie farby pixlu výsledného obrázku sa pri spätnom ray-tracingu používajú štyri základné typy lúčov: primárne (pixlové) lúče – nesú svetelnú informáciu priamo do oka pozorovateľa cez pixel výsledného obrázku, iluminačné alebo tiež tieňové lúče – pomocou nich možno zistiť osvetlenie povrchu objektu, odrazové lúče a lomené lúče (anglicky sa síce tieto lúče nazývajú „transparency rays“, avšak pri prechode lúču iným materiálom nastáva často lom svetla a preto sa nazývajú lomené), ktoré sa využívajú na zistenie svetelnej informácie objektov v prípade odrazu alebo lomu svetla. Samozrejme matematicky sú toto všetko rovnaké lúče, avšak z výpočtových a najmä optimalizačných dôvodov je lepšie použiť túto klasifikáciu [1].

Hlavnou časťou celého algoritmu je zistenie aké svetlo prichádza do daného bodu povrchu objektu a pokračuje do oka pozorovateľa. Toto svetlo sa rozdeľuje na dva komponenty: osvetlenie daného bodu inými zdrojmi svetla a vyžarovanie svetla z objektu v určitom smere. Určenie vyžarovaného svetla v danom bode spočíva v zistení osvetlenia v tomto bode a zvážení ako daným povrchom svetlo prechádza. Každý povrch časť svetla absorbuje, časť odráža a časť objektom prechádza – preto sa zavádzajú odrazové a lomené lúče.

Tieňové lúče

Tieňové lúče sa samozrejme používajú na zistenie tieňov, a to tak, že po zistení priesečníku objektu s lúčom, vyšleme z tohto bodu ďalší lúč a pokiaľ v jeho ceste nie sú žiadne nepriehľadné objekty, určite nejaké fotóny (a teda svetlo) prichádza do daného bodu. Ak sú v ceste lúča nepriehľadné objekty, potom žiadne svetlo neprichádza priamo zo zdroja svetla a teda daný bod je v tieni vzhľadom na toto svetlo (viď obrázok 1).



Obrázok 1: Tieňové lúče (prevzaté z [2])

Odrazové lúče

Pri zisťovaní svetla, ktoré je odrazené z určitého bodu do smeru dopadajúceho svetla (dopadajúcim je pri spätnom ray-tracingu myslený v skutočnosti odrazený lúč) sa používajú odrazové lúče pre daný bod a smer. Tento lúč nesie svetelnú informáciu k povrchu, ktorý bude dokonale odrazený do smeru dopadajúceho svetla [1]. Na určenie farby sledujeme lúče, až kým nenájdeme objekt z ktorého bolo svetlo vyžiarené. Na výpočet odrazeného lúča sa používa nasledujúci vzorec [1]:

$$R = I - 2N(I \cdot N)$$

$I - dopadajúci\ lúč$

$R - odrazený\ lúč$

$N - normála\ povrchu\ v\ danom\ bode$

Pre realistickejšie výsledky sa zväčša používa viac ako jeden odrazený (alebo aj lomený) lúč, pričom sú vysielané v rôznych, avšak vhodne zvolených smeroch a výsledná farba je priemerom farieb jednotlivých lúčov.

Lomené lúče

Tak ako existuje jeden smer z ktorého môže byť svetlo dokonale odrazené, tiež existuje jeden smer z ktorého je svetlo dokonale lomené. Pri výpočte lomeného lúča sa používa Snellov zákon, a preto je potrebné poznať relatívne indexy lomu materiálu z ktorého lúč prichádza a zároveň materiálu v ktorom sa lúč láme.

1.2 Výpočet priesečníkov objektov s lúčom

Najdôležitejšou časťou ray-tracingu je nájdenie priesečníku lúča s objektmi v scéne, preto sú v tejto kapitole prezentované algoritmy, poprípade algebraické rovnice na ich výpočet. V nasledujúcich podkapitolách sa uvažujú lúče definované ako:

$$R_{zdroj} \equiv R_O \equiv [X_O, Y_O, Z_O]$$
$$R_{smer} \equiv \overrightarrow{R_D} \equiv [X_D, Y_D, Z_D]$$
$$plat\acute{i}\ X_D^2 + Y_D^2 + Z_D^2 = 1$$

Pomocou tejto definície je možné lúč popísať nasledovne: $R(t) = R_O + \overrightarrow{R_D} * t$, pre $t > 0$ (jedná sa teda o polpriamku). Body pre ktoré je $t < 0$ sa nachádzajú pred počiatkom lúča. Taktiež bod v ktorom $t = 0$, sa väčšinou neuvažuje kvôli problémom s presnosťou pri výpočtoch s desatinnou čiarkou.

Priesečník lúča a gule

Guľa je definovaná pomocou stredu a polomeru:

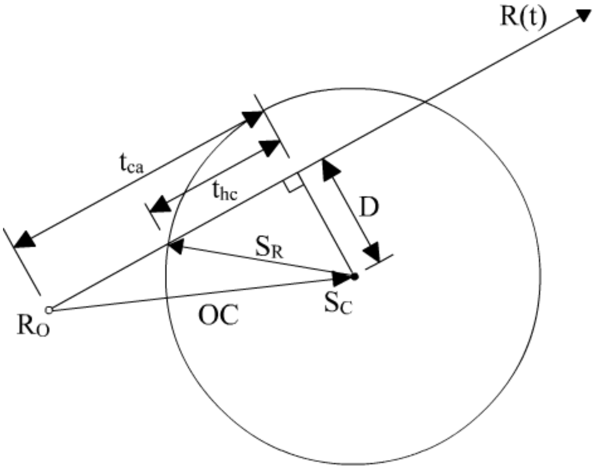
$$S_{stred} \equiv S_C \equiv [X_C, Y_C, Z_C]$$
$$S_{polomer} \equiv S_R$$

Pre body na kružnici (x, y, z) teda platí: $(x - X_C)^2 + (y - Y_C)^2 + (z - Z_C)^2 = S_R^2$. Na zistenie priesečníku sa dosadí rovnica lúča do rovnice gule a je riešená pre t. Geometricky možno vyjadriť riešenie nasledovne (znázornenie je na obrázku 2):

$$\overrightarrow{OC} = S_C - R_O$$
$$t_{ca} = OC \cdot \overrightarrow{R_D}$$
$$t_{2hc} = S_R^2 - |OC|^2 + t_{ca}^2$$
$$t = t_{ca} \pm \sqrt{t_{2hc}}$$

Pokiaľ $t_{2hc} < 0$ lúč guľu nepretína. Samozrejme stále platí, že korene musia byť väčšie ako 0, inak lúč pretína guľu pred svojim počiatkom. Potom, čo sú nájdené korene t, je možné vypočítať samotný priesečník ($r_i = [x_i, y_i, z_i]$) dosadením do rovnice lúča a pomocou neho vypočítať normálu povrchu v tomto bode:

$$N = \left[\frac{(x_i - X_C)}{S_R}, \frac{(y_i - Y_C)}{S_R}, \frac{(z_i - Z_C)}{S_R} \right]$$



Obrázok 2: Geometria priesečníka gule (podľa [1])

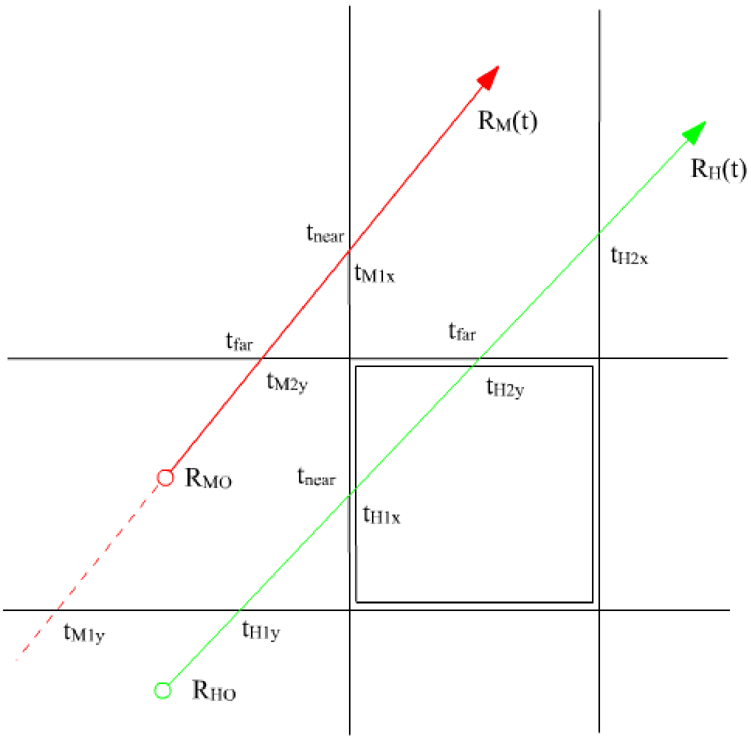
Priesečník lúča a osového kvádra

Jedna z najčastejšie používaných primitív v ray-tracingu je osový kváder (kváder ktorého steny sú rovnobežné s rovinami tvorenými súradnicovými osami) – takýto objekt sa používa nielen ako viditeľný objekt v scéne ale aj ako hraničný objekt (napríklad pri použití CSG, alebo delení scény na podpriestory). Kváder s požadovanými vlastnosťami sa dá popísať dvoma bodmi – spodnou a hornou medzou.

$$B_l = [X_l, Y_l, Z_l]$$
$$B_h = [X_h, Y_h, Z_h]$$

Princíp algoritmu, ktorý prezentovali Kay a Kajiya v [3] spočíva v hľadaní vzdialeností t_{near} a t_{far} k priesečníku lúča a roviny kvádra (znázornenie na obrázku 3). Nasledujúci algoritmus vracia logickú hodnotu TRUE ak lúč pretína kváder:

- Nastav $t_{near} = -\infty$, $t_{far} = \infty$.
- Pre každý pár rovín asociovaných s X, Y, Z (nasleduje znázornenie pre X):
    - Ak je smer lúča X_D rovný nule – lúč je rovnobežný s rovinami vráť FALSE.
    - Inak vypočítaj vzdialenosti priesečníka rovín:
    - $t_1 = (X_l - X_O) / X_D$, $t_2 = (X_h - X_O) / X_D$
    - Ak $t_1 > t_2$, vymeň t_1 a t_2.
    - Ak $t_1 > t_{near}$, nastav $t_{near} = t_1$.
    - Ak $t_2 < t_{far}$, nastav $t_{far} = t_2$.
    - Ak $t_{near} > t_{far}$, vráť FALSE.
    - Ak $t_{far} < 0$, lúč pretína kváder pred počiatkom – vráť FALSE.
- Všetky testy prešli - vráť TRUE.



Obrázok 3: Určenie priesečníku lúča a kvádra (podľa [1])

Priesečník lúča a trojuholníka

Pri určovaní priesečníku lúča s trojuholníkom treba najskôr zistiť priesečník lúča s rovinou trojuholníka a následne či tento bod je vnútri trojuholníka. Rovinu je definovaná normálou P_N a jej vzdialenosťou od počiatku súradnicového systému D. Parameter t, v ktorom lúč pretne rovinu je potom možné vyjadriť ako:

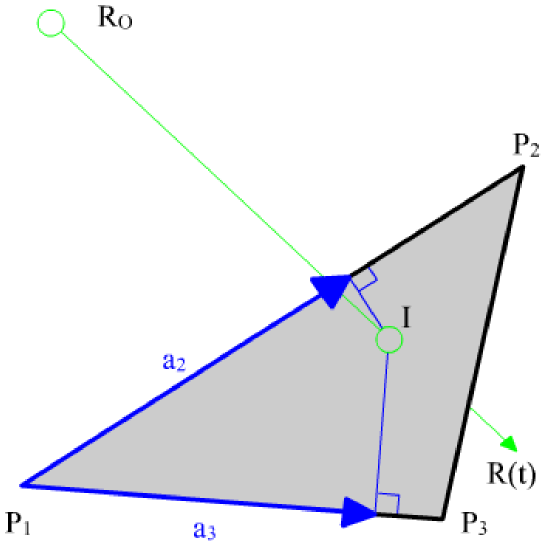
$$t = \frac{-(P_N \cdot R_O + D)}{P_N \cdot R_D}$$

Samozrejme pokiaľ vyjde t záporné, opäť platí, že lúč pretína rovinu pred svojim počiatkom a teda priesečník neexistuje. Keď poznáme priesečník ($I = R_O + R_D * t$) bodu s rovinou, zisťujeme či tento bod leží vnútri trojuholníka, čo je možné napríklad pomocou barycentrických súradníc.

Riešenie pomocou barycentrických súradníc

Priesečník I sa dá vyjadriť pomocou barycentrických súradníc ako: $I = a_1P_1 + a_2P_2 + a_3P_3$. Vzhľadom na to, že $a_1 + a_2 + a_3 = 1$, je možné tiež I zapísať: $I - P_1 = a_2 (P_2 - P_1) + a_3 (P_3 - P_1)$.

Po premietnutí trojuholníka do 2D nie je problém vypočítať a_2 a a_3, ktoré pokiaľ ležia vnútri trojuholníka musia spĺňať podmienky $a2 > 0$, $a3 > 0$ a zároveň $a2 + a3 <= 1$. Bližšie informácie viď [4]. Znázornenie možno nájsť na obrázku 4.



Obrázok 4: Parametrické vyjadrenie priesečníka (podľa [16])

1.3 Textúrovanie

Veľmi dôležitou časťou pri renderovaní je aj textúrovanie. Pomocou textúr priradzujeme rôznym povrchom premenlivé vlastnosti, okrem samotnej farby povrchu objektu to môže byť aj napríklad odraz okolia na povrchu (environment mapping), zmena geometrie objektu skutočným posunom bodov povrchu (displacement mapping), parametre povrchu ako je difúzia a reflexia, zmena normály povrchu (bump mapping) a iné [5].

Nanesenie textúry predstavuje hľadanie vhodnej projekčnej funkcie, pričom ak je povrch popísaný analytickou funkciou (čo je pri použití geometrických primitív v ray-tracingu veľmi časté) je možné použiť jej inverznú funkciu ak existuje. Ak povrch telesa nie je popísaný plochou rozvinuteľnou do roviny, potom po nanesení textúry na primitívu dochádza k skresleniu textúry (napríklad pri mapovaní na guľu).

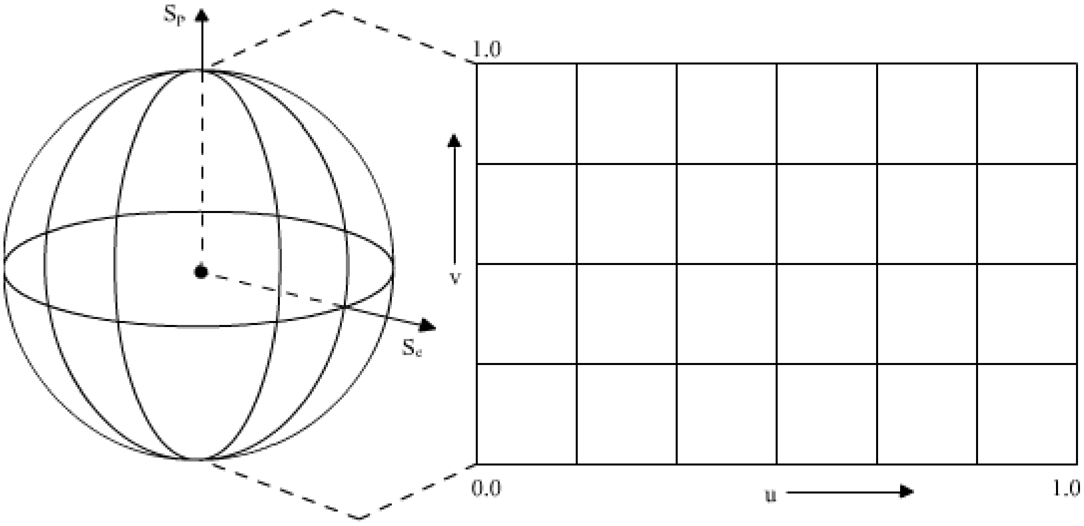
V nasledujúcich podkapitolách sú uvedené mapovanie povrchu gule a valca na parametre u a v ($u, v \in \langle 0,1 \rangle$), ktoré sú nutné na nanesenie textúry.

Inverzné mapovanie povrchu gule

Jedným so spôsobov ako previesť inverzné mapovanie povrchu gule predstavuje prevod kartézskych súradníc priesečníka lúča a gule na polárne súradnice tohto bodu. Definujeme si vektory S_p (jednotkový vektor od stredu gule k pólu) a S_e (jednotkový vektor od stredu gule k rovníku), pričom platí, že skalárny súčin týchto vektorov je rovný nule, a teda sú kolmé. Pokiaľ má priesečník normálu S_N, vyjadrujú sa parametre u a v nasledovne:

$$\phi = \arccos(-S_N \cdot S_p)$$
$$v = \phi / \pi$$

$$\theta = \frac{\arccos((S_e \cdot S_N)/\sin(\phi))}{2 * \pi}$$
$$u = \theta \quad pre\ (S_p \otimes S_e) \cdot S_N > 0$$
$$u = 1 - \theta \quad pre\ (S_p \otimes S_e) \cdot S_N \leq 0$$

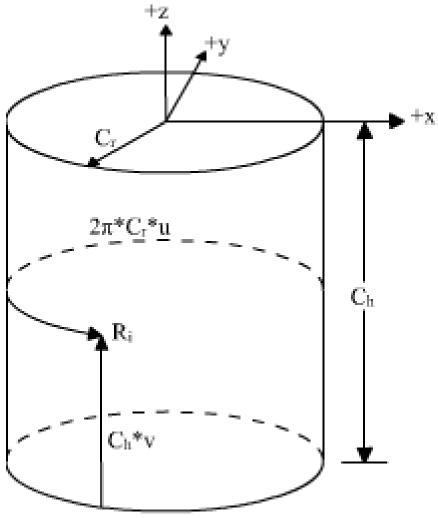


Obrázok 5: Inverzné mapovanie na povrch gule (podľa [1])

Inverzné mapovanie na valec

Pri inverznom mapovaní na povrch valca je výpočet ešte jednoduchší ako v prípade mapovania na guľu. Definujme valec s polomerom C_r a výškou C_h ako $X_C^2 + Y_C^2 = C_r^2$, pričom $0 \leq Z_C \leq C_h$. Ak poznáme priesečník Ri, je možné vypočítať parametre *u,v* (*u* začína v +X a smeruje k +Y, *v* začína v základni valca a smeruje k vrcholu, viď obrázok 6) pomocou:

$$v = Z_i / C_h$$
$$\theta = \frac{\arccos(X_i / C_r)}{2 * \pi}$$
$$u = \theta \quad pre\ Y_i \geq 0$$
$$u = 1 - \theta \ pre\ Y_i < 0$$



Obrázok 6: Inverzné mapovanie na povrch valca (podľa [1])

1.4 Osvetľovanie

Časť renderovania pri ktorom sa na základe uhla a vzdialenosti medzi bodom povrchu objektu a svetlom mení farba objektu kvôli vytvoreniu fotorealistického efektu sa nazýva osvetľovanie.

Osvetlenie daného bodu je funkciou rôznych vlastností povrchu a zdrojov svetla. Medzi používané osvetľovacie modely patria Lambert, Phong, Oren-Nayar, Torrance-Sparow, Cook-Torrance a iné.

Phongov osvetľovací model

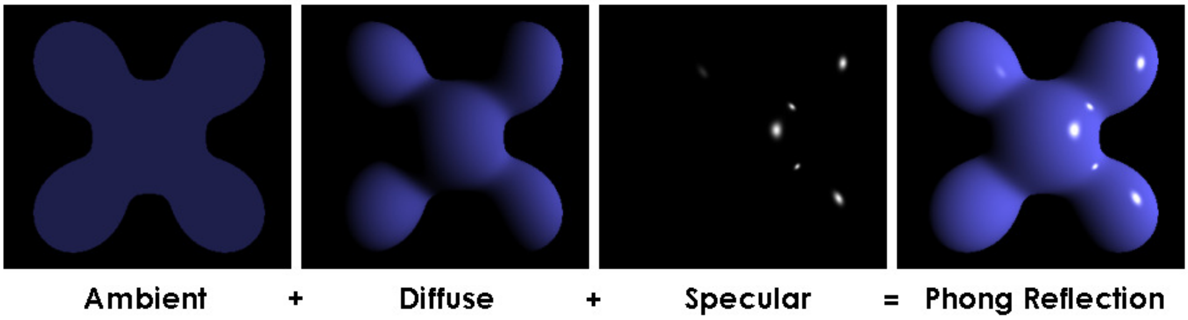
Phongov osvetľovací model kombinuje odrazovú zložku svetla z povrchov s Lambertovskou difúziou a pridáva ambientnú zložku, čo simuluje že slabé svetlo sa typicky nachádza takmer v celej scéne. Táto metóda je založená na pozorovaní, že pre lesklé povrchy je odrazová zložka veľmi malá a jej intenzita sa rapídne znižuje, kým pre matné povrchy je väčšia a znižuje sa pomalšie.

Pomocou Phongovho modelu sú charakterizované svetlá komponentmi i_S (intenzita odrazovej zložky) a i_D (intenzita difúznej zložky), globálna zložku ambientného svetla je i_A a jednotlivé materiály sú definované nasledujúcimi vlastnosťami:

- odrazová zložka k_S
- difúzna zložka k_D
- ambientná zložka k_A
- lesklosť α (pre väčšinu povrchov je lesklosť rovná približne 50, zrkadlové povrchy ju majú ešte vyššiu)

Ak poznáme normálu povrchu N v danom bode, smerový vektor L z bodu povrchu k svetlu, smerový vektor R ktorý je rovný ideálne odrazenému lúču a vektor V, ktorý smeruje k pozorovateľovi, definuje sa osvetlenie v danom bode ako:

$$I_p = k_A i_A + \sum_{svetlá} \left(k_D (L \cdot N) i_D + k_S (R \cdot V)^{\alpha} i_S \right)$$



Obrázok 7: Phongov osvetľovací model (prebrané z [17])

Blinn-Phongov osvetľovací model

Blinn prezentoval v [6] obmenu Phongovho osvetľovacieho modelu, kedy nahradil skalárny súčin vektorov *R* a *V* skalárnym súčinom vektorov *H* a *N*, kde *H* je takzvaný „halfway“ vektor, ktorý možno vyjadriť $H = (L+V) / | L+V |$, čo je výpočetne menej náročné, keďže nie je nutné zisťovať smer ideálne odrazeného lúča. Vzhľadom na to, že uhol medzi vektormi *H* a *N* je menší ako medzi *R* a *V*, treba prispôsobiť lesklosť α, aby produkoval podobné zvýraznenia ako Phongov model.

Zaujímavosťou je, že podľa [7] produkuje tento zjednodušený model vďaka použitiu halfway vektoru presnejšie zvýraznenia empiricky určených BRDF funkcií ako Phongov model pre niektoré typy materiálov.

1.5 Delenie na podpriestory

Pri ray-tracingu je z hľadiska rýchlosti kritické obmedziť počet objektov s ktorými je potrebné hľadať priesečník, a preto je dobrou stratégiou rozdeliť priestor scény na menšie časti – podpriestory, a pri prechode lúča hľadať priesečníky iba s objektmi, ktoré sa nachádzajú v podpriestoroch obsahujúcich samotný lúč. Narozdiel od využitia hierarchie obalových objektov, ktoré obaľuje skupiny objektov inými objektmi, ktoré umožňujú jednoduchšie testovanie priesečníku s lúčom, používa delenie na podpriestory inú filozofiu – síce tiež používa jednoduché hraničné telesá objektov v scéne na identifikovanie vhodných kandidátov na prienik, ale podpriestory sú budované rozdeľovaním priestoru obklopujúceho objekty, namiesto uvažovania objektov samotných [1]. Jednotlivé podpriestory sa označujú voxely a sú to zväčša osové kvádre, avšak je možné použiť aj iné tvary. Voxely sa konštruujú pred samotným procesom ray-tracingu, tak aby obsahovali celú scénu a spôsob akým sú voxely vytvárané vedie k najvýznamnejším variáciám – hlavným rozdielom je či sú vytvárané voxely uniformne alebo neuniformne – obom metódam sa venujem v nasledujúcich podkapitolách.

Neuniformné delenie podpriestorov

Neuniformné techniky delenia podpriestorov rozdeľujú scénu na oblasti s rozličnými veľkosťami aby sa prispôsobili charakteristikám prostredia scény – táto metóda umožňuje, aby sa priestor delil viac v miestach, kde sa nachádza väčší počet objektov a na druhej strane dovoľuje, aby veľké voxely pokrývali priestor ktorý je riedko zaplnený objektmi [1].

Jedným zo spôsobov ako neuniformne deliť priestor je pomocou oktalových stromov, ktoré sa konštruujú rekurzívnym delením pravouhlých telies na osem menších, až pokiaľ výsledné uzly stromu nespĺňajú určitú podmienku. Problémom oktalových stromov je to, že väčšinou jednotlivé uzle nenesú ukazovatele na všetkých osem ďalších uzlov, a preto sa používa napríklad unikátne označenie jednotlivých uzlov (čísla 1-8 pre každý uzol s prefixom čísla rodičovského uzlu), čo však prináša so sebou potrebu použitia hašovacej tabuľky pri prechode stromom. Znázornenie oktalového stromu je na obrázku 8.

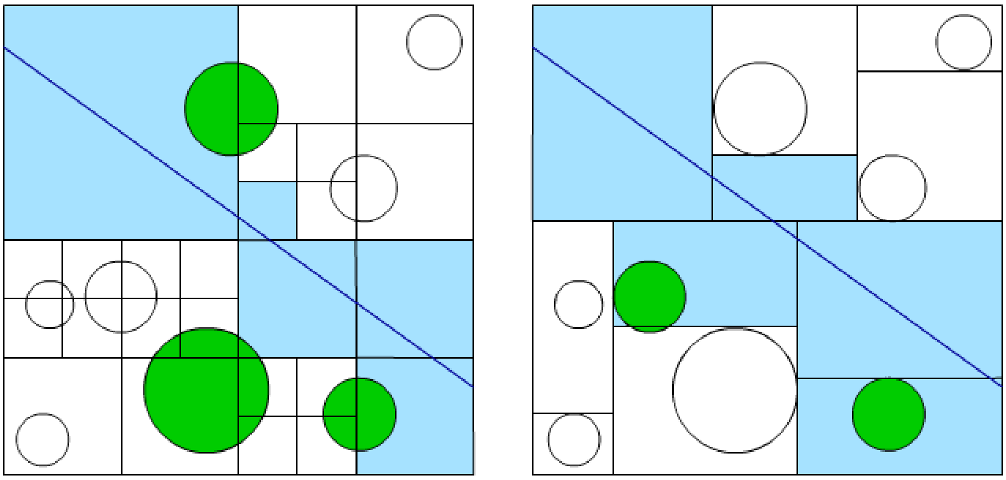
Uniformné delenie podpriestorov

Použitie uniformného delenia podpriestoru, kedy sú voxely organizované v pravidelnej 3D mriežke prináša dva zásadné rozdiely oproti neuniformnému deleniu – delenie je úplne nezávislé na štruktúre scény, avšak k voxelom, cez ktoré lúč prechádza, je možné efektívne pristupovať a prechádzať cez ne. Pokiaľ je rozhodujúca rýchlosť prístupu k voxelom môže tento prístup priniesť značný výkonnostný zisk oproti neuniformnému deleniu. V 3D mriežke sa k prechodu používa obdoba DDA algoritmu – takzvaný 3DDDA. Ďalšou výhodou je efektívna konštrukcia kandidátnych zoznamov jednotlivých uzlov [1]. Príklad uniformného delenia priestoru je na obrázku 9.

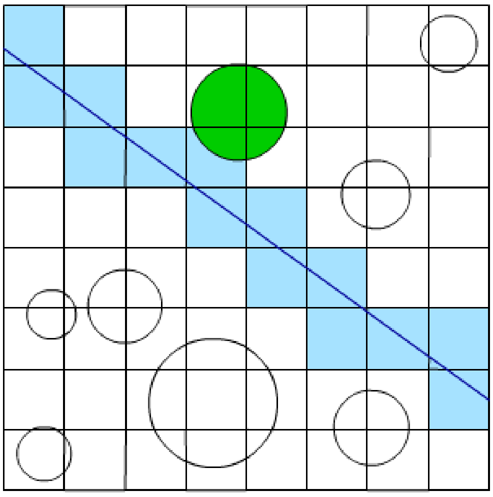
Výhody tohto prístupu však nie vždy kompenzujú jeho slabú prispôsobivosť, pričom hlavnou nevýhodou je náročné prechádzanie prázdneho priestoru a ďalšou vysoká pamäťová náročnosť so zvyšujúcim sa „rozlíšením“ mriežky.

Ďalšie metódy delenia podpriestorov

Mimo oktalových stromov a uniformných mriežok sa používajú aj iné delenia – napríklad BSP stromy a najčastejšie ich špeciálny typ – *k*-dimenzionálny strom (skrátene *k*d-strom), ktorý rozdeľuje priestor iba na dve časti, pričom jednou z informácií ktoré uzol *k*d-stromu nesie je súradnicová osa, ktorá bola použitá na delenie a ukazovatele na ľavý a pravý uzol. Pri použití *k*d-stromu je hlavným problémom algoritmus na určenie správnych deliacich pozícií, čo je najkritickejšie pre správne fungujúci kd-strom. Na určovanie deliacich pozícií sa využívajú rôzne heuristiky. Príklad kd-stromu je znázornený na obrázku 8.



Obrázok 8: Znázornenie oktalového (vľavo) a kd-stromu (vpravo) premietnutých do 2D, zelenou farbou sú označené objekty, ktorých priesečník s lúčom bude hľadaný



Obrázok 9: Priestor rozdelený do pravidelnej mriežky (podľa [1])

2 Analýza a návrh riešenia

Základnou funkcionalitou programu by malo byť vygenerovanie obrázku z popisu objektov v 3D scéne s čo najväčšou mierou fotorealizmu. Implementovaná aplikácia by mala spracovať definíciu scény a na základe použitých objektov, ich materiálov a textúr, polohy svetiel a ďalších faktorov za pomoci rôznych optimalizácií rasterizuje pohľad pozorovateľa a výsledný obrázok uloží do súboru. Vzhľadom na rozšíriteľnosť ray-tracingového algoritmu by mal byť program plne objektovo orientovaný a umožňovať jednoduché doplnenie funkcionality v prípade potreby.

Základným formátom súboru, ktorý bude program podporovať je PPM (Portable Pixmap) verzie P6 (binárne dáta s 24 bitmi na pixel). Tento formát bol zvolený kvôli jeho jednoduchej implementácií a podpore na rôznych platformách.

2.1 Základný algoritmus

Jadro ray-traceru predstavuje veľmi jednoduchý rekurzívny algoritmus. Pozostáva z týchto krokov:

- Nájdi najbližší priesečník medzi sledovaným lúčom a objektmi scény
- Ak priesečník neexistuje, vráť farbu pozadia
- Vyšli tieňové lúče z priesečníku ku všetkým zdrojom svetla
- Vyhodnoť súčet osvetľovacích modelov v priesečníku
- Ak nie je prekročená hĺbka rekurzie, vyšli odrazený lúč a lomený lúč z priesečníku, inak vráť farbu osvetlenia
- Vráť súčet farieb osvetlenia, odrazeného a lomeného lúča

2.2 Optimalizácie

Napriek rýchlosti súčasných počítačov je ray-tracing veľmi náročný – pre typickú scénu v rozlíšení 1024x768 sa môže sledovať niekoľko miliónov lúčov (tento počet závisí na tom, aký presný výsledok požadujeme), pričom každý lúč prechádza scénou, testuje priesečníky s objektmi, aplikuje na jednotlivé body osvetlenie atď. Preto je nutné používať rôzne optimalizácie aby bolo renderovanie dokončené v rozumnom čase. V tejto kapitole prezentujem rôzne optimalizačné techniky, ktoré využijem v rámci svojej implementácie.

Využitie trojuholníkových modelov

S ohľadom na architektúru súčasných grafických kariet je prirodzené, že veľká časť používaných 3D modelov je väčšinou v trojuholníkovej reprezentácií, preto som sa rozhodol podporovať najmä trojuholníkové modely, naviac hľadanie priesečníku trojuholníka a lúču je veľmi rýchle.

Program bude podporovať načítanie trojuholníkového modelu z formátu PLY, ktorého štruktúra je popísaná v [11] a zároveň formát 3DS, v ktorom je na internete obrovský počet trojuholníkových modelov (formát je analyzovaný v [12]). Vzhľadom na veľký počet objektov sú trojuholníkové modely tiež vhodné na testovanie algoritmov deliacich scénu na podpriestory.

Algoritmy na hľadanie priesečníkov

V odborných článkoch bolo publikovaných veľké množstvo algoritmov zaoberajúcich sa optimalizovaným hľadaním priesečníkov lúča s rôznymi typmi telies.

Pre hľadanie priesečníku lúča a gule budem používať geometrické riešenie, ktoré bolo bližšie popísané v kapitole 1.2.

Efektívne hľadanie priesečníku lúča a kvádra bolo publikované v [13]. Ide o algoritmus veľmi podobný algoritmu z kapitoly 1.2 s malou optimalizáciou, kedy sa použijú iba tri delenia na získanie prevrátenej hodnoty smerového vektoru (keďže delenie je jedna z najpomalších CPU operácií) a následne sa používa iba násobenie s touto hodnotou.

Doktor Wald sa mimo iného zaoberal optimalizovaním hľadania priesečníku lúča a trojuholníku [14], pričom jeho riešenie využíva barycentrické súradnice a množstvo predpočítaných dát ako je dominantná osa trojuholníku a vektory premietnutých hrán trojuholníku.

V [15] bol tiež predstavený efektívny algoritmus na určenie, či sa trojuholník pretína s osovým kvádrom.

Delenie na podpriestory

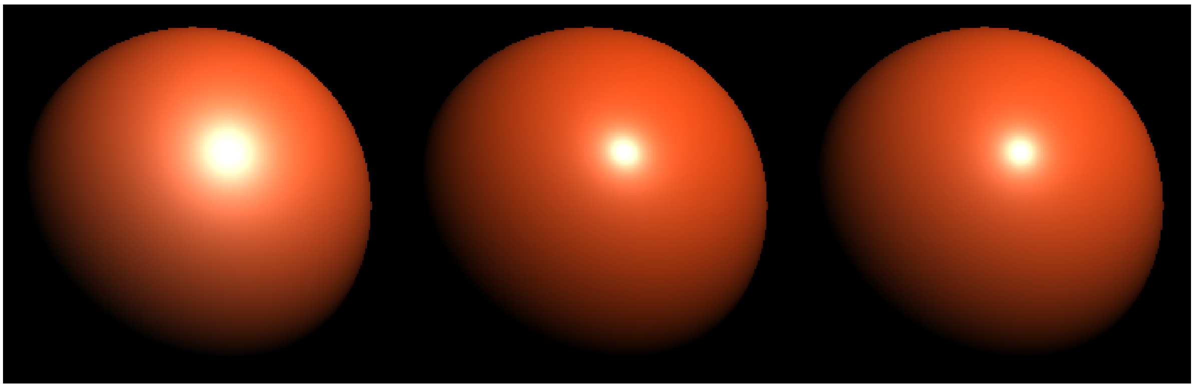
Problematike delenia na podpriestory pre využitie v ray-tracingu sa venoval doktor Havran v [8]. Z jeho záverov vyplýva, že vo väčšine prípadov je najvýhodnejšie použiť kd-strom, aj keď v niektorých prípadoch dosahuje lepšie výsledky uniformná mriežka, avšak iba v umelo vytvorených pravidelných scénach. Napriek tomu rozdiely vo výkonnosti vo všetkých prípadoch sú veľmi malé. Preto som sa rozhodol implementovať výlučne kd-strom pre komplexné scény.

Najčastejšie používaná heuristika pri použití kd-stromu je založená na výpočte povrchu jednotlivých hraničných telies (označovaná tiež SAH – „Surface Area Heuristic“), nevýhoda tohto prístupu je, že čas potrebný na vytvorenie stromu je celkom vysoký – preto som sa rozhodol využívať pri konštrukcií stromu zjednodušené testy prieniku objektu scény a hraničného telesa uzlu stromu, kedy sa v skutočnosti testuje prienik hraničného telesa objektu a hraničného telesa uzlu. Až keď sa už zistí, že uzol kd-stromu bude list, vykoná sa skutočný prienik objektu scény a hraničného telesa uzlu, kedy je možné objekt odstrániť zo zoznamu objektov daného uzlu ak prienik nenastane.

Osvetľovacie modely

Na každý bod je pri ray-tracingu aplikovaný osvetľovací model, preto je rozumné, aj keď nie nevyhnutne nutné, použiť optimalizácie aj pri tomto procese.

Zväčša čas strávený výpočtom osvetľovacieho modelu je zanedbateľný oproti samotnému prechodu podpriestorovým stromom a hľadaniu priesečníkov, avšak napríklad použitie Blinn-Phongovho osvetľovacieho modelu (viď kapitola 1.4.2) neprináša len optimalizáciu ale aj vyššiu mieru realizmu odrazových zvýraznení [7]. Blinn-Phongov model však využíva „halfway" vektor, čo spôsobuje, že odrazové zvýraznenia sú pri použití tohto modelu väčšie, avšak ako je prezentované v [10], jednoduchým riešením tohto problému je vynásobenie parametru lesklosti štyrmi. Rozdiely medzi Phongovým a Blinn-Phongovým modelom je vidieť na obrázku 10.



Obrázok 10: Porovnanie Blinn-Phongovho a Phongovho modelu. (vľavo Blinn-Phong, v strede Phong, vpravo Blinn-Phong so 4-násobnou lesklosťou)

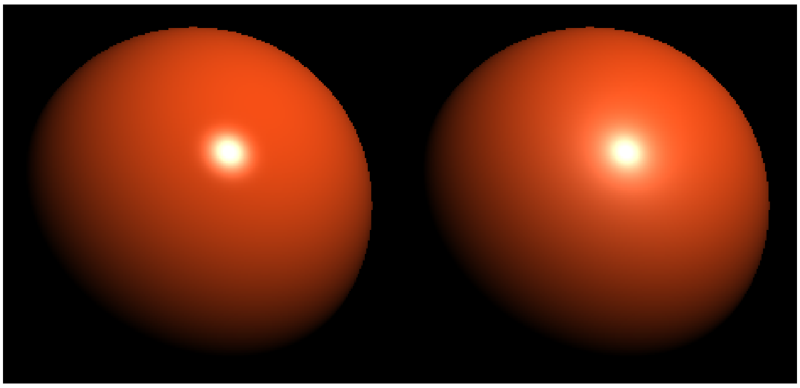
Schlickova optimalizácia

Pri vyhodnocovaní Phongovho osvetľovacieho modelu sa na výpočet odrazovej zložky používa umocnenie, pričom exponent býva rádovo okolo 50 a viac, preto Schlick navrhol (publikované v [10]) použiť namiesto mocniny aproximačnú funkciu:

$$S_n(t) = \frac{t}{n - nt + t}$$

V tomto prípade t je skalárny súčin vektorov R a V z pôvodného Phongovho modelu a n je konštanta lesklosti materiálu (porovnanie modelov viď obrázok 11).

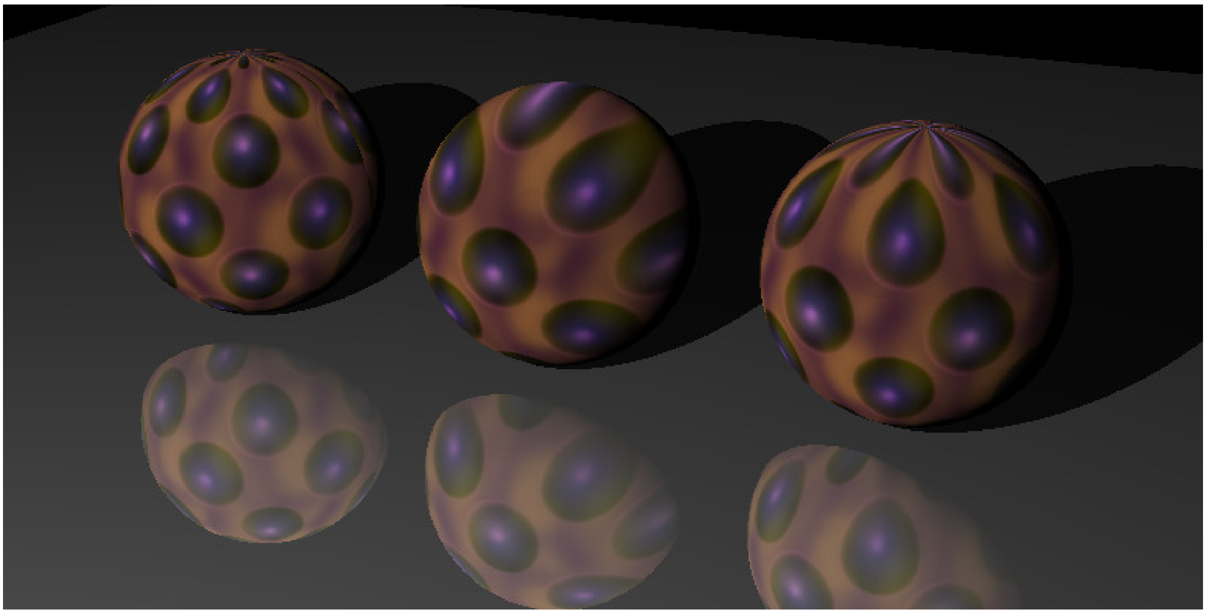
Je síce pravda, že táto optimalizácia neprinesie žiadny úžitok pre scény s malým počtom objektov s odrazovými zvýrazneniami, avšak jednoduchý test rýchlosti výpočtu päťdesiatej mocniny náhodnej hodnoty trvá pri šiestich miliónoch výpočtov o sekundu dlhšie ako pri použití Schlickovej aproximačnej funkcie (testované na AMD Opteron 2216). Stále, táto optimalizácia sa prejaví iba ak scéna obsahuje veľký počet zrkadliacich objektov a na výslednom obrázku zaberú objekty s odrazovými zvýrazneniami veľkú plochu.



Obrázok 11: Porovnanie Phongovho modelu a Schlickovej aproximácie

2.3 Textúrovanie

Program bude podporovať štandardné typy mapovania textúr – sférické, cylindrické a rovinné. Ich porovnanie vidno na obrázku 12.



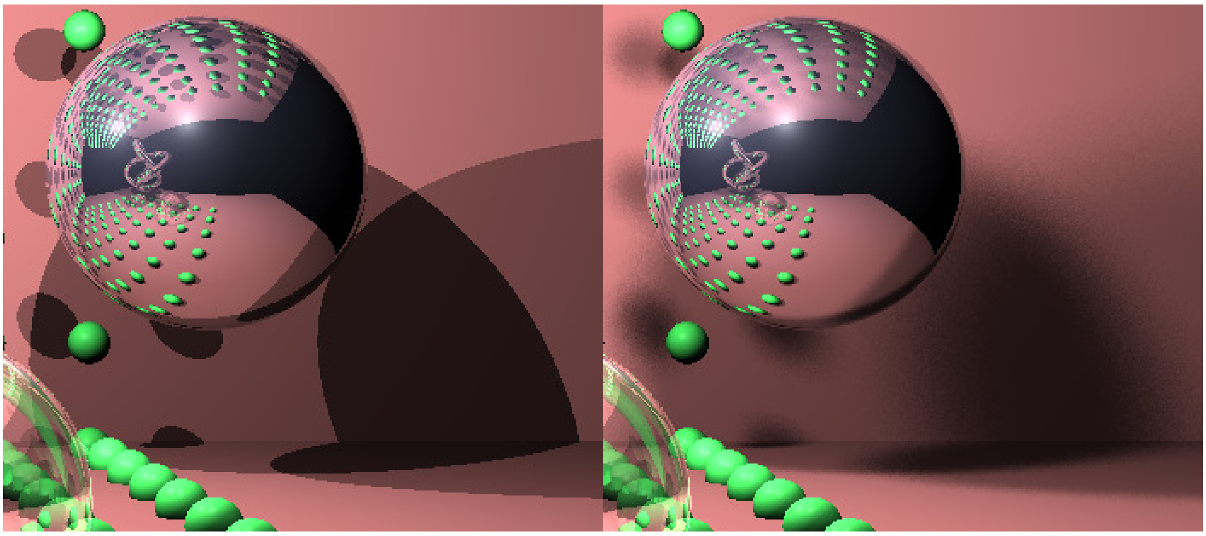
Obrázok 12: Mapovanie textúr (vľavo sférické mapovanie, v strede rovinné, vpravo cylindrické)

2.4 Monte-Carlo metódy

Pre „pokročilé“ funkcie sa pri ray-tracingu často využíva metóda Monte-Carlo, ktorá sa využíva keď je nemožné alebo výpočetne príliš náročné použiť deterministický algoritmus, preto sa použije náhodné vzorkovanie, a keby sa počet vzoriek rovnal nekonečnu dosiahneme presný výsledok.

Monte-Carlo metódy sa je možné využiť napríklad na určenie mäkkých tieňov – vyšle sa určitý počet tieňových lúčov do náhodných pozícií v rámci svetla a výsledné osvetlenie sa rovná priemeru všetkých tieňových lúčov. Rovnako je možné použiť náhodné vzorkovanie na difúzne odrazy, antialiasing atď.

Problémom Monte-Carlo metód je to, že pre kvalitný výsledok treba použiť veľký počet vzorkov – pri ray-tracingu teda zvyčajne lúčov, čo môže mať obrovský dopad na rýchlosť celého programu, avšak produkujú obrázky s väčšou mierou fotorealizmu. Porovnanie obrázkov vygenerovaných s použitím tejto metódy a bez nej je možné vidieť na obrázku 13.



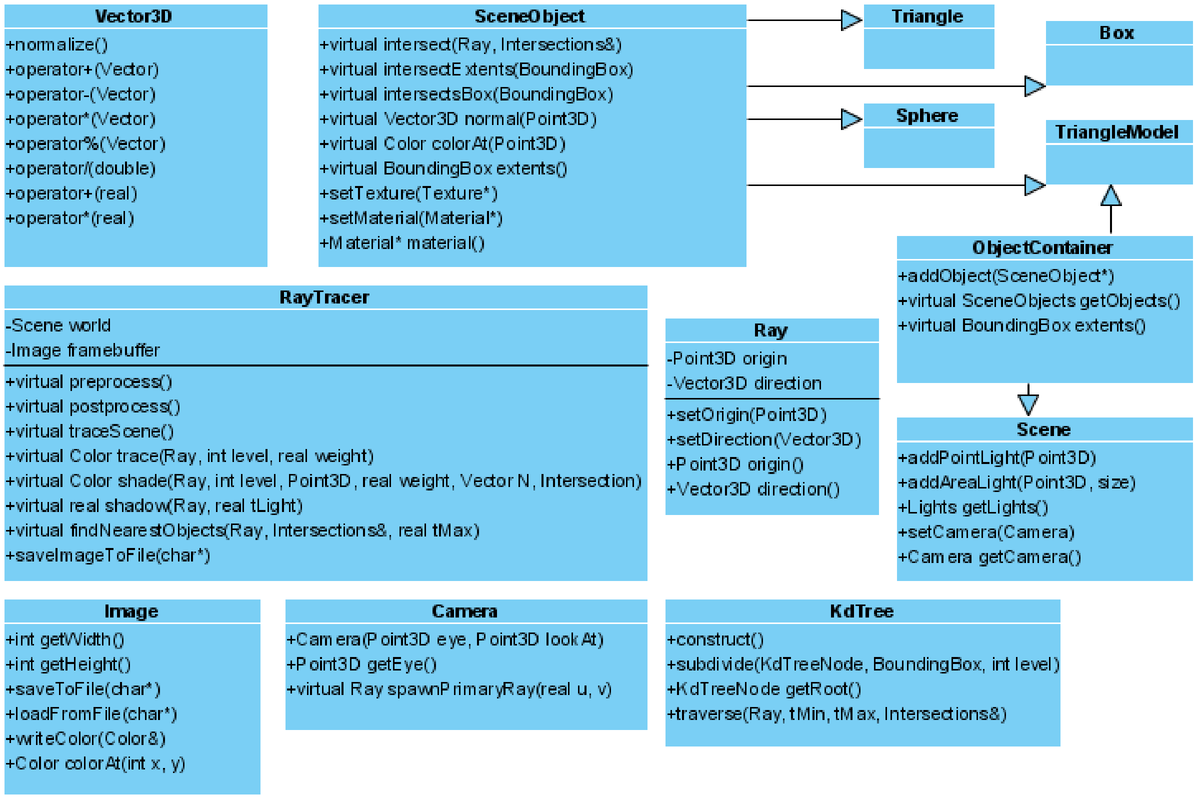
Obrázok 13: Porovnanie tieňov. Pravá scéna využíva Monte Carlo metódu na mäkké tiene (použitých 64 tieňových lúčov na priesečník)

3 Implementácia a testovanie

S ohľadom na vykonanú analýzu a návrh riešenia som program implementoval. Táto kapitola obsahuje detailnejší popis implementácie a charakteristiku použitých funkcií.

3.1 Objektový návrh

Pri návrhu jednotlivých tried programu som sa snažil zamerať na rozšíriteľnosť, preto využívam dedičnosť a kompozíciu. Prehľad najzákladnejších tried spolu s podstatnými verejnými metódami je možné vidieť na obrázku 14.



Obrázok 14: Prehľad tried

Jadro programu predstavuje trieda *RayTracer*, pracujúca s triedou *Scene*, v ktorej sú uložené popisy jednotlivých objektov scény a zároveň zdroje svetla. Trieda *RayTracer* poskytuje základné metódy ako sú sledovanie jednotlivých lúčov, hľadanie najbližšieho priesečníku s objektmi v scéne, aplikovanie osvetlenia atď. Pre potreby rozšírení sú takmer všetky metódy virtuálne a je ich možné nahradiť vlastnými algoritmami v odvodených triedach. Jednotlivé objekty scény sú odvodené od abstraktnej triedy *SceneObject* a musia implementovať metódy *intersect* a *normal*, taktiež by mali správne nastaviť rozmery svojho hraničného telesa (chránená premenná *extent*). Voliteľne môžu implementovať aj metódu *intersectsBox*, čo pomôže pri budovaní kd-stromu.

3.2 Vektorová trieda

Na prácu s vektormi a celkovú reprezentáciu bodov v trojrozmernom priestore slúži trieda *Vector3D*, ktorú je možné skompilovať v dvoch variantoch. Pri použití makra USE_SSE sa využije rozšírenie prekladača gcc, kedy operácie s touto triedou budú využívať SIMD rozšírenia ktoré sú dostupné v súčasných procesoroch, avšak SSE inštrukcie vyžadujú aby všetky triedy, ktoré sú zložené z jedného alebo viacerých členov triedy *Vector3D* boli zarovnané na pamäťových adresách ktoré sú deliteľné bez zvyšku šestnástimi – toto obmedzenie však kompilátor štandardne nedodržuje pokiaľ alokuje objekt na halde, preto pri použití makra USE_SSE sa predefinujú operátory new tried *SceneObject* a *Texture* na verziu, ktorá používa na alokáciu funkciu *posix_memalign* umožňujúcu nadefinovať zarovnanie alokovaného objektu. Preto pokiaľ procesor nepodporuje SSE inštrukcie alebo operačný systém nepodporuje funkciu *posix_memalign* je potrebné nedefinovať makro USE_SSE. Taktiež, niekedy réžia využívania SSE inštrukcií prevyšuje výkonnostný zisk, tak je možné makro vypnúť, čo spôsobí, že sa bude používať klasická reprezentácia vektoru troma desatinnými číslami. Samozrejme navonok má trieda *Vector3D* stále rovnaké rozhranie bez ohľadu na to, či sa využíva verzia podporujúca SSE alebo nie.

3.3 Vytvorenie scény

Scéna, ktorú bude program renderovať je popísaná v samostatnom zdrojovom súbore *scene_def.cpp*, kde je pripravených niekoľko vzorových scén, ktoré som konštantne využíval pri testovaní programu. Každá scéna je definovaná unikátnym číslom a výber konkrétnej scény spočíva v nastavení premennej *currentScene*.

Jednotlivé objekty scény sa vytvárajú z tried, ktoré sú odvodené z abstraktnej triedy *SceneObject* (teda objekty tried *Sphere*, *Box* alebo *Triangle*) jednoduchým zavolaním konštruktoru objektu, s prípadným nastavením textúry objektu zavolaním metódy *setTexture*.

Textúru je možné priradiť objektu vytvorením inštancie objektu ktorý dedí z abstraktnej triedy *Texture*, čiže buď použitím inštancie objektu *SimpleTexture*, čo umožní nastavenie jednej konkrétnej farby pre celý objekt, alebo použitím inštancie objektu *BitmapTexture*, ktorá má ako parameter inštanciu triedy *Image* a teda dokáže načítať bitovú mapu zo súboru formátu PPM. Pokiaľ užívateľ nenastaví objektu textúru, bude použitá implicitne biela farba.

Objektu typu *SceneObject* je tiež možné definovať materiál (metódou *setMaterial*). Materiál sa definuje objektom triedy *Material* s niekoľkými parametrami ako sú koeficienty pre Phongov osvetľovací model, teda difúzna zložka, odrazová zložka a lesklosť (ambientná zložka je definovaná globálne pre všetky objekty v metóde *RayTracer::shadow*), ďalej je tu parameter drsnosti povrchu, ktorý používa Oren-Nayar difúzny shader a nakoniec sú to koeficienty pre samotný ray-tracer, teda miera zrkadlenia a priepustnosti svetla spolu s indexom lomu a parameter difúzneho zrkadlenia.

Po vytvorení objektu scény a nastavení jeho vlastností stačí už len zavolať metódu *Scene::addObject*, ktorá má ako parameter novovytvorený objekt. Na pridanie zdroja svetla do scény slúžia metódy *Scene::addPointLight* a *Scene::addAreaLight*, ktoré vytvoria na zadaných súradniciach bodový respektíve plošný zdroj svetla, a to vytvorením inštancie objektu *Light* a jeho následným pridaním do zoznamu svetiel v scéne.

Pre nastavenie polohy pozorovateľa slúži metóda *Scene::setCamera* s parametrom referencie na triedu typu *Camera*, kde sa dá nastaviť samotná poloha pozorovateľa v scéne, bod na ktorý sa pozorovateľ díva a uhol záberu („field of view").

Načítanie trojuholníkového modelu

Trieda *ModelLoader* zaisťuje načítanie trojuholníkových modelov zo súborov formátu 3DS poprípade PLY. Na vytvorenie objektu triedy *ModelLoader* je potrebné poskytnúť objekt odvodený z triedy *ObjectContainer*, čo znamená buď inštanciu triedy *Scene* alebo *TriangleModel*, pri načítavaní modelu bude použitá metóda *ObjectContainer::addObject* na pridanie jednotlivých trojuholníkov do zoznamu objektov týchto tried. *ModelLoader* poskytuje metódy na zmenu pozície a veľkosti načítaného modelu (*setTranspose* a *setScale*), a zároveň je možné definovať textúru a materiál (metódy *setTexture* a *setMaterial*), ktoré budú taktiež nastavené jednotlivým trojuholníkom obsiahnutých v modeli.

Pomocná funkcia triedy *ModelLoader* je vytvorenie roviny pomocou dvoch trojuholníkov na zadaných súradniciach metódou *createPlane*, ktorej použitie prináša značné urýchlenie oproti využitiu triedy *Box* na takýto účel.

Použitie inštancie triedy *TriangleModel*, čo je vlastne virtuálny objekt scény obsahujúci trojuholníky z modelu, je vhodné na experimentálne účely, pretože pre scénu sa javí ako jediný samostatný objekt a vnútorne používa vlastný kd-strom na určenie priesečníku s niektorým z trojuholníkov.

3.4 Samotný ray-tracing

Najpodstatnejšou časťou celého programu je trieda *RayTracer*, ktorá vykonáva samotné renderovanie a následné uloženie do obrázku.

Na vytvorenie objektu triedy *RayTracer* je nutné poskytnúť základné parametre, čo sú referencia na objekt typu *Scene* – teda scéna, ktorá sa bude renderovať, ďalej rozmery výsledného obrázku v pixloch a maximálnu hĺbku rekurzie samotného algoritmu. Po vytvorení inštancie stačí zavolať metódu *traceScene*, ktorá vykoná samotný ray-tracing a uloží výsledné dáta do pamäte, takže po ukončení renderovania treba využiť metódu *saveImageToFile*, čo vykoná metódu *Image::saveToFile* interného objektu triedy *Image*. Program štandardne generuje obrázok s názvom obsahujúcim dátum a čas spustenia programu, takže výsledný obrázok bude uložený do aktuálneho adresára s názvom napríklad „April 10 091453.ppm“ (čo znamená, že obrázok sa začal renderovať 10.apríla o 9:14:53).

Metóda *traceScene* volá metódu *preprocess*, ktorú si môžu nadefinovať odvodené triedy a môže byť použitá napríklad na dvojfázové spracovanie scény, konštrukciu podpriestorov atď. Po skončení tejto fáze nasleduje cyklus, kde sa pre všetky pixely výsledného obrázku pomocou metódy *Camera::spawnPrimaryRay* generujú primárne lúče do scény a spätne sa trasujú metódou *trace*, ktorá vracia výslednú farbu lúča. V metóde *traceScene* je tiež možné využiť makro SQUARE_RENDER, vďaka ktorému sa budú lúče generovať nie lineárne ale pomocou malých mriežok (veľkosti 16x16), čo dokáže prekvapivo zvýšiť rýchlosť renderovania o niekoľko percent keďže generované lúče sú viac koherentné, a tak je väčšia šanca že potrebné dáta sú už v cache pamäti procesora.

Vnútri metódy *trace* sa hľadá najbližší priesečník lúča a objektov scény volaním *findNearestObjects*, ktorá testuje či lúč pretína ktorýkoľvek z objektov scény metódou *SceneObject::intersect*. Pokiaľ sa priesečník nájde, vypočíta sa normála v tomto bode (pomocou *SceneObject::normal*) a zavolá sa metóda *shade*, aplikujúca na daný bod osvetľovací model, pričom vysiela tieňové, odrazové a lomené lúče do scény a rekurzívne na ne volá *trace*.

Aplikovanie osvetľovacieho modelu

Metóda *shade* triedy *RayTracer* vyhodnocuje osvetlenie v danom bode v prvej fáze vysielaním jedného alebo viacerých tieňových lúčov (na základe typu použitých svetiel v scéne) ku všetkým zdrojom svetla, na čo slúži metóda *shadow* – tá pre konkrétny tieňový lúč vracia hodnotu 1 pokiaľ bod z ktorého lúč vychádza nie je v tieni žiadneho iného objektu, poprípade konštantu ambientného osvetlenia ak má lúč v ceste nepriehľadný objekt alebo inú hodnotu ak je v ceste lúču polopriehľadný objekt. Na vyhľadanie objektov medzi zdrojom svetla a počiatkom tieňového lúča tiež používa skôr zmienená metóda *findNearestObjects*. Pokiaľ boli použité v scéne plošné zdroje svetla, použije sa metóda Monte-Carlo – vysiela sa niekoľko tieňových lúčov (maximálne 64) do náhodných súradníc plošne definovaného svetla.

Po vyhodnotení tieňa sa v danom bode aplikuje difúzny shader. Ak je definované makro PHONG_ONLY použije sa klasický Lambertovský model, v prípade že makro PHONG_ONLY nie je definované, používa sa Oren-Nayar shader, ktorý berie do úvahy aj parameter drsnosti materiálu. Farba samotného materiálu sa zisťuje metódou *SceneObject::colorAt*, ktorá volá metódu *Texture::colorAt* textúry asociovanej s objektom alebo vráti bielu farbu ak textúra nie je definovaná.

V ďalšej fáze sa vyhodnocuje odrazová zložka Phongovho modelu, kedy sa k výslednej farbe pričíta biela farba, v závislosti na definícií makra PHONG_ONLY sa na výpočet použije buď klasický Phongov model, alebo Blinn-Phongov.

V poslednej fáze sa generujú odrazové a lomené lúče ak nebola prekročená hĺbka rekurzie, pričom ak mal materiál nastavené difúzne odrazy, tak sa opäť použije metóda Monte-Carlo na určenie výslednej farby odrazu, pri čom sa používa až 128 odrazených lúčov, ktoré majú približný smer ideálne odrazeného lúča. Po skončení trasovania nových lúčov je návratová hodnota súčtom osvetľovacieho modelu a farieb týchto lúčov.

3.5 kd-stromy

Základná trieda *RayTracer* používa naivnú metódu hľadania priesečníkov, kedy pri každom zavolaní metódy *findNearestObjects* hľadá priesečník so všetkými objektmi v scéne. Tento prístup síce je použiteľný pre scény s malým počtom objektov, avšak pre komplexné scény s obrovským počtom objektov (čo platí najmä pri použití trojuholníkových modelov) by renderovanie trvalo veľmi dlho, preto je implementovaná trieda KdTree, ktorá zapuzdruje operácie s kd-stromom. Najdôležitejšie metódy tejto triedy sú *construct*, *subdivide* a *traverse*.

Reprezentácia uzlu kd-stromu

Uzol kd-stromu implementovaný triedou *KdTreeNode* je v zhustenej forme, kedy sa na všetky dáta uzlu – pozícia rozdelenia, deliaca osa, ukazovatele na ľavý a pravý uzol, ukazovateľ na zoznam objektov v uzle a bitová hodnota či je daný uzol listom, používa iba 8 bajtov. Prvé štyri bajty sú použité na uloženie pozície rozdelenia a zo zvyšných 4 bajtov sa 2bity používajú určenie deliacej osi, jeden bit na určenie, či je daný uzol listom a na základe tejto informácie sú zvyšné bity použité ako ukazovateľ na ľavý uzol (ak uzol nie je listom), pričom pravý uzol sa alokuje zároveň s ľavým, takže jeho adresa je vždy ľavý uzol + 1 (čo zároveň zaručuje, že adresa takýchto uzlov bude vždy zarovnaná na 8 bajtov). Ak je daný uzol listom zvyšné dáta reprezentujú zoznam objektov asociovaných s týmto uzlom.

Takáto zhustená forma dát zaisťuje, že sa do cache pamäte procesora dá nahrať viac uzlov jedným čítaním z hlavnej pamäte a preto je prechod kd-stromom s takto usporiadanými dátami rýchlejší.

Konštrukcia kd-stromu

Pre skonštruovanie kd-stromu stačí zavolať metódu *construct* s referenciou na objekt triedy *ObjectContainer*, ktorá pridá všetky objekty do zoznamu koreňového uzlu kd-stromu a zavolá rekurzívnu metódu *subdivide* na koreňový uzol.

Metóda *subdivide* je jadrom konštrukcie celého kd-stromu – algoritmus najskôr vyberie dominantnú osu hraničného telesa ktoré obklopuje objekty v danom uzle a pripraví zoznam kandidátnych pozícií na rozdelenie do ďalších uzlov stromu. Následne zistí počet telies ktoré by sa nachádzali v ľavom a pravom uzle pre vybrané kandidátne pozície (volaním *SceneObject::intersectsExtents*) a nakoniec pomocou heuristiky založenej na povrchu hraničných telies a počte objektov v uzle vyberie najvýhodnejšiu kandidátnu pozíciu na rozdelenie uzlu a pokiaľ je výsledok tejto heuristiky nižší ako počet objektov v aktuálnom uzle a nebol prekročený limit delenia tak vytvorí nový ľavý a pravý uzol, pre ktoré určí objekty na základe vybranej deliacej pozície (opäť volaním *SceneObject::intersectsExtents*), a pokračuje sa delenie rekurzívnym volaním *subdivide* na novovytvorené uzly. Ak sa zistilo že sa uzol už nebude ďalej deliť, tak sa v cykle prejdú všetky objekty aktuálneho uzlu a volaním *SceneObject::intersectsBox* sa zistí či má objekt skutočne prienik s hraničným telesom daného uzlu, čo odstráni zo zoznamu objektov tohto uzlu objekty ktorých ohraničenie síce zasahuje do hraničného telesa uzlu avšak v skutočnosti sa samotný objekt a teleso nepretínajú.

Prechod kd-stromom

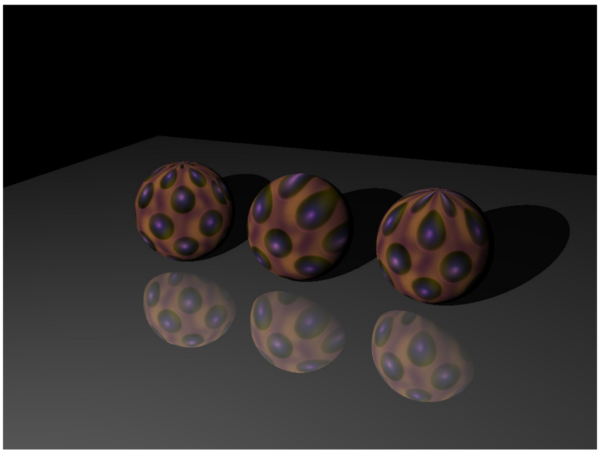
Metóda *traverse*, ako názov napovedá, prechádza stromom a hľadá najbližší priesečník s lúčom. Metóda implementuje algoritmus TA^B_{rec} z [8]. Na využitie kd-stromu pri ray-tracingu slúži trieda KdTreeRayTracer, ktorá má ako privátneho člena kd-strom a pri zavolaní metódy *preprocess* sa skonštruuje samotný strom. Aby trieda strom nielen vytvorila, ale aj využila má tiež prepísanú metódu *findNearestObjects*, kde sa volá zmienený *KdTree::traverse*.

4 Testovanie

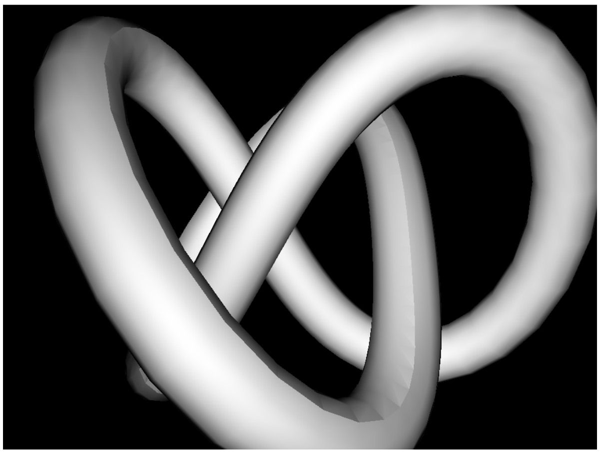
Pri testovaní aplikácie som sa zameriaval najmä na čas potrebný na vybudovanie kd-stromu scény a samotné renderovanie testovacích scén. V tejto kapitole uvádzam časy potrebné na vygenerovanie obrázkov jednotlivých scén v rozlíšení 1024x768 pixelov.

Čas potrebný na renderovanie obrázkov s rôznym počtom objektov scény (testované s použitím prekladača GCC 4.2.3 s prepínačom –O3 na procesore AMD Opteron 2216 pod operačným systémom Linux):

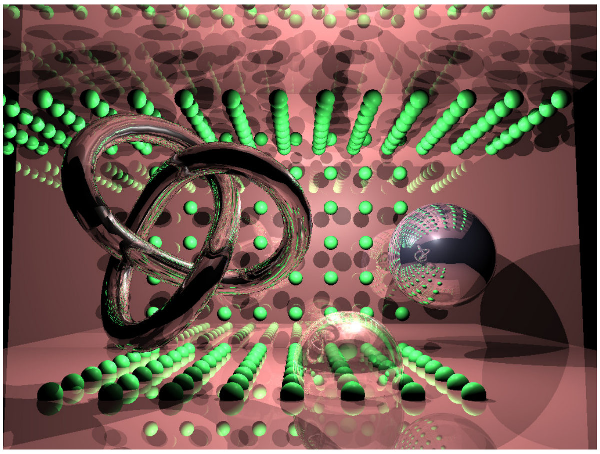
	Počet objektov	Čas renderovania	Počet lúčov	Lúčov za sekundu
Scéna s troma textúrovanými guľami a zrkadlovou plochou [obrázok 15]	5	1.7 sekundy	1 857 435	1 092 608
Jednoduchá scéna s trojuholníkovým modelom [obrázok 16]	2600	3.3 sekundy	1 229 501	372 576
Kombinovaná scéna s trojuholníkovým modelom a guľami [obrázok 17]	2856	15 sekúnd	7 789 890	519 326
Scéna s modelom auta s použitím bodového zdroja svetla [obrázok 18]	124 897	19 sekúnd	3 065 297	286 225
Scéna s modelom auta s použitím plošného zdroja svetla (až 64 tieňových lúčov)	124 897	119 sekúnd	43 081 537	362 029



Obrázok 15: Scéna s troma textúrovanými guľami a zrkadlovou plochou

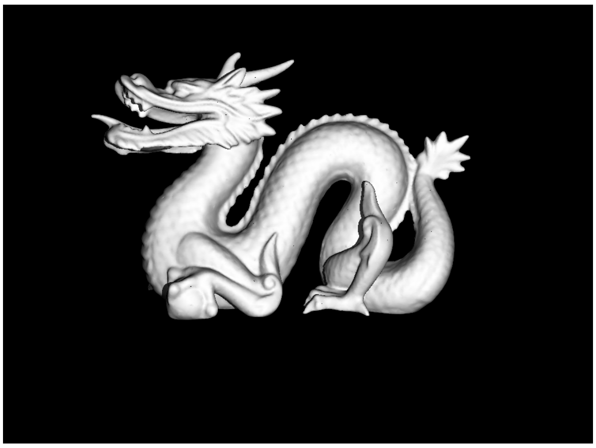


Obrázok 16: Jednoduchá scéna s trojuholníkovým modelom



Obrázok 17: Kombinovaná scéna s trojuholníkovým modelom a guľami

Obrázok 18: Scéna s modelom auta



Obrázok 19: Jednoduchá scéna s modelom draka

Z tabuľky je možné vidieť, že vďaka použitiu kd-stromu sa rýchlosť aplikácie nedegraduje tak výrazne s rastúcim počtom objektov v scéne a výkonnosť záleží najmä na použitých materiáloch v scéne a teda počtu lúčov ktoré treba generovať naviac.

Rýchlosť budovania kd-stromu a renderovania bez použitia skutočného prieniku s objektami (testované s použitím prekladača GCC 4.2.3 s prepínačom –O3 na procesore AMD Opteron 2216 pod operačným systémom Linux):

Počet objektov	Čas načítania modelu a vybudovania kd-stromu	Čas renderovania s vybudovaným kd-stromom
2 600 [obrázok 16]	0.06 sekundy	4.3 sekundy
202 520 [obrázok 19]	8.6 sekundy	4.5 sekundy
871 414 [obrázok 19 – s presnejším modelom]	38.5 sekundy	4.1 sekundy

Rýchlosť budovania kd-stromu a renderovania pri použití optimalizácie s prienikmi hraničných telies a následnom odstránení objektov bez skutočného prieniku (testované s použitím prekladača GCC 4.2.3 s prepínačom –O3 na procesore AMD Opteron 2216 pod operačným systémom Linux):

Počet objektov	Čas načítania modelu a vybudovania kd-stromu	Čas renderovania s vybudovaným kd-stromom
2 600 [obrázok 16]	0.17 sekundy	3.18 sekundy
202 520 [obrázok 19]	9.4 sekundy	2.5 sekundy
871 414 [obrázok 19 – vo vyššom rozlíšení]	40 sekúnd	2.1 sekundy

Z tabuliek vyplýva, že ak sa na budovanie kd-stromu použije metóda, ktorú som navrhol a teda objekty, ktoré nemajú priesečník s hraničným telesom uzlu sa až na konci budovania stromu odstránia z listov stromu predĺži sa mierne čas vytvárania stromu, avšak zrýchli sa následné renderovanie scény, pretože v jednotlivých listoch stromu bude menej objektov, a teda je potrebných menší počet zisťovania priesečníkov.

Vyhodnotenie používania Monte-Carlo metód v scéne s 2856 objektmi – viď obrázok 17 (testované s použitím prekladača GCC 3.4.2 s prepínačom –O3 na procesore Intel Pentium M 1.73 GHz pod operačným systémom Windows):

	Počet lúčov	Čas renderovania
Ideálne zrkadlové odrazy, bodové zdroje svetla – bez akéhokoľvek Monte-Carlo renderovania	7.8 miliónov	28 sekúnd
Ideálne zrkadlové odrazy, plošné zdroje svetla – na mäkké	221 miliónov	9 minút

tiene použitých až 64 tieňových lúčov		
Difúzne odrazy, plošný zdroj svetla – na mäkké tiene použitých až 64 tieňových lúčov, na difúzne odrazy až 128 odrazových lúčov	1 947 miliónov	78 minút

Z poslednej tabuľky je jasné, že používanie Monte-Carlo metód síce produkuje oveľa prirodzenejšie obrázky, avšak za cenu veľmi vysokého počtu potrebných lúčov a tým pádom aj času stráveného samotným renderovaním.

5 Záver

Cieľom tejto bakalárskej práce bolo spracovanie problematiky ray-tracingu, overenie možnosti delenia scény na podpriestory, urýchlenie výpočtu priesečníku lúču s objektmi, bol predstavený návrh a implementácia jednoduchého, no veľmi ľahko rozšíriteľného ray-traceru, ktorý umožňuje implementáciu rôznych techník sledovania lúča a mapovania textúr, takže zadanie bolo splnené.

Taktiež bolo ukázané, že je možné zrýchliť renderovanie obrázkov tým, že pri budovaní kd-stromu sa použijú iba prieniky hraničných telies, a až keď je strom vybudovaný stačí využiť skutočné prieniky objektov a hraničného telesa uzlu stromu, takže sa neuskutočňuje zbytočné testovanie priesečníkov pri samotnom renderovaní.

Osobne bola pre mňa práca na tomto projekte veľmi zaujímavá a pri jej vypracovaní som si prehĺbil moje znalosti z oblasti počítačovej grafiky a zároveň som sa zdokonalil v jazyku C++.

V budúcnosti by bolo možné ďalej rozširovať funkčnosť programu a zároveň zvýšiť jeho rýchlosť rozdelením ray-tracingu do viacerých vlákien, čo by vôbec nebol problém (všetky používané metódy pri hľadaní priesečníkov sú konštantné a teda neprepisujú pamäť objektov scény a nemôže dôjsť k súbežnému čítaniu a zápisu pamäte) a výkon by mal teoreticky vzrásť lineárne s počtom jadier procesoru. Taktiež doktor Wald uvádza v [14] spôsob ako trasovať s využitím SIMD naraz 4 lúče čo by opäť mohlo priniesť teoreticky až 3-násobné zrýchlenie programu, čím by sa program posunul bližšie k real-time renderovaniu. Tiež by bolo možné implementovať ďalšie spôsoby mapovania textúr – napríklad mapovanie na celé trojuholníkové modely. Zaujímavé by tiež bolo implementovať využívanie textúr na bump mapping – teda zmenu normály objektu na základe textúry a veľmi dobrým rozšírením by bola implementácia globálneho osvetľovacieho modelu.

Literatúra

[1] Glassner, A.: *An Introduction to Ray Tracing*. San Francisco, USA, Morgan Kaufmann 1989, ISBN 0-12-286160-4.

[2] Wikipedia, The Free Encyclopedia: *Ray-Tracing* [online]. Dostupné na URL: http://en.wikipedia.org/wiki/Ray_tracing (máj 2008)

[3] Kay, T.L., Kajiya, J.T.: *Ray Tracing Complex Scenes*. SIGGRAPH‘86 Proceedings, New York, USA, ACM 1986, s. 269-278.

[4] Bikker, J.: *Raytracing: Theory and Implementation* [online]. Dostupné na URL: http://www.devmaster.net/articles/raytracing_series/part1.php (máj 2008)

[5] Kršek, P.: *Studijní opora k předmětu Základy počítačové grafiky*. Brno, Česká republika, FIT VUT 2003.

[6] Blinn, J.F.: *Models of light reflection for computer synthesized pictures*. SIGGRAPH’77 Proceedings, New York, USA, ACM 1977, s. 192-198.

[7] Ngan, A., Durand, F., Matusik, W.: *Experimental Validation of Analytical BRDF Models*. Technical Sketch, SIGGRAPH 2004.

[8] Havran, V.: *Heuristic Ray Shooting Algorithms*. Prague, Czech Republic, Czech Technical University 2000.

[9] Using the GNU Compiler Collection (GCC): *Vector Extensions* [online]. Dostupné na URL: http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html#Vector-Extensions (máj 2008)

[10] Heckbert, P.S.: *Graphic Gems IV*. Chestnut Hill, USA, AP Professional 1994, ISBN 0-120336155-9.

[11] Bourke, P.: *PLY – Polygon File Format* [online]. Dostupné na URL: http://local.wasp.uwa.edu.au/~pbourke/dataformats/ply/ (apríl 2008)

[12] Pitts, J.: *3DS Format* [online]. Dostupné na URL: http://www.whisqu.se/per/docs/graphics56.htm (apríl 2008)

[13] Williams, A., Barrus, S., Morley, R.K., Shirley, P.: *An efficient and robust ray-box intersection algorithm*. New York, USA, ACM 2005.

[14] Wald, I.: *Realtime Ray Tracing and Interactive Global Illumination*. Saarland University 2004.

[15] Akenine-Möller, T.: *Fast 3D Triangle-Box Overlap Testing* [online]. Dostupné na URL: http://jgt.akpeters.com/papers/AkenineMoller01/ (apríl 2008)

[16] Funkhouser, T.A.: Ray-Triangle Intersection II [online]. Dostupné na URL: http://www.cs.princeton.edu/courses/archive/fall00/cs426/lectures/raycast/sld019.htm (máj 2008)

[17] Wikipedia, The Free Encyclopedia: *Phong Shading* [online]. Dostupné na URL: http://en.wikipedia.org/wiki/Phong_shading (máj 2008)

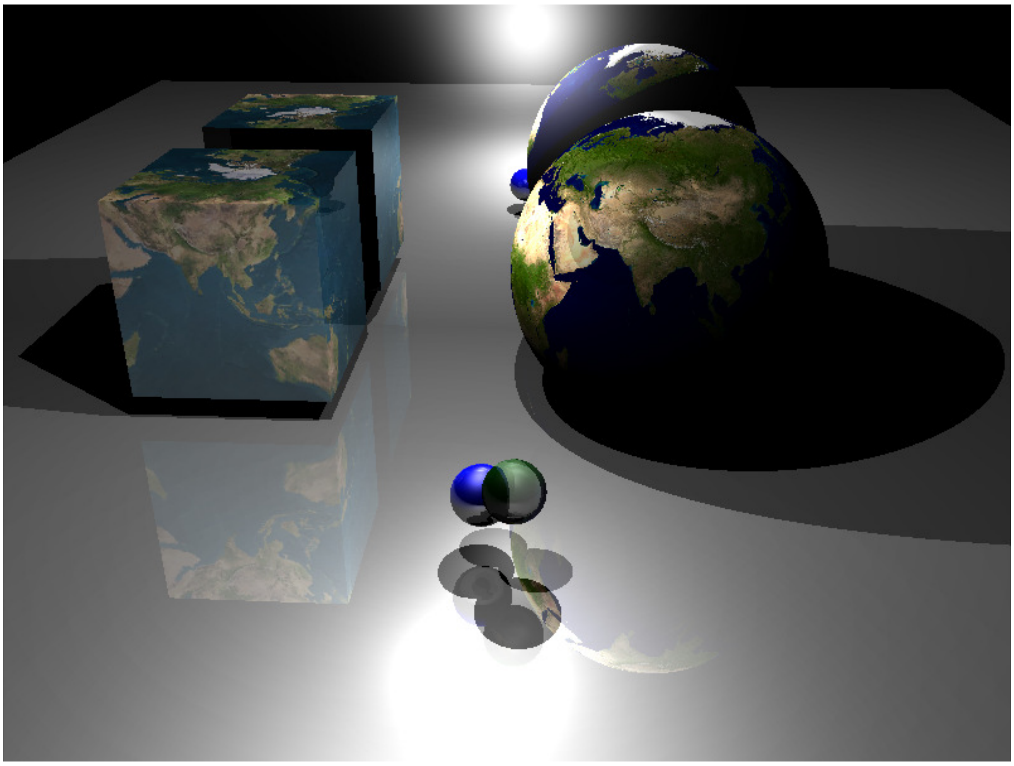
Zoznam príloh

Príloha 1. Príklad scén renderovaných programom.

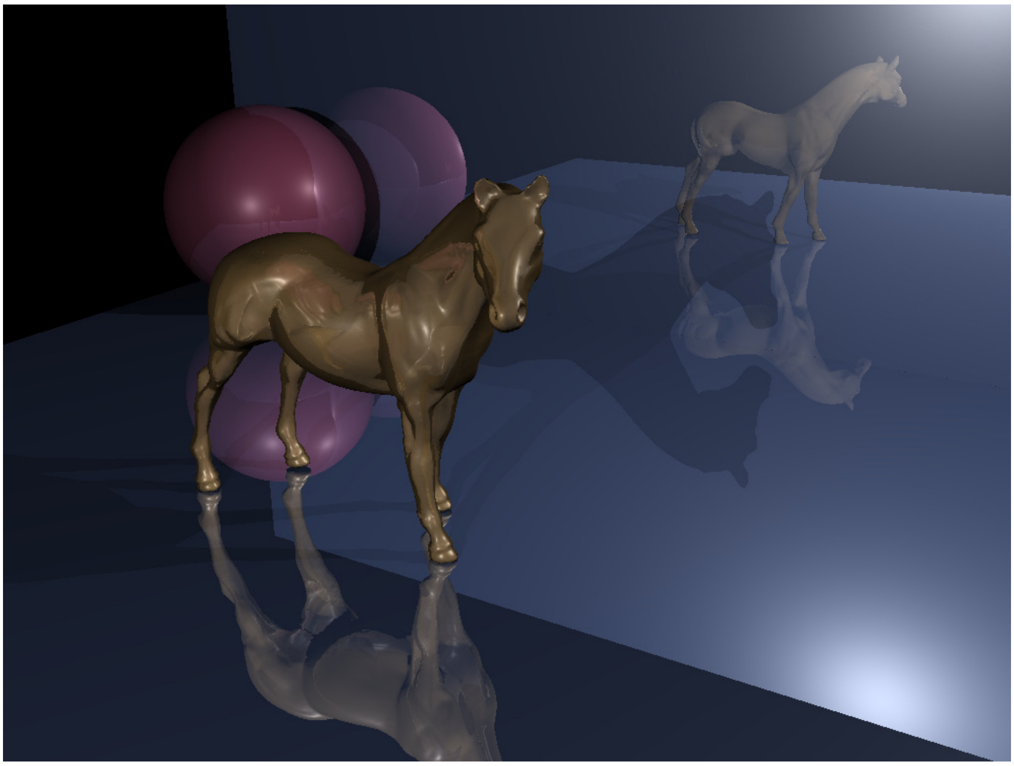
Príloha 2. CD zo zdrojovými textami a rozširujúcou dokumentáciou.

Príloha 1

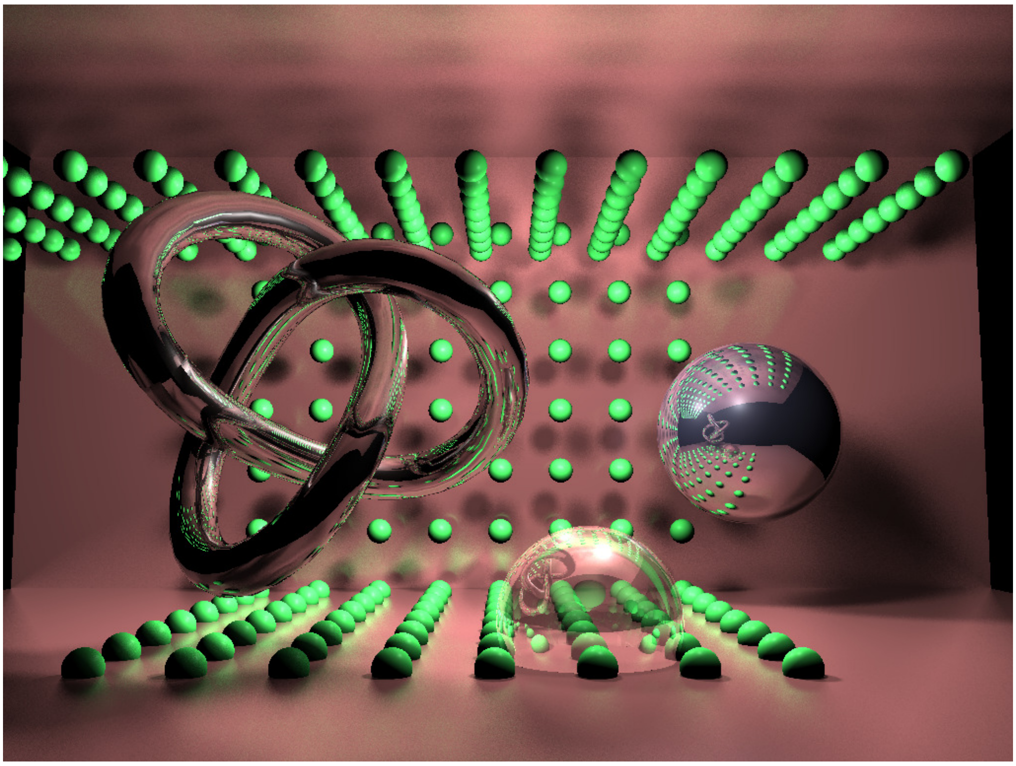
Príklad scén renderovaných programom



Ukážka mapovania sférickej textúry na kocku.



Scéna s 96 966 objektmi. Využíva odrazy a lom svetla, ostré tiene.



Scéna s 2 856 objektmi, použité difúzne odrazy a mäkké tiene.

Scéna s 124 897 objektmi, použité mäkké tiene.