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

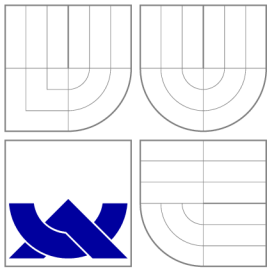
RAYTRACING VIRTUÁLNÍCH GRAFICKÝCH SCÉN

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

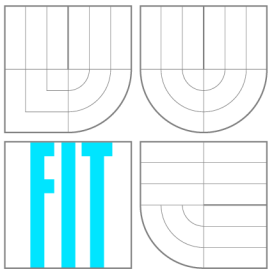
AUTOR PRÁCE
AUTHOR

MATEJ KENDRA

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RAYTRACING VIRTUÁLNÍCH GRAFICKÝCH SCÉN

RAYTRACING OF VIRTUAL GRAPHICS SCENES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MATEJ KENDRA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN PEČIVA, Ph.D.

BRNO 2013

Abstrakt

Cílem této práce bylo nastudovat knihovnu OpenSceneGraph a algoritmy používané pro metody sledování paprsku. Z nastudovaných materiálů pak vytvořit aplikaci umožňující zobrazovat akcelerovanou a raytracingovou metodou se zaměřením na zrcadlící se povrchy.

Abstract

The goal of this work was to learn OpenSceneGraph library and algorithms of raytracing methods. Then, from these knowledge, create an aplication for displaying a scene via accelerating and raytracing method targeted on reflecting surfaces.

Klíčová slova

sledování paprsků, osg, OpenSceneGraph, 3D, virtuální scéna, zrcadlení

Keywords

raytracing, osg, OpenSceneGraph, 3D, virtual scene, reflection

Citace

Matej Kendra: Raytracing virtuálních grafických scén, bakalářská práce, Brno, FIT VUT v Brně, 2013

Raytracing virtuálních grafických scén

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Pečivy Ph.D.

.....
Matej Kendra
14. května 2013

Poděkování

Touto cestou bych chtěl poděkovat panu Ing. Janu Pečivovi Ph.D. za jeho pomoc, odborné rady a poznámky při tvorbě této bakalářské práce.

Obsah

Kapitola 1

Úvod

V dnešním moderním světě potřebujeme zobrazovat obrovské množství vizuálních dat, ať už se jedná o zobrazování průmyslových výrobků, či jejich technologií, počítačové hry a virtuální reality nebo zobrazování lékařských výsledků. Ve všech těchto oblastech potřebujeme, či chceme člověku zobrazit data v daných scénách co nejefektivněji a s ohledem na účel co nejvěrohodněji.

Projekcí, rozebranou v kapitole 2.1, umožňujeme člověku vnímat trojrozměrné scény na dvourozměrných prostředcích, jakými jsou monitory a televize. Kromě samotné projekce ovšem potřebujeme věrohodně napodobit mnoho reálných vlastností – světla, stíny, odrazy a další, třeba i složitější vlastnosti, jako například mlhu či transparenci objektů aj. Všechny takovéto vlasnosti mohou řešit různé metody jiným způsobem.

Existuje spousta způsobů, jak člověku virtuální scény zobrazovat, od metod akcelerovaných přes raytracingové, až po metody fyzikálně přesné. V této práci jsou nejznámější z těchto metod popsány v kapitole 2.2. Zvláště pak je detailně rozebrána metoda sledování paprsku – raytracing 2.3, která je stěžejní pro tuto práci.

Výsledkem této práce by tak měla být aplikace, která uživateli umožní zobrazit scénu akcelerovanou metodou a zároveň dokáže nad touho scénou aplikovat metodu sledování paprsku.

Kapitola 2

Zobrazování virtuálních scén

Tato kapitola se zabývá projekcí virtuálních scén a dále pak metodami jejich zobrazení. Zobrazování může probíhat přes hardwarové prvky nebo jej provádí přímo procesor počítače. Popsány jsou metody OpenGL, Microsoft DirectX, ray-casting a NVIDIA OptiX spolu se samotným raytracingem.

2.1 Projekce

Zobrazení všech trojrozměrných scén se provádí na dvourozměrných zobrazovacích prostředcích (monitory, televize apod.), proto potřebujeme transformovat z trojrozměrného do dvojrozměrného zobrazení. Tato transformace se dle [21] nazývá promítání nebo-li projekce. Při těchto tranformacích však dochází ke ztrátě prostorových informací, výsledky promítání jsou transformacemi zkreslovány. Jelikož ne všechny obory potřebují promítat stejně, existuje více druhů projekce, jež umožňují danému oboru zlepšit vnímání objektů ve scéně.

Promítání probíhá na průmětnu[1] za pomocí promítacího paprsku. Tento paprsek je polopřímkou vycházející z promítacího (prostorového) bodu a jeho směr závisí na dané promítací metodě. Z praktického hlediska se počítačové projekce provádí na rovinu průmětny, protože paprsky se nemusí transformovat. Pro rovinné promítání rozlišujeme hlavně tyto dvě metody:

- rovnoběžnou (paralelní) projekci
- středovou (perspektivní) projekci

V projekci je nutné mít stanovenou transformaci prostorového bodu na průmětnu (samotné promítání) a také souřadný systém, ve kterém promítáme. Souřadný systém můžeme mít světový, anglicky World Coordinate System (WCS) nebo souřadnicový systém průmětny – Viewing Coordinate System (VCS). V obou případech však musíme mít možnost tyto souřadnice transformovat do systému druhého. Z tohoto důvodu musí být vypočítána transformační matice, popřípadě bude matic vypočítáno více.

2.1.1 Paralelní projekce

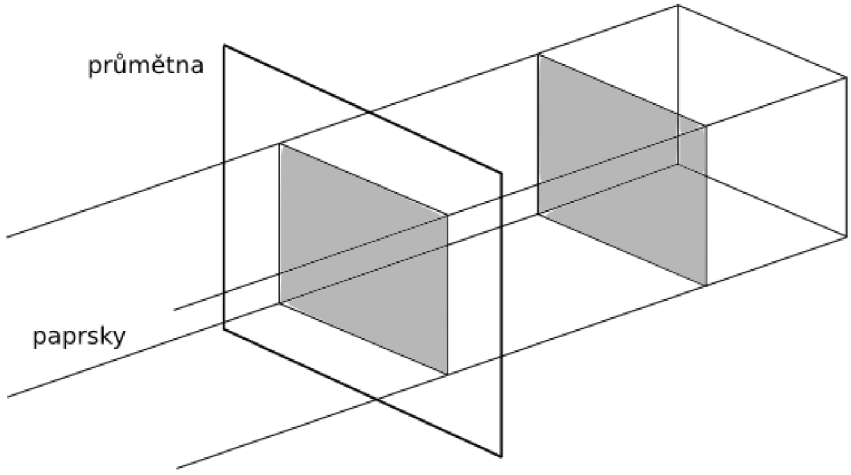
Paralelní projekce vysílá rovnoběžné promítací paprsky kolmé na průmětnu (viz obr. 2.1). Všechny paprsky mají tedy stejný směr. Hlavní vlastností tohoto promítání je zachování

[1] plocha v prostoru, na kterou dopadají promítací paprsky a vytváří tak obraz (průmět)

rovnoběžnosti úseček (hran) ve scéně. Nejpoužívanějšími průmětnami bývají hlavní roviny yz, xz nebo yz.

Použijeme-li obecnou průmětnu, která není rovnoběžná s hlavními osami, jedná se o axonometrické promítání. O kosoúhlém promítání mluvíme v případě, když zkombinujeme jak axonometrické promítání, tak jedno znázornění hlavní průmětny z Mongeova[2] promítání.

Tato projekce se používá převážně v architektuře, nezkracuje totiž vzdálenosti v rovinách rovnoběžných s průmětnou, ani v případě, že se vzdaluje pozorovatel (kamera).



Obrázek 2.1: Znázornění paralélní projekce.

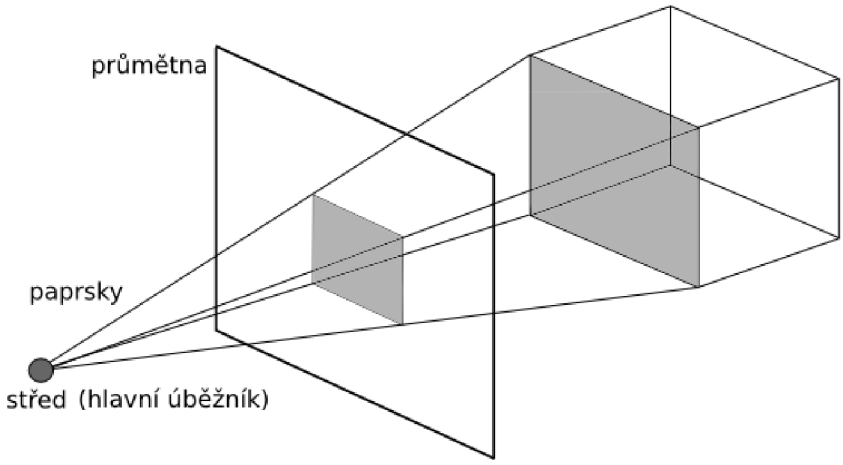
2.1.2 Perspektivní projekce

Perspektivní promítání využívá svého středu či středů, ze kterých vysílá paprsky různým směrem. Ikdyž můžeme mít průmětnu s libovolnou polohou, rozlišujeme nejčastěji tři hlavní případy:

- jednobodovou perspektivu – průmětna protíná jednu osu, všechny úsečky míří do hlavního úběžníku (viz obr. 2.2)
- dvoubodovou perspektivu – průmětna protíná dvě osy, dostaneme dva hlavní úběžníky
- trojbodovou perspektivu – průmětna protíná všechny tři osy, dostaneme tři hlavní úběžníky

Jedná se o metodu, která se podobá optickému modelu našeho oka. Se zvětšující se vzdáleností od scény se nám v této projekci také mění velikost promítaných předmětů, což koresponduje s naším prostorovým vnímáním.

[2]Promítání ve více směrech (směrech hlavních os) do průměten. Zahrnuje nárys a dále bokorysy, půdorys, pohled zespodu a zezadu, podle potřeby.



Obrázek 2.2: Znázornění perspektivní projekce s jedním hlavním úběžníkem.

Pro příklad si můžeme uvést aplikace CAD, sloužící pro design v různých oborech (CAM ve strojírenství, AutoCAD v architekruře či Google SketchUp). Tyto aplikace využívají paralelní projekce z důvodu zachování rovnoběžnosti – přesnost modelů. Kdežto modelovací softwary typu Blender či komerční Cinema4D nebo Autodesk Maya, které slouží převážně k tvorbě herních či filmových modelů, používají perspektivní projekci. U těchto softwarů však existuje většinou možnost mezi projekcemi přepínat.

2.2 Akcelerované metody a metody využívající raytracing

Akcelerované metody již ze svého názvu naznačují, že jejich hlavní funkcí je akcelerovat. Myslí se tím využívání hardwarových součástí – jde tedy o tzv. hardwarovou akceleraci. Výhodou těchto metod je, že výpočet je prováděn přímo na hardwaru bez nutnosti použít CPU, tím se urychluje výpočet nejen konkrétní aplikace, ale celého systému. Z hlediska zobrazování se akcelerované metody zabývají hlavně akcelerací za pomocí grafických karet a grafických procesorů (GPU).

Nejdůležitější vlastností GPU procesoru je jeho schopnost paralelizovat práci s různými typy dat. Práce je pak rozdělena mezi více výpočetních vláken, které dokáží pracovat asynchronně. V těchto vláknech se tak může řešit vícero akcí zároveň, počítání osvětlovacích modelů, stínování, simulace fyzikálních jevů apod.

Příklady stěžejních nástrojů, využívajících hardwarovou akceleraci, OpenGL a DirectX, jsou popsány níže.

Raytracingové metody

U metod využívající raytracing – sledování paprsků od pozorovatele, se výpočet děje ve většině případů na procesoru CPU a bude podrobně popsán v kapitole 2.3. Kromě raytracingu a metody ray-casting (2.2.3) však existují i pokročilejší metody sledování paprsků. Patří mezi ně například pathtracing [6] a jeho obousměrná (angl. bi-directional) metoda [7], metoda fotonových map [5] či metoda „Metropolis Light Transport“ [16].

Vlastností raytracingových metod je sice velká reálnost zobrazovaných scén, ale také velká výpočetní náročnost. Procesor CPU je tímto výpočtem velmi zatížen a rychlost těchto

metod nebývá nejrychlejší. Avšak i tyto metody procházejí vývojem a velké úsilí je věnováno hlavně zrychlování či případné akceleraci. Zajímavostí je tak technologie OptiX firmy NVidia popsána v 2.2.4.

2.2.1 OpenGL

OpenGL je primární multiplatformní prostředí pro tvorbu interaktivních 2D a 3D aplikací [4]. Od uvedení v roce 1992 se stalo průmyslově nejrozšířenějším rozhraním při vytváření a podpoře grafických aplikací. Aplikace dokáží využívat velkého výkonu, který jim toto rozhraní umožňuje, ať už jde o CAD či CAM aplikace, lekářské zobrazování, hry či virtuální realitu.

Hlavními vlastnostmi OpenGL jsou:

- jde o **průmyslový standard** o který se stará konsorcium nezávislých firem[3]
- **stabilita** a **spolehlivost** – kontroly, aktualizace a zpětná kompatibilita
- **dostupnost** a **přenositelnost** – různá zařízení a platformy
- neustálý **vývoj** spolu s hardwarem
- **dokumentace** – velice dobrá dostupnost materiálů (knih, tutoriálů nebo ukázek aplikací)

Paleta platforem a systémů, na kterých OpenGL můžeme využít, je veliká. Podpora věškerých UNIXových, Windows i Mac stanic, spolu s jejich hlavními systémy: Linux, Windows 95 až Windows NT a MacOS, jsou toho důkazem. OpenGL nám také poskytuje podporu hlavních zobrazovacích systémů – X-Window, Win32 a MacOS.

Nemalé je také portfolio programovacích jazyků, v nichž OpenGL můžeme využívat. Jedná se zejména o C, C++, Python, Perl a Javu.

2.2.2 DirectX

DirectX je aplikační rozhraní (API), vytvořené firmou Microsoft. Toto API umožňuje vývojářům her, simulací atp. přímé ovládání hardwaru. Snahou je docílit co nejvyššího výkonu, abychom jej mohli použít i v realtimových scénách. Jelikož se jedná o komerční produkt, jeho použitelnost zaostává za OpenGL. Použít jej můžeme pouze na počítačích s operačním systémem Windows. DirectX se skládá z více menších částí – knihoven, které nám umožňují práci s různými rozhraními (Direct3D pro vykreslování 3D grafiky, DirectSound pro zvuk nebo DirectDraw pro kreslení 2D grafiky), proto také název Direct*X* [19].

Vývoj DirectX začal již v roce 1995, kdy vyšla verze 1.0. První verze byla použita v operačním systému Windows 95. Důvodem k vývoji bylo, že pokud chtěl vývojář hardware využívat v systému MS-DOS, musel k němu složitě přistupovat. A proto vzniklo rozhraní pro zlepšení přístupu, které komunikuje přímo s ovladači hardwaru. S dalšími verzemi přibývaly i další použitelné části rozhraní.

Nejnovější verzí, vydanou v roce 2009, je DirectX 11. Největším posunem této verze jsou výpočty prostřednictvím GPGPU tzv. DirectCompute, zlepšení se dočkalo také použití vláken pro vícejádrové procesory a jednou z největších výhod se stala teselace[4] – důsledkem

[3] mezi nejznámější firmy patří AMD, Apple, Intel, NVidia, Samsung či Sony viz [3])

[4] proces, který umožuje přeměnit obecný polygon na nepravidelnou trojúhelníkovou síť (TIN - triangular irregular network)

je, že dokážeme hardwarově vypočítávat větší detaily bez nutnosti předávat grafické kartě velké objemy dat. Tuto verzi podporuje systém Windows 7, Windows 8 má již zabudouvanou verzi 11.1.

Nevýhodou DirectX, kromě omezeného použití platforem, je skutečnost, že jej musí podporovat grafická karta. U starších grafických karet tak novější verze DirectX použít nelze, popřípadě může utrpět funkčnost systému.

2.2.3 Ray-casting

Jak již bylo zmíněno na začátku kapitoly, ray-casting je jednou z metod využívající sledování paprsku. Z praktického hlediska se jedná o raytracing prvního řádu, nedokáže se rekurzivně zanořovat a řešit tak průhledné či zrcadlící se materiály. Sledované paprsky jsou vyslány z kamery (může se jednat o perspektivní i paralelní projekci) a při prvním kontaktu s objektem, který je umístěn ve scéně se vrací barva tohoto objektu. Případné stíny se tak jako u raytracingu řeší vysláním stínového paprsku z místa dopadu primárního paprsku směrem ke zdrojům světla (viz 2.3.2).

2.2.4 NVIDIA OptiX

OptiX je raytracingový engine vyvinutý firmou NVIDIA, který umožňuje zobrazovat scény za pomoci akcelerovaného sledování paprsků. Na grafických procesorech NVIDIA s použitím technologie CUDA trvá výpočet řádově několikrát méně než při obyčejném raytracingu. Na oficiálních internetových stránkách firmy NVIDIA [8] se uvádí, že co trvá za normálních podmínek minuty, engine OptiX stíhá spočítat za milisekundy.

Výhodou tohoto enginu určitě bude rychlost s jakou počítá komplikované scény. Avšak i přes neustálý vývoj a zlepšování v oblasti akcelerace raytracingu je hlavní nevýhodou cena hardwaru. NVIDIA uvádí, že ideálními nástroji pro tyto výpočty jsou grafické karty řady Quadro či Tesla. U modelu Tesla se však ceny pohybují v řádech desetitisíců korun za kus[5]. Řada Quadro je přece jen dostupnější, ceny se pohybují v řádech tisíců.

Pro vývojáře je k dispozici engine verze 3.0, která jim dovoluje díky volné licenci produkovat kvalitní a rychlý raytracing. Podmínkou použití je ovšek kromě překladače C/C++ a nástroje `CMAKE` také `CUDA Toolkit 2.3`.

2.3 Raytracing

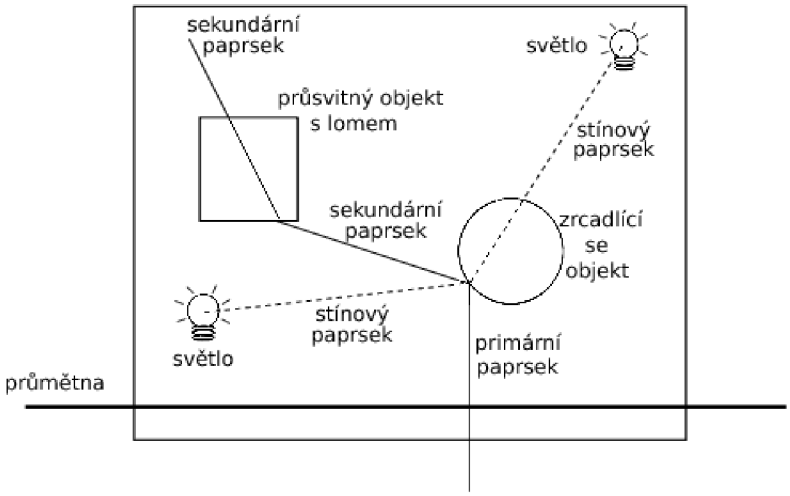
Raytracing je jednou z nejznámějších metod sledování paprsků. Jedná se o metodu, kterou vyvinul Turner Whitted v roce 1980 [18]. Navázal tak na algoritmus Arthura Appela z roku 1968 [1]. Jde o empirický model, který dostaneme zjednodušením reálných vlastností. Hlavním zjednodušením raytracingu je, že počítáme s konečným počtem přímých paprsků světla, které se nemůžou ohýbat, dalším zjednodušením je zanedbání zákonu zachování energie.

V reálném světě putují paprsky světla z jeho zdroje skrz prostředí do oka pozorovatele. Na této cestě se paprsky můžou různě ohýbat, odrážet či lomit a výsledný vjem se promítne v mozku. Z hlediska sledování takovýchto paprsků je však nemožné sledovat všechny tyto paprsky směřující od světelného zdroje, proto se u raytracingu využívá opačného mechanismu – zpětné sledování paprku (viz [21] kapitola Metody vycházející od pozorovatele). Paprsky jsou tak vysílány z místa pozorovatele, kdy je místo oka použita kamera a sleduje

[5] na portálu Heureka.cz dne 15.4.2013: HP Tesla C2075 6GB za nejlevnější cenu od 50 000,- Kč

se cesta těchto paprsků až ke zdrojům světla. Výsledný obraz je pak promítnut na průmětnu a zobrazen pozorovateli.

Počet vyslaných paprsků je ovlivněn rozlišením sledované průmětny/kamery, kdy přes každý pixel na průmětně vyšleme právě jeden primární paprsek (angl. primary-ray). Viditelnost je tak vyřešena již tímto vysláním paprsků, jelikož nemusíme ořezávat žádné objekty ve scéně ani počítat, kde nám scéna končí či začíná. Pokud totiž primární paprsek nezasáhne žádný objekt výslednou barvou je barva pozadí. Naopak, zasáhne-li primární paprsek objekt ležící ve scéně, vrací se barva materiálu tohoto objektu ovlivněná osvětlovacím modelem a případnými odrazy či lomy (viz obr 2.3).



Obrázek 2.3: Vyslání primárního paprsku do scény skrz průmětnu.

2.3.1 Osvětlovací model

Oko vnímá barvu jako vjem. Tento vjem je vyvolán dopadem světelného paprsku na objekt, jehož barva se nám tak promítne v oku. Abychom tedy mohli vůbec vnímat barvu potřebujeme mít zdroj světelných paprsků a musíme umět spočítat, jakým způsobem ovlivňují tyto parsky barvu okolních objektů ve scéně. Tento výpočet obstarává osvětlovací model. Ve většině aplikací je tento model počítán za pomoci shaderů v grafických kartách, avšak v případě této práce tomu tak není. Implementace osvětlovacího modelu a stínování je probrána detailně v kapitole 4.3.2.

Druhů osvětlovacích modelů je více, mezi nejznámější patří – Lambertův a Phongův osvětlovací model. Tyto modely jsou rozdílné pouze u spekulární[6] složky barvy. Zatímco u Lambertova osvětlovacího modelu se nepočítá s odlesky či odrazy světla, ale jen s difusní[7] a ambientní[8] složkou materiálu, Phongův osvětlovací model, který v roce 1973 vymyslel ve své disertační práci Bui Tuong Phong a jež se dostala do [12], počítá již i s odrazovou složkou.

Zvolený osvětlovací model počítá, do jaké míry ovlivňují světelné zdroje[9] barvu v daném bodě pro který je počítán. Vzdálenost bodu od zdroje, na který paprsek dopadá, ovlivňuje

[6] složka simulující odrazy od lesklých povrchů
[7] odraz od matného materiálu vyzařovaný do všech směrů od objektu
[8] simuluje rozptyl světla od objektu
[9] světelné zdroje mohou být bodové, kuželové či plošné

výslednou barvu také. Tato závislost se popisuje jako útlum (angl. attenuation) rovnicí:

$$attenuation = \frac{1}{a + b * d + c * d^2}$$

,v níž a představuje konstaní útlum světla, b lineární útlum světla, c kvadratický útlum světla a d je vzdálenost bodu od světelného zdroje.

U kuželových světel kromě vzdálenosti, počítá osvětlovací model ještě i směr v jakém je paprsek světla vyslán. Čím kolmější je směr paprsku se směrem kuželového zdroje světla (angl. light direction), tím je barva tmavší. Tato vlastnost je popsána Lambertovým kosinovým pravidlem:

$$\mathbf{N}\,\mathbf{L} = |N||L|\cos\alpha = \cos\alpha$$

,kde je N normalizovaný vektor normály v bodě dopadu, L je normalizovaný vektor ke světelnému zdroji a α úhel mezi těmito dvěma vektory. Normalizace je pro výpočet kosinova pravdidla nezbytná ať už se jedná o jakýkoliv vektor. Phongův model je podle [21] popsán nasledujícími vzorci:

$$I_a = I_A r_a$$

Ambientní složka I_a je vyjádřena z globálního množství okolního světla ve scéně I_A a koeficientu r_a, který značí schopnost povrchu objektu odrážet světlo.

$$I_d = I_L r_d(\vec{n} \cdot \vec{l})$$

Koeficient difuzního odrazu objektu r_d tvoří spolu s barevnou složkou dopadajícího světelného paprsku I_L složku difuzní barvy I_d. Ta je ovšem závislá na směru světla. Čím blíže je tento směr normále v bodě dopadu, tím větší tato složka je. Matematicky vyjádřeno $(\vec{n} \cdot \vec{l})$, pokud je však úhel záporný – difuzní složka je černá, protože bod dopadu je odvrácený.

$$I_s = I_L r_s(\vec{r} \cdot \vec{v})^n$$

Spekulární složka I_s je tvořená složkou paprsku I_L, koeficientem spekulární složky objektu r_s a úhlem daným vektorem odrazu světla a normálou $(\vec{r} \cdot \vec{v})$ umocněným koeficientem ostrosti odrazu – skalárem n.

Vektor odrazu světla R se vypočítá podle rovnice $R = 2(L \cdot N)N - L$, kde L je normalizovaný vektor ke zdroji světla a N je normalizovaná normála v bodě dopadu.

Výsledná barva I je pak určena součtem všech předem zmíněných složek $I = I_a + I_d + I_s$. Ve scénách, kde máme více světel se však tento vzorec nepoužívá. Místo něj je použit vzorec následující:

$$I = I_a r_a + \sum_{i=1}^{M} I_{L_i}(r_d(\vec{n} \cdot \vec{l_i}) + r_s(\vec{r_i} \cdot \vec{v})^n)$$

,kde počítáme sumu difuzních a spekulárních složek všech světel M. Tato suma spolu s ambientní složkou scény tvoří výslednou barvu. Dle OpenGL specifikace [14] by však měl být osvětlovací model implementován takto:

$$I = E_{mat} + A_{glob}A_{mat} + \sum_{i=1}^{M}(attenuation_i)(spot_i)(A_{L_i}A_{mat} + D_{L_i}D_{mat}(\vec{n}\cdot\vec{l_i}) + S_{L_i}S_{mat}(\vec{r_i}\cdot\vec{v})^n)$$

,kde E_{mat} je složka vyzařována materiálem, A_{glob} je globální ambientní složka scény, útlum tvoří složka *attenuation* a pro kuželová světla i složka *spot*. Zbývající složky se počítají stějně jako předcházející rovnice.

2.3.2 Stíny

Jestliže chceme mít raytracingem simulovány stíny, musí být z místa dopadu primárního paprsku vyslán takzvaný stínový paprsek – anglicky shadow-ray (2.3). Stínovým paprskem se zjistí, zda-li svítí zdroj světla přímo na místo dopadnu primárního paprsku. V případě, že ano, jsou k výsledné barvě připočítány všechny barevné složky ovlivněné osvětlovacím modelem. V opačném případě se výpočet osvětlovacího modelu vůbec neprovádí.

Pro každý zdroj světla se musí vyslat jeden stínový paprskek, v případě, že bude ve scéně zdrojů světla přibývat, časová náročnost se bude úměrně s počtem těchto světel zvětšovat také. Základní verze raytracingu umí pracovat pouze s bodovými zdroji světla, proto jsou výsledné stíny pouze ostré, nedokáží simulovat plynulý přechod mezi osvětlenou a neosvětlenou částí.

2.3.3 Texturování a antialiasing

Textura je vlastnost povrchu materiálu, která dokáže definovat jeho barvu, strukturu a vylepšit tak kvalitu zobrazovaného objektu [21]. Jedná se o levnější a rychlejší možnost jak vykreslit složitý model. Jednoduchý objekt s kvalitní texturou se vykresluje totiž lépe než geometricky složitý objekt. Proto je texturování dnes tak hojně využívané. Textury můžeme dělit podle dimenze na:

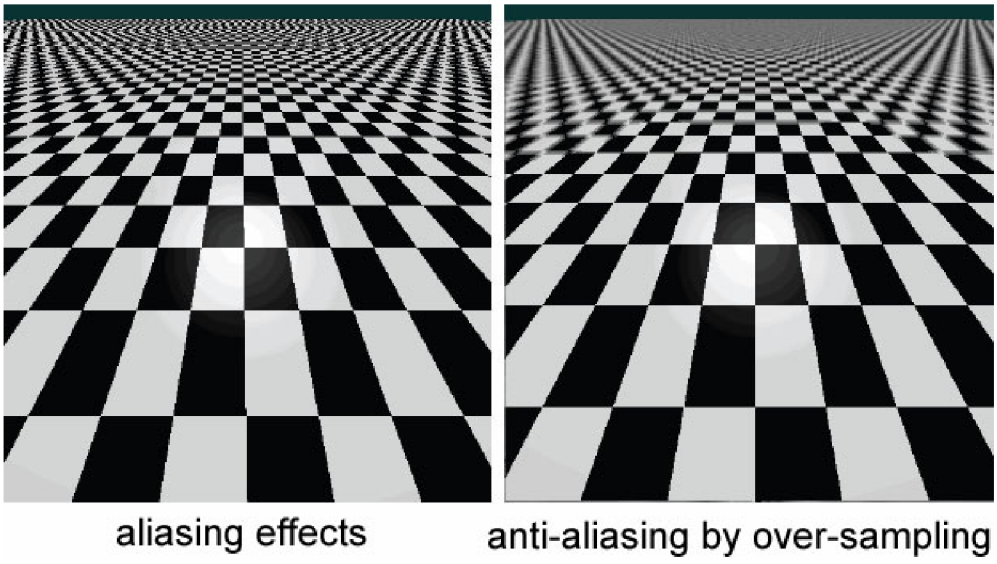
- jednorozměrné – opakující se podélné vzorky (čáry, atp. . . .)
- dvourozměrné – obrázky se souřadnicemi u a v
- trojrozměrné – obrázky mající souřadnice u, v a r pro hloubku
- čtyřrozměrné – trojrozměrné textury měnící se v čase

Mapování

Základní jednotkou textury je `texel`, v případě trojrozměrných textur je to `voxel`. Pokud chceme objektu přiřadit určitou dvourozměrnou texturu, musíme použít tzv `mapování`. Nejčastěji se používá mapování inverzní – povrch objektu je rozprostřen do plochy, na kterou se inverzní rovnicí daného objektu nanáší texely textury. Netriviální objekty však nemusí mít inverzní rovnici, v takovémto případě se musí provést jiná metoda mapování. Namapovaná textura může být větší než objekt, pak se nám nevykreslí celá. Pokud však bude menší můžeme ji nechat opakovat `REPEAT`, zarovnat na barvu okraje `CLAMP TO EDGE` či ukončit úplně `CLAMP`.

Mapování není podmíněno vzdáleností objektu od pozorovatele. Proto budeme-li mít objekt dostatečně vzdálený může se nám projevit aliasing. Aliasing je způsoben vysokými

frekvencemi v obrázku, která se projevuje tak, že sousední pixely na obrazovce nejsou sousedními texely v textuře. Dochází tak k šumu, kterého se chceme zbavit, potřebujeme antialiasing viz obr. 2.4. Základními prvky jsou buď filtry, které nám odfiltrují vysoké frekvence anebo prvky, které nám umožní vzorkovat větší frekvencí než je frekvence obrázku. Mezi takovéto metody patří např. FSSA, HRAA, Quincunx, ... ad.



Obrázek 2.4: Ukázka aliasingu a antialiasingu.
Zdroj: http://www.cs.berkeley.edu/~sequin/CS184/IMGS/anti-aliasing.jpg

Mipmapping

Nejčastěji se však používá metoda `mip-mappingu`. Jedná se o časově nenáročnou metodu, kdy jsou filtry předpočítány zmenšeniny základního obrázku textury. Ty ovšem po uložení všech rozlišení zabírají více paměti. Jednotlivé mipmapy se snažíme vytvořit ve velikosti mocnin 2, pro jednodušší použití filtrů, a abychom mohli mezi úrovněmi jednoduše interpolovat. Nejmenší úroveň mipmapy tvoří textura o velikosti 1x1 pixelů, největší rozlišení není omezováno. S narůstající či zmenšující se vzdáleností od pozorovatele se tak počítá úroveň (angl. level) mipmapy. Metod na počítání úrovně mipmapy je dle [15] více:

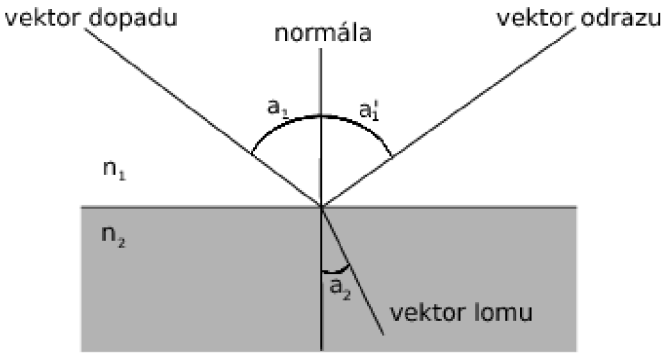
- derivační metody – derivative-based methods
- metoda derivování invariantu – Invariant derivatives methods
- metoda určení plochy – Area estimation
- metoda inverzního W – Inverse W method

Obrázek textury na dané úrovni je následně mapován a výsledná barva textury je dána texelem z tohoto obrázku.

2.3.4 Odrazy a lomy světla

Další z vlastností raytracingu je, že dokáže věrohodně zobrazovat průsvitné či zrcadlící se materiály. Pokud bychom však vraceli pouze barvu primárních paprsků nikdy toho

nemůžeme docílit. Proto potřebujeme z místa dopadu primárního paprsku vysílat ještě sekundární paprsky (angl. secondary-rays). Tyto paprsky můžeme rozdělit podle toho, zda se odráží či lámou.



Obrázek 2.5: Znázornění odrazu a lomu světla.

Pokud se bude sekundární paprsek lámat (angl. refraction), spočítá se lom podle Snellova zákona z [2] takto:

$$n_1 \sin a_1 = n_2 \sin a_2$$

Kde n_1 a n_2 jsou indexy lomu[10], které mají pro různé materiály různou hodnotu. Například vakuum má index lomu $n = 1$ a index lomu vody je $n = 1,333$ (viz [20]).

V případě odrazu (angl. reflection) se daný odraz spočítá z úhlu dopadu podle zákona odrazu:

$$a_1 = a_1^{'}$$

Kde je úhel dopadu a_1 roven úhlu odrazu $a_1^{'}$, který se odráží v rovině dopadu dané úhlem dopadu a kolmicí dopadu (normálou). Zákon odrazu je zjednodušením Snellova zákona (2.3.4), kdy paprsek nemusí procházet rozhraní více materiálů a indexy lomu můžeme tudíž z rovnice vypustit. Sinusy a jejich úhly jsou pak shodné.

Obrázek 2.5 nám znázorňuje, jak takovýto odraz či lom vypadá. Tyto paprsky se však v určitých scénách mohou odrážet/lámat do nekonečna, například dvě naproti sobě stojící zrcadla. Proto se u raytracingových metod musí použít zarážka v podobě hlouky zanořovací rekurze (odrazu). Tato zarážka nám v určitém okamžiku řekne, že další odrážený/lomený paprsek již vysílat nebudeme.

2.3.5 Algoritmus

Níže uvedený pseudo kód popisuje, způsob jakým lze metodu zpětného sledování pa-

[10] bezrozměrná veličina, která nám popisuje šíření světla

prsků jednoduše implementovat.

Algoritmus 1: Pseudo–algoritmus raytracingové metody.

foreach *pixel na obrazovce* **do**
    aplikuj osvětlovací model a stínování na získaný bod;
    barva bodu = Vrať barvu bodu();
    if *objekt je zrcadlo && rekurze > 0* **then**
        barva bodu += Vyšli sekundární paprsek odrazu(rekurze–);
    if *objekt průsvítá && rekurze > 0* **then**
        barva bodu += Vyšli sekundární paprsek lomu(rekurze–);
end foreach
return ***barvu bodu***

Kapitola 3

Nástroje

V této kapitole se budeme zaobírat nástroji, které byly využity k implementaci aplikace umožňující zobrazovat scény akcelerovanou i raytracingovou metodou. Nástroje byly vybírány tak, ať jde aplikace jednoduše použít na více platformách a aby případný další vývoj této aplikace nemusel být vyvíjen na jiných nástrojích.

Kapitola se dělí na tři podkapitoly, které se zaměřují na rozdílné části aplikace. V první části je uveden framework Qt 3.1 s jehož pomocí bylo vytvořeno uživatelské rozhraní pro výslednou aplikaci. Další částí je knihovna OpenScenegraph 3.2, která obstarává veškerou práci s virtuálními scénami. V poslední podkapitole 3.3 je popsána příprava prostředí pro tvorbu této práce.

3.1 Qt

Původní zadání výsledné aplikace se nezmiňovalo o tom, jakým způsobem má být uživateli umožněno s aplikací pracovat. Počáteční vývoj směřoval směrem terminálové aplikace. Po pár iteracích vývoje se však naskytl problém se vstupními argumenty, převážně s načítáním uživatelova 3D modelu a zadáváním hloubky zanořovací rekurze. Z tohoto důvodu se od dané chvíle začala aplikace zaměřovat i na jednoduché uživatelské rozhraní, které by umožnilo uživateli jednoduše zadávat model, který chce zobrazit a také jednoduše měnit požadovanou hloubku rekurze odrazu. Vzhledem k multiplatformnímu použití, dobré zdokumentovanosti, jednoduchému vývoji uživatelského prostředí a dlouhodobé podpoře byl zvolen framework Qt [13].

Qt je multiplatformní framework, který nám umožňuje využít svých nástrojů pro rychlou a efektivní tvorbu aplikací a uživatelských rozhraní. V různých verzích podporuje jak stolní, vestavěné tak i mobilní platformy. Framework tvoří intuitivní aplikační rozhraní pro jazyk C++ či jiné CSS/JavaScript-ové programování. Qt podporuje tyto platformy:

- Linux / Embedded Linux
- Windows / Windows Embedded
- Mac
- Solaris
- v nových verzích pak bude podporovat již i – Android a iOS

Pro instalaci frameworku stačí stáhnout volně dostupnou verzi z oficiálních stránek [1] a spustit instalátor. Knihovny a popřípadě i IDE jsou pak jednoduše spustitelné a použitelné.

Qt Creator IDE

Qt framework obsahuje integrované vývojové prostředí zvané Qt Creator IDE, ve kterém jsou knihovny potřebné pro vývoj pod tímto frameworkem. Spolu s vybavením na tvorbu GUI Qt Designer, tvoří mocný nástroj.

Toto vývojové prostředí umožňuje jednoduchou a přehledou správu více souborových projektů a jejich grafických rozhraní. Spouštění a případné ladění dokážeme provést a analyzovat přímo v IDE.

3.2 OpenSceneGraph

OpenSceneGraph je knihovna pro tvorbu výkonných 3D grafických scén [9]. Jedná se o otevřený software (angl. open source) jehož licence je popsána v sekci 3.2. Aplikace, které využívají tuto knihovnu, se dnes uplatňují v mnoha směrech. Například ve vědeckých, zdravotnických a vizuálních simulacích, ve hrách, virtuálních realitách nebo při modelování těchto scén.

Vývoj knihovny OpenSceneGraph částečně probíhá i na Fakultě informačních technologií VUT v Brně, takže již v zadání této práce bylo použití této knihovny vyžadováno. Hlavní výhodou této knihovny však (kromě multiplatformního využití) je, že dokáže jednoduše zobrazovat scény pomocí OpenGL. Další výhodou je skutečnost, že její vývoj pořád probíhá a knihovna je relativně dobře zdokumentovaná.

Na oficiální stránkách OpenSceneGraph je poslední vývojářskou verzí ke stažení verze 3.1.5 z roku 2013. Na Fakultě informačních technologií byla však začátkem letního semestru 2012/2013 k dispozici verze 3.1.0. Příprava prostředí této fakultní verze je popsána v kapitole 3.3.

Licence

Knihovna OpenSceneGraph je vydána pod „OpenSceneGraph Public License“ licencí, která je odvozena od licence „Less GNU Public License“ (LGPL). Tato licence dovoluje použít aplikaci pro komerční použití v případě statického linkování nebo vestavěných systémů.

LGPL licence umožňuje vývojářům využít a integrovat software s LGPL licencí do svých vlastních prací. Takovéto práce mohou být dokonce chráněné autorským právem aniž by vývojář musel vydat zdrojové kódy nebo vytvořené části softwaru.

Avšak software pod LGPL licencí musí být přístupný koncovému uživateli, musí mu být poskytnut přístup k modifikacím takového softwaru. V případě chráněných děl se tak oddělují části vytvořené a části přejaté LGPL softwarem.

3.2.1 Historie

Projekt začal jako komponenta grafu scény do simulátoru Hanga Glidinga, kterou vytvořil Don Burns. V roce 1999 se k vývoji simulátoru připojil Robert Osfield a společně s Burnsem pak začali vytvářet otevřený software pro simulování grafů scény.

[1] http://qt-project.org/downloads

Během prvního roku od uvedení první veřejné stránky a alfa verze softwaru, začali členové experimentovat se softwarem. V dalších etapách vývoje se software zdokonaloval až do podoby profesionálního zobrazovacího simulátoru.

První oficiální verzí byla verze 1.0 z roku 2005. Následovala rozšířená verze 2.0 (rok 2007), která přinesla možnost využít multi-jádrové systémy, další důležité nástroje pro práci s objekty, ale hlavně multiplatformní sestavování zdrojových souborů za pomoci aplikace `CMAKE`. V témže roce vyšli i první knížky s OpenSceneGraph tématikou. Vývoj pokračoval až do dnešní podoby, kde je k použití verze 3.0.1 vydaná roku 2011.

Nyní čítá základna přispívajících vývojářů k stabilní výstupní verzi zhruva 480 členů. Na oficiálních emailových seznamech jsou však záznamy tisíců dalších vývojářů, kteří nepřímo přispívají k vývoji.

3.2.2 Graf scény

Graf scény je n-ární stromová struktura, pro kterou platí, že každý uzel má jednoho předchůdce. Tyto struktury tvoří kořen scény (angl. root) na vrcholu stromu a listy stromu (angl. leaves) v jeho spodní části.

Kořen v knihovně OpenSceneGraph je objekt, který zahrnuje celou virtuální 2D nebo 3D scénu. Scéna se pak směrem k listům rozděluje do hierarchií objektů, které můžou představovat různá uskupení. Uskupení můžeme rozdělit několika způsoby – podle pozice objektů, animací objektů, definic objektů či logických vazeb mezi objekty. Listy grafu jsou pak již fyzické objekty samotné – vykreslitelné (angl. drawable) geometrie s vlastními materiálovými vlastnostmi.

Výhody použití grafů scény jsou popsány v níže uvedených sekcích, čerpáno z [10] a [21].

Výkon

Graf scény umožňuje využít vysokého výkonu díky dvěma klíčovým technikám – redukování objektů, které nejsou ve scéně viditelné, nemusíme tedy procházet takové množství dat a řazení vlastností podle stavů. Shodné objekty se stejnou texturou či materiálem jsou pak vykresleny najednou, tento postup šetří propustnost grafické karty – nemusíme přepínat stav grafického zařízení.

OSG podporuje různé druhy redukcí, například pohledovou či absorbční. Dále podporuje úrovně detailu ve scéně (angl. Level Of Detail – LOD), řazení podle OpenGL stavu nebo jazyk GLSL (OpenGL Shader Language).

Produktivita

Knihovna OpenSceneGraph umožňuje jednoduše psát výkonné grafické aplikace díky správě grafických částí přes aplikační rozhraní OpenGL. Namísto tisíců řádků kódu OpenGL stačí zavolat pár knihovních funkcí. Podporou dalších rozšíření umí knihovna optimalizovat zobrazování tříděním nebo redukováním scény.

Přenositelnost

Díky zapouzdření OpenGL poskytuje OpenSceneGraph i práci s nižší vrstvou zobrazování, načítání a ukládání dat. Tím snižuje/odstraňuje potřebu specifikace výstupní platformy.

Jádro je tak nezávislé od zobrazovacích oken, což dovoluje uživatelům přidávat vlastní knihovny pro použití oken. Knihovna `osgViewer` dovoluje zobrazovat nativní okna pod

operačnímy systémy Windows – systém Win32, UNIX – X11 a OSX – Carbon. K této knihovně můžeme přidat a integrovat i další nástroje jako Qt (viz kapitola 4.1.1), GLUT, FLTK, SDL, WxWidget, Cocoa a MFC. OpenSceneGraph obsahuje ukázky těchto integrací přímo ve své distribuci.

3.2.3 Technologie

Knihovna je napsána ve standardním jazyce C++ a také OpenGL (viz 2.2.1). Využívá také standardní knihovnu šablon (angl. standard template library – STL) pro kontajnery. Kromě C++ je OpenSceneGraph vyvíjen komunitou i pro jazyky Java, Lua a Python. Knihovna podporuje starší typy hardwaru a operačních systému i novější mobilní zařízen. Důvodem je podpora OpenGL verze 1.0 až 4.2, OpenGL ES 1.1 a 2.0. Můžeme ji tak využívat na platformách:

- Linux
- Windows
- FreeBSD
- Mac OS
- Solaris
- IRIX, HP-UX, AIX
- PlayStation2
- iOS, Android

Knihovna využívá návrhových vzorů pro snadnější pochopení tříd a jejich použitelnost. Jednotlivé moduly jsou uživatelem lehce rozšiřitelné.

Práce s databázemi

Pro načítání či ukládání dat slouží knihovna `osgDB`, která podporuje široké spektrum formátů a rozšířitelných dynamických zásuvných modulů. Distibuce obsahuje 55 různých modulů pro načítání 3D dat či obrázků.

3D data lze načítat například z formátů Alias Wavefront (.obj), 3D Studio MAX (.3ds), AutoCAD (.dxf), Quake Character Models (.md2). Direct X (.x), nativní ASCII formát .osg a také Inventor Ascii 2.0 (.iv)/ VRML 1.0 (.wrl).

OpenSceneGraph dokáže pracovat s obrázkovými formáty .rgb, .gif, .jpg, .png, .tiff, .pic, .bmp, .dds (včetně komprimovaných mipmap obrázků), .tga a quicktime (pod operačním systémem OSX).

Práce s uzly

OpenSceneGraph má sadu nástrojů pro práci s uzly/objekty přes různé oddělené knihovny, které dokážou být zkompilovány, anebo použity přímo. Nástroje jsou popsány v následující tabulce.

Knihovna	Použití
osgParticle	částicové systémy
osgText	antialiasingové fonty
osgFX	framework speciálních efektů
osgShadow	framework stínů
osgManipulator	interaktivní ovládání ve 3D
osgSim	vizuální simulace
osgTerrain	zobrazování terénů
osgAnimation	animace postav a neohebných částí
osgVolume	objemové vykreslování (podpora zdrav. modulu Dicom)

Tabulka 3.1: Nástroje pro práci s uzly/objekty.

3.2.4 Důležité třídy

Knihovna OpenSceneGraph obsahuje přes tisícovku tříd. V této podkapitole je přehled těch, které byly pro tuto práci použity.

osgQt::GraphicsWindowQt

Tato třída je odvozena od kořenové třídy `osgViewer::GraphicsWindow`, která poskytuje aplikačního rozhraní pro přístup k tvorbě a ke správě grafických oken a jejich událostí. `osgQt::GraphicsWindowQt` přidává frontu událostí na vrchol kontextu (`osg::GraphicsContext`), tím vytváří mechanismus k přizpůsobení okenních událostí.

Implementaci této třídy, jak již název napovídá, z velké části tvoří propojení událostí grafu scény a Qt a také tvorbu tzv widgetu. Tento widget je třídy `osg::GLWidget` a přímo dědí vlastnosti Qt komponenty `QGLWidget`.

Do takto vytvořeného widgetu pak lze již zobrazovat, za pomocí `osgViewer::View`, graf scény.

osgViewer::View

Třída `osgViewer::View` slouží ke správě pohledu na scénu a jeho zobrazování za pomoci OpenGL. Má na starosti hlavní kameru pohledu (sekce `osg::Camera`) a případně seznam dalších podřízených kamer, které jsou vzhledem k hlavní kameře relativní. Pokud není žádná z vedlejších kamerem připojena k pohledu, hlavní kamera řídí ovládání a zobrazování scény je implementováno v této kameře. Nadruhou stranu, jestliže jsou k pohledu připojeny další kamery, zobrazování je implementováno v podřízených kamerách.

Nejdůležitější vlastností třídy je však skutečnost, že obsahuje informace samotné scény, hlavního světla a také že umožňuje nastavovat data scény a její ovládací prvky (viz sekce `osgGA::GUIEventHandler`).

osg::Camera

`osg::Camera` je podtřídou `osg::Transform`, která reprezentuje nastavení kamery.

Transformace je uzel skupiny, jejíž potomci jsou transformovány maticí 4x4. Používá se pro pozicování objektů ve scéně, animaci a umožňuje také funkcionalitu ovládání – přibližování apod. Nemá žádné nastavovací funkce, obsahuje pouze rozhraní pro definici transformační matice. V případě kamery se tak jedná o projekční matici (matici pohledu).

Kamera implementuje práci s tranformacemi, nastavuje případné matice podle typu projekce či samotné rysy a geometrii kamerového pohledu. Nejdůležitější funkcí je však možnost nastavení widgetu, do kterého bude kamera promítat.

osgGA::GUIEventHandler

Tato třída představuje základní rozhraní, které nám umožňuje spravovat události vyvolané v GUI. Události jsou vyvolávány třídou `osgGA::GUIEventAdapter`, zpětnou vazbu obstarává třída `osgGA::GUIActionAdapter`. Tento mechanismus pak třídě `osgGA::GUIEventHandler` dovoluje reagovat na příchozí událostí implementovanou odezvou.

osg::NodeVisitor

Třída slouží k typově bezpečným operacím nad objekty typu `osg::Node`. `NodeVisitor` umožňuje procházet graf scény a aplikovat změny na vybrané uzly. Vzor funguje na principu dvojího voláni, kdy se ve visitoru volá funkce `apply()` pro aplikováni na určitý objekt. Spuštění celého procházaní se ale aplikuje příkazem `node→ accept(*NodeVisitor)`.

osg::Node

Základní třída pro všechny objekty ve scéně, která poskytuje rozhraní pro nejpoužívanější operace. Nejdůležitějšími vlasnostmi uzlu jsou přítomnost dvou proměnných – tzv `NodePath` a `StateSet`, obě v knihovně `osg`. První z nich je seznam předků daného uzlu, který potřebujeme například pro transformaci lokálních souřadnic do světových.

Ve druhé proměnné je uložený stav materiálu daného uzlu. Tento stav má vlastní atributy `osg::StateAttribute`, které uchovávají různé další vlastnosti. Jako příklad může sloužit textura, nastavená světla či samotný materiál.

osg::Drawable

Třída Drawable slouží jako základní třída pro tzv vykreslitelné geometrie. V knihovně OpenSceneGraph jsou všechny renderovatelné objekty odvozeny od této třídy, neobsahuje však žádná primitiva. Ta jsou k dispozici až v podtřídách např. `osg::Geometry`.

Důležité také je, že `Drawable` není uzlem, takže nejde procházet přímo visitorem ani přímo přídávat do scény. Pro přídávání se musí zaobalit do třídy `osg::Geode`.

Renderovací vlasnosti jsou uloženy jako v případě uzlu v `osg::StateSet-u`, tento stav může být sdílen více uzly či drawably. Pro lepší výkon jsou dokonce mezi geody některé drawably sdílené.

U podtřídy `osg::Geometry` je důležité, že máme přístup k polím obsahujícím vrcholy, normály či jiné indexy, ku příkladu index do pole barev apod.

osg::LightSource

Tato třída reprezentuje koncový uzel, který definuje světlo ve scéně. Jeho nejdůležitější vlastností je, že obsahuje atribut `osg::Light` a že je odvozen z třídy `osg::Node`, takže jej lze procházet visitorem.

osg::Light

Stav světla je uložen ve třídě `osg::Light`. Tato třída zapouzdřuje funkcionalitu `glLight()` z OpenGL. Světel může být i více a lze je definovat jménem. Všehna světla pak ještě můžou být zapnuta/vypnuta nastavením `glEnable`/`glDisable`.

Světlo samotné má pak tyto parametry:

- `GL_AMBIENT` – ambientní („všudypřítomná“) složka světla
- `GL_DIFFUSE` – difuzní složka světla
- `GL_SPECULAR` – spekulární (odrazová) složka světla
- `GL_POSITION` – pozice světla
- `GL_SPOT_DIRECTION` – směr kuželového světla
- `GL_SPOT_EXPONENT` – intenzita kuželového světla
- `GL_SPOT_CUTOFF` – maximální úhel kuželového světla
- `GL_CONSTANT_ATTENUATION` – konstantní útlum světla
- `GL_LINEAR_ATTENUATION` – lineární útlum světla
- `GL_QUADRATIC_ATTENUATION` – kvadratický útlum světla

osgUtil::IntersectionVisitor

Tato třída se používá pro testování průsečíků ve scéně, prochází scénou a vyhodnocuje je použitím obecné třídy `osgUtil::Intersector`, která implementuje samotné počítání průsečíků.

osgUtil::LineSegmentIntersector

`LineSegmentIntersector` je odvozen od obecné třídy `osgUtil::Intersector`. Jedná se již o konkrétní třídu implementující paprskové průsečíky v grafu scény, musí však být použita spolu se třídou `osgUtil::IntersectionVisitor`, pro procházení scénou.

osgUtil::LineSegmentIntersector::Intersection

Tato struktura reprezentuje již získaný průsečík v daném bodě. Tvoří jej tyto složky:

- nodePath – proměnná typu `osg::NodePath` reprezentující seznam rodičů trefeného uzlu/drawablu
- drawable – ukazatel na zasažený objekt
- matrix – ukazatel na matici objektu
- localIntersectionPoint – zasažený trojrozměrný bod v lokálních souřadnicích
- localIntersectionNormal – trojrozměrný vektor, který reprezentuje normálu v bodě
- indexList – seznam indexů pro získání vrcholů zasažené primitivy

- ratioList – seznam „mír“, pro interpolaci bodu v rámci trefené primitivy
- primitiveIndex – index určuje kolikátou primitivu (trojúhelník) musel `osgUtil::LineSegmentIntersector` projít, než našel výsledný průsečík
- ratio

3.3 Příprava prostředí

Tato kapitola vychází z [17] a pojednává o přípravě prostředí pro operační systém Ubuntu, na kterém byla tato práce vyvíjena. Jelikož poskytnuté zdrojé soubory nebyly z oficiálního zdroje, ale jednalo se o fakultní verzi, bylo potřeba nejen zkompilovat knihovnu OpenSceneGraph, ale také přidat modul pro podporu formátu `.iv`. Tento modul je potřeba pro otvírání modelů tohoto formátu knihovnou OpenSceneGraph.

Open Inventor

`Open Inventor` je objektově-orientovaná, multiplatformní 3D sada nástrojů určených pro vývoj průmyslových aplikací v jazycích C++, .NET nebo Java. Jedná se o jednoduché API s rozšířitelnou architekturou, které má svůj nativní formát datových souborů (`.iv`).

Jeho vývoj firmou Visualization Sciences Group (VSG) však není věřejný. Náhradou tak je níže uvedený `Coin3D`.

Coin3D

Tento nástroj je založen na OpenGL a jedná se o 3D grafickou knihovnu, která vychází z `OpenInvertor-u 2.1`. `Coin3D` je s tímto invertorem plně kompatibilní, takže dokáže bez potíží načítat nativní soubory `.iv`.

`Coin3D` je od roku 2011 vedený jako otevřený software a ke stažení je na oficálních stránkách[2].

CMAKE

Tento nástroj je využívám k multiplatformnímu překladu a kompilování zdrojových souborů. Pokud má `CMAKE` k dispozici konfigurační soubor, vygeneruje podle jeho nastavení `Makefile`, kterým lze na dané platformě zkompilovat nakonfigurované zdrojové soubory. Na oficiálních webových stránkách[3] lze stáhnout verzi 2.8.10 vydanou v roce 2011.

Pro interakci skrze uživatelské rozhraní je možné spustit program `CCMAKE`. Tímto nástrojem lze jednoduše nastavovat konfigurační soubor. Přídávat například další zásuvné moduly a jiná vylepšení.

3.3.1 Překlad

Zdrojové kódy knihovny OpenSceneGraph jsou dodávány již s předpřipraveným konfiguračním souborem pro nástroj `CMAKE`. Pro obyčejnou kompilaci stačí spustit `./configure` skript.

[2] https://bitbucket.org/Coin3D/coin/wiki/Home
[3] http://www.cmake.org/

Pokud chceme kompilaci upravit je nejjednodušší způsob použití nástroje `CCMAKE`. Ve složce se zkompilovanými zdrojovými soubory jej tedy spustíme příkazem `CCMAKE .` . Pokud `CCMAKE` některé nainstalované nástroje jako Qt či Coin3D rozpozná, automaticky je přidá do knihovny `osgPlugins`. V opačném případě musíme po otevření uživatelského okna, nastavit patřičná makra:

- INVENTOR_INCLUDE_DIR – cesta k souborům `Coin3D`
- INVENTOR_LIBRARY_DEBUG – cesta ke knihovně
- INVENTOR_LIBRARY_RELEASE – cesta ke knihovně

Po přidání zásuvného modulu `Coin3D` do konfiguračního souboru, pak již nastavení uložíme a spustíme kompilaci. `CMAKE` vygeneruje z takto nastaveného konfiguračního souboru `Makefile`, který stačí spustit. Spuštění se provádí příkazem `make` ve zdrojové složce, následuje příkaz `make install`.

Abychom měli knihovnu zkompilovánu pro všechny uživatele například ve složce `/usr/local/lib` musíme tento příkaz spustit pod administrátorským účtem. V opačném případě se knihovna zkompiluje do domovského adresáře konkrétního uživatele.

Po dokončení kompilace knihovny a její instalaci musíme však ještě přidat cestu k této knihovně do systémové proměnné `LD_LIBRARY_PATH`. Jedním ze způsobů jak toho dosáhnout je zapsat zvolenou cestu ke knihovně do souboru `/etc/ld.so.conf.d`. Po zapsání a uložení toho souboru pak stačí už jen potvrdit změny příkazem `ldconfig`, po kterém se promítnou změny.

Pro použití v Qt se musí ještě cesta ke knihovně OSG dopsat ke knihovnám používaných v Qt projektu. Do souboru `Projekt.pro` se tedy připíše `LIBS += -L/`*`cesta_k_OSG`* a dále také všechny knihovny, které chceme použít. Knihovny jsou oddělény zpětným lomítkem například `-losg \`.

Knihovna OpenSceneGraph po tomto postupu je již k dispozici pro vývoj i s modulem `Coin3D`.

Kapitola 4

Raytracer

Kapitola popisuje implementaci aplikace, která zobrazuje virtuální scény za pomocí knihovny OpenSceneGraph. Výsledná aplikace by kromě zobrazení akcelerovanou metodou měla umět zobrazovat i raytracingem – metodou sledování paprsku.

V kapitole 4.1 je popsáno uživatelské rozhraní výsledné aplikace a také implementace propojení knihovny OpenSceneGraph a frameworku Qt. V další části 4.2 je popsáno spuštění renderování raytracingovou metodou. Kapitola 4.3 pak popisuje implementaci konkrétní třídy, která samotné paprsky vyhodnocuje.

Poslední kapitolou je vyhodnocení výsledků.

4.1 Uživatelské rozhraní

Uživatelské rozhraní této práce je vytvořene přes framework Qt. Tento nástroj je popsán v kapitole 3.1.

Hlavní smyčka

Hlavní program se tak, jak je zvykem u Qt aplikací, skládá pouze z vytvoření jednoho objektu `QApplication` a jednoho hlavního okna `MainWindow`. Takto vytvořený objekt aplikace spustíme metodou `exec()`, která nastartuje hlavní smyčku programu. Ve smyčce se zobrazí vytvořené hlavní okno a zároveň se zpřístupní sloty k přijetí signálů vyvolaných stisknutím tlačítek v menu.

Základní okno

Hlavní okno programu `MainWindow` je odvozeno od třídy `QMainWindow`. Oknu je přiřazeno GUI formou formuláře vytvořeného nástrojem Qt-Designer. Tento `Ui::MainWindow` formulář v sobě obsahuje kromě informací o samotném oknu, ještě položky menu a hlavní (centrální) widget.

Menu samotné je jednoduché a intuitivní. Obsahuje pouze nezbytné prvky, které by neměly rušit. V menu tak uživatel najde jen možnosti pro otevření souboru (položka `Open`), nastavení zanořovací rekurze `Settings`, spuštění nápovědy (`About`) a v poslední řadě možnost vypnutí tlačítkem `Exit`.

`Open` vyvolá signál s názvem souboru pro otevření. Tento signál se pošle centrálnímu widgetu, který daný soubor načte do svých dat.

Zmáčknutí `Settings` se zobrazí dialog implementovaný třídou `depthDialog`, ve kterém lze vidět hloubku rekurze a zároveň ji lze posuvníkem měnit. Implicitní hodnota rekurze je 2. Změna se signálem posílá rovnou i centrálnímu widgetu, který si ji uloží a může s ní dále pracovat.

4.1.1 Propojení Qt a OSG

Centrální widget hlavního okna je složitější, protože v něm potřebuje aplikace spravovat zobrazování knihovnou OpenSceneGraph. Pro účely vytvoření toho widgetu je napsána třída `ViewerWidget`.

Tato třída ve svém konstruktoru vytvoří `QWidget`, jehož geometrie je přes celou centrální plochu hlavního okna. Dále se nastaví jednovláknový model, kterým specifikuje počet průchodů při zobrazování. A vytvoří perspektivní kameru (třída `osg::Camera`), která je vytvořena metodou `createCamera` podle velikosti widgetu. Její kontext je v této metodě vytvořen, tak jak je uvedeno v kapitole 3.2.4 třídou `osgQt::GraphicsWindowQt`. Jelikož se jedná o Qt widget, musíme jej vykreslovat pravidelně po určitých intervalech, k tomu slouží `QTimer`, který se spustí na konci konstruktoru.

Samotný widget je pak vytvořen metodou `addViewWidget`, ve kterém je vytvořen objekt `osgViewer::View`. Tomuto objektu se přiřadí vytvořená kamera a nastaví se ovládání `osgGA::TrackballManipulator`. Co je ovšem stěžejní pro výslednou aplikaci je schopnost přiřazovat objektu `osgViewer::View` data scény. Po příchodu signálu z hlavního okna, že má být načten uživatelem vybraný model, se soubor s tímto modeleme nahraje do toho objektu.

Důležitý je také fakt, že objektu `osgViewer::View` můžeme nastavit správce událostí. Ten je detailně popsán v podkapitole 4.1.2.

4.1.2 Spuštění raytracingu

Jak již bylo zmíněno dříve, centrální widget má v sobě zakomponovaný objekt třídy `osgViewer::View`. Tomuto objektu lze nastavit správce událostí. Pro správu vlastních událostí a reakcí na ně je implementována třída `renderHandler`, která je odvozena z `osgGA::GUIEventHandler`.

Třída `renderHandler` v konstruktoru ukládá ukazatele na samotný `View` a ukazatel na centrální (rodičovský) widget. Správce implementuje reakce na stisk kláves mezerník a enter. Po stisku těchto dvou kláves se zavolá statická metoda `createRender(this)`, kdy se v parametru funkce předává ukazatel na třídu `renderHandler`. V této metodě se vytvoří instance třídy `render`, která se spustí metodou `startRender()`, tím se započne samotné renderování (viz 4.2).

Toto řešení ještě neimplementuje renderování ve více vláknech, avšak volání statických metod je u nich vyžadováno. Proto je metoda `createRender()` statická, pro pozdější rozšíření na vícevláknové renderování.

4.2 Renderer

Samotné renderování je prováděno objektem třídy `render`. Konstruktor objektu si uloží předané ukazatele na `osgViewer::View` a `ViewerWidget`. Uloží si také hodnotu zanořovací rekurze, která je uložena v centrálním widgetu a vytvoří objekt pro procházení průsečíků ve scéně konstruktorem třídy `osgUtil::IntersectionVisitor`.

Spuštění renderování se provádí metodou `startRender()`. V ní renderer zkopíruje existující kameru z centrálního widgetu. Nad touto kopií pak provede vytvoření vektorů (detailně v podkapitole 4.2.1).

Až renderer vytvoří všechny potřebné vektory, skopíruje pohled kamery do obrázku a otevře výsledné okno „Result“ viz 4.2.2, které se inicializuje zobrazením uloženého obrázku.

Následuje vysílání paprsků přes všechny pixely obrazovky skrz celou průmětnu scény. Každý paprsek je vytvořen jako instance třídy `intersectionHit`, které je předána struktura paprsku `osgUtil::LineSegmentIntersector::Intersection` získaná za pomoci objektu `osgUtil::LineSegmentIntersector`.

Objekt `intersectionHit`, jehož detailnější implementace je popsána v 4.3, obsahuje metodu `getColor(depth)`. Tato metoda vrátí barvu průsečíku dle zadané rekurze v parametru `depth`. Následně se vytiskne získaná barva do výsledného obrázku okna „Result“.

4.2.1 Vytvoření vektorů

Renderer potřebuje ještě před vysláním prvního paprsku načíst pro urychlení některá data. Jedná se zejména o vektory světel a zrcadel, dále je potřeba vytvořit mipmapy všech textur a spustit optimalizaci. Vektory jsou pak uloženy v rendereru tak, aby k nim měly přístup všechny vyslané paprsky.

Vektor světel

Získání tohoto vektoru implementuje třída `lightsVisitor`, která dědí vlastnosti `osg::NodeVisitor`-u. Aplikováním metodou `accept(*lightsVisitor)` na kameru se projdou všechny `osg::LightSource` objekty ve scéně a uloží se do vektoru `_sourceVector`.

Jelikož má zdroj světla jako svou proměnnou stav světla `osg::Light` musíme vytvořit vektor `_lightVector`, který je obsahuje. K výslednému vektoru se ještě přidá globální světlo scény získané z `osgViewer::View` metodou `getLight()`.

Metoda `lightsVisitor::getLightVector()`, vrací získaný vektor světel, ten pak uložíme do rendereru.

Vektor zrcadel

Vektor zrcadel je získán podobně jako vektor světel. Třída `mirrorVisitor` prochází všechny uzly ve scéně a kontroluje zda v jejich uživatelských datech není proměnná typu string s hodnotou `Photorealism: Material.reflectiveColor R G B`. Kde R, G a B určuje barvu odrazu zrcadla.

V případě, že tam takováto proměnná existuje a je nenulová, uloží se ukazatel na daný uzel a jeho barva odrazu do struktury `mirror`. Uzlem může být `osg::Drawable`, v tom případě se ukládá ukazatel na drawable, v ostatních případech se ukládá ukazatel na základní třídu `osg::Object`.

Vektor zrcadel následně musíme uložit do rendereru, aby k němu měli přístup všechny ním vyslané paprsky. Získáme jej metodou `mirrorVisitor::getMirrors()`.

Mipmapy textur

Průchod grafem scény a vytváření mipmap provádí `textureVisitor`. Pro každý uzel scény, který má jako svůj atribut texturu, získá obrázek textury a předá jej jako parametr funkci

`createMipMaps(image)`.

V této funkci dojde k alokaci nového obrázku, jehož velikost je dostatečně velká na uložení všech úrovní mipmap. Obrázek je typu `osg::Image`, který umožňuje vložit mipmapy jako vektor offsetů funkcí `setMipmapLevels(vector)`.

Základní obrázek je zkopírován a další úrovně jsou vytvořeny za pomoci metody `scaleImage(res, res, 1)` vždy z úrovně předcházející. Tato metoda zmenší obrázek, který ji zavolal, na zadané rozlišení – šířka, výška, hloubka. Zmenšování se provádí na původním obrázku, data jsou pak překopírována do obrázku nově alokovaného, který je poté nastaven jako hlavní obrázek textury.

Tím se docílí, že všechny materiály s texturou budou mít nastavený stejný obrázek jako původní scéna, avšak v něm již budou k dispozici i všechny úrovně mipmap.

Optimalizace

Částečná optimalizace výpočtu je již vytvořením předchozích vektorů provedena. Jednotlivé průsečíky totiž nemusí znovu procházet či přepočítávat světelné zdroje a zrcadla, ale stačí jim jen porovnávat vytvořené vektory.

Znatelně rychlejšího počítání průsečíků se dosáhne procházením grafu scény využitím KD stromů. KD stromy jsou speciální případy binárních stromů určené k popisu libovolného k-dimenzionálního prostoru. Jak se píše v [21], řezné roviny jsou však vždy kolmé na na některou souřadnou osu. Knihovna OpenSceneGraph implicitně využívá KD stromů k urychlení, pokud ovšem existují. Pro jejich použití tedy stačí pouze KD strom dané scény vytvořit. Vytvoření KD stromu se skládá ze dvou částí – vytvoření objektu třídy `osg::KDTreeBuilder` a průchod tímto objektem přes všechny uzly ve scéně. Jelikož je `KDTreeBuilder` odvozen od třídy `osg::NodeVisitor`, dá se tento průchod aplikovat použitím metody
`accept(*KDTreeBuilder)` u kořenové kamery.

4.2.2 Zobrazování výsledků

Po spuštění renderování se vytvoří výsledné okno s názvem „Result“, které přenese pohled z hlavní kamery do toho okna skopírováním obrázku pohledu kamery. Tento obrázek je uložen jako bitmapa `QPixmap`, jejíž velikost je shodná s velikostí pohledu kamery.

Vlastností pixelové mapy je, že můžeme přímo přistupovat k jednotlivým pixelům. Výsledek je tedy zobrazován průběžně vykreslováním jednotlivých pixelů do této pixmapy. Barvu vykreslovaného pixelu získáme zpracováním primárního paprsku v `intersectionHit`.

4.3 Zpracování průsečíků

Třída `intersectionHit` slouží pro získání barvy uzlu, který průsečík zasáhl. V jejím konstruktor je uložen předaný ukazatel na renderer 4.2, pro přístup k vektorům (viz 4.2.1). Dále je konstruktoru předán `osgUtil::LineSegmentIntersector::Intersection` – ukazatel na průsečík vypočítaný knihovnou OpenSceneGraph, ve kterém jsou již spočítány potřebné údaje.

Hlavní funkcí je získání barvy uzlu průsečíku zasaženého paprsku, k tomu slouží metoda `getColor(depth)`, která do určité hloubky rekurze `depth` vrátí výslednou barvu.

Geometrie

Jednou z proměnných průsečíku OSG je uzel drawable, který má vlastní geometrii. Geometrie `osg::Geometry` uchovává informace o vrcholech a barvách. Barva geometrie se určuje podle typu vazby barvy na objekt (angl. bind):

- BIND_OFF – žádná barva, výsledná barva bude šedá
- BIND_OVERALL – barva geometrie je jenom jedna, získá se jako první položka pole barev
- BIND_PER_PRIMITIVE – geometri se skládá z primitiv, například trojúhelníků, každé primitivum má tak vlastní barvu (barva prvního vrcholu)
- BIND_PER_PRIMITIVE_SET – ve skupině je více primitiv, skupina má pak nastavenu jednu barvu (první primitivy)
- BIND_PER_VERTEX – barva bodu je určena interpolací barev vrcholů zasažené primitivy

Metodou `getGeomColor()` se určí typ vazby, která je uzlu nastavena a podle typu je výsledná barva geometrie získána z pole barev.

Obrázek 4.1: Vazba barvy `PER_PRIMITIVE_SET` – primitivu tvoří dva trojúhelníky čtverce.

Stav uzlu

Pro získání barvy uzlu se musí spočítat napřed `osg::StateSet` stav objektu. Nelze totiž použít pouze stav daného uzlu, ale musí se projít i rodičovské uzly, kvůli možné dědičnosti nastavení. Výpočet celkového stavu uzlu provádí funkce `getNodeStateSet()`.

V případě, že uzel nějaký stav má, získají se z něj tři základní atributy – textura, materiál a světelný model (`osg::LightModel`). Ze světelného modelu se uloží globální ambientní složka funkcí `osg::LightModel::getAmbientIntensity()`.

Materiál

Další nutnou položkou je získání barvy materiálu `osg::Material`. Z nastavení uzlu tedy dostaneme, který tyto vlastnosti má. Metoda `getMaterialColor(material)` uloží ambientní, difusní i spekulární složku materiálu. Pokud bude nastavená položka `ColorMode`, která indikuje, že má barva geometrie přednost před barevnou složkou materiálu, barva této složky se přepíše barvou geometrie.

Textura

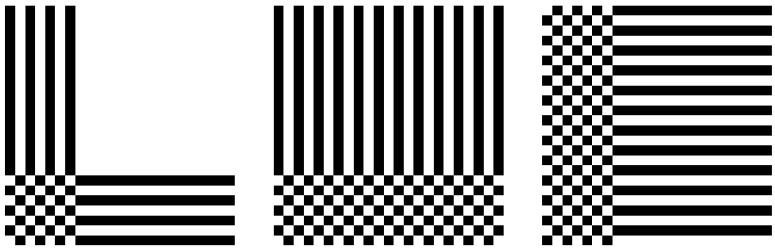
Barvu povrchu nemusíme použít pouze z geometrie objektu nebo jeho materiálu, ale také z přidělené textury. Textura `osg::Texture` se získá jako atribut stavu `osg::StateSet` a je předána funkci `getTexColor(texture)`. Výsledkem této metody je barva zasaženého texelu textury.

První důležitým výpočtem je výpočet souřadnic texelu. Ty jsou spočítány za pomocí vrcholů zasažené primitivy a souřadnicovým polím, ve kterých jsou uloženy informace, na jaké pozici texelu leží daný vrchol. Podobně jako tomu bylo u interpolace barvy `BIND_PER_VERTEX`, tak i v tomto případě se musí souřadnice bodu interpolovat ze souřadnic vrcholů.

Interpolovaný bod by v této chvíli ještě však nešel použít pro získání barvy. Jak je uvedeno v sekci Mapování kapitoly 2.3.3, texturu lze opakovat, ukončit či zarovnat na hranici. Proto je nutné ještě přetransformovat interpolovaný bod do souřadnic [0-1;0-1]. Knihovna OpenSceneGraph rozlišuje několik typů mapování pro každou souřadnici (s, t, r). Metoda `osg::Texture::getWrap()` může vrátit tyto typy:

- REPEAT – opakování textury dané souřadnice (`GL_REPEAT`)
- MIRROR – opakování textury zrcadlením (`GL_MIRRORED_REPEAT_IBM`)
- CLAMP – ořezání souřadnice (`GL_CLAMP`)
- CLAMP_TO_EDGE – zarovnání souřadnice na hranu textury (`GL_CLAMP_TO_EDGE`)
- CLAMP_TO_BORDER – zarovnání souřadnice na hranici (barvu rámečku) textury (`GL_CLAMP_TO_BORDER_ARB`)

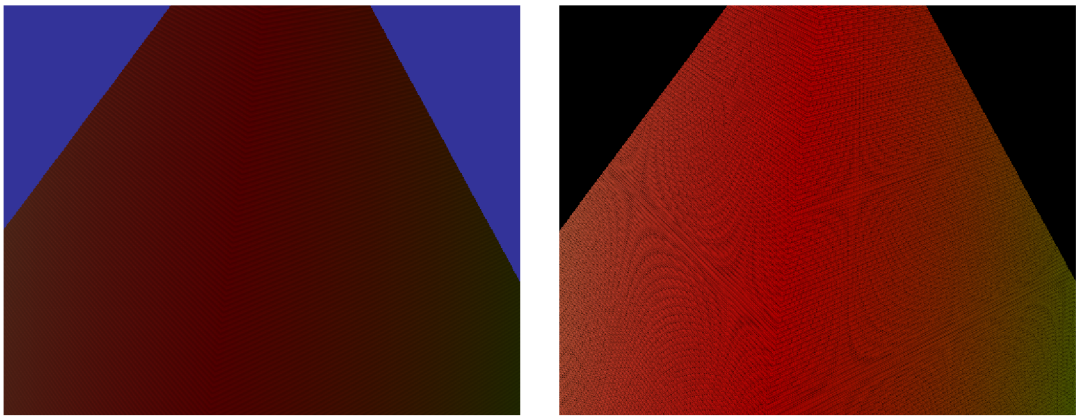
Avšak poslední tři možnosti jsou nahrazeny pouze zarovnáním na hranu textury. Možnost `CLAMP_TO_BORDER` je totiž zastaralá a `CLAMP` se prakticky nevyužívá, protože chceme mít texturu vždy přes celý objekt.



Obrázek 4.2: Zleva: Ořezání souřadnic pomocí `CLAMP`, opakování souřadnic v ose s a ose t.

Po transformaci interpolované souřadnice, ji lze již použít pro získání barvy texelu. Třída `osg::Image` umožňuje získat barvu metodou `getColor(texCoord)`, kdy se ji jako parametr předává souřadnice texelu. Výslednou barvou textury na daném průsečíku je pak barva tohoto texelu.

Výběr obrázku, který se má použít pro získání barvy texelu ovšem závisí na aliasingu. Je potřeba vybrat správnou úroveň mipmapy, popřípadě vybrat mipmapy rovnou dvě a ty interpolovat mezi sebou. Pak se jedná o trilineární interpolaci. Metod pro výpočet této úrovně je více (viz 2.3.3), avšak jejich výpočet lze na metodu sledování parsku jen složitě implementovat, popřípadě je jejich efektivnost velice nízká.



Obrázek 4.3: Obrázek vlevo ukazuje správné vyhodnocení mipmap, obrázek vpravo nedořešený aliasing.

4.3.1 Modulace barev

Máme-li objekt, který má barvu materiálu/geometrie a naneseme na něj ještě texturu, výsledná barva musí být jejich kombinací. V metodě `modulateColor()` se kombinují barvy zdrojové (angl. source), cílové (angl. destination) a barva nastaveného prostředí. Implicitně je zdrojem materiál a cílem textura. Ještě před samotnou modulací musí být na materiál aplikován osvětlovací model, jinak by se neprojevily světelné efekty. Na obrázku **??** lze vidět modulaci barvy geometrie a textury.

Knihovna OpenSceneGraph a její třída `osg::TexEnv`, která je atributem textury a obsahuje potřebnou barvu prostředí, dovoluje kombinovat barvy těmito způsoby:

- DECAL – použije se jen cílová barva
- MODULATE – kombinace obou barev
- BLEND – zesvětlení barev
- REPLACE – nahrazení jedné nebo druhé barvy
- ADD – sečtení barev

Podle specifikace OpenGL [14] lze barvy kombinovat více způsoby, avšak ty OSG nenabízí. Najdeme v ní také rovnice pro popis všech těchto kombinací.

4.3.2 Osvětlovací model a stínování

Pro implementaci osvětlení byl zvolen Phongův osvětlovací model, jehož popis je v kapitole 2.3.1. Výpočet modelu se spustí metodou `PhongModel()`.

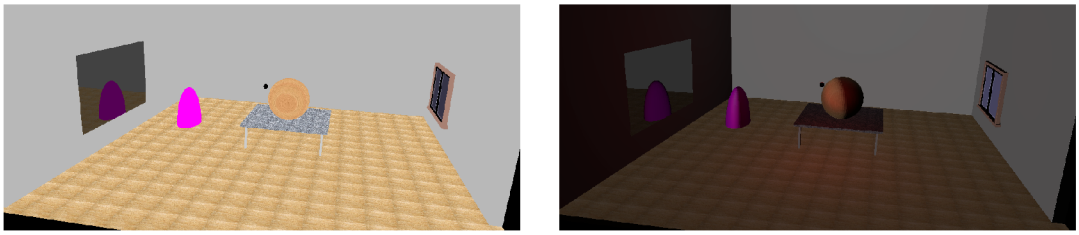
Uvnitř metody se získá bod průsečíku a normála v tomto bodě. Oba 3D vektory ve světových souřadnicích. Vypočítá se vektor k pozorovateli, kdy je místo oka použita buď pozice kamery anebo bod odrazu zrcadla. Pozice do proměnné `position` se získá funkcí `osg::Camera::getViewMatrixAsLookAt(position,center,up)`.

Z rodičovského renderu se získá počet světel, ovšem ještě před smyčkou procházející přes všechna světla se vytvoří nový `osgUtil::LineSegmentIntersector`. Tento objekt bude

počítat průsečíky stínových paprsků a je nastaven do `osgUtil::IntersectionVisitor`, který je k dispozici v renderu.

Dále se vytvoří `worldVisitor`, který slouží pro získání matice pro převod lokálních souřadnic na světové.

Ve smyčce pro všechny světla ve scéně pak metodou `osg::Light::getPosition()` zjistíme pozici světla. Tuto pozici převedeme do světových souřadnic a vypočítáme vektor ke světlu jako rozdíl pozice světla a bodu dopadu paprku. Ze světla ještě získáme jeho směr použitý pro kuželová světla.



Obrázek 4.4: Znázornění scény zleva: Bez osvětlovacího modelu, s osvětlovacím modelem bez počítání stínů.

Převod do světových souřadnic

Převod probíhá tak, že se na zdroj světla (`osg::LightSource`) aplikuje visitor implementovaný ve třídě `worldVisitor`. Tento visitor prochází předky daného zdroje světla a uloží si potřebnou matici pro tranformaci. Metodou `osg::computeLocalToWorld(getNodePath()` u posledního předka vytvoří matici podle cesty grafem tvořenou předky. Jakmile je procházení u konce, nalezená matice je použita pro převedení lokálních souřadnic pozice světla do světových.

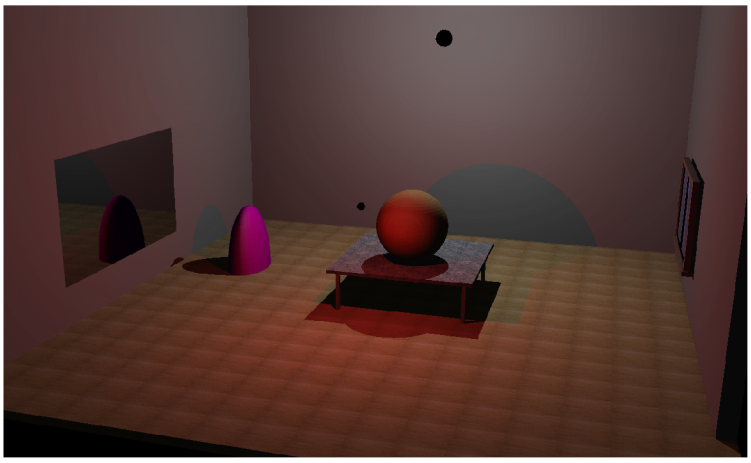
Stíny

Pokračuje se nastavením začátku a konec stínového paprsku a jeho vysláním do scény. V případě, že paprsek trefí nějaký objekt musíme zjistit jestli se nejdená o světlo. Tento výpočet je implementován ve funkci `notInShadow(picker)`, kde `picker` představuje ukazatel na `osgUtil::LineSegmentIntersector`, který vypočítal průsečíky.

Tak jako má zrcadlo data `Photorealism: Material.reflectiveColor R G B`, některé objekty mohou mít nastaveno `Photorealism: Material.castShadow X`. V případě, že je $X = 0$, průsečík s tímto materiálem se zanedbává a prohledává se další průsečík daného paprsku. Proto je nutné zjisti jestli je tento řetězec v uživatelských datech, k tomu slouží metoda `haveZeroCastShadow(object)`. Parametrem je `osg::Object` u kterého se přítomnost řetězce kontroluje.

Nestačí ovšem zjistit jestli je řetězec u objektu zasaženého průsečíkem. Kvůli dědičnosti musíme projít i všechny rodiče objektu.

Stíny jsou implementovány tak, že pokud nemá objekt ani jeho předci `zeroCast` a paprsek nedoletěl až ke světlu, osvětlovací model se nemusí počítat. V opačném případě, kdy paprsek úspěšně doletěl až ke světlu se Phongův model počítá.



Obrázek 4.5: Výsledný render cény s počítanými stíny.

Osvětlení

Samotné osvětlení světly je počítáno podle OpenGL 2.1 specifikace 2.3.1. Je tedy spočítán útlum světla podle vzdálenosti světla od místa dopadu paprsku a k němu je přidán i vliv kuželového světla.

Dále je podle rovnic počítána ambientní, difuzní a spekulární složka každého světla, ke kterým se přinásobuje jejich útlum. Tyto složky se kumulují a na konci smyčky se k výsledné sumě ještě přičte globální ambientní složka a emisní složka materiálu.

Jelikož OpenSceneGraph počítá i s intenzitou světla, mohl by nám rozsah složky barvy světla přetéct přes hodnotu 1 (stačí když bude intenzita světla > 1). Z tohoto důvodu na konci osvětlovacího modelu musíme dostat hodnoty do rozmezí $0 - 1$. Pro dosažení patřičného výsledku stačí zavolat metodu `normalize()` pro objekt `modColor`, který je typu `osg::Vec4`. `modColor` je pak použit pro pozdější modulaci barev materiálu a textury.

4.3.3 Odrazy a lomy

Jak je uvedeno v algoritmu 1, nejprve se spočítá barva zasaženého bodu a poté se zjišťuje, jestli je paprsek objektem odrážen nebo lomen.

Základní podmínkou je tedy zjistit, jestli zasažený objekt je či není zrcadlo. Jelikož má paprsek – objekt `intersectionHit`, k dispozici ukazatel na svého rodiče typu `render`, má přístup i k vektoru zrcadel ve scéně. Pro porovnání jestli paprskem zasažený objekt je zrcadlo, musí být porovnány tedy všechny objekty z vektoru. Jak již bylo zmíněno několikrát, graf scény dovoluje dědit nastavení. Proto v případě neúspěšného nalezení zrcadla zasaženého objektu, musíme hledat i v jeho předchůdcích.

Pokud bude nalezeno zrcadlo, musí se spočítat vektor od pozorovatele a vektor odrazu. Vektor od pozorovatele `toEye` je spočítán jako: pozice kamery $-$ bod dopadu paprsku. Vektor odrazu je spočítán tak, jak je tomu u vektoru odrazu spekulární složky Phongova osvětlovacího modelu (kapitola 2.3.4), jen je namísto vektoru ke světlu použit vektor k pozorovateli.

Kosinus podle Lambertova pravidla získáme v knihovně OSG obyčejným vynásobením dvou stejně velkých vektorů. Ekvivalentem v OpenGL je funkce `dot(X,Y)`, která počítá skalární součin mezi vektory X a Y.

Po výpočtu odrazového vektoru se dá spočítat, kam poslat sekundární paprsek. Bod dopadu paprsku posuneme směrem odrazového vektoru až na okraj scény. Pro tento případ se tak načte kořenový uzel scény a zjistí se velikost průměru koule, která obaluje scénu. Metodou `osg::Node::getBound()` se získá koule typu `osg::BoundingSphere`, která spustí metodu `radius`. Z takto získaného poloměru se vypočítá průměr scény a tak je nalezen maximální možný posun konečného bodu sekundárního paprku.

Paprsek je tedy vyslán z bodu dopadu primárního paprsku k bodu na hranici scény ve směru odrazu. Pokud paprsek narazí, jeho průsečík se použije pro vytvoření nové instance `intersectionHit`, která rekurzivně zavolá metodu `getColor(depth)`, ale se zmenšenou hloubkou rekurze `depth`.

V případě lomů by byl postup téměř totožný, jenže k vzorci pro výpočet odrazového vektoru by se musel spočítat podle Snellova zákona i vektor lomu. Protože však žádný z dostupných testovaných modelů nepodporuje nastavení hodnot indexů lomů nejsou v této aplikaci implementovány.

Obrázcích 4.4 a 4.5 jsou zobrazeny scény se zapnutými zrcadly.

4.4 Vyhodnocení

Jak bylo předpokládáno, metoda sledování paprsků je pomalá. V přehledné tabulce jsou zhodnoceny výsledky pro model `Mirror light intensity`, který obsahuje jen jedno zrcadlo a čtyři světla. Ve scéně je nutno vytvořit mipmapy u třech textur. Dvě z nich s rozlišením 256x256 pixelů, třetí 512x512.

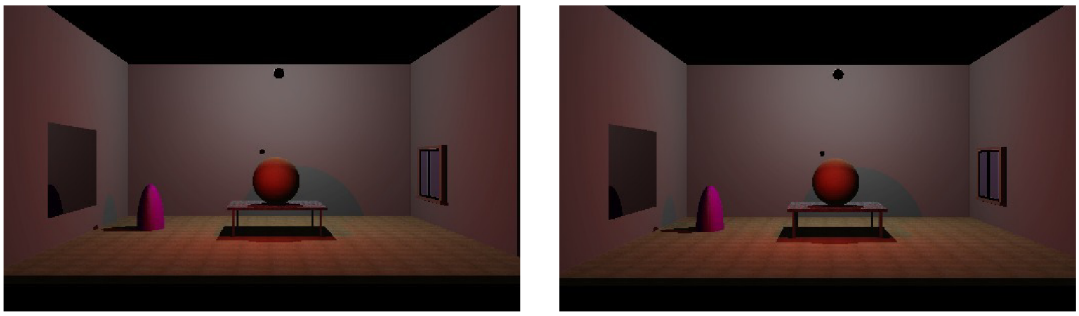
Porovnávány jsou časy pro obrázek stejného rozlišení 510x302 pixelů pouze s rozdílem použití kD stromů. Sloupec `Vektory` udává čas potřebný pro vyhledání a uložení všech světel a zrcadel, ale také pro tvorbu mipmap – tento čas by měl být pro obě verze přibližně stejný.

Ve sloupci `Inicializace` je uveden čas potřebný pro vytvoření výsledného okna a načtení obrázku z kamery, je tedy úměrný velikosti rozlišení. Ve sloupci `Render` je uveden čas samotného renderování primárních paprsků. V posledním sloupci je celkový čas, zaznamenaný po uložení výsledného obrázku.

Verze	**Vektory** [s]	**Inicializace** [s]	**Render** [s]	**Celkem** [s]
s kD stromy	0,31	0,16	353,23	353,7
bez kD stromů	0,29	0,18	404,78	405,25
s kD (1240x754)	0,34	0,8	25383,56	25384,7

Tabulka 4.1: Porovnání raytracingu s/bez použití kD stromů.

S ohledem na použití kD stromů se nám renderování zrychluje, jak je vidět v tabulce 4.4. Ovšem rychlost aplikace je i tak dosti pomalá. Pokud bychom testovali větší rozlišení dostaly bychom řádově jiná čísla (viz řádek „s kD (1240x754)“), kdy render trval 7 hodin. Důvodem je, že render s větším rozlišením musel vyslat 6x více primárních paprků. Trvání renderování pro verzi bez kD stromu pro toto větší rozlišení trvalo přibližně ještě 2,5-krát tak dlouho.



Obrázek 4.6: Výsledné obrázky testovaných verzí aplikace – vlevo obr. využívající kD stromů, vpravo obrázek bez použití kD stromů.

S ohledem na tuto tabulku můžeme říct, že rychlost aplikace úměrně klesá s počtem paprsků. Rychlost pak ovlivňuje nejen rozlišení, ale i počet světel ve scéně a velká plocha zrcadel spolu s vysokou rekurzivní hloubkou.

Vizuální výsledky

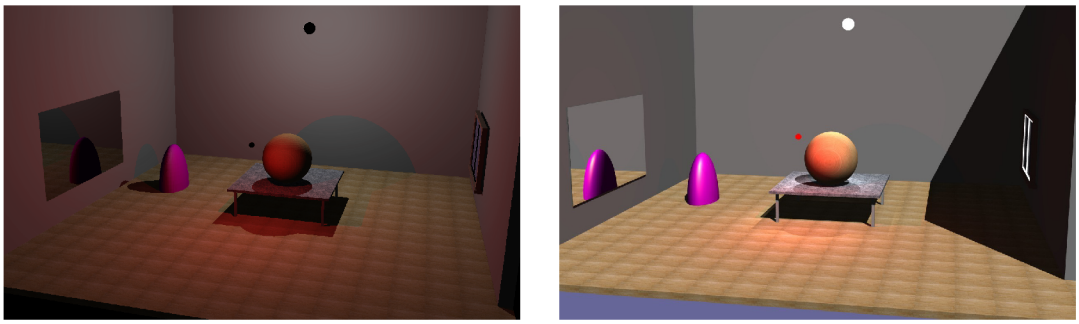
Jak můžeme vidět na obrázcích 4.7 nebo 4.6, výsledný render není úplně vyhovující, ikdyž jsou stíny na místech, kde být mají a osvětlovací model je počítán správně. Konstantní stínování, způsobené neinterpolováním normály knihovnou OpenSceneGraph, sráží výsledný dojem. Výsledky také ukazují, že kuželová světla nevytvářejí očekávaný výsledek, jaký by měla. Naopak zrcadlící se materiál je vidět velmi zřetelně i se správným zobrazením stínů.

Bez možnosti použít mipmapy ovšem aplikace ztrácí jednu z vlastností, kterou by měla mít. Ikdyž jí však nemá v aplikaci přímo, jsou mipmapy vytvářeny a lze je snadno použít. V metodě `getMipMapTexel(textureImage,texCoord)` je naznačen výpočet barvy texelu mipmapy dané určitou úrovní. Výpočet úrovně však implementován nebyl.

Porovnání s Lexolights

Lexolights (viz [11]) je 3D vizualizační software pro CAD systémy tam, kde je vyžadována vysoká věrnost vizualizace k realitě. Výsledek je dosahován Lexolights shaderem a stínovacím enginem, který vyhodnocuje osvětlení modelu přes jednotlivé pixely. Aplikace Lexolights také využívá knihovny OpenSceneGraph a Qt.

Na obrázcích 4.7 lze vidět výsledky dvou aplikací. První obrázek byl vyrenderován aplikací vytvořenou v této práci, a přestože aplikace renderovala obrázek s menším rozlišením, čas renderování (přibližně 7 hodin) je hodně za očekáváním. Druhý obrázek byl aplikací Lexolights vyrenderován zhruba za 5 minut. Obrázky nemají totožný model co se týče pozic a intenzit světel.



Obrázek 4.7: Výsledné porovnání vytvořené aplikace (vlevo) a programu Lexolights.

Propastný rozdíl byl nejspíše způsoben tím, že testovací platformou aplikace Lexolights byl nativní systém WindowsXP. Naopak vytvořená aplikace byla spuštěna na virtuálním stroji pod platformou Linux, která měla k dispozici znatelně menší zdroje.

Jeden z dalších možných rozdílů je aplikací Lexolights využívaný POV-Ray[1] a dlouhodobý vývoj této aplikace.

[1] „The Persistence of Vision Raytracer“ je velice kvalitní, svobodný software na vytváření třídimenzionální grafiky. K dispozici jsou verze jak pro Windows, Mac OS/Mac OS X a Linux.

Kapitola 5

Závěr

V bakalářské práci bylo mým cílem nastudovat knihovnu OpenSceneGraph a algoritmy používané metodami sledování paprsků. Za pomocí těchto znalostí jsem následně měl implementovat aplikaci umožňující zobrazovat virtuální scény jak akcelerovanou, tak raytracingovou metodou.

Nastudované způsoby jak zobrazovat, ať už akcelerovaně či raytracingově, jsou popsány v kapitole 2. Nástrojům, které jsem použil pro implementaci výsledné aplikace, je věnována kapitola 3, jedná se zejména o framework Qt pro tvorbu uživatelského rozhraní a již zmíněnou knihovnu OSG.

K implementaci, popsané v kapitole 4, jsem využil existujících tříd knihovny OpenSceneGraph tak, aby byla výsledná aplikace jednoduše rozšiřitelná a nemuselo se při dalším vývoji zasahovat do funkčních částí. Zdrojové kódy aplikace a všech pro ni vytvořených tříd jsou k dispozici na přiloženém DVD, spolu se zdrojovými soubory knihovny OpenSceneGraph.

Vytvořená aplikace disponuje jednoduchým uživatelským rozhraním, které dovoluje uživateli zadat model a zobrazit jej za pomoci OpenGL. Aplikace uživateli také umožňuje spustit renderování za pomoci metody raytracingu. Jak je ale z výsledků v kapitole 4.4 patrné, dlouhodobě vyvíjené aplikaci Lexolights nemůže rychlostně konkurovat.

Dalším vhodným rozšířením této práce by bylo zprovoznění vícevláknového renderování, doplnění výpočtu úrovně mipmapy či zlepšení optimalizace. Pro urychlení výpočtu by pak bylo výhodné použít jednu z pokročilejších metod sledování paprsku.

Literatura

[1] Appel, A.: Some techniques for shading machine renderings of solids. *Spring Joint Computer Conference*, 1968: s. 37–45.

[2] FORSYTH, D. A.; PONCE, J.: *Computer Vision: A modern approach*. New Jersey, USA: Pearson Education, Inc., první vydání, 2003, ISBN 0-13-191193-7.

[3] The Khronos Group: Adopter Companies: OpenGL ES. [online], 2013, [vid. 6.4.2013]. URL http://www.khronos.org/conformance/adopters/conformant-companies/#opengles

[4] The Khronos Group: OpenGL Overview. [online], 2013, [vid. 20.11.2012]. URL http://www.opengl.org/about

[5] Jensen, H. W.; Christensen, N. J.: Photon Maps in Bidirectional Monte Carlo Ray Tracing of Complex Objects. *Computers & Graphics*, ročník 19, č. 2, 1995: s. 215–224.

[6] Kajiya, J. T.: The Rendering Equation. *SIGGRAPH*, ročník 20, č. 4, 1986: s. 143–150.

[7] Lafortune, E. P.; Willems, Y. D.: Bi-Directional Path Tracing. *Proceedings of Computer graphics*, 1993: s. 145–153.

[8] NVIDIA Corporation: OptiX: Interactive ray tracing on NVIDIA Quadro professional graphics solutions. [online], 2013, [vid. 12.4.2013]. URL http://www.nvidia.com/object/optix.html

[9] openscenegraph: OpenSceneGraph. [online], 2012, [vid. 2.10.2012]. URL http://openscenegraph.org/

[10] openscenegraph: Knowledge Base: What is a Scene Graph? [online], 2013, [vid. 15.4.2013]. URL http://www.openscenegraph.com/index.php/documentation/knowledge-base/36-what-is-a-scene-graph

[11] Pečiva, J.; Starka, T.: Lexolights. [online], 2013, [vid.3.5.2013]. URL http://www.fit.vutbr.cz/research/prod/index.php.cs?id=276&notitle=1

[12] Phong, B. T.: Illumination for computer generated pictures. *Communications of the ACM*, ročník 18, č. 6, 1975: s. 311–317.

[13] Digia plc: Qt Project. [online], 2013, [vid. 14.4.2013]. URL http://qt-project.org/

[14] Segal, M.; Akeley, K.: *The OpenGL Graphics System:A Specification (Version 2.1)*. Silicon Graphics, Inc., 2006.

[15] Shirman, L.; Kamen, Y.: A new look at mipmap level estimation techniques. *Computers & Graphics*, ročník 2, č. 23, 1999: s. 223–231.

[16] Veach, E.; Guibas, L. J.: Metropolis Light Transport. *SIGGRAPH*, 1997: s. 65–76.

[17] Wang, R.; Qian, X.: *OpenSceneGraph 3.0 : Beginner's guide*. Birmingham, UK: Packt Publishing Ltd., první vydání, 2010, ISBN 978-1-849512-82-4.

[18] Whitted, T.: An Improved Illumination Model for Shaded Display. *Communications of the ACM*, ročník 23, č. 6, 1980: s. 343–349.

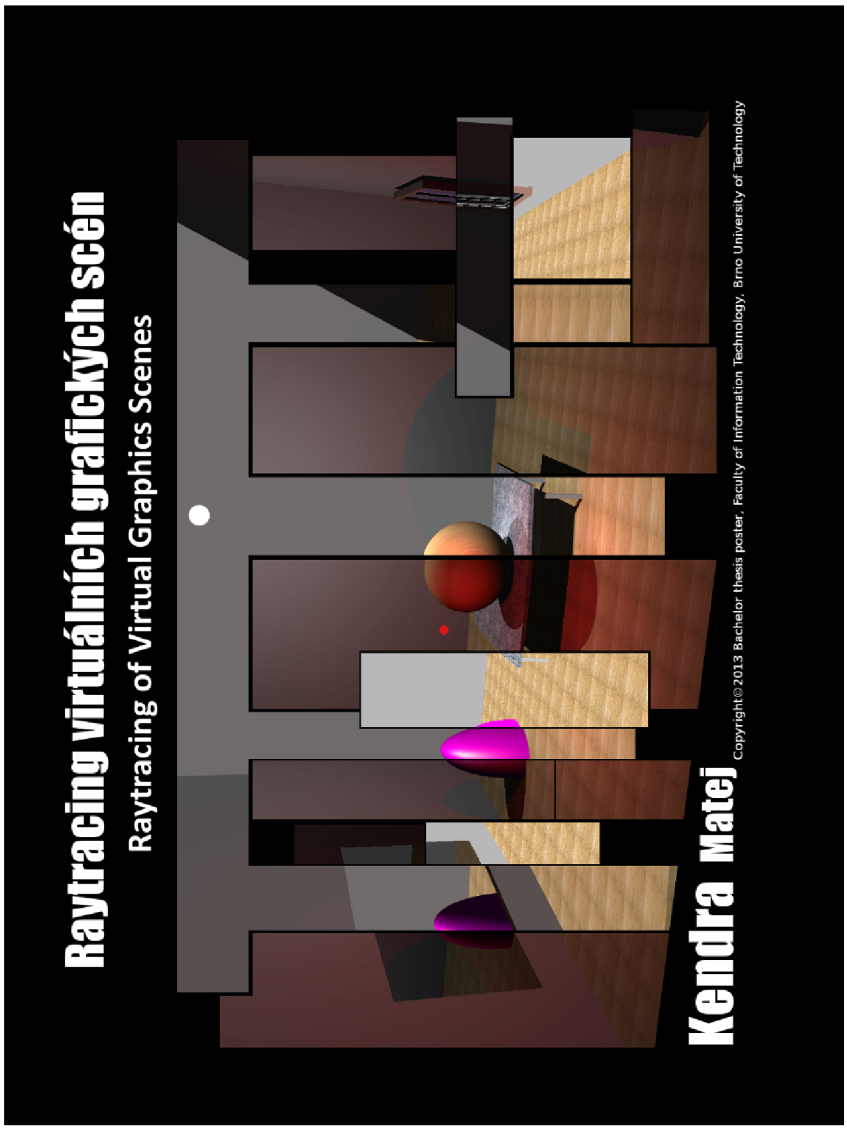
[19] Wikimedia Foundation, Inc.: Microsoft DirectX. [online], 2013, [rev. 15.4.2013][vid. 13.4.2013].
URL http://en.wikipedia.org/wiki/DirectX

[20] Wikimedia Foundation, Inc.: Refractive index [Index lomu]. [online], 2013, [rev. 5.4.2013][vid. 3.1.2013].
URL http://en.wikipedia.org/wiki/Refractive_index

[21] ŽÁŇA, J.; BENEŠ, B.; SOCHOR, J.; aj.: *Moderní počítačová grafika*. Brno, Czech republic: Computer Press, druhé vydání, 2004, ISBN 80-251-0454-0.

Příloha A

Plakát



Obrázek A.1: Plakát k této bakalářské práci.

Příloha B

Obsah DVD

Na DVD lze nalézt uvedené adresáře/soubory:

- **OSG** – obsahuje zdrojové soubory knihovny OpenSceneGraph verze 3.1.0
- **Coin3D** – adresář se soubory knihovny `Coin3D`
- **src** – adresář se zdrojovými soubory aplikace
- **Raytracer** – adresář s výslednou aplikací, spolu s modely
- **Obrázky** – adresář se všemi použitými a vyrenderovanými obrázky
- **tex** – adresář se zdrojovými soubory LaTeXu k vytvoření tohoto textu
- soubor **README.txt**
- plakát – soubor **Plakat-xkendr00.bmp**