UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Zobrazování 3D scény metodou raytracingu

Pavel Lokvenc

Bakalářská práce
2014

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2013/2014**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Pavel Lokvenc**
Osobní číslo:	**I11526**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Zobrazování 3D scény metodou raytracingu**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Cílem práce je nastudování metody raytracingu a její vlastní implementace.
V teoretické části BP bude detailně popsán princip raytracingu, použité objekty pro tvorbu scény a jejich vlastnosti a chování (světla, základní tvary, barvy, materiály), mapování textur a načtení modelu. Teoretická část se bude dále zabývat možnostmi optimalizace výpočtu.
V praktické části BP bude implementován jak samotný algoritmus výpočtu metody sledování paprsku, tak i testovací modulární raytracingový systém, sloužící k vizualizaci načteného modelu. Implementace bude provedena pomocí jazyka C++. V rámci praktické části bude provedeno srovnání výkonnostních a kvalitativních parametrů vlastního řešení s existujícími nástroji.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

Seznam odborné literatury:

PHARR, Matt, HUMPHREYS, Greg. Physically based rendering: from theory to implementation. 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2010. 1167 p. ISBN 01-237-5079-2.
SUFFERN, Kevin. Ray tracing from the ground up. Wellesley, Mass.: A K Peters, 2007. 762 p. ISBN 15-688-1272-8
SHIRLEY, Petr, MORLEY, R., Keith. Realistic ray tracing. 2nd ed. Wellesley: Ak Peters, 2009. ISBN 978-156-8814-612.

Vedoucí bakalářské práce: **Ing. Petr Veselý**
Katedra softwarových technologií

Datum zadání bakalářské práce: **20. prosince 2013**
Termín odevzdání bakalářské práce: **9. května 2014**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 31. března 2014

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 30. 4. 2014 Pavel Lokvenc

Poděkování

Chtěl bych poděkovat vedoucímu bakalářské práce Ing. Petru Veselému za jeho ochotnou pomoc. Dále bych chtěl poděkovat především rodině za podporu.

Anotace

Práce pojednává o zobrazovací metodě raytracing, která vyhází z geometrické optiky a využívá se pro vykreslování trojrozměrných scén. Práce obsahuje teoretický popis řešených problémů. V praktické části je provedena implementace vlastního vykreslovacího systému.

Klíčová slova

3D, počítačová grafika, sledování paprsku, vykreslování

Title

Rendering 3D scene with method of raytracing.

Annotation

Main goal of bachelor thesis is about rendering technique called ray tracing. Ray tracing is based on geometric optics and it is used for rendering three dimensional scenes. Thesis contains theoretic background of solved problems. In practice part is implemented ray tracing system.

Keywords

3D, computer graphics, ray tracing, rendering

Obsah

Seznam zkratek

1D	one dimenstional, jednorozměrný
3D	three dimensional, trojrozměrný
2D	two dimensional, dvojrozměrný
BMP	Windows bitmap
BRDF	bidirectional reflection distribution function
BSDF	bidirectional surface distribution function, sada BRDF
BSP	binary space partitioning
BVH	bounding volume hierarchy
CSG	constructive solid geometry
GUI	graphical user interface
JPEG	Joint Photographic Experts Group
NURBS	non-uniform racional bezier surface
OBJ	Wavefront object file
OpenGL	Open graphics library
OpenMP	Open Multi-Processing
PNG	portable network graphics
SIMD	single instruction, multiple data
XML	eXtensible markup language

Seznam obrázků

Seznam tabulek

Úvod

3D grafika nás především v posledních letech obklopuje na každém kroku. Najdeme ji použitou všude, od reklamních letáků přes filmové efekty a počítačové hry až po architekturu a medicínu.

Rendering je jedna z možností jak vytvořit výstup z 3D scén, v počítači reprezentovaných datovými strukturami modelů a dalších objektů. Při renderingu dochází k projekci 3D scény do 2D obrázku. Tento proces je možné si představit jako pořizování fotografií v reálném světě.

K řešení tohoto problému lze přistoupit pomocí metody projekce, jako knihovna OpenGL. Tyto metody jsou vhodné pro zobrazování v reálném čase. Další možností je využití metod stavících na fyzice a především potom na geometrické optice jako právě popisovaný raytracing a jemu podobné metody. Tyto metody se vyznačují vysokou kvalitou, jsou však náročné na výpočet.

Tato bakalářská práce se věnuje právě problematice vykreslování 3D scény pomocí metody raytracingu. Na začátku práce je stručné seznámení s jednotlivými vykreslovacími metodami a také některé základní pojmy z matematiky.

V práci je detailně popsána metoda raytracingu, včetně detailů jako je tvorba odražených a lomených paprsků a výpočet stínů. Také jsou zde popsány algoritmy pro výpočet průsečíků mezi paprskem a tělesem.

Dále je v práci řešeno odstranění aliasingu na výsledném obrázku, který vzniká tím, že se používá jeden paprsek pro celou oblast pixelu. Různé části této oblasti však mohou mít barvu různou a vzniká tak aliasing.

Práce také popisuje, jak vytvořit různé typy kamer, kterými lze na scénu pohlížet. Je zde řešen problém rovnoběžné a perspektivní kamery.

V práci je popsáno několik typů světel, která osvětlují scénu a vrhají v ní stíny.

Také je popsána tvorba a chování materiálů. Materiály se definují pomocí BRDF. To jsou funkce, které určují pravděpodobnost, že se světlo odrazí daným směrem. Pro definování komplexních materiálů je použito několika různých BRDF.

Poslední kapitola teoretické části je věnována optimalizaci výpočtu tak, aby bylo možné vykreslovat složitější modely a scény v přijatelnějším čase.

V rámci bakalářské práce byla také vytvořena implementace renderovacího systému, který využívá právě metody raytracingu. K implementaci bylo využito jazyka C++.

1 Přehled metod řešení zobrazovací rovnice

V této části práce bude zběžně představeno několik zásadních metod pro řešení zobrazovací rovnice. Zobrazovací rovnice pomocí integrace popisuje chování světla ve scéně a byla představena v roce 1986 Jamesem Kajiyou [1]:

$$L_{\mathrm{o}}(\boldsymbol{x}, \omega_{\mathrm{o}}, \lambda, t) = L_e(\boldsymbol{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\boldsymbol{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\boldsymbol{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \boldsymbol{n}) \mathrm{d}\omega_{\mathrm{i}} \qquad (1.1)$$

kde: λ – je konkrétní vlnová délka světla (každá vlnová délka se může jinak odrážet),
t – je čas,
$\boldsymbol{x}$ – je místo ve scéně,
ω_{o} – je směr odchozího světla,
ω_{i} – je obrácený směr příchozího světla,
$L_{\mathrm{o}}(\boldsymbol{x}, \omega_{\mathrm{o}}, \lambda, t)$ – je všechna odchozí záře ve směru ω_o, o vlnové délce λ, v t a v **x**,
$L_e(\boldsymbol{x}, \omega_{\mathrm{o}}, \lambda, t)$ – emitovaná záře,
Ω – je jednotková polokoule obsahující všechny hodnoty ω_{o},
$\int_{\Omega} \ldots \mathrm{d}\omega_i$ – integrál přes jednotkovou polokouli Ω,
$f_{\mathrm{r}}(\boldsymbol{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t)$ – BRDF, pravděpodobnost, že se světlo odrazí daným směrem,
$L_{\mathrm{i}}(\boldsymbol{x}, \omega_{\mathrm{i}}, \lambda, t)$ – světlo přicházející ze směru ω_{i},
$(\omega_{\mathrm{i}} \cdot \boldsymbol{n})$ – faktor zeslabení světla v závislosti na úhlu mezi vektorem ω_i a normálou.

Vzhledem k tomu, že rovnice obsahuje integrál přes polokouli, což je nekonečně velká množina přípustných směrů, tak není možné ji tedy přesně vyřešit. Vždy je možné se k výsledku pouze přiblížit. Následující metody se tedy pokoušejí s větším či menším úspěchem přiblížit k řešení této rovnice.

1.1 Ray tracing

Metoda sledování paprsků byla představena T. Whittedem. Vychází z geometrické optiky a umožňuje vykreslit libovolný objekt, pro který je možné vypočítat průsečík s paprskem.

Jako výhody této metody můžeme považovat to, že vychází z fyzikálních zákonů, její poměrně snadnou implementaci a možnost poměrně snadné paralelizace.

Naopak nevýhodami je značná výpočetní náročnost, nemožnost zobrazovat měkké stíny a rozostřené odrazy, rozklad světla pod povrchem materiálu, aliasing výsledného obrázku. Je zřejmé, že se jedná o hrubou aproximaci zobrazovací rovnice, například vůbec se nepočítá s difúzními odrazy světla, lomem světla pod povrchem objektu a dalšími.

Tato metoda je podrobně popsána v této bakalářské práci.

1.2 Distributed raytracing

Tuto metodu představil v roce 1984 na konferenci SIGGRAPH Rob Cook a vychází z metody klasického ray tracingu. Princip spočívá v tom, že k řešení integrálu zobrazovací rovnice je využito vzorkování jak v prostoru (antialiasing, měkké stíny, neostrý odraz a lom světla, hloubka ostrosti), tak v čase (rozmazání pohybem).

1.3 Radiozita

Radiozita je metoda, která slouží k výpočtu globálního osvětlení scény, k výpočtu využívá metodu konečných elementů. Plochy ve scéně jsou adaptivně rozděleny na menší. Každá ploška potom ovlivňuje své okolí podle toho, kolik energie přijme a kolik energie vyzáří. Více podrobností je možné nalézt v [2].

Zásadní nevýhodou tohoto algoritmu je, že dovede pracovat jen s polygonovou reprezentací těles a nedovede pracovat s texturami, průhlednými objekty a zrcadlovými odrazy, zpracovává pouze difúzní odrazy, nedovede si tedy poradit s kaustickými efekty.

K zobrazení výsledků radiozity je možné využít OpenGL, nebo ray tracing, čímž můžeme přidat zrdcadlové odrazy a další vlastnosti povrchů.

Radiozita je dnes již spíše historickou záležitostí, vzhledem k výše uvedeným záporům. Další nevýhodou je, že její implementace není oproti ostatním zde zmíněným metodám úplně triviální záležitostí.

1.4 Path tracing

Algoritmus publikoval v toce 1986 James Kajiya spolu se zobrazovací rovnicí. Věrně simuluje globální osvětlení v trojrozměrné scéně. Zobrazovací rovnici řeší pomocí Monte Carlo integrace. Pro každý pixel sleduje možné cesty paprsků světla, vycházející ze světelných zdrojů, které na něj po interakci se scénou dopadají.

Sledování cest fyzikálně přesně simuluje efekty globálního osvětlení, jako jsou lomy a odrazy světla, color bleeding, kaustika, měkké stíny, ale také hloubku ostrosti a rozmazání pohybem.

K dokonale vypadajícímu obrázku je však třeba provést velké množství simulací, jinak je obrázek zatížen viditelným šumem.

Tato metoda byla dále rozšířena metodami Bidirectional path tracing a Metropolis light transport. V současné době také existují implementace na grafické kartě, které podávají poměrně solidní výsledky v reálném čase.

Více informací o této metodě je možné nalézt v [3].

1.5 Photon mapping

Metodu publikoval Henrik Wann Jensen. Rendering a probíhá ve dvou částech. V první se předpočítá fotonová mapa a ve druhé části se obrázek vyrenderuje pomocí distribuovaného raytracingu, kde se k výsledné záři přičítá ještě příspěvek nejbližších fotonů.

Pomocí této metody lze počítat veškeré efekty zmiňované v části o path tracingu. Mapování fotonů je silné především při výpočtu kaustiky. Při nedostatečném počtu fotonů jsou výsledné obrázky flekaté, takže je nutné tuto metodu rozšířit o další části výpočtu.

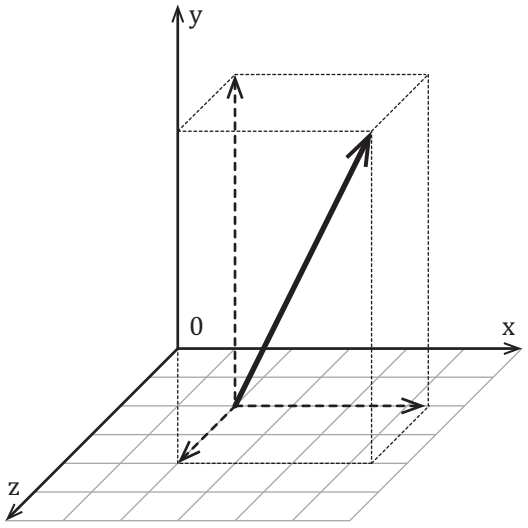
Více informací o této metodě je možné nalézt v [3].

2 Matematické pojmy

2.1 Souřadný systém

Stejně jako v jiných odvětvích počítačové grafiky i v případě sledování paprsku je třeba definovat souřadný systém. Samotné souřadnice však nestačí, je třeba je vztahovat k počátku a bázovým vektorům.

V obecném *n*-rozměrném prostoru určuje počátek bod $\boldsymbol{p_0}$ a *n* lineárně nezávislých vektorů vektorový prostor. V tomto prostoru může být jakýkoliv vektor $\boldsymbol{v}$ vyjádřen jako lineární kombinace těchto bázových vektorů.



Obrázek 1 – Pravotočivý ortonormální prostor

Ve 3D je vhodné používat ortonormální prostor s počátkem $\boldsymbol{p_0} = [0,0,0]$ a bázovými vektory $\boldsymbol{x} = (1,0,0)$, $\boldsymbol{y} = (0,1,0)$ a $\boldsymbol{z} = (0,0,1)$. Tomuto prostoru potom říkáme světové souřadnice (*world space*).

2.1.1 Levotočivý a pravotočivý systém

Pokud definujeme souřadný systém, máme možnost umístit třetí bázový vektor dvěma způsoby tak, že vektor směřuje buď do jednoho, nebo druhého poloprostoru rozděleného rovinou mezi vektory **x** a **y**. Pro rozpoznání lze použít pravidlo pravé ruky[1].

Ve většině dnešních programů se využívá pro *world space* pravotočivý souřadný systém. Rozdíly však bývají v tom, jaká osa se používá jako svislá. Některé programy značí svislou osu Z (3ds max, AutoCad, stavařské programy) a jiné jako Y (Maya, Softimage, Blender)[2]. Pro zobrazovací souřadný systém (např. výstup na monitoru) se naopak používá levotočivý

[1] http://cs.wikipedia.org/wiki/Pravidlo_prav%C3%A9_ruky
[2] http://www.autodesk.com/, http://www.blender.org/

souřadnicový systém, kde poté třetí bázový vektor je komý na zobrazovací plochu a míří směrem od pozorovatele (do monitoru).

2.2 Bod, vektor, normála

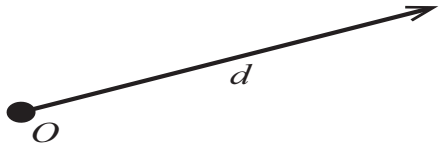
Všechny tyto tři objekty jsou reprezentovány v prostoru pomocí souřadnic x, y, z a pomocí operací, které lze nad nimi provádět. Při implementaci je možné je považovat za stejný typ objektu, nebo také na ně pohlížet jako na tři různé typy a nadefinovat jim operace odpovídající konvencím známým z analytické geometrie a lineární algebry (například $bod - bod = vektor$, $bod + vektor = bod$).

2.3 Paprsek

Paprsek si lze představit jako polopřímku v prostoru. Tu lze definovat pomocí počátku a směrového vektoru. Díky tomu je možné výhodně počítat průsečíky s tělesy ve scéně. Bod na polopřímce lze potom vyjádřit pomocí vektorové rovnice:

$$\boldsymbol{P} = \boldsymbol{O} + \boldsymbol{d}t \tag{2.1}$$

kde $\boldsymbol{O}$ je počátek naší polopřímky, $\boldsymbol{d}$ je směrový vektor a t je parametr v intervalu $\langle 0; \infty)$. Při výpočtu průsečíku s objektem je potom hledán právě parametr t.



Obrázek 2 – Polopřímka

2.4 Interpolace

Díky interpolaci je možné odhadnout hodnotu funkce, je-li její hodnota známá pouze v některých bodech. Vypočtená hodnota je však pouze přibližná, velikost chyby záleží především na tom, jak vhodně je zvolená interpolační funkce. Je nutné, aby interpolační křivka procházela zvolenými body.

Pro interpolaci mezi dvěma známými funkčními hodnotami se zpravidla používá lineární interpolace (proložení přímkou). Hodnotu $f(x)$ v konkrétním bodě x, lze nalézt pomocí vzorce:

$$f(x) = f(x_0) + \frac{f(x_0) - f(x_1)}{x_0 - x_1}(x - x_0) \tag{2.2}$$

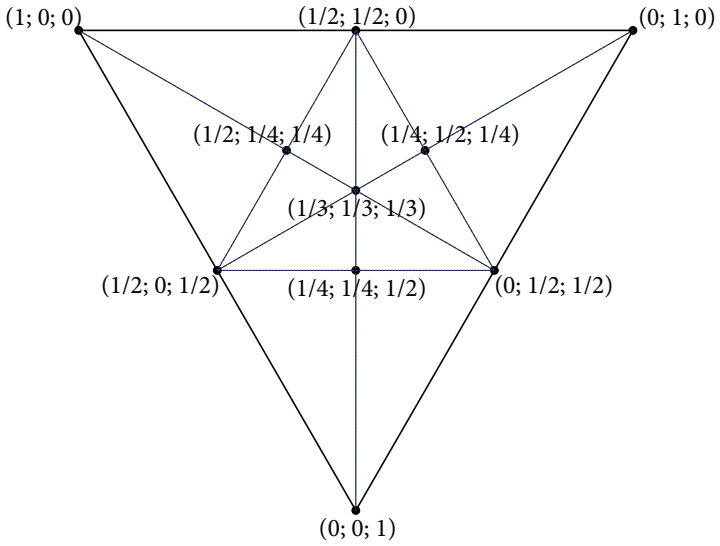
kde x_0 resp. x_1 jsou hraniční body, kde jsou známé funkční hodnoty, a kde platí, že $x_0 < x < x_1$.

Při vyšším počtu interpolovaných bodů se používají polynomy vyšších stupňů nebo metody interpolace jako např. Lagrangeova [2] nebo Newtonova interpolace [3].

V počítačové grafice je interpolace velice často využívána. Jako typický příklad lze uvést například dopočítávání barevné informace při transformaci rastrových obrázků, kde může být použita bilineární interpolace [4], což je aplikace principu lineární interpolace ve dvou směrech dle osy x a y.

2.5 Barycentrické souřadnice

Barycentrické souřadnice [5] jsou souřadným systémem, kterým lze určit polohu bodu uvnitř libovolného simplexu [6] (úsečka, trojúhelník, čtyřstěn). Pro potřeby této bakalářské práce je nejdůležitějším objektem trojúhelník.



Obrázek 3 – Barycentrické souřadnice v trojúhelníku

Pro bod nacházející uvnitř trojúhelníku, musí platit, že žádná ze souřadnic není menší než 0 a zároveň součet všech souřadnic je menší nebo roven 1.

Pokud jsou známé polohy bodů ve světových souřadnicích je možné hledaný bod dopočítat pomocí barycentrické interpolace:

$$P = aP_0 + bP_1 + cP_2 \tag{2.3}$$

kde a, b, c jsou jednotlivé složky zadané barycentrické souřadnice a P_0, P_1, P_2 jsou k nim příslušné vrcholy ve světových souřadnicích. Toto lze využít například pro dopočítávání mapovacích souřadnic a normál při výpočtu průsečíku s trojúhelníkovým modelem viz kapitola 6.2. Dalším příkladem může být dělení trojúhelníků pomocí metody konečných prvků při použití radiozity viz kapitola 1.3.

3 Ray tracing

V roce 1968 publikoval Arthur Appel metodu zvanou ray casting. Ta pracovala pouze s primárními paprsky. Pomocí této metody tedy nebylo možné zobrazovat stíny, odrazy a lomy světla. Oproti stávajícím metodám však dokázala velice jednoduše zobrazovat zakřivené plochy (implicitní plochy, NURBS plochy aj.) bez nutnosti jejich triangulace, CSG objekty. Pro možnost zobrazení dané plochy stačí, aby bylo možné vypočítat průsečík dané plochy a paprsku.

V roce 1979 rozšířil tuto metodu Turner Whitted, a dal tak vzniknout ray tracingu. Ten již dokázal zobrazovat stíny, a lom a odraz světla.

Jde o globální renderovací metodu, která vychází z geometrické optiky a snaží se simulovat chování světla v reálném světě. Základní verze algoritmu vypadá následovně:

```
Vytvoř objekty, kameru a světla

Pro každý pixel
  Vytvoř paprsek procházející ve směru průmětny středem pixelu

  Pro každý objekt scény
    Pokud paprsek protíná objekt
      Je vzdálenost průsečíku menší než nejblíže nalezený průsečík?
          Vypočítej a nastav nejbližší průsečík

  Pokud nebyl protnut žádný objekt
    Vyplň pixel barvou pozadí
  Jinak
    Vyšli stínovací paprsek pro každé světlo
    Pokud je objekt odrazivý vyšli odražený paprsek
    Pokud je objekt průhledný vyšli refrakční paprsek
    Vypočítej stínovací funkcí výslednou barvu B
    Barvou B vyplň pixel
```

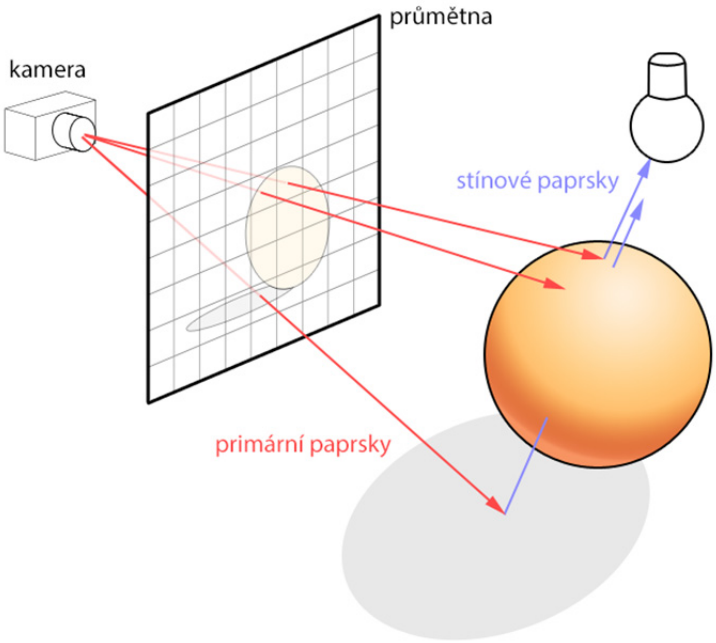
Jak je z tohoto zápisu vidět, při ray tracingu se pracuje s paprsky obráceně než v přírodě. V přírodě paprsky putují směrem od světelného zdroje a po odražení od objektů dopadnou na sítnici v našem oku (průmětnu). Tento přístup se ovšem vzhledem ke své náročnosti a nízké efektivitě nepoužívá. Paprsků, které dopadnou na průmětnu je totiž velice malé množství, ovšem výpočty by musely být provedeny pro všechny vytvořené paprsky. Proto se volí způsob, kdy paprsky vycházejí z oka a hledáme nejbližší průsečík.

3.1 Typy paprsků

V praxi se rozlišuje mezi třemi základními typy paprsků:

- **Primární paprsek (primary ray)** – tak jsou nazývány ty paprsky, které jsou vrhnuty z kamery do scény. Pro tyto paprsky je nutné kromě nalezení průsečíku také vypočítat další hodnoty jako konkrétní hodnotu průsečíku, normálu v daném vrcholu, souřadnice pro mapování textur aj.

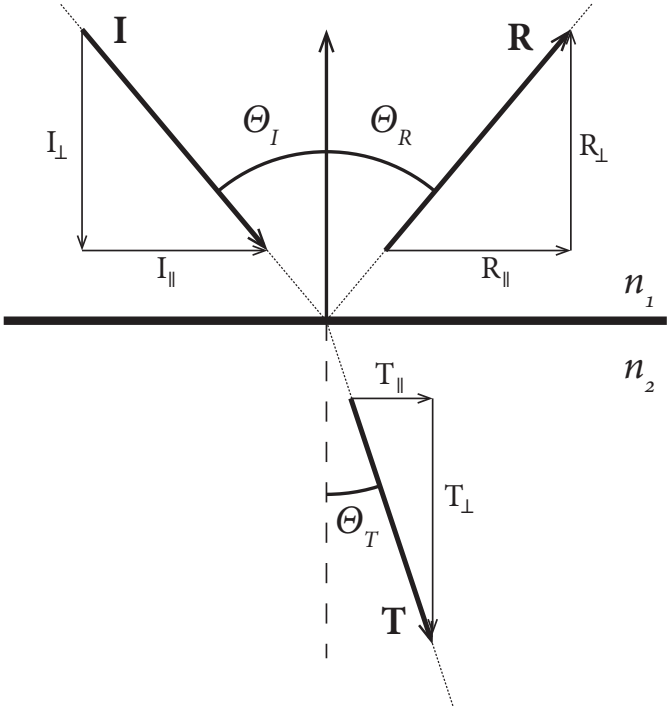
- **Sekundární paprsek (secondary ray)** – tak nazýváme paprsky, které vytvoříme po dopadu primárního nebo sekundárního paprsku na objekt. Simulujeme případ, kdy se předchozí paprsek odrazí zpět do prostoru scény, nebo případ, kdy pronikne do průhledného, či průsvitného tělesa. Počet sekundárních paprsků může být při větší hloubce rekurze mnohokrát vyšší než počet paprsků primárních. Je to dáno tím, že vznikají i při dopadu sekundárních paprsků na objekt ve scéně.
- **Stínový paprsek (shadow ray)** – paprsek, který je vytvořený z bodu, kam dopadl primární nebo sekundární paprsek, směrem ke všem zdrojům světla ve scéně. Jeho úkolem je zjistit, jestli v cestě ke světlu neleží objekt, který by mohl od daného světla vrhat do bodu stín. Těchto paprsků bývá ve scéně zpravidla mnohem více než sekundárních a primárních paprsků. Jejich výhodou však je, že stačí nalézt jakýkoliv průsečík (nemusí být nejbližší) a dále nemusíme provádět další výpočty, jako je výpočet normál, mapovacích souřadnic atd. Je tedy výhodné při implementaci na to myslet a vytvořit pro tento případ speciální metodu.



Obrázek 4 – Základní princip ray tracingu

3.2 Odraz a lom světla

Odraz a lom vznikají, když se paprsek světla dostane na rozhraní s prostředím, které odráží dopadené paprsky resp. paprsky procházejí skrz něj. V obou případech je třeba vytvořit sekundární paprsky, díky kterým je možné korektně vypočítat světelný příspěvek v daném bodě. Je třeba správně vypočítat směr sekundárního paprsku.



Obrázek 5 – Odraz a lom světla

Obrázek 5 znázorňuje situaci, ke které dochází na rozhraní dvou prostředí. Prostředí mají různé absolutní indexy lomu n_1 a n_2.

Směr příchozího paprsku je $\boldsymbol{I}$, směr odraženého paprsku je $\boldsymbol{R}$, a směr lomeného paprsku je $\boldsymbol{T}$. Dalším je pro výpočet k dispozici normála $\boldsymbol{N}$. Důležitou podmínkou je, že se jedná o jednotkové vektory, takže platí:

$$|\boldsymbol{I}| = |\boldsymbol{R}| = |\boldsymbol{T}| = |\boldsymbol{N}| = 1 \tag{3.1}$$

Jednotlivé operace s vektory budou nyní předvedeny na ukázkovém vektoru $\boldsymbol{v}$ a v další části práce se bude toto značení používat již na konkrétních vektorech.

Vektory směrů paprsků je možné rozložit na jejich složky $\boldsymbol{v}_\perp$, která je kolmá s rozhraním resp. s tečnou plochou v místě dopadu s rozhraním a $\boldsymbol{v}_\parallel$, která je vodorovná s rozhraním resp. s tečnou plochou v místě dopadu s rozhraním. Složku $\boldsymbol{v}_\perp$ je potom možné vypočítat jako:

$$\boldsymbol{v}_\perp = (\boldsymbol{v} \cdot \boldsymbol{n})\boldsymbol{n} \tag{3.2}$$

Složka $\mathbf{v}_\parallel$ je potom rozdílem mezi $\mathbf{v}$ a $\mathbf{v}_\perp$:

$$\boldsymbol{v}_\parallel = \boldsymbol{v} - \boldsymbol{v}_\perp \tag{3.3}$$

Nyní je možné spočítat goniometrické funkce pro úhly ve vzniklém pravoúhlém trojúhelníku jako:

$$\sin\Theta_v = \frac{|\boldsymbol{v}_\perp|}{|\boldsymbol{v}|} = |\boldsymbol{v}_\perp|$$
$$\cos\Theta_v = \frac{|\boldsymbol{v}_\parallel|}{|\boldsymbol{v}|} = |\boldsymbol{v}_\parallel| \tag{3.4}$$

3.2.1 Směr odraženého paprsku

Díky výše uvedeným rovnicím je možné již snadno vypočítat směr odraženého paprsku. Výpočet vychází ze zákona odrazu [1], který říká, že úhel dopadu je roven úhlu odrazu:

$$\Theta_I = \Theta_R \tag{3.5}$$

Pokud jsou stejné úhly, musí být stejné také hodnoty goniometrických funkcí:

$$|\boldsymbol{I}_\parallel| = \cos\Theta_I = \cos\Theta_R = |\boldsymbol{R}_\parallel|$$
$$|\boldsymbol{I}_\perp| = \sin\Theta_R = \sin\Theta_R = |\boldsymbol{R}_\perp| \tag{3.6}$$

Potom se složky vektoru odraženého paprsku $\boldsymbol{R}$ rovnají:

$$\begin{gathered}\boldsymbol{R}_\perp = -\boldsymbol{I}_\perp \\ \boldsymbol{R}_\parallel = \boldsymbol{I}_\parallel\end{gathered} \tag{3.7}$$

Směr odraženého paprsku $\boldsymbol{R}$ je potom roven:

$$\begin{gathered}\boldsymbol{R} = \boldsymbol{R}_\perp + \boldsymbol{R}_\parallel \\ \boldsymbol{R} = \boldsymbol{I}_\perp - \boldsymbol{I}_\parallel \\ \boldsymbol{R} = [\boldsymbol{I} - (\boldsymbol{I}\cdot\boldsymbol{N})\boldsymbol{N}] - (\boldsymbol{I}\cdot\boldsymbol{N})\boldsymbol{N} \\ \boldsymbol{R} = \boldsymbol{I} - 2(\boldsymbol{I}\cdot\boldsymbol{N})\boldsymbol{N}\end{gathered} \tag{3.8}$$

3.2.2 Směr lomeného paprsku

Při lomu paprsku se vychází ze Snellova zákona [1]:

$$\eta_1 \sin\Theta_I = \eta_2 \sin\Theta_T$$
$$\sin\Theta_T = \frac{\eta_1}{\eta_2}\sin\Theta_I \tag{3.9}$$

kde musí platit, že $\sin\Theta_I \leq \frac{\eta_2}{\eta_1}$. V případě porušení této podmínky by vyšlo, že $\sin\Theta_T > 1$ a takový výsledek nepatří do oboru hodnot funkce sinus. V tomto případě poté dochází k totálnímu odrazu světla.

Lomený paprsek se opět bude rovnat součtu svých složek $\boldsymbol{T} = \boldsymbol{T}_\parallel + \boldsymbol{T}_\perp$.

V rovnici Snellova zákona je možné nahradit sinus a je tak možné vypočítat složku lomeného paprsku $\boldsymbol{T}_\parallel$:

$$\left|\mathbf{T}_{\parallel}\right| = \frac{\eta_1}{\eta_2}\left|\mathbf{I}_{\parallel}\right|$$
$$\boldsymbol{T}_{\parallel} = \frac{\eta_1}{\eta_2}\boldsymbol{I}_{\parallel} = \frac{\eta_1}{\eta_2}(\mathbf{I} + \cos\Theta_I \cdot \boldsymbol{N}) \tag{3.10}$$

Druhou složku vektoru je možné vypočítat pomocí Pythagorovy věty:

$$\boldsymbol{T}_{\perp} = -\sqrt{1 - \left|\boldsymbol{T}_{\parallel}\right|^2}\boldsymbol{N} \tag{3.11}$$

Výsledný směr lomeného paprsku potom lze vypočítat jako:

$$\boldsymbol{T} = \boldsymbol{T}_{\parallel} + \boldsymbol{T}_{\perp} = \frac{\eta_1}{\eta_2}\boldsymbol{I} + \left(\frac{\eta_1}{\eta_2}\cos\Theta_I - \sqrt{1 - \sin^2\Theta_T}\right)\boldsymbol{N} \tag{3.12}$$

kde ze Snellova zákonu:

$$\sin^2\Theta_T = \left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2\Theta_I = \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - \cos^2\Theta_I) \tag{3.13}$$

Vzhledem k tomu, že v reálném světě světlo není přímka, ale vlnění, tak při lomu světla dochází také k jeho částečnému odrazu, díky Huygensovu principu [1]. Pokud je úhel mezi světly vyšší než kritický úhel, k lomu nedochází vůbec, všechno světlo je odraženo.

Ke zjištění, jaká část světla bude odražena a jaká část bude lomena, se využívá Fresnelových rovnic [1], ze kterých lze získat koeficienty transmise a reflexe. Platí zde, že:

$$t + r = 1 \tag{3.14}$$

Koeficient reflexe se dále rozkládá na dva koeficienty, podle polarizace světla:

$$r_s = \left[\frac{\eta_1\cos\Theta_I - \eta_2\cos\Theta_T}{\eta_1\cos\Theta_I + \eta_2\cos\Theta_T}\right]^2$$
$$r_p = \left[\frac{\eta_1\cos\Theta_T - \eta_2\cos\Theta_I}{\eta_1\cos\Theta_T + \eta_2\cos\Theta_I}\right]^2 \tag{3.15}$$

kde:

$$\cos\Theta_T = \sqrt{1 - \left(\frac{\eta_1}{\eta_2}\sin\Theta_T\right)^2} \tag{3.16}$$

Výsledné koeficienty potom vypočítáme jako:

$$r(\Theta) = \begin{cases} \frac{r_s + r_p}{2}, & pokud\ nejde\ o\ totální\ odraz \\ 1, & pokud\ se\ jedná\ o\ totální\ odraz \end{cases} \tag{3.17}$$

$$t(\Theta) = 1 - r(\Theta) \tag{3.18}$$

Vzhledem k tomu, že výpočet Fresnelových rovnic je výpočetně velice náročný, lze je nahradit vhodnou aproximací jako je například Schlickova aproximace [2], která se často využívá v grafických aplikacích:

$$\begin{aligned} r(\Theta) &= r_0 + (1 - r_0)(1 - \cos\Theta_I)^5 \\ r_0 &= \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2 \end{aligned} \tag{3.19}$$

Jako důsledek těchto jevů v přírodě může být například pohled na vodní hladinu, kde při změně pozorovacího úhlu buď je vidět skrz hladinu, anebo naopak vidíme odraz. Dalším příkladem může být také jev zvaný Snellovo okno[3], které je vidět při pohledu zpod vodní hladiny a nebo jako vinětace objektivu u fotografických přístrojů.

3.3 Omezení rekurze

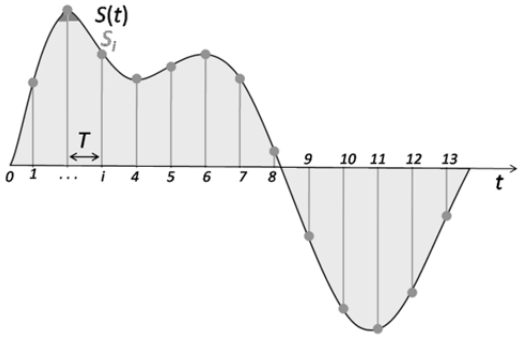
Vzhledem k tomu, že sekundární paprsky mohou vytvářet další sekundární paprsky, je třeba nějak omezit hloubku rekurze, jinak bude výpočet probíhat velice pomalu (a narůst kvality výstupu nebude znatelný), v extrémním případě může dojít k chybě přetečení zásobníku při rekurzivním volání funkce (*StackOverflow*).

První možností je zadáním pevné hloubky rekurze, která nesmí být překročena. Druhou možností je výpočet ukončit, pokud by poměr mezi posledními dvěma kroky rekurze byl menší než nějaká námi zadaná hodnota a určovat tak rozdíl adaptivně, dle významnosti příspěvku k barvě.

3.4 Další možnosti zlepšení kvality

Jedním z problémů zobrazování pomocí ray tracingu je vznik aliasu. To je způsobeno převodem spojité informace, v tomto případě scény, na informaci diskrétní, tedy pixel obrázku. Hodnota spojité funkce se vypočítává pouze v pravidelných intervalech a informace, kterou funkce přináší mezi těmito intervaly je zanedbána.

[3] http://en.wikipedia.org/wiki/Snell's_window



Obrázek 6 – Vzorkování funkce [9]

Alias výstupu lze odstranit zvýšením počtu vzorků na jeden pixel. Tato technika by šla přirovnat k aplikaci metody Monte-Carlo [8], která má rozličné využití od ekonomie, přes počítačovou simulaci až po výpočty v počítačové grafice. Jedná se o výpočetní metodu přibližnou, která ovšem při použití dostatečně velkého počtu vzorků podává celkem přesné výsledky.

Výpočet barevného příspěvku k pixelu je vypočítán podle vzorce:

$$\boldsymbol{C}_{\mathrm{rgb}} = \frac{\sum_0^n \boldsymbol{c}_{\mathrm{n}}}{n} \tag{3.20}$$

kde n je počet vzorků (vržených paprsků) a c_n je barva paprsku.

Na rozdíl od antialiasingu v projekčních metodách, nebo ve 2D, je antialiasing při ray tracingu velice časově a výpočetně náročný proces. Vykreslovací čas při zvyšování počtu paprsků stoupá přibližně lineárně, pokud připustíme, že čas výpočtu jednotlivých vzorků v jednom pixelu je pravděpodobně stejný. Antialiasing je také použit tam, kde ho není třeba, například na souvislých barevných plochách.

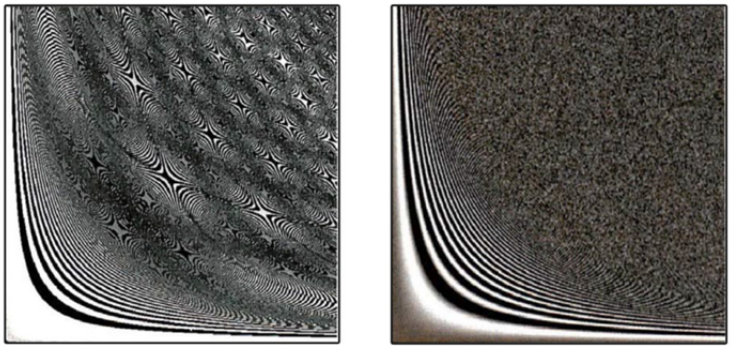
Při rozmísťování paprsků ve výsledném pixelu je třeba dobře zvážit strategii, kterou použijeme. Je možné definovat několik možných přístupů:

- Pravidelné rozmístění pixelů. Pixel je rozdělen na sub-pixely, skrz jejichž střed je vržen paprsek. Výhodou je pokrytí všech částí daného pixelu. Naopak nevýhodou je, že pokrytí pixelu je příliš rovnoměrné a ve výsledném obrazu tak mohou vznikat moiré patterns [9].
- Náhodné rozmístění pixelů. Paprsky jsou uvnitř pixelu rozmístěny pomocí pseudonáhodného generátoru čísel. Výhodou tohoto vzorkování je, že odstraňuje vzory, které vznikají při použití předchozí metody, a místo nich se v obrazu objevuje šum. Ten je lidským okem lépe přijímán. Nevýhodou této metody však je, že šum se objevuje i tam, kde by neměl. Při hranách objektu mohou také vznikat nepřesnosti a artefakty. Může docházet k nechtěnému shlukování vzorků.
- Roztřesené rozmístění pixelů. Pixel je rozdělen na sub-pixely, paprsky jsou však uvnitř těchto sub-pixelů rozmístěny náhodně. Jedná se o kombinaci dvou předchozích metod. Je zde uplatněná jak rovnoměrnost rozmístění pixelů, tak jistá náhoda.

Nemůže zde docházet ke shlukování vzorků v tak velké míře. Šum není tak výrazný po celém obrázku a moiré patterns tu také nejsou v tak velké míře.

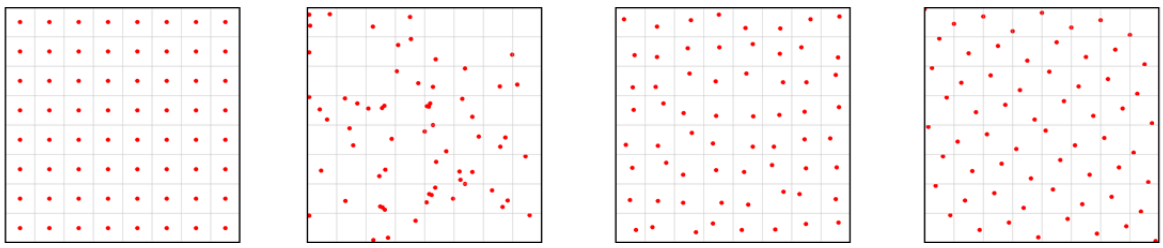
- Použití speciálních algoritmů, jako Hammersley points [4], které zajišťují rovnoměrné rozvržení vzorků v ploše. Tyto algoritmy mohou být také deterministické, což je vhodné pokud se jedná o animaci, protože poté nedochází ke vzniku šumu vzhledem k času.

Na obrázku 7 je ukázáno použití různých typů vzorkování. Vlevo je použito rovnoměrné vzorkování a jsou zde vidět moiré patterns. Vpravo je zase použito náhodné rozmístění vzorků a objevuje se zde místo moiré patterns šum.



Obrázek 7 – Ukázka artefaktů při použití různých metod vzorkování [3]

Vzorkování je jedním ze základních postupů při vylepšování výsledků vykreslování. Lze jej uplatnit při tvorbě pokročilých efektů, jako jsou rozmazané odrazy světla, rozmazaný lom světla, plošná světla, hloubka ostrosti kamery, nebo třeba rozmazání pohybem (vzorkování času). Implementaci vzorkování je tedy vhodné provést velice obecně. Jedním z postupů jak dosáhnout obecnosti je vzorkovat prostor jednotkového čtverce (interval $\langle 0; 1 \rangle$), resp. jednotkové koule pro vzorkování v prostorovém úhlu (odraz, lom). Získané souřadnice je potom možné transformovat do světových souřadnic pro další výpočty.



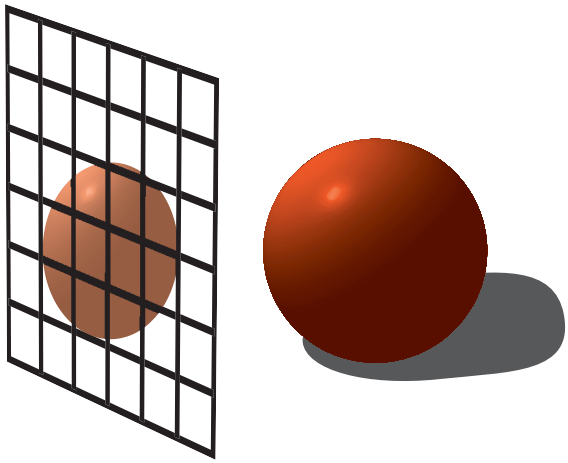
Obrázek 8 – Ukázky různých vzorkovacích algoritmů při 64 vzorcích

Na obrázku 8 je ukázáno různé rozmístění vzorků. Bráno zleva jde o rovnoměrné, náhodné, rozstřesené, Hammersley points. Obrázek je vytvořen jednoduchou aplikací napsanou v Javě. Aplikace je přiložena na CD.

4 Kamery a projekce

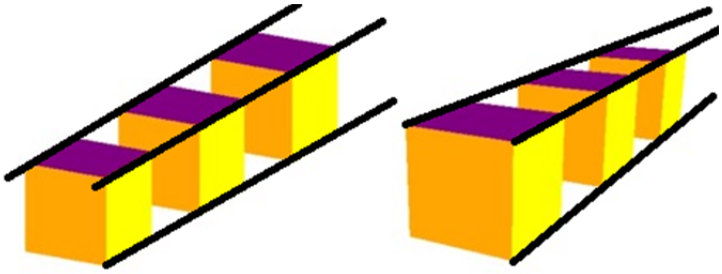
Pro zobrazení trojrozměrné scény na dvourozměrném plátně je třeba provést příslušnou transformaci. Při zobrazování pomocí projekčních metod jako je OpenGL nebo DirectX je zapotřebí scénu transformovat pomocí projekčních transformačních matic. Při použití metody sledování paprsku je však řešení snadnější. Paprsky, které vysíláme do scény, stačí jen transformovat tak, aby odpovídaly reálnému chování. Inspiraci lze najít v tom, jak funguje lidské oko, nebo fotografické přístroje a objektivy.

K záznamu barevných dat je vhodné vložit do scény pomocný objekt, kterým bude představovat film ve fotoaparátu. V zahraniční literatuře je možné tento objekt nalézt pod názvem *view plane*. Tato struktura také usnadní správnou transformaci primárních paprsků, které směřují ven z kamery.



Obrázek 9 – View plane umístěný ve scéně

Parametry obecného *view plane* jsou potom výška a šířka. Konkrétní implementace si jejich definici řeší po svém. Pro metodu sledování paprsku je vhodné si *view plane* rozdělit na pixely, kde výšku a šířku lze spočítat jako součin počtu pixelů a velikosti hrany pixelu ve scéně ve směru příslušné osy.



Obrázek 10 – Vlevo rovnoběžné a vpravo perspektivní promítání

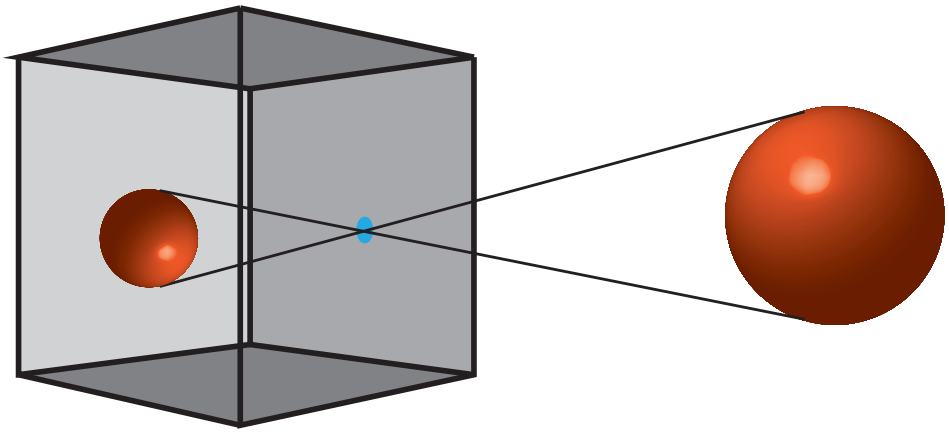
Existuje mnoho různých typů promítání, většina jich je používána ve stavařské a strojařské praxi. Pro potřeby této bakalářské práce jsou asi nejdůležitější rovnoběžné a perspektivní promítání viz obrázek 10. Vzhledem k tomu, že ray tracing je založen na geometrické opti-

ce, je možné modelovat také speciální typy projekce jako je rybí oko (*fisheye*) nebo panoramatická projekce.

4.1 Rovnoběžná kamera

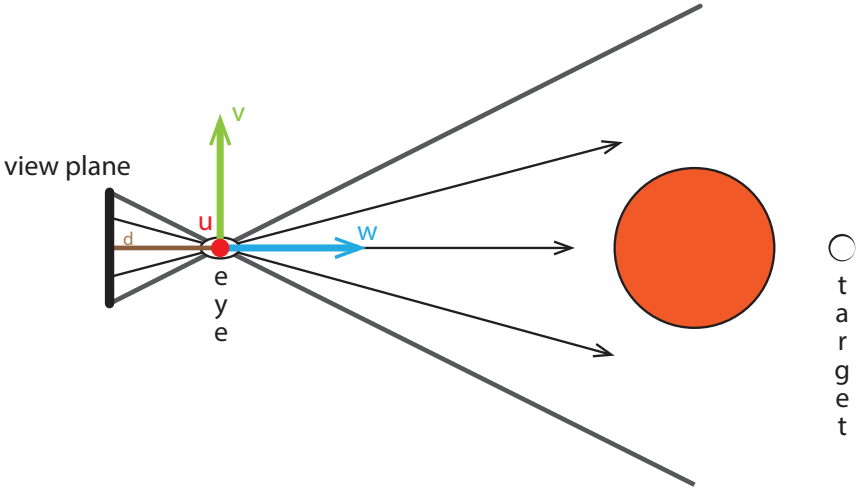
View plane se umístí do prostoru, a nasměruje. Potom jsou z jednotlivých pixelů vystřelovány paprsky, které jsou kolmé na *view plane*. Tato kamera je velice jednoduchá na výslednou implementaci.

4.2 Perspektivní kamera



Obrázek 11 – Dírková kamera

V případě perspektivní kamery lze vycházet z dírkové kamery[4]. Paprsky procházejí směrem od pozorovaného objektu skrz dírku v krabici a dopadají na film. Na filmu je potom objekt promítnut převrácený. Při vyvolávání je potom třeba film převrátit. Tento systém lze vylepšit soustavou čoček podobně jako moderní fotoaparáty. Takto také funguje lidské oko.



Obrázek 12 – Řešení perspektivní kamery

[4] Ruské fotoaparáty LOMO, funkční papírový model vycházel jako vystřihovánka v časopisu ABC

Perspektivní kameru definujeme pomocí *view plane*, bodů ***eye*** a ***target*** a pomocí vektorů *up*, který definuje natočení kamery kolem vlastní osy a skalární veličiny *d*, která představuje vzdálenost mezi *view plane* a ***eye***. Vektor $\boldsymbol{w}$ je potom vypočítán z bodů ***eye*** a ***target*** a následně normalizován. Na obrázku 12 je vektor $\boldsymbol{v}$ shodný s vektorem ***up***.

Z vektorů $\boldsymbol{w}$ a ***up*** se potom vypočítá ortonormální báze složená z vektorů $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$. Výsledný směr paprsku lze poté spočítat jako:

$$\boldsymbol{R_d} = P_x\boldsymbol{u} + P_y\boldsymbol{v} + d\boldsymbol{w} \tag{4.1}$$

kde P_x resp. P_y jsou souřadnice pixelů transformované do souřadnicového systému scény a vektory $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$ tvoří ortonormální bázi.

Počátek paprsku se potom rovná bodu ***eye***. Není potom nutné řešit, zdali objekt, se kterým se paprsek protne, náhodou neleží v prostoru mezi *view plane* a bodem ***eye***.

5 Světla a stíny

Ve scéně musí být umožněno definovat světla k osvětlení vykreslované scény. V přírodě jsou všechna světla plošná, ať se jedná o vlákno žárovky nebo o povrch hvězdy jako Slunce. Nasimulovat však plošné světelné zdroje je výpočetně náročné, proto se často volí aproximace pomocí jednodušších typů světel.

5.1 Ambientní světlo

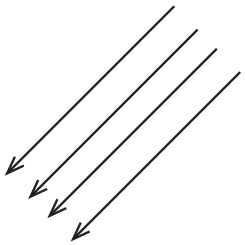
Ambientní světelný zdroj nemá žádný svůj reálný protějšek. Slouží k plošnému přidání určitého světelného příspěvku v libovolném bodě scény. Světlo nemá žádný směr. Není tedy možné ho zahrnout do výpočtů stínovacích algoritmů a také s ním vypočítávat stíny. Pokročilejší metody, které se zabývají výpočtem globálního osvětlení, toto světlo nevyužívají a nahrazují ho přesnějšími výpočty. Intenzitu příspěvku lze vypočítat jako:

$$\boldsymbol{L} = \boldsymbol{C}_{\mathrm{RGB}} \cdot I \tag{5.1}$$

kde C_{RGB} je barva světla v barevném prostoru RGB a I je jeho intenzita. Výslednou hodnotou je opět vektor, který reprezentuje barevný prostor RGB.

5.2 Směrové světlo

Směrové světlo také nemá v reálném světě svůj protějšek. Je však možné tímto světelným zdrojem aproximovat takové světelné zdroje, které jsou velice vzdálené, a lze u nich zanedbat, že jejich paprsky vyzařují pod určitým úhlem. Jako příklad je možné si představit polední slunce, které je vysoko na obloze. To sice je zdroj plošný resp. při dostatečné vzdálenosti zdroj bodový, že jej lze nahradit směrovým světelným zdrojem.



Obrázek 13 – Směrové světlo

Směr světla vzhledem ke konkrétnímu místu je stejný jako směrový vektor, kterým je světlo definováno.

Intenzitu příspěvku lze vypočítat jako:

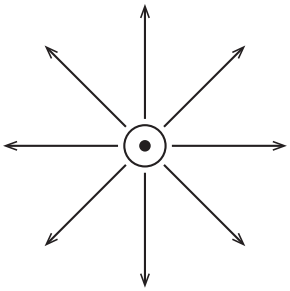
$$\boldsymbol{L} = \boldsymbol{C}_{\mathrm{RGB}} \cdot I \tag{5.2}$$

kde C_{RGB} je barva světla a I je jeho intenzita.

Vzhledem k tomu v jakých situacích se směrový světelný zdroj používá, tak není třeba uvažovat útlum intenzity vzhledem k vzdálenosti.

5.3 Bodové světlo

Bodový zdroj světla si lze představit jako svíčku, žárovku apod. K jeho definování je třeba poloha světelného zdroje, jeho barva a intenzita.



Obrázek 14 – Bodové světlo

Směr světla pro konkrétní bod ve scéně je možné spočítat jako rozdíl mezi polohou zdroje světla P_L a bodem ve scéně P_0:

$$\boldsymbol{D} = \boldsymbol{P_L} - \boldsymbol{P_0} \tag{5.3}$$

Intenzitu světla je možné potom vypočítat jako:

$$\boldsymbol{L} = \boldsymbol{C}_{\mathrm{RGB}} \cdot I \tag{5.4}$$

U tohoto typu světelného zdroje je vhodné umožnit pracovat s útlumem intenzity v závislosti na vzdálenosti bodu od světelného zdroje. Vhodnou funkcí je například útlum rostoucí se čtvercem vzdálenosti, který je možné přesněji řídit pomocí činitele útlumu [5]:

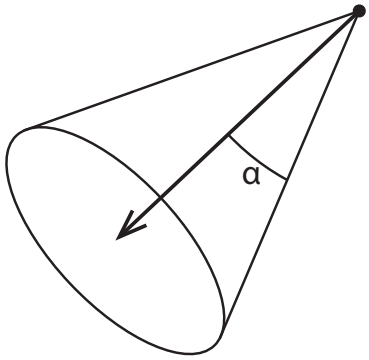
$$I = \frac{I_0}{1 + \alpha d^2} \tag{5.5}$$

kde α je činitel útlumu intenzity a d je vzdálenost od zdroje světla.

5.4 Kužel světla

Pod kuželovým zdrojem světla si lze představit divadelní reflektory, nebo reflektory automobilu. Opět se jedná o zjednodušení chování bodového zdroje, kde by bylo výpočetně náročné počítat odrazy od zrcadla uvnitř reflektorů pomocí kaustiky.

K definování toho světelného zdroje je třeba znát jeho polohu, směrový vektor, barvu, intenzitu a nakonec také úhel resp. sinus úhlu mezi směrovým vektorem a okrajem světelného kužele viz obrázek 15.



Obrázek 15 – Kužel světla

Směr daného světla lze vypočítat jako:

$$\boldsymbol{D} = \begin{cases} \boldsymbol{P_L} - \boldsymbol{P_0}, & \sphericalangle|D,V| < \alpha \\ \text{neexistuje}, & \sphericalangle|D,V| \geq \alpha \end{cases} \tag{5.6}$$

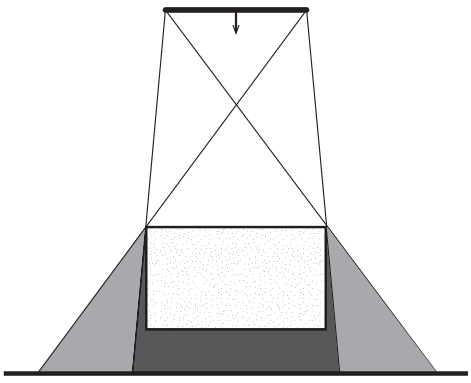
Intenzitu světla je možné vypočítat opět jako:

$$\boldsymbol{L} = \boldsymbol{C}_{\text{RGB}} \cdot I \tag{5.7}$$

Stejně jako u bodového zdroje je zde možné uvažovat útlum intenzity v závislosti na vzdálenosti (5.5). Nově je zde možné také uvažovat útlum intenzity v závislosti na úhlu paprsku vzhledem ke směrovému vektoru.

5.5 Plošné světlo

Plošné zdroje světla se od předchozích značně liší. Jako příklad takovýchto zdrojů v reálném světě lze uvést zářivky, žárovková světla rozptýlená mléčným sklem apod. Jejich klíčovou vlastností je vykreslování měkkých stínů, tak jako je to znázorněno na obrázku 16.



Obrázek 16 – Plošný zdroj spolu se stíny

K definici plošných zdrojů je zapotřebí mít objekt, který bude reprezentovat dané světlo. Na tento objekt je poté třeba pomocí vzorkování rozmístit dostatečný počet bodových zdrojů světla. Intenzita každého bodového zdroje potom bude rovna podílu intenzita plošného zdroje a počtu vzorků resp. bodových zdrojů. Zpravidla se k definici používá jedno-

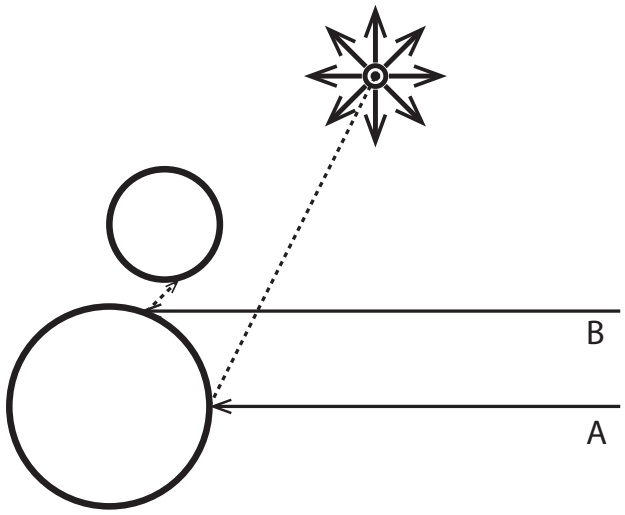
duchých tvarů jako je plocha, trojúhelník, disk nebo koule, z toho důvodu, že tyto objekty je poměrně jednoduché vzorkovat.

Čím více vzorků je použito, tím je výstup kvalitnější, ale také časově náročnější. Výsledek je také závislý na použitém vzorkovacím algoritmu viz kapitola **Chyba! Nenalezen zdroj odkazů.**.

Ne všechny typy světel byly nakonec implementovány v praktické části bakalářské práce. Je však vhodné zde jejich popis uvést. Jejich implementace je potom poměrně rychlou a jednoduchou záležitostí.

5.6 Stíny

Výpočet stínů je při použití ray tracingu poměrně snadnou záležitostí. Z místa průsečíku primárního resp. sekundárního paprsku se vyšle stínový paprsek ke každému světlu a následně se zjišťuje, zdali v jeho cestě nestojí nějaká překážka. Pokud ne, tak bude světelný příspěvek zahrnut ve výsledné barvě pixelu. Jednotlivé typy paprsků byly popsány v kapitole 3.1



Obrázek 17 – Ukázka stínových paprsků. Stínové paprsky jsou znázorněny čárkovaně

Na obrázku 17 je znázorněna scéna se dvěma koulemi a jedním bodovým světelným zdrojem. Stínový paprsek vytvořený mezi místem dopadu primárního paprsku A a bodovým světelným zdrojem nemá ve své cestě žádnou překážku, tudíž dojde k osvětlení místa dopadu paprsku A. Pro stínový paprsek od primárního paprsku B byl ovšem nalezen průsečík s menší z obou koulí, proto je místo průsečíku na větší zastíněno menší koulí.

6 Tělesa ve scéně a jejich popis

Základním stavebním kamenem scény jsou tělesa, se kterými je možné analyticky vypočítat průsečík s paprskem. Příkladem těchto objektů může být koule, trojúhelník, válec, plocha, nebo také NURBS plocha. Pomocí těchto těles je potom možné definovat složitější struktury (trojúhelníková síť, CSG modelování), kterými lze popsat složitější objekty.

Je vhodné, aby hladké oblé povrchy (koule, válec aj.) byly definovány analyticky namísto reprezentace pomocí trojúhelníkové sítě, kdy je třeba k vytvoření iluze hladkého zaobleného tělesa velké množství trojúhelníků.

6.1 Koule

K popisu koule je třeba znát její střed a poloměr, jedná se tedy o paměťově nenáročné těleso.

Pro zjištění průsečíku koule s paprskem lze vyjít z analytické rovnice koule ve vektorovém tvaru:

$$(\boldsymbol{P} - \boldsymbol{S}) \cdot (\boldsymbol{P} - \boldsymbol{S}) - r^2 = 0 \tag{6.1}$$

kde $\boldsymbol{S}$ je střed koule, r je její poloměr a $\boldsymbol{P}$ je hledaný bod, který představuje substituci $\boldsymbol{P} = \boldsymbol{O} + \mathbf{d}t$, která vychází z definice paprsku resp. polopřímky v kapitole 2.3. Rovnici potom lze zapsat jako:

$$(\boldsymbol{O} + \boldsymbol{d}t - \boldsymbol{S}) \cdot (\boldsymbol{O} + \mathbf{d}t - \boldsymbol{S}) - r^2 = 0 \tag{6.2}$$

A následně upravit:

$$(\boldsymbol{d} \cdot \boldsymbol{d})t^2 + [2(\boldsymbol{O} - \boldsymbol{S}) \cdot \boldsymbol{d}]t + (\boldsymbol{O} - \boldsymbol{S}) \cdot (\boldsymbol{O} - \boldsymbol{S}) - r^2 = 0 \tag{6.3}$$

Rovnice je tedy kvadratickou rovnicí parametru t:

$$at^2 + bt + c = 0 \tag{6.4}$$

kde:

$$\begin{gathered} a = (\boldsymbol{d} \cdot \boldsymbol{d}) \\ b = 2(\boldsymbol{O} - \boldsymbol{S}) \cdot \boldsymbol{d} \\ c = (\boldsymbol{O} - \boldsymbol{S}) \cdot (\boldsymbol{O} - \boldsymbol{S}) - r^2 \end{gathered} \tag{6.5}$$

Diskriminant je potom:

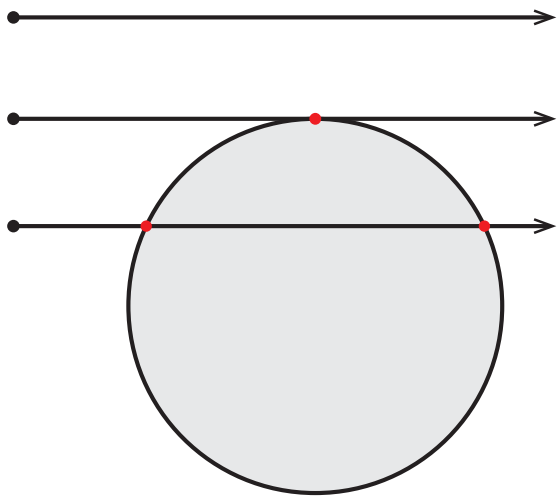
$$D = b^2 - 4ac \tag{6.6}$$

Na obrázku 18 jsou vidět tři možnosti polohy paprsku vůči kouli. Pokud je $D < 0$, rovnice nemá řešení, paprsek tudíž kouli neprotíná. Pokud je $D = 0$ paprsek se dotýká koule v jednom bodě a řešení je:

$$t = \frac{-b}{2a} \tag{6.7}$$

Pokud je $D > 0$, paprsek protíná kouli ve dvou místech a je třeba určit menší ze dvou získaných parametrů t. Řešení je potom následující:

$$t_{12} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{6.8}$$



Obrázek 18 – Tři možnosti průsečíku paprsku s koulí

Normálu $\boldsymbol{N}$ v místě průsečíku lze snadno vypočítat jako:

$$\boldsymbol{N} = \boldsymbol{P} - \boldsymbol{S} \tag{6.9}$$

kde P je nalezený průsečík paprsku s tělesem a S je střed koule.

6.2 Trojúhelník

Trojúhelník je v moderní počítačové grafice jakýmsi základním stavebním kamenem pro vytváření složitějších trojrozměrných modelů. K jeho základní definici postačují tři body v prostoru. Dále je ještě vhodné uchovávat pro jednotlivé body jejich mapovací souřadnice. Takovým bodům potom říkáme vertexy.

Další věcí, kterou je nutno řešit, je výpočet normály v místě průsečíku s paprskem. Existují tři možnosti, jak problém řešit [10] [13].

- Tři body v prostoru definují rovinu. Normála je potom vektor kolmý k této rovině. Tento způsob se nazývá *flat shading*, trojúhelníky po vykreslení jsou ploché.

- Vypočítají se normály pro jednotlivé vertexy součtem normál trojúhelníků, ke kterým vertex patří. Normála v místě průsečíku s paprskem se potom vypočítá jako interpolace pomocí barycentrických souřadnic z těchto tří normál. Tento způsob se nazývá *smooth shading*, a s použitím stínování se potom vytváří iluze toho, že je trojúhelník zaoblený.
- Poslední způsob vychází z předcházejícího s tím rozdílem, že normály se nevypočítávají, ale jsou přímo zadané. Řešení je více flexibilní, dovoluje kombinovat dvě předchozí metody.

Pro zjištění průsečíku mezi trojúhelníkem a paprskem existuje řada různě složitých algoritmů. Vzhledem k tomu, že u složitých modelů a scén může být počet trojúhelníku v řádu (desítek) milionů, je vhodné, aby tento algoritmus byl co nejrychlejší.

Jako příklad a zároveň jakýsi standard by se dal uvést algoritmus Möller-Trumbore [4] z roku 1997. Jeho výhoda spočívá také v tom, že při výpočtu se využívají barycentrické souřadnice průsečíku uvnitř trojúhelníku, které mohou být dále použity pro výpočet konkrétní normály a mapovací souřadnice. Na obrázku 19 je ilustrováno řešení problému algoritmem Möller-Trumbore. Nejprve problém transformuje do počátku souřadného systému a následně transformací M^{-1} transformuje do barycentrických souřadnic.

Algoritmus vychází z toho, že barycentrické souřadnice umožnují parametrizovat trojúhelník za dvěma proměnnými b_1 a b_2:

$$\boldsymbol{p}(b_1, b_2) = (1 - b_1 - b_2)\boldsymbol{V_0} + b_1\boldsymbol{V_1} + b_2\boldsymbol{V_2} \tag{6.10}$$

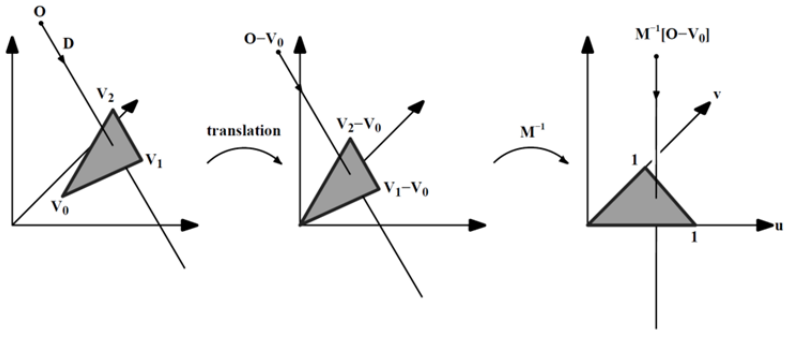
kde $b_1 \geq 0 \wedge b_2 \geq 0 \wedge b_1 + b_2 \leq 1$. Pokud barycentrické souřadnice průsečíku neodpovídají těmto podmínkám, pak paprsek neprotíná trojúhelník.

Hledaný bod je nahrazen rovnicí paprsku:

$$\mathbf{O} + \mathbf{d}t = (1 - b_1 - b_2)\boldsymbol{V_0} + b_1\boldsymbol{V_1} + b_2\boldsymbol{V_2} \tag{6.11}$$

Po úpravě rovnice potom dostáváme rovnici ve tvaru:

$$[-\boldsymbol{d} \quad \boldsymbol{V_1} - \boldsymbol{V_0} \quad \boldsymbol{V_2} - \boldsymbol{V_0}] \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \boldsymbol{O} - \boldsymbol{V_0} \tag{6.12}$$



Obrázek 19 – Ilustrace algoritmu Möller-Trumbore [4]

Provedeme substituci $\boldsymbol{e_1} = \boldsymbol{V_1} - \boldsymbol{V_0}$, $\boldsymbol{e_2} = \boldsymbol{V_2} - \boldsymbol{V_0}$ a $\boldsymbol{s} = \boldsymbol{O} - \boldsymbol{V_0}$ a následně tuto rovnici můžeme řešit pomocí Cramerova pravidla:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{|-\mathbf{d} \quad \mathbf{e_1} \quad \mathbf{e_2}|} \begin{bmatrix} |\mathbf{s} \quad \mathbf{e_1} \quad \mathbf{e_2}| \\ |-\mathbf{d} \quad \mathbf{s} \quad \mathbf{e_2}| \\ |-\mathbf{d} \quad \mathbf{e_1} \quad \mathbf{s}| \end{bmatrix} \tag{6.13}$$

Z lineární algebry je známé, že výraz $|\boldsymbol{A} \quad \boldsymbol{B} \quad \boldsymbol{C}| = -(\boldsymbol{A} \times \boldsymbol{C}) \cdot \boldsymbol{B} = -(\boldsymbol{C} \times \boldsymbol{B}) \cdot \boldsymbol{A}$. Rovnici je potom možné přepsat do tvaru:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\boldsymbol{m} \cdot \boldsymbol{e_1}} \begin{bmatrix} \boldsymbol{n} \cdot \boldsymbol{e_2} \\ \boldsymbol{m} \cdot \boldsymbol{s} \\ \boldsymbol{n} \cdot \boldsymbol{d} \end{bmatrix} \tag{6.14}$$

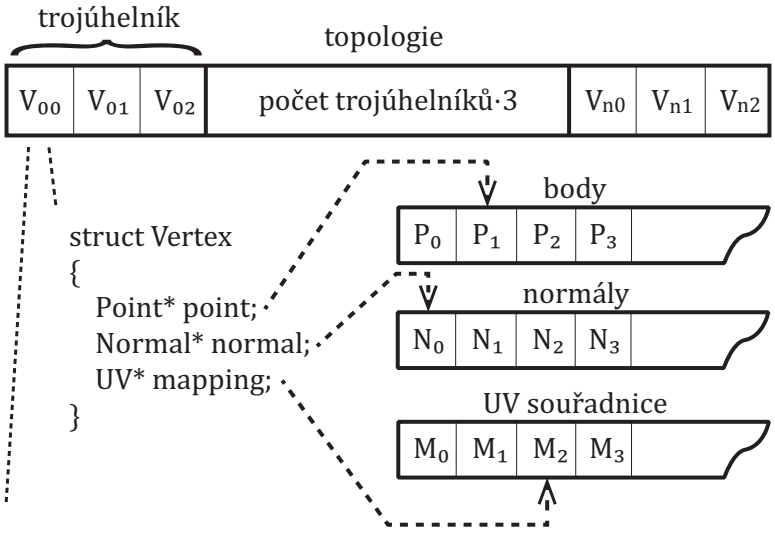
kde $\boldsymbol{m} = \boldsymbol{d} \times \boldsymbol{e_2}$ a $\boldsymbol{n} = \boldsymbol{s} \times \boldsymbol{e_1}$.

Pokud ukládáme hodnoty normál pro každý vertex, potom její hodnotu v místě průsečíků získáme interpolací ve tvaru $\boldsymbol{N} = (1 - b_1 - b_2)\boldsymbol{N_0} + b_1\boldsymbol{N_1} + b_2\boldsymbol{N_2}$, viz kapitola 2.5. Obdobně lze interpolovat také mapovací souřadnice.

6.3 Trojúhelníková síť

Trojúhelníková síť je datová struktura, která umožňuje pomocí trojúhelníků vytvářet složitější modely. Vzhledem k tomu, že sousední trojúhelníky spolu sdílí body, normály, případně mapovací souřadnice, je také vhodné, aby s tím datová struktura počítala a dokázala tyto duplicity, z důvodu šetření paměti, odstranit. Pozor jedná o sdílení dat o bodech apod. nikoliv o struktury trojúhelníku jako například *strip* a *fan* jako v OpenGL a podobných metodách.

Vzhledem k tomu, že při zobrazování pomocí ray tracingu není třeba provádět složitější výběr hran, zjišťovat sousední trojúhelníky, jako například při modelování v modelovacích programech, je možné strukturu pojmout relativně jednoduše.



Obrázek 20 – Ilustrace datové struktury trojúhelníkové sítě

Základem datové struktury jsou buffery, které uchovávají body, normály a mapovací souřadnice (volitelné). Dále je vhodné zavést pomocnou datovou strukturu Vertex, která uchovává ukazatele na bod, normálu a mapovací souřadnici, které jsou uloženy v bufferech. Topologii modelu potom uchováme v bufferu, jehož velikost je $početTrojúhelníků \cdot 3$. Ta uchovává vertexy po třech pro každý trojúhelník. Tento buffer je nakonec použit pro výpočet průsečíků.

6.4 Další možnosti

Tímto přehledem jistě nejsou vyčerpány všechny možnosti, jak definovat objekty ve scéně. Analyticky lze vyjádřit také další tvary jako například válec, kruh, hyperboloid, paraboloid, kužel, toroid, nekonečná plocha atd. Další možnosti poskytují křivky a NURBS plochy pomocí nich vytvořené. Popis těchto metod je ovšem nad rámec bakalářské práce.

Techniky souhrnně označované jako subdivision surface, potom umožňují během vykreslování zjemňovat trojúhelníkové sítě. Na rozdíl od klasické zjemnění dostupné přímo v programech, většinou pod názvem (mesh)smooth, je možné definovat při tomto postupu váhy jednotlivých vrcholů resp. hran. Zajímavostí k tomuto je fakt, že tyto nástroje byli v programu Autodesk Maya[5] dostupné delší dobu, v nedávné době však přestali být podporovány.

Další samostatnou oblastí je zobrazování částicových efektů, objemových těles, jako je kouř a dým. Tato oblast je poměrně komplikovaná. V [6] je tomuto tématu věnována jedna kapitola, která nabízí úvod do této problematiky.

Obrázek 21 – Složitější model vykreslený pomocí implementovaného rendereru

[5] http://www.autodesk.com/products/autodesk-maya/overview

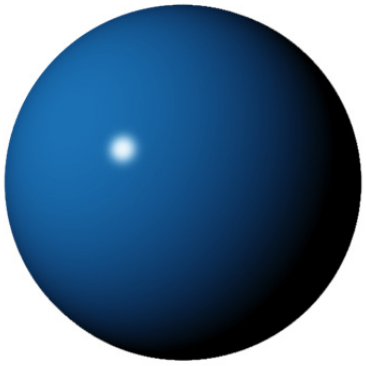
7 Stínování a materiály

Při vykreslování scény je třeba vzít v potaz, že různé předměty v reálném světě mohou být vyrobeny z různých materiálů, jako je dřevo, kov, plast, papír a každý z nich má rozdílné vlastnosti jako barvu, lesklost apod. Tyto vlastnosti vypovídají především o tom, jak daný materiál odráží dopadající světlo.

Vzhledem k tomu, že povrch tělesa není nikdy dokonale hladký a je tvořen jemnou strukturou plošek a krystalků, tak může paprsek po dopadu opustit povrch dvěma způsoby. Buď se na povrchu několikrát odrazí a zlomí a působí potom jako difúzní složka a nebo se zrcadlově odrazí a potom je to takzvaně složka zrcadlová.

Difúzní složka (*diffuse*) nezávisí na směru pohledu. Paprsek se po mnohočetném odrazu a lomu odrazí do náhodného směru se stejnou pravděpodobností všech směrů. Intenzita této složky závisí jen na úhlu dopadu světla.

Naproti tomu zrcadlová složka (*specular*) je závislá také na směru pozorování. Směrovost této složky je tím větší, čím je povrch hladší. Světlo odražené od dokonalého zrcadla má pouze tuto složku. V praxi tato složka slouží především pro vytvoření iluze použití plošných světel, pokud jsou používány zdroje bodové. Pro tvorbu odrazů se potom používá sekundárních paprsků.



Obrázek 22 – Ukázka difúzní a zrcadlové složky

Materiály v trojrozměrné počítačové grafice jsou struktury, pomocí kterých lze modelovat, jak odrážejí světlo pod určitými úhly dopadu. Tyto modely lze rozdělit na:

- Empirické, reprezentované vhodně zvolenými matematickými funkcemi. Typičtí zástupci tohoto přístupu jsou Lambertův osvětlovací model, Phongův osvětlovací model, a z pokročilejších například Oren-Nayar model, který lépe modeluje matné povrchy než Lambertův model, nebo Torrance-Sparrowův model, který je založen na modelu mikroplošek [12].

- Naměřené modely pomocí speciálního přístroje, známém jako gonioreflektometr[6]. Naměřená data jsou potom tabelována a hodnoty pro konkrétní hodnoty úhlů a barev světla jsou získávána pomocí interpolace.

7.1 BRDF

Základem materiálů je BRDF ([12], [4]), která udává subkritickou hustotu pravděpodobnosti, že se světlo, které dopadne na povrch, odrazí daným směrem:

$$f_r(\omega_i, \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)\cos\theta_i\,\mathrm{d}\omega_i}[sr^{-1}] \tag{7.1}$$

kde ω_r je odchozí směr světla a ω_i je příchozí směr světla, definované vůči normále. Jednotkou je sr^{-1} a vyjadřuje poměr odražené diferenciální záře k ozáření.

BRDF je vzhledem k záři lineární, takže příspěvky jednotlivých světel lze sčítat. Dále platí Helmholzelův princip reciprocity, který vychází ze zákona odrazu a říká, že hodnota funkce zůstane stejná, pokud zaměníme příchozí a odchozí směr:

$$f_r(\omega_i \to \omega_r) = f_r(\omega_r \to \omega_i) \tag{7.2}$$

Nakonec zde platí také zákon zachování energie, který říká, že poměr odraženého zářivého toku k příchozímu zářivému toku musí být menší než 1:

$$\frac{\mathrm{d}\Phi_r}{\mathrm{d}\Phi_i} \leq 1 \tag{7.3}$$

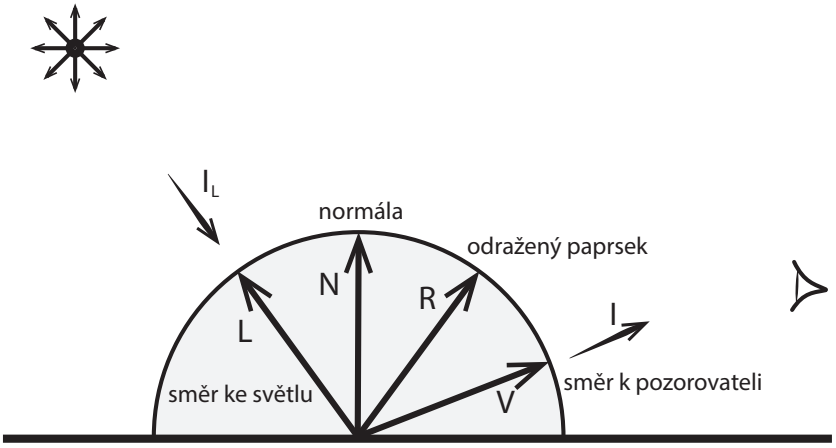
7.2 Phongův osvětlovací model

Jedním z často používaných osvětlovacích modelů byl navržen Bui Tuong Phongem [13]. Model připomíná materiál plastu. Odražené světlo se skládá ze tří částí:

$$I = I_d + I_s + I_a \tag{7.4}$$

kde I_d je difuzní složkou, I_s je zrcadlovou složkou a I_a je ambientní složkou. Ambientní složka se přidává z důvodu jednoduché simulace okolního světla. V dnešní době je tato metoda však překonaná technikami pro výpočet globálního osvětlení. Z tohoto důvodu se také nejedná o BRDF, protože je porušen zákon zachování energie.

[6] http://en.wikipedia.org/wiki/Gonioreflectometer



Obrázek 23 – Odraz světla od povrchu tělesa

Difuzní složka se vypočítá podle Lambertova zákona:

$$I_d = I_L r_d \boldsymbol{C} \cos(\sphericalangle \boldsymbol{LN}) \\ I_d = I_L r_D \boldsymbol{C}(\boldsymbol{L} \cdot \boldsymbol{N}) \tag{7.5}$$

kde I_L je světelný příspěvek světla, r_d je koeficient difuzního složky, $\boldsymbol{C}$ barva materiálu a $\boldsymbol{L}$ je směr ke světlu a $\boldsymbol{N}$ je normála povrchu. Intenzita příspěvku I_d je tedy přímo úměrná velikosti funkce kosinus úhlu, který mezi sebou svírají vektory $\boldsymbol{L}$ a $\boldsymbol{N}$.

Zrcadlová složka I_s se vypočítá jako:

$$I_s = I_L r_s (\boldsymbol{V} \cdot \boldsymbol{R})^{exp} \tag{7.6}$$

kde I_L je světelný příspěvek světla, r_s je koeficient zrcadlové složky, $\boldsymbol{V}$ je směr pohledů a $\boldsymbol{R}$ směr odraženého paprsku, exp je potom Phongův exponent. Čím je tento exponent větší, tím se odlesky stávají menší a ostřejší. Vektor $\boldsymbol{R}$ lze vypočítat z vektoru $\boldsymbol{L}$ pomocí rovnice (3.8).

Ambientní složka se potom vypočítá jako:

$$I_a = I_A r_a C \tag{7.7}$$

kde I_A je barva okolního světla, r_a je koeficient odrazu okolního světla a $\boldsymbol{C}$ je barva povrchu.

V případě klasického Phongova osvětlovacího modelu se nejedná o fyzikálně korektní BRDF, nesplňuje podmínku reciprocity při záměně ω_i a ω_o, nesplňuje také zákon zachování energie, protože díky složce I_a může být výsledné $I > 1$.

Jednotlivé složky materiálu je vhodné implementovat jako samostatné BRDF, zvýší se variabilita použití.

V praktické části je implementováno několik materiálu od matného, který je napsán podle Lambertova zákona, přes Phongův materiál až po materiály, které pracují s odrazem a lomem světla.

8 Optimalizace

Uvažujme scénu, která obsahuje 2000 různých objektů, 2 světla se zapnutým vrháním stínů, a tato scéna je vykreslována do rozlišení 800x800. Pokud je použita základní implementace metody sledování paprsku, bude počet provedených testů průsečíku paprsek-objekt:

$$800 \cdot 800 \cdot 2000 = 1{,}28 \cdot 10^9 \tag{8.1}$$

V tomto motivačním případě není zahrnuta nutnost výpočtu sekundárních a stínových paprsků.

Z této úvahy a také například z měření pomocí profileru takovéto implementace vyplyne, že právě metody pro provádění testů průsečíků paprsek-objekt zabírají nejvíce procesorového času. Tyto metody je třeba optimalizovat.

Na optimalizaci se dá pohlížet jako:

- optimalizaci kritických metod,
- minimalizaci počtu provádění kritických metod.

V prvním případě se využívají, případně navrhují, úspornější algoritmy pro dosažení cíle. Dále je vhodné se vyvarovat využívání matematických funkcí typu sinus a cosinus, které jsou implementované v programovacích jazycích a pokud to lze nahrazovat je prací s vektory. Pokud je to možné tak se vyhnout iterativním algoritmům pro přibližné řešení rovnic. Z pokročilejších metod je účinné cílené používání SIMD instrukcí [14] (jedna instrukce prováděná zároveň nad více daty) pro realizaci vektorových operací. Nakonec, pokud bude bráno jako kritérium výsledný čas renderingu, tak lze mezi optimalizace započítat také paralelizaci výpočtu. Tyto metody poskytují jisté možnosti urychlení, nebude s nimi však dosaženo řádových rozdílů.

Ve druhém případě jsou využívány takzvané akcelerační struktury, které zmenšují počet nutných vykonávání kritických funkcí.

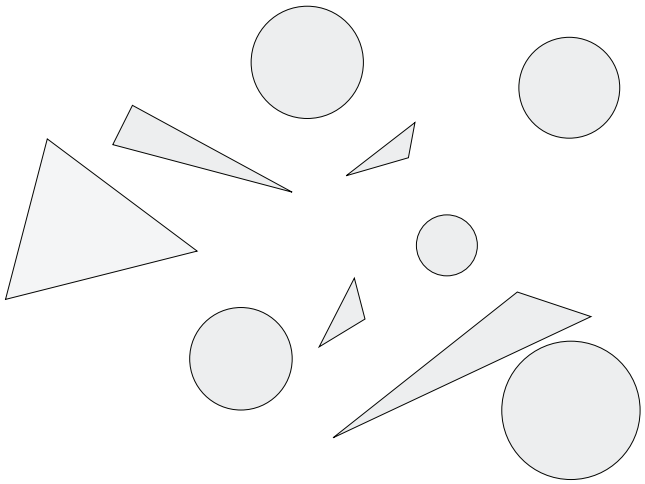
Za správně implementovanou akcelerační strukturu lze považovat takovou strukturu, která podává stejné výsledky jako výpočet provedený bez ní, samozřejmě ale v kratším čase.

Akcelerační struktury je možné rozdělit z hlediska toho, jak daný problém řeší na ty, co:

- **dělí objekty** – dělí objekty buď na logické kusy (židle → nohy, opěradlo, sedadlo), nebo na části, které leží poblíž sebe. Typický zástupce této kategorie je *bounding volume hierarchy* [12], která se kromě akcelerace raytracingu používá také v počítačových hrách k detekci kolizí.
- **dělí prostor** – rozdělí prostor scény a do takto rozděleného prostoru potom přiřazuje objekty. Sem patří například struktura *grid* [12], *kd-tree* [12], *octree* [4] případně jejich úplné zobecnění v podobě *binary space partitioning* [15] (využívá se také

poměrně často za stejným účelem v počítačových hrách, jednu z takových aplikací můžeme najít například ve hře Doom).

V následující části bude popsáno několik typů akceleračních struktur. Demonstrace bude pro přehlednost uváděna pouze ve 2D prostoru na následující scéně.



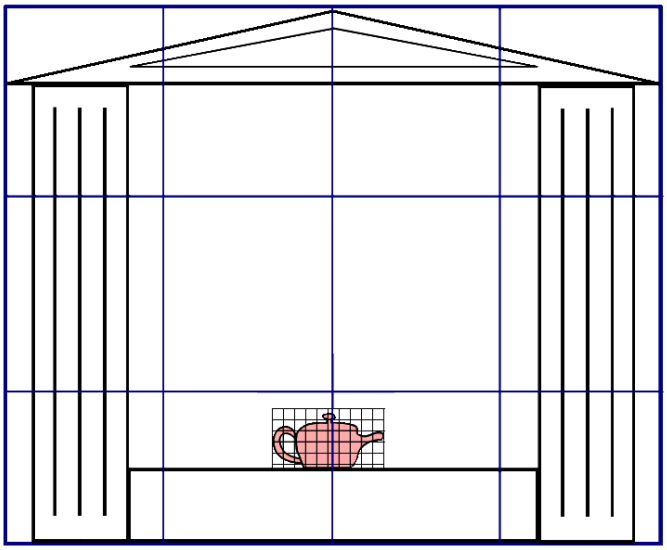
Obrázek 24 – Ukázková scéna

8.1 Grid (mřížka)

Princip této akcelerační struktury spočívá v tom, že scéna se obalí trojrozměrnou mřížkou tvořenou z takzvaných voxelů (analogie pixelů ve 2D). K procházení struktury (hledání průsečíku paprsku s tělesem) se používá DDA algoritmus modifikovaný pro 3D prostředí.

Výhodou této struktury je její poměrně snadná implementace a poměrně rychlé vybudování celé struktury. Naopak horší je to s paměťovou náročností, kde je třeba mít v každém voxelu poměrně hodně informací a zároveň je voxelů velký počet, takže je struktura poměrně náročná na paměť[7]. To lze řešit například tak, že při budování struktury budeme alokovat pouze ty voxely, u kterých bude jisté, že budou obsahovat nějaká tělesa. Při průchodu strukturou toto řešení nijak neomezuje, jen je potřeba kontrolovat, zda byl voxel alokován. Dále je vzhledem k vysokému počtu voxelů vhodné používat alokování paměti po vetších blocích a z těchto bloků potom přidělovat paměť pro jednotlivé voxely (*region based allocating*). Toto opatření minimalizuje nutnost žádat systém o přidělení nového bloku paměti a přináší úsporu času během vytváření struktury. Vzhledem k tomu, že budování struktury však není nejdéle trvající úlohou při vykreslení (pohybuje se řádově v sekundách), tak je dosaženo jen nepatrného zrychlení, které má smysl řešit až u produkce delší animace, kde je třeba strukturu často přepočítávat.

[7] V přiložené implementaci je to při mřížce o rozměru 64x64x64 $64^3 \cdot 24 byt\mathring{u} = 6Mb$ a to zde nejsou započítány ukazatele na tělesa.



Obrázek 25 – Špatný příklad rozdělení prostoru

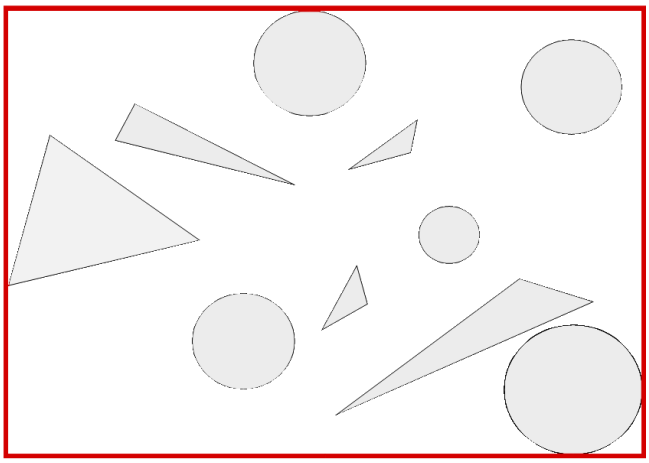
Na Obrázek 25 lze naopak vidět velkou nevýhodu této struktury. Pokud scéna obsahuje objekty, které jsou velikostně velmi rozdílné, je možné se dostat do situace, která se v anglické literatuře označuje jako *„teapot in stadiun“* případně *„teapot in temple“*. Ve scéně velkých rozměrů je umístěno relativně složité těleso reprezentované například trojúhelníkovou sítí. V tomto případě se nám může stát, že stadion bude rozdělen v sítí tak, jak by měl být, kdežto relativně složitá konvička bude celá v jednom voxelu. Řešit to lze buď zvýšením hustoty mřížky. Zde je ale vhodné zvážit riziko, že výpočty prováděné při průchodu sítí zvýší renderovací čas zbytku obrázku a čas získaný na detailním malém tělesu nebude v rovnováze s časem, který ztratíme na zbytku scény. Druhým řešením je přistupovat k akcelerační struktuře jako k tělesu, takže je možné pro tyto detailní objekty vytvořit samostatnou akcelerační strukturu a tu následně vložit do akcelerační struktury celé scény. Tím lze vytvářet takzvané *„nested grids“*. Zde záleží opět na uživatelských zkušenostech a citu, jak tuto možnost využít.

8.1.1 Vytvoření mřížky

Vstupními parametry, které potřebné k vytvoření mřížky jsou:

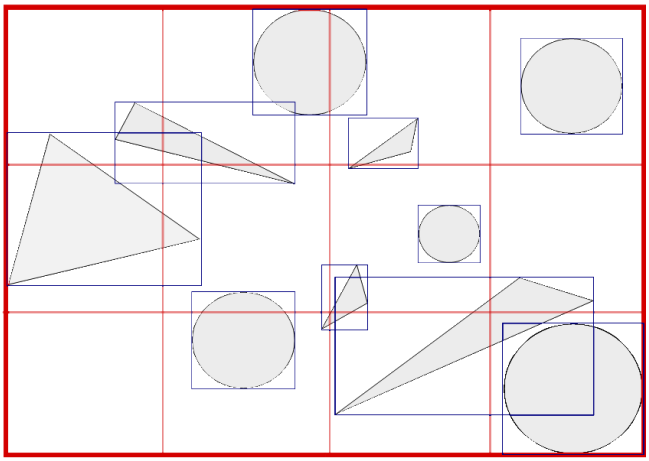
- seznam objektů, pro které má být struktura vytvořena,
- maximální počet voxelů v nejdelším rozměru,
- další parametry, které určují, zdali bude možné vytvářet „nested grids“, a dále parametry, které souvisí s konkrétní implementací.

Algoritmus začne tím, že si zjistí obalovou krychli objektů zadané scény.



Obrázek 26 – Obalový kvádr scény

Dále se vzniklý prostor rozdělí na voxely, ke kterým jsou přiřazeny ukazatele na objekty, u kterých se alespoň část obalové kvádru nachází uvnitř voxelu.



Obrázek 27 – Rozdělení prostoru

Pokud vytváříme *„nested grids“* tak je třeba učinit rozhodnutí, jestli se struktury hierarchicky níže budou vytvářet v tuto chvíli (a spotřebovávat tak paměť), přestože nemusí být během výpočtu nikdy použity, nebo je budeme vytvářet až v případě potřeby. Toto je třeba zvážit, zvláště v případě, že bychom chtěli výpočet paralelizovat. V tom případě by bylo nutné po dobu vytváření zamezit přístup do struktury například pomocí mutexu, jinak by struktura nemusela být v konzistentním stavu.

8.1.2 Hledání průsečíku pomocí mřížky

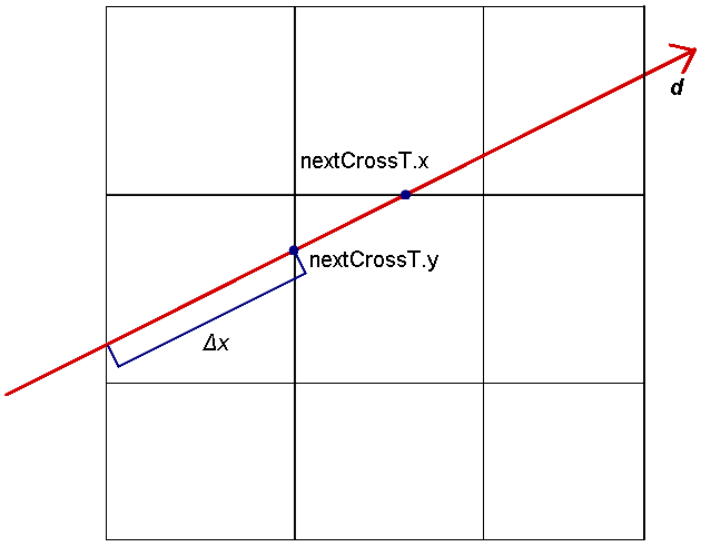
Hledání nejbližšího průsečíku probíhá v mřížce pomocí modifikovaného algoritmu DDA, který se ve 2D grafice používá ke kreslení úseček. Nejprve je vhodné zjistit, zdali paprsek vůbec protíná mřížku, tudíž jestli protíná nějaký z našich těles.

Poté je nutné vypočítat parametry potřebné pro samotný průchod:

- $\Delta x, \Delta y, \Delta z$ – délka kroku v jednotlivých krocích,

- $\vec{x}, \vec{y}, \vec{z}$ – směr kroku, který je roven signum$(ray.dx)$[8], resp. signum$(ray.dy)$, signum$(ray.dz)$,
- $outX, outY, outZ$ – číslo posledního voxelu, je zde kvůli ukončení procházení,
- $nextCrossT[3]$ – souřadnice dalšího průsečíku mezi voxely pro každou z os.

Poté může začít samotný průchod strukturou. Ten je již analogický standardnímu DDA algoritmu jen s tím rozdílem, že je třeba postupovat po voxelech a mít přitom na paměti směr v každé ose.



Obrázek 28 – Průchod strukturou

[8] Funkce signum vrací -1, pokud je parametrem záporné číslo, 0 pokud je parametrem 0 a 1 pokud je parametrem kladné číslo.

9 Přehled stávajících řešení

9.1 mental ray

mental ray[9] je komerční renderovací software byl vyvíjen od roku 1986 společností Mental Images, v roce 2009 byl odkoupen společností NVIDIA. Podporuje distribuovaný raytracing, pro kaustiku a globální osvětlení používá photon mapping. Dokáže pracovat s polygonovou reprezentací, subdivision surface a plochami NURBS. V případě použití speciálních materiálu podporuje také spektrální rendering (s barvou světla se počítá jako s vlněním). Firma NVIDIA přidala v poslední době také součást iRay[10], která podporuje renderování v reálném čase.

Je distribuován spolu s velkými balíky pro tvorbu 3D grafiky jako je Autodesk 3ds max, Autodesk Maya, Autodesk Softimage, Houdiny a další.

V roce 2003 dostala firma Mental Images Oscara za přínos filmu.

9.2 V-Ray

V-Ray[11] je komerční renderovací software vyvíjený od roku 1997 bulharskou společností Chaos Software. Pro výpočet globálního osvětlení je možné zvolit mezi path tracingem, photon mappingem, irradiance mapami, a nebo přímý výpočet hrubou silou.

Je k dispozici pro programy Autodesk 3ds max, Autodesk Maya, SketchUp, Rhino, Cinemu4D a Blender.

9.3 Maxwell render

Maxwell render[12] mladý komerční renderer je vyvíjen španělskou firmou Next Limit Technologies. Jedná se o první masově užívaný unbiased renderer. Je v něm použit algoritmus Metropolis light transport pro výpočet globálního osvětlení. Je možné ho propojit s řadou balíků pro tvorbu 3D grafiky, nebo využít jeho součást Maxwell studio. Také podporuje možnost interaktivního renderování v reálném čase.

9.4 POV-Ray

Opensource ray tracer POV-Ray[13], používají k popisu scény specializovaný jazyk SDL. Podporuje CSG tělesa, radiozitu, mapování fotonů. Je možnost propojit ho s programem Blender, jinde se moc nepoužívá.

[9] http://www.nvidia-arc.com/mentalray.html
[10] http://www.nvidia-arc.com/iray.html
[11] http://www.vray.com/
[12] http://www.maxwellrender.com/
[13] http://www.povray.org/

9.5 LuxRender

Opensource renderer LuxRender[14], který podporuje snad vše, na co si člověk vzpomene. Za zmínku stojí například GPU akcelerace pro path tracing, plně spektrální rendering, rozptyl světla pod povrchem objektu a spolupráce více počítačů při renderingu po síti. K dispozici je úplné propojení s programy Blender a Autodesk 3ds max.

[14] http://www.luxrender.net/en_GB/index

10 Implementace vlastního raytraceru

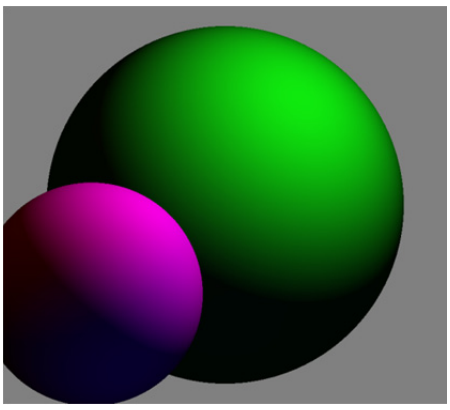
V rámci bakalářské práce byl implementován vlastní ray tracingový rendererovací systém. Základní vlastnosti tohoto rendererovacího systému jsou:

- modulárnost ve smyslu možnosti definování rozhraní pro jednotlivé části rendereru. Není tedy problém nahradit nebo přidat téměř libovolnou součást,
- multiplatformnost programu,
- paralelizace programu pomocí OpenMP[15],
- implementování vlastní tříd pro práci s vektory,
- podpora trojúhelníkových modelů a základních analyticky popsatelných tvarů,
- scénu lze definovat pomocí XML souboru, trojúhelníkové modely jsou potom načítány z formátu OBJ,
- byla implementována akcelerační datová struktura mřížky. Díky tomu je rychlost vykreslování řádově vyšší,
- implementace několika základních typů materiálů,
- lom a odraz světla,
- perspektivní kamera,
- uživatelské rozhraní.

10.1 Volba jazyka a platformy

Před samotným výběrem programovacího jazyka bylo nejprve jednoduché jádro experimentálně implementováno v jazycích C++, C# a Java. Staticky definovaná scéna obsahovala 4 bodová světla, kameru s rovnoběžným promítáním a dvě koule.

Implementace v jazyce C++ probíhá pomaleji, je třeba spravovat paměť, k tvorbě GUI je zapotřebí externích knihoven. C++ má možnost přetěžovat téměř libovolné operátory, což pokud je důsledně používáno může vést k výraznému zpřehlednění kódu. Testovací scénu byla vykreslena za čas okolo 1 sekundy.



Obrázek 29 – Scéna pro experimentální implementace

[15] http://openmp.org/wp/

V C# probíhala tato jednoduchá implementace velice rychle. Jazyk poskytuje velké možnosti, z nichž praktické je přetěžování operátorů. Bohužel na testovací scéně byl čas potřebný k vykreslení kolem 8 sekund, který lze nejspíše přičíst tomu, že C# kód běží nad virtuálním strojem a také integrovanému *garbage collectoru*. Další nevýhodou je, že jazyk C# a knihovna .NET není multiplatformní (pokud nebude uvažována knihovna Mono).

Java se ukázala pro řešení jako velice nevhodná, protože podstatnou část aplikace tvoří vektorové výpočty. Java nepodporuje přetěžování operátorů a tak se výpočty s vlastními třídami pro vektory, body a normály staly velice nepřehledné. Z tohoto důvodu také nebyla dokončena ani experimentální jednoduchá implementace.

Pro implementaci byl nakonec zvolen jazyk C++ s multiplatformní knihovnou *wxWidgets 2.9.4*[16], ze které byly využity části pro tvorbu GUI, načítání ze souborů XML, ukládání a načítání obrázků ve formátech JPG, BMP a PNG. Zatím není podporována verze knihovny *wxWidgets 3.0.0* a novější z důvodu zpětné nekompatibility některých částí.

K vývoji bylo použito vývojových prostředí Microsoft Visual Studio 2012 (Windows) a Eclipse s rozšířením CDT (Linux). Při vývoji byl použit verzovací systém git. Repozitář se zdrojovými soubory a projekty pro vývojová prostředí bakalářské práce je volně dostupný na adrese https://github.com/polygoncz/raytracer. Pro vytvoření vývojářské dokumentace ze zdrojových kódů byl použit program Doxygen[17].

10.2 Adresářová struktura projektu

```
Raytracer
│
├──build
│   ├──eclipse
│   └──MSVC
│
└──src
    ├──agreggates
    ├──brdf
    ├──cameras
    ├──core
    ├──import
    ├──integrators
    ├──ligths
    ├──materials
    ├──renderers
    ├──shapes
    └──wxgui
```

V adresářové struktuře jsou odděleny složky se zdrojovými kódy (*src*) a samotnými soubory projektů vývojových prostředí (*build*). To přináší složitější nastavení a samotnou praci s vývojovými prostředími, například vkládání nových souborů, mazání. Výhodou je, že vývoj není vázán na jedno konkrétní vývojové prostředí.

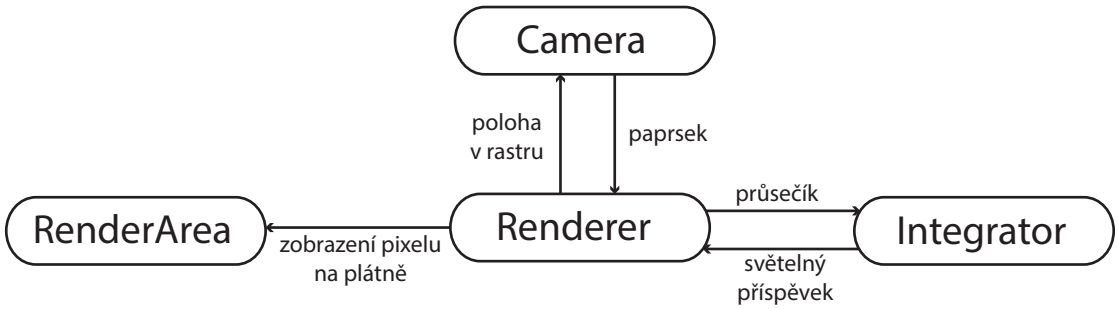
[16] https://www.wxwidgets.org/
[17] http://www.stack.nl/~dimitri/doxygen/

Ve složce *src/core* jsou implementace základních tříd, jako jsou například vektory, normály, barvy, scéna, statistiky. Dále se ve složce nacházejí hlavičkové soubory s rozhraními a bázovými třídami pro jednotlivé komponenty.

V ostatních složkách jsou potom hlavičkové a zdrojové soubory, které implementují jednotlivá rozhraní, vše přehledně roztříděné do složek.

Složka *src/wxgui* obsahuje zdrojové a hlavičkové soubory programu s grafickým uživatelským rozhraním.

10.3 Architektura renderovacího systému



Obrázek 30 – Architektura raytraceru

Předtím, než budou uvedeny implementační detaily, je nutné pochopit, jak mezi sebou komunikují stěžejní části renderovacího systému. Na schématu architektury na obrázku 30 jsou použity přímo názvy rozhraní, tak jak jsou potom implementovány ve zdrojových kódech.

Centrálním prvkem je *Renderer*. Ten postupuje pixel po pixelu výsledného obrázku. Souřadnice pixelu pošle objektu *Camera*, odkud následně dostane paprsek, který prošel transformací kamery.

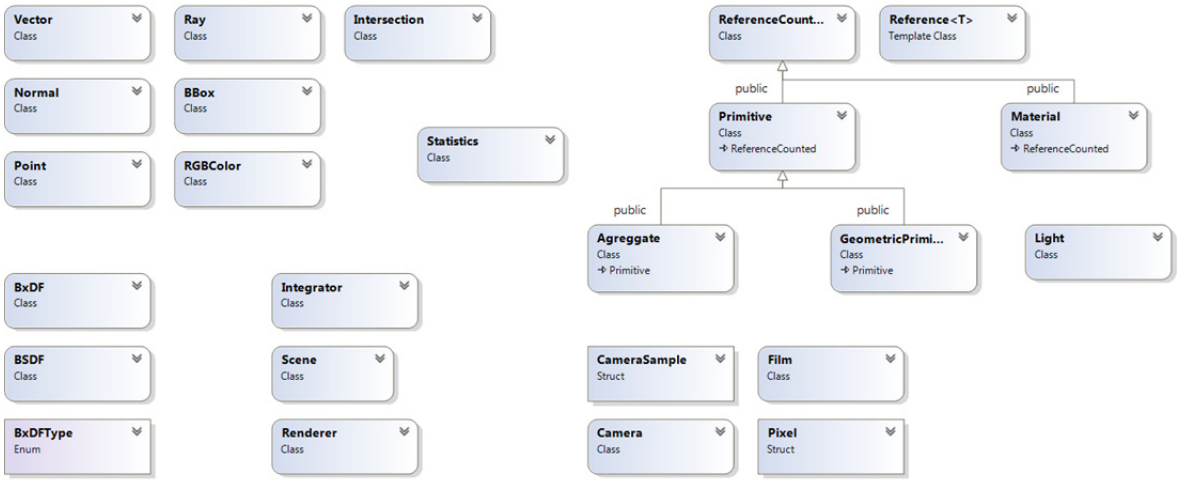
Renderer potom najde průsečík tohoto paprsku s nejbližším viditelným objektem ve scéně. Poté předá informace o průsečíku objektu *Integrator*. Ten pomocí stínovacích algoritmů vypočítá světelné příspěvky od světel ve scéně, případně, pokud podporuje, tak lomy a odrazy světla. Celkový světelný příspěvek potom odešle *Rendereru*.

Renderer ze světelných příspěvků ze vzorků v pixelu určí výslednou barvu pixelu. Tu potom pošle objektu *RenderArea*, který vykreslí pixel na obrazovku.

Toto je pouze zjednodušené shrnutí fungování, každý z těchto prvků, využívá ke svému chodu další subsystémy.

10.4 Jádro

Jádro systému obsahuje implementaci prvků, které se hojně využívají ve všech částech systému, jako vektorové prvky, barvy, a také rozhraní a bázové třídy jednotlivých částí renderovacího systému. Jeho soubory se nachází ve složce *src/core*.



Obrázek 31 – Přehled tříd a rozhraní jádra renderovacího systému

Jsou zde zahrnuty třídy *Vector*, *Point*, *Normal*. Ty obsahují data o pozici v prostoru a dále převážně přetížené operátory pro práci s nimi. Přestože je možné tyto tři typy objektů reprezentovat jednou třídou, byl zvolen tento přístup. Na první pohled je zřejmé co daná instance představuje. Je také možné definovat vazby mezi typy objektů tak, jak jsou zažité v analytické geometrii a lineární algebře.

Třída *Ray* potom reprezentuje paprsek. Ten v sobě kromě počátku a směru ještě uchovává interval $\langle tmin; tmax \rangle$, epsilon (kvůli korekci chyb při výpočtech s desetinnými čísly) a aktuální hloubku rekurze.

BBox potom reprezentuje obalovou krychli, zarovnanou s osami souřadného systému. Kromě metod pro sjednocení, průniky, je také implementována metoda pro výpočet průsečíku s paprskem.

Třída *RGBColor* reprezentuje strukturu RGB barvy. Jednotlivé barevné složky jsou reprezentovány jako desetinná čísla. Díky tomu je možné pracovat s high dynamic range obrazem. Před zobrazením na obrazovku je však třeba provést transformaci pomocí speciální mapovací funkce. V tomto systému je zahrnuto pouze mapování pomocí oříznutí na mezní hodnoty.

Třída *Intersection* slouží k hromadnému uchování informací o průsečíku paprsku s tělesem tak, aby bylo možné snadno provádět výpočty stínování, stínovacích a sekundárních paprsků. Je zde využit návrhový vzor přepravka (crate).

Třída *Scene* v sobě nese informace o scéně (světla, objekty, akcelerační strukturu, kameru) a poskytuje rozhraní pro její načtení, výpočet průsečíku atd.

Třída *Renderer* je bázová třída pro třídy, které implementují hlavní vykreslovací smyčku a ukládání výsledků na výstup.

Třída *Integrator* je rozhraní pro třídy, ve kterých je implementováno, jak bude vypočítána barevná informace z průsečíku paprsku s objektem ve scéně.

Třída *BSDF* reprezentuje sadu *BxDF* objektů aby bylo možné definovat materiál, který bude reprezentován více *BxDF* modely (jako příklad si lze představit lesklý plast s nepříliš výrazným odrazem okolí).

Třídy *Primitive*, *GeometricPrimitive* a *Aggregate* jsou bázové třídy pro objekty, které reprezentují objekty ve scéně. Zobrazitelné objekty jako jsou koule, trojúhelníková síť aj. by měly dědit ze třídy *GeometricPrimitive*, která má v sobě zahrnutou práci s referencí třída *Material*, která slouží k definování povrchu daného objektu. Třída *Aggregate* je zde pouze ze sémantických důvodů, neimplementuje nic navíc a slouží jako bázová třída pro akcelerační struktury.

Třída *Statistics* slouží k vedení statistik o počtu trojúhelníků, počtu vypočtených průsečíků. Je implementována pomocí statických atributů a metod, které jsou obaleny pomocí maker preprocesoru, aby bylo dále v kódu vidět, že se nejedná o výkonný kód.

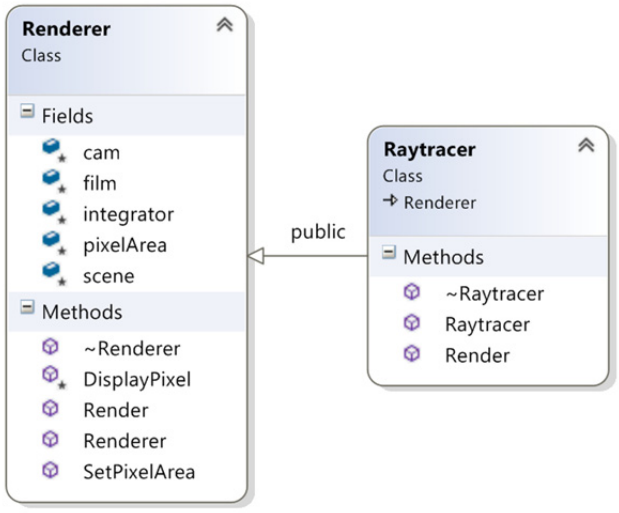
Zbytek prvků na Obrázek 31 jsou potom bázové třídy dalších prvků systému, nebo pomocné jednoduché datové struktury, které zpravidla využívají pouze jednotlivé třídy.

10.5 Vybrané komponenty systému

V této části bakalářské práce budou představeny vybrané implementované komponenty. Z ukázek zdrojových kódu byly pro lepší čitelnost odstraněny dokumentační komentáře.

10.5.1 Raytracer

Hlavní vykreslovací smyčka je implementována v souboru *renderers/raytracer.cpp(.h)*.



Obrázek 32 – Diagram třídy *Raytracer*

Třída implementuje vykreslovací smyčku tak, že výpočet probíhá pixel po pixelu výsledného obrázku. V každém pixelu jsou potom rozmístěny vzorky, ze kterých jsou potom vrženy paprsky do scény. Tam proběhne výpočet, zdali mají nějaký průsečík s tělesem ve scéně. Pokud ano tak se informace o průsečíku odešlou objektu Integrator, který provede

výpočet výsledné barvy vzorku. Pokud paprsek nemá žádný průsečík, bude vzorku přidělena barva pozadí. Nakonec se barva vzorků zprůměruje a barva se přiřadí pixelu.

```
void Raytracer::Render() const {
    #pragma omp parallel for schedule(guided)
    for (int r = 0; r < film->height; r++) {
        for (int c = 0; c < film->width; c++) {
            RGBColor color;
            CameraSample sample;
            Ray ray;

            int n = 3;
            float diff = 1.f / n;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    sample.x = c + i * diff;
                    sample.y = r + j * diff;
                    cam->GenerateRay(sample, &ray);

                    Intersection inter;
                    if (scene->Intersect(ray, inter)) {
                        color += integrator->L(ray, *scene, inter);
                    } else {
                        color += scene->background;
                    }
                }
            }
            color /= (n*n);

            #pragma omp critical
            {
                DisplayPixel(c, r, color);
            }
        }
    }
}
```

Ve zdrojovém kódu je možné vidět použití OpenMP pro paralelizaci vykreslovací smyčky. Pro práci s OpenMP se využívá direktiv preprocesoru *#pragma omp* a možné použít také několik pomocných funkcí. V ukázce je vidět jejich použití pro paralelizaci cyklu for a pro řešení kritické sekce kolem metody *DisplayPixel*.

10.5.2 Integrator podle T. Whitteda

Integrator je komponenta renderovacího systému, která zajistí správné vypočítání a složení jednotlivých komponent, ze kterých výsledná barva. Tento integrátor dokáže skládat barvu z ambientní, difúzní, zrcadlové složky a potom se složek, které vzniknou výpočtem sekundárních paprsků.

```
RGBColor WhittedTracer::L(const Ray& ray, const Scene& scene,
                          Intersection& inter) const
{
    if (ray.depth == maxDepth)
        return scene.background;

    RGBColor l;
    BSDF* bsdf = inter.material->GetBSDF(inter.normal, -inter.ray.d);
    for (uint32_t i = 0; i < bsdf->NumComponents(); ++i) {
        const BxDF* bxdf = bsdf->operator[](i);
```

```
        if (bxdf->MatchesType(BSDF_AMBIENT)) {
            l += bxdf->F(scene.ambient->GetDirection(inter),
                 inter.ray.d, inter.normal) * scene.ambient->L(inter);
        } else if (bxdf->MatchesType(BSDF_DIFFUSE)
                    || bxdf->MatchesType(BSDF_SPECULAR)) {
            for (uint32_t i = 0; i < scene.lights.size(); i++) {
                Light* light = scene.lights[i];
                Vector shDir = light->GetDirection(inter);
                STATS_ADD_SHADOW_RAY();

                Ray shadowRay(inter.hitPoint + inter.normal *
                              inter.ray.rayEpsilon, shDir, 0.f,
                              INFINITY, inter.ray.rayEpsilon);
                if (!scene.IntersectP(shadowRay)) {
                    float ndotwi = Dot(inter.normal, shDir);
                    if (ndotwi > 0.f)
                        l += bxdf->F(shDir, -inter.ray.d, inter.normal)
                             * light->L(inter) * ndotwi;
                }
            }
        } else if (bxdf->MatchesType(BSDF_REFLECTION)
                    || bxdf->MatchesType(BSDF_TRANSMISSION)) {
            Vector wo;
            const Vector wi = inter.ray.d;
            RGBColor fL = bxdf->SampleF(wi, wo, inter.normal);

            Intersection newInter;
            Ray newRay(inter.hitPoint + inter.normal *
                       inter.ray.rayEpsilon, wo, 0.0f, INFINITY,
                       EPSILON, ray.depth + 1);
            if (scene.Intersect(newRay, newInter))
                l += fL * L(newRay, scene, newInter);
            else
                l += fL * scene.background;
        }
    }
    delete bsdf;
    return l;
}
```

Je zde vidět, že každý typ BRDF se zpracovává jinak. BRDF ambientního typu se zpracovává tak, že jen vynásobíme výstup BRDF funkce s intenzitou okolního světla. Difúzní a zrcadlová BRDF se vypočítají podle zobrazovací rovnice (1.1), takže se tam počítá s útlumem světla v závislosti na úhlu mezi směrem přicházejícího světla a normálou. BRDF funkce, které popisují lom a odraz světla, využívají metodu SampleF pro vytvoření nového paprsku, který je potom vrhnut do scény a následně rekurzivně vypočítána jeho hodnota.

Tento integrátor zajišťuje právě onu rekurzi celého algoritmu. Tučně zvýrazněná místa zdrojového kódu zajišťují, aby nevznikaly při stínování trojúhelníkových sítích s interpolovanou normálou artefakty. Princip spočívá v tom, že vypočítaná normála neodpovídá normále skutečného trojúhelníku. Může se tak stát, že pro stínovací algoritmus je daný bod v polostínu, kdežto pro stínové paprsky na světle nebo ve stínu. Na tomto přechodu pak vznikají zuby, které kopírují geometrii tělesa. Tento artefakt lze v rozumné míře odstranit tak, že bod, ze kterého vychází nový paprsek, posuneme ve směru vypočítané normály o velmi malou vzdálenost. V implementaci bylo využito hodnoty epsilon, které slouží pro eliminaci chyby při výpočtu s desetinnou čárkou. Tato hodnota se dynamicky vypočítává v závislosti na vzdálenosti průsečíku od kamery.

10.5.3 Čítač referencí

Třída *Reference<T>* je šablonová třída, která implementuje nejjednodušší garbage collector v podobě čítače referencí. Jako parametr konstruktoru přijímá ukazatel na objekt, který je potomkem třídy *ReferenceCounted*. Dále má třída přetížené operátory přířazení, porovnání, reference, dereference a kopírovací konstruktor. Zásadní nevýhodou tohoto řešení je nemožnost vytvořit křížové reference mezi dvěma objekty, je tedy si při používání na toto brát ohled, jinak se čítač referencí může chovat nekorektně. Pro účely tohoto systému jsou však vlastnosti čítače referencí dostatečné.

Samotný čítač referencí bylo nutné upravit tak, aby bylo možné jej využívat při paralelním zpracování. Úprava spočívá v tom, že operace *zvětši počet referencí o jednu* a *sniž počet referencí o jednu* je třeba provádět atomicky. Pokud se tak neděje, může dojít ke snížení čítače na nulu vymazání paměti v jednom vlákně i v případě, že jiné vlákno právě zvyšuje čítač. To je možné řešit pomocí speciálních funkcí překladače, které jsou však závislé na platformě a překladači. Pro zajištění přenositelnosti programu by bylo třeba zajistit volání správných funkcí například pomocí direktiv preprocesoru. V přiložené implementaci je využito direktivy knihovny OpenMP *#pragma omp atomic* (viz ukázka níže).

```
Reference(T *_ptr = NULL)
    : ptr(_ptr)
{
    if (ptr) {
        #pragma omp atomic
        ptr->count++;
    }
}
```

Celý kód je poměrně dlouhý, v případě zájmu je doporučeno studium bohatě komentovaného zdrojového kódu.

10.5.4 TriangleMesh a Triangle

Tyto třídy se nacházejí v souboru *shapes/trianglemesh.cpp(.h)*. Tyto struktury jsou implementovány tak jak je uvedeno v kapitolách 0 a 6.3.

```
struct Vertex {
    int p; //Index of point
    int n; //Index of normal
};

class TriangleMesh: public GeometricPrimitive {
public:
    TriangleMesh(int nf, int nv, int nn, const Vertex *topo, Point *P,
                 Normal *N, Reference<Material>& mat);

    TriangleMesh(int nf, vector<Vertex>& topo, vector<Point>& P,
                 vector<Normal>& N, Reference<Material>& mat);
    virtual ~TriangleMesh();
    bool CanIntersect() const;
    virtual BBox Bounds() const;
    virtual void Refine(vector<Reference<Primitive> > &refined);
    friend class Triangle;
```

```
private:
    int nfaces, nverts, nnorms;
    Vertex *topology;
    Point *p;
    Normal *n;
};

class Triangle: public GeometricPrimitive {
public:
    Triangle(TriangleMesh* m, int n);
    virtual ~Triangle();
    virtual bool Intersect(const Ray& ray, Intersection& sr);
    virtual bool IntersectP(const Ray& ray);
    virtual BBox Bounds() const;
private:
    Normal InterpolateNormal(const float beta, const float gamma);

private:
    TriangleMesh* mesh;
    Vertex *v;
};
```

Při načítání modelu je vytvořen jen objekt typu *TriangleMesh*. S ním ovšem nelze vypočítat průsečíky s paprsky. Před prováděním tohoto výpočtu, nebo před přidáním do akcelerační struktury je nejprve třeba vytvořit objekty typu *Triangle*, které je již možné použít k výpočtům. Pro třídu *TriangleMesh* je třída Triangle označena jako přátelská a tudíž je z ní možné přistupovat k atributům instancí *TriangleMesh*. K definování trojúhelníku potom stačí jen dva ukazatele, první na instanci třídy *TriangleMesh* a druhý je ukazatel na strukturu *Vertex* z bufferu topologie. Další informace je potom možné získat pomocí ukazatelové aritmetiky. Na první pohled se řešení zdá nepřehledné, ale na druhou tímto řešením lze dosáhnout podstatné úspory paměti tím, že je možné sdílet mezi jednotlivými trojúhelníky data o společných vrcholech.

10.5.5 Ostatní

Podrobnosti o implementaci ostatních tříd a jejich metod a atributů jsou k dispozici na přiloženém médiu komentované zdrojové kódy. Také je dodána vývojářská dokumentace vygenerovaná ze zdrojových kódů pomocí nástroje Doxygen.

10.6 Zadávání vstupních dat

Vstupní data se v programu zadávají pomocí XML souboru, který popisuje scénu. Samotné 3D modely se potom importují z formátu OBJ, který lze exportovat z většiny aplikací pro tvorbu trojrozměrných modelů. Pro úspěšné načtení OBJ souboru je třeba, aby v něm uložené plochy byly trojúhelníky (většina programů umožňuj triangulovat plochy při exportu) a aby v něm byly uloženy informace o normálách v bodech (opět má většina programů tuto volbu). OBJ soubory pak musí být ve stejné složce jako XML soubor popisující scénu.

K parsování XML souboru bylo využito tříd z knihovny *wxWidgets*. Jejich popis je však mimo rozsah bakalářské práce, proto je doporučeno ke studiu problematiky využít dokumentaci k této knihovně, kterou lze nalézt na webových stránkách http://docs.wxwidgets.org/. Třídy, které jsou využity pro import těchto souborů jsou potom v souborech *import/xmlsceneimporter.cpp(.h)* a *import/objimporter.cpp(.h)*.

Ukázku možností zápisu scény v XML dokumentu naleznete v příloze A.

10.7 Příloha na médiu

Na přiloženém médiu se nacházejí zdrojové kódy a zároveň projekty pro prostředí MSVC 2012 a Eclipse. Ke zdrojovým kódům není přiložena kvůli velké velikosti knihovna *wxWidgets*. Tu je nutné pro překlad projektů dodatečně stáhnout a zkompilovat a v projektech nastavit správnou cestu k hlavičkovým souborům knihovny a v případě MSVC nastavit také složku se soubory **.lib*. V případě kompilace v operačním systému Linux je třeba mít nainstalovaný vývojářský balíček. Linkování knihoven se potom provede pomocí parametru. Na stránkách http://wiki.wxwidgets.org/Main_Page jsou návody, jak knihovnu přeložit a také jak nastavit různá vývojová prostředí.

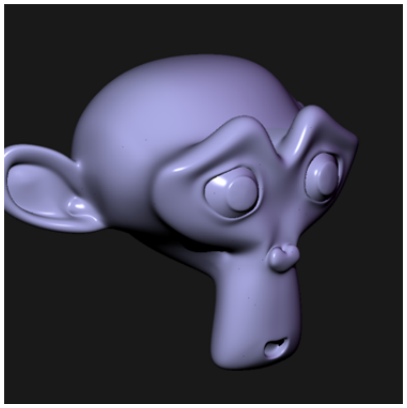
11 Měření a porovnání s jinými řešeními

V této části bakalářské práce jsou prezentována měření a porovnání některých vlastností implementovaného renderovacího systému a porovnání s některými ze stávajících dostupných řešení.

Měření byla provedena na sestavě:

- AMD Phenom II N870 2,3GHz (3 jádra),
- 4096 MB RAM,
- operační systém Microsoft Windows 7 64-bit,
- 32-bit sestavení programu s podporou SSE2 instrukcí,
- překlad proveden pomocí překladače v MSVC 2012.

11.1 Porovnání času vykreslení za použití akcelerační struktury



Obrázek 33 – Testovací model opice

Toto měření porovnává na stejné scéně čas potřebný k vykreslení. V prvním případě je výpočet proveden hrubou silou, při výpočtu průsečíku se kontrolují všechny trojúhelníky ve scéně. Ve druhém případě je použita akcelerační struktura mřížky popsaná v kapitole 8.1. Scéna byla vykreslována jedním vláknem, při jednom vzorku na pixel.

Tabulka 1 – Porovnání časů vykreslení bez použití akcelerační struktury a za použití mřížky

Model	Výpočet hrubou silou [HH:MM:SS]	Mřížka [HH:MM:SS]
Low-poly opice (3 936 troj.)	00:05:15	00:00:04
Hi-poly opice (1 007 616 troj.)	cca 14 hodin	00:00:13

Toto měření dokazuje tvrzení uvedené v kapitole 8, že akcelerační struktura dokáže zkrátit řádově čas potřebný k vykreslení scény.

11.2 Vliv počtu jader na čas vykreslení

Toto měření bylo prováděno na scéně, kterou můžete vidět v Příloze B na Obrázek 36 – Busta skřeta v zrcadle. Tato scéna obsahuje dvě světla, jedno hlavní, druhé vedlejší. Dála pak model skřeta, model podstavce, a model zrcadla s aplikovanými materiály. Bylo použito 4 vzorků na jeden pixel při rozlišení 1024 na 800 pixelů. Toto poměrně náročné nastavení bylo zvoleno, aby byly na první pohled viditelné rozdíly.

Tabulka 2 – Porovnání časů v závislosti na počtu vláken

Počet vláken	Čas vykreslování [HH:MM:SS:MS]	Zrychlení proti 1 vláknu [%]
1	00:05:30:957	-
2	00:03:39:434	44%
3	00:02:57:712	57%
6	00:02:52:783	58%

Je vidět, čas neklesá úměrně s počtem vláken, které vykonávají výpočet. Je to dáno tím, že pokud je použito pouze jedno knihovna OpenMP neřeší kritické sekce a atomické sčítání. Čím je počet vláken větší, tím jsou tyto rozdíly patrnější. Počet tří vláken by měl být asi nejvhodnější pro danou sestavu. Šest vláken bylo měřeno jen jako pokus, výsledek je však poměrně zajímavý, autor práce očekával, že výsledný čas začne opět narůstat, protože musí být provedeno více context-switchů.

11.3 Porovnání s ostatními řešeními

Pro porovnání byly vybrány renderery mental ray a V-Ray v prostředí programu Autodesk Maya. Měření probíhalo na naprosto identických scénách se stejným nastavením parametrů. Vzhledem k tomu, že tyto renderery neimplementují akcelerační strukturu mřížky, byla u nich použita výchozí akcelerační mřížka založená na BSP stromech. Kvalitativní porovnání nemá smysl dělat, výsledné obrázky jsou téměř identické, liší se jen mírně odlišným odstínem barev, protože Maya pracuje odlišně s nastavením intenzity světla.

Tabulka 3 – Porovnání výkonnosti s existujícími řešeními

	Výsledné časy [HH:MM:SS:MS]		
Model	**V-Ray**	**mental ray**	**vlastní řešení**
Tars Tarkas (914 617 troj.)	00:02:14	00:03:14	00:04:17
Predator (986 719 troj.)	00:01:55	00:04:11	00:04:24

Výsledné obrázky je možné vidět v Příloze B, na Obrázek 37 a Obrázek 38. Obě scény obsahují stejnou soustavu světel. Jedno hlavní světlo, a potom soustavu světel s malou intenzitou rozmístěnou kolem modelu, které slouží k navození dojmu, že je použito globální osvětlení. Model Tars Tarkase používá materiál, který vychází z Phongova osvětlení, a který používá sekundární paprsky k výpočtu odrazu světla. Model Predátora používá matný materiál, který vychází z Lambertova zákona.

Z měření vychází absolutně nejlépe renderer V-Ray, který lze v dnešní době označit za jedno z nejvýkonnějších řešení dnešní doby. Největší rozdíl přineslo nejspíše v použití

různých akceleračních struktur. U V-Ray také používá jiné techniky k vystřelování stínových paprsků, protože dle zápisů v logu byla vystřelena pouze jedna čtvrtina stínových paprsků oproti ostatním dvěma řešením.

Závěr

Zadání bakalářské práce bylo splněno, především potom v implementační části jsem zadání rozšířil, například podporou více vláken.

V bakalářské práci byl implementován renderovací systém, který staví na myšlence o ray-tracingu pana T. Whitteda. Zároveň byl připraven základ v podobě pomocných struktur, které umožňují použití také s jinými algoritmy pro vykreslování trojrozměrných scén.

V bakalářské práci byla vytvořena správa paměti v podobě počítání referencí. Po implementování paralelního zpracování bylo nutné tento jednoduchý garbage collector upravit pro podporu více vláken.

Spokojen jsem také s návrhem řešení materiálů a BRDF funkcí. Tento návrh dokáže skládat materiály z jednotlivých BRDF a tak je při implementaci nového typu materiálu je možné použít již implementované BRDF.

Při porovnávání s již existujícími řešeními byla sice moje implementace nejpomalejší, Nicméně se nejednalo o řádové rozdíly ale spíše o drobnou ztrátu. Jako hlavní nevýhody oproti existujícím řešením vidím slabší akcelerační strukturu a slabiny ve vrhání sekundárních a stínových paprsků. Zde se silným ukázal být především V-Ray.

V průběhu tvorby bakalářské práce jsem velice rozšířil své znalosti nejen v oblasti trojrozměrné počítačové grafiky, ale také počítačové grafiky obecně, fyziky, a uplatnil znalosti získané v matematice a lineární algebře. Dále jsem také rozšířil své znalosti jazyka C++ a programování obecně.

Časem bych se rád pokusil o implementaci pokročilejší metody zobrazování jako je path tracing nebo photon mapping, které umožňují simulovat globální osvětlení. Tyto metody by již stály za to pokusit se je implementovat na grafické kartě vzhledem k množství na sobě nezávislých výpočtů.

Při studiu materiálů mě také zaujali možnosti využití SIMD instrukcí pro průchody speciálně navrženými akceleračními strukturami. Kvalitní a rychlá akcelerační struktura je klíčem k rychlému vykreslení při použití výše zmíněných metod, které pracují s globálním osvětlením.

Pro snadnější a interaktivní vytváření scény bych systém v budoucnu velice rád provázal s některým modelovacím programem jako je Blender nebo Autodesk Maya pomocí zásuvného modulu. Potom by došlo k nepotřebě vytvořeného datového formátu XML pro popis scény, který není příliš uživatelsky přívětivý.

Literatura

[1] KAJIYA, James T. The rendering equation. *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86*. New York, New York, USA: ACM Press, 1986, s. 143-150. DOI: 10.1145/15922.15902.

[2] Lagrangeova interpolace. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2014 [cit. 2014-04-18]. Dostupné z: http://cs.wikipedia.org/wiki/Lagrangeova_interpolace

[3] Newtonova interpolace. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2014 [cit. 2014-04-18]. Dostupné z: http://cs.wikipedia.org/wiki/Newtonova_interpolace

[4] ŽÁRA, Jiří, Bedřich BENEŠ a Petr FELKEL. *Moderní počítačová grafika*. Vyd. 1. Praha: Computer Press, 1998, xvi, 448 s. ISBN 80-722-6049-9.

[5] WEISSTEIN, Eric W. Barycentric Coordinates. *MathWorld: A Wolfram Web Resource* [online]. 2014 [cit. 2014-04-25]. Dostupné z: http://mathworld.wolfram.com/BarycentricCoordinates.html

[6] WEISSTEIN, Eric W. Simplex. *MathWorld: A Wolfram Web Resource* [online]. 2014 [cit. 2014-04-25]. Dostupné z: http://mathworld.wolfram.com/Simplex.html

[7] KREMPASKÝ, Julius. *Fyzika*. Bratislava: Alfa, 1987.

[8] SCHLICK, Christophe. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*. 1994, vol. 13, issue 3, s. 233-246. DOI: 10.1111/1467-8659.1330233.

[9] Sampling (signal processing). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-04-18]. Dostupné z: http://en.wikipedia.org/wiki/Sampling_(signal_processing)

[10] SUFFERN, Kevin G. *Ray tracing from the ground up*. Wellesley, Mass.: A K Peters, 2007, xxi, 762 p. ISBN 15-688-1272-8.

[11] WEISSTEIN, Eric W. Moiré Pattern. *MathWorld: A Wolfram Web Resource* [online]. 2014 [cit. 2014-04-25]. Dostupné z: http://mathworld.wolfram.com/MoirePattern.html

[12] ŽÁRA, Jiří. *Počítačová grafika principy a algoritmy: principy a algoritmy*. 1. vyd. Praha: Grada, 1992, 446 s., [8] s. barev. il. ISBN 80-856-2300-5.

[13] SHIRLEY, Peter. *Realistic ray tracing*. Vyd. 1. Massachusetts: Natick, 2003, 225 s. ISBN 15-688-1198-5.

[14] MÖLLER, Tomas a Ben TRUMBORE. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*. 1997, vol. 2, issue 1, s. 21-28. DOI: 10.1080/10867651.1997.10487468.

[15] PHARR, Matt a Greg HUMPHREYS. *Physically based rendering: from theory to implementation*. 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2010, xxvii, 1167 p. ISBN 01-237-5079-2.

[16] PHONG, Bui Tuong. Illumination for computer generated pictures. *Communications of the ACM*. vol. 18, issue 6, s. 311-317. DOI: 10.1145/360825.360839.

[17] SIEWERT, Sam. Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms. *Intel: Developer Zone* [online]. 2009 [cit. 2014-04-25]. Dostupné z: https://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms

[18] SHIMER, Carl. Binary Space Partition Trees. WORCHESTER POLYTECHNIC INSTITUTE. *Advanced Topics in Computer Graphics* [online]. 2005 [cit. 2014-04-25]. Dostupné z: http://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html

[19] REKTORYS, Karel. *Přehled užité matematiky*. Praha: SNTL, 1973.

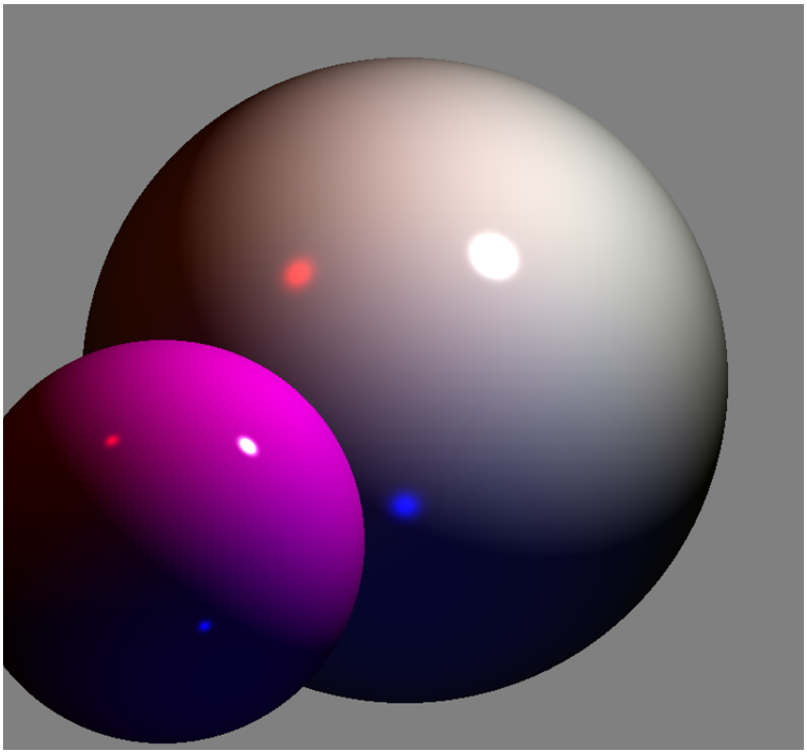
[20] DE GREVE, Bram. Reflections and Refractions in Ray Tracing. 2006. Dostupné z:http://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf

Příloha A – formát vstupních dat

```
<?xml version="1.0" encoding="UTF-8" ?>

<scene> <!-- kořenový element -->
  <background r="0.1" g="0.1" b="0.1" /> <!-- pozadí a jeho barva -->
  <ambient r="1.0" g="1.0" b="1.0"
           intensity="0.1" /> <!-- ambientní světlo a jeho barva -->

  <lights> <!-- element, který obaluje světla -->
    <point intensity="4.0"> <!-- bodové světlo a jeho intezita -->
      <color r="1.0" g="1.0" b="1.0" /> <!-- barva světla -->
      <position x="-10.0" y="7.5" z="10.0" /> <!-- jeho pozice -->
    </point>

    <point intensity="0.4">
      <color r="0.8" g="0.0" b="0.8" />
      <position x="10.0" y="5.5" z="5.0" />
    </point>
  </lights>

  <!-- akcelerační struktura, typy mohou být grid a brute -->
  <aggregate type="grid" />

  <objects> <!-- element, který obaluje modely ve scéně -->
    <!-- načtení modelu s názvem demon.obj ze stejného adresáře -->
    <triangle_mesh src="demon.obj">
      <!-- materiál phong s koeficienty složek a exponentem -->
      <material type="phong" c_ambient="0.1" c_diffuse="0.9" exp="10.0">
          <!-- barva povrchu -->
          <base_color r="0.8" g="0.0" b="0.8" />
          <!-- barva zrcadlové složky -->
          <specular_color r="0.5" g="0.5" b="0.5" />
      </material>
    </triangle_mesh>

    <triangle_mesh src="plane.obj">
      <!-- matný materiál -->
      <material type="matte" c_ambient="0.1" c_diffuse="0.9">
        <base_color r="0.8" g="0.0" b="0.8" />
      </material>
    </triangle_mesh>

    <triangle_mesh src="cube.obj">
      <!-- odrazivý materiál, vychází z phongova -->
      <material type="reflective" c_ambient="0.1" c_diffuse="0.9"
                exp="10.0">
        <base_color r="0.8" g="0.0" b="0.8" />
        <specular_color r="0.5" g="0.5" b="0.5" />
        <reflective_color r="0.8" g="0.0" b="0.8" />
      </material>
    </triangle_mesh>

    <triangle_mesh src="torus.obj">
        <!-- materiál skla, jako parametr je index lomu -->
        <material type="glass" ior="1.333">
          <refraction_color r="0.0" g="0.99" b="0.0" />
        </material>
    </triangle_mesh>
  </objects>

  <!-- velikost filmu a šířka pixelu ve scéně -->
  <film width="800" height="800" size="0.008" gamma="1.0" />
```

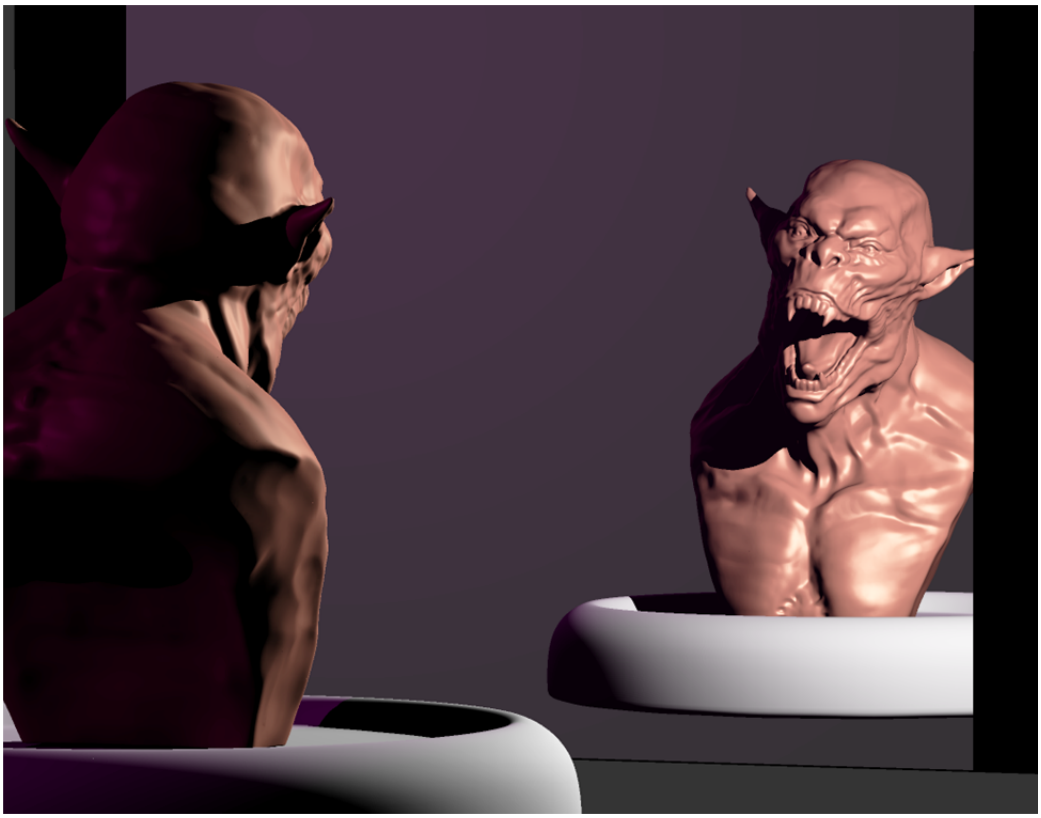
```
  <!--typ rendereru, počet vzorků na pixel, počet vláken-->
  <renderer type="raytracer" aa_samples="1" num_threads="1" />

<!-- perspektivní kamera, vzdálenost filmu od bodu eye, expozice(WIP)-->
  <camera type="perspective" d="50.0" exposure="1.0">
    <eye x="-26.0" y="3.5" z="35.0" /> <!-- bod eye -->
    <target x="0.0" y="0.0" z="0.0" /> <!-- bod target -->
    <up x="0.0" y="1.0" z="0.0" /> <!-- vektor natočení up -->
  </camera>
</scene>
```

Příloha B – výstupy

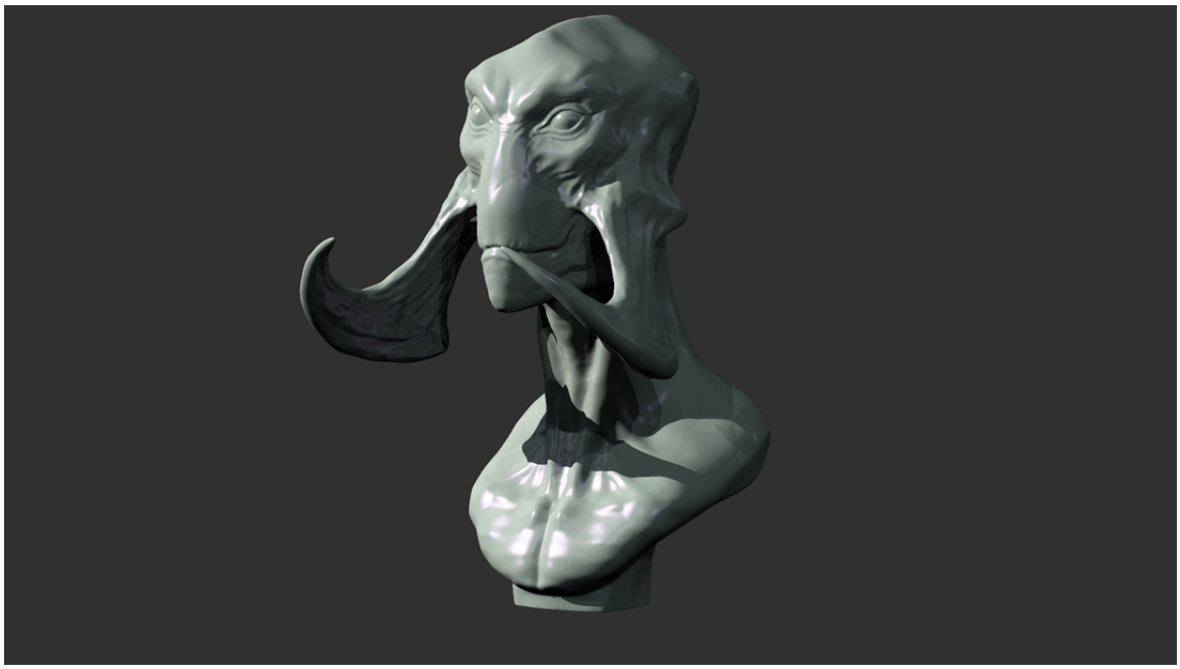


Obrázek 34 – Jedna z prvních testovacích scén

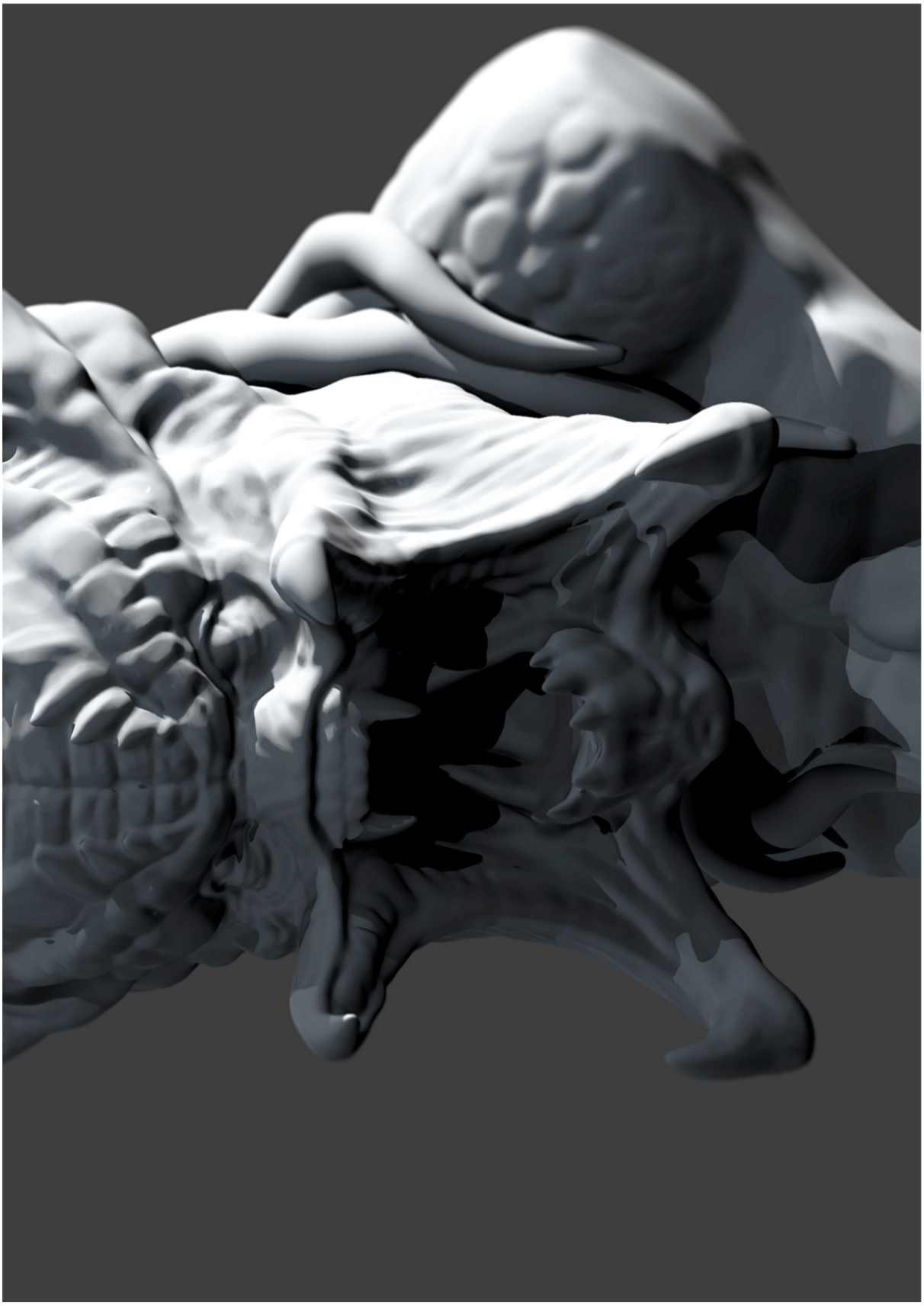
Obrázek 35 – Busta skřeta (model z turbosquid.com)



Obrázek 36 – Busta skřeta v zrcadle



Obrázek 37 – Tars Tarkas, inspirováno filmem John Carter (model z turbosquid.com)



Obrázek 38 – Predátor (model z turbosquid.com)