VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DISTRIBUTED RAY TRACING V ROZUMNÉM ČASE

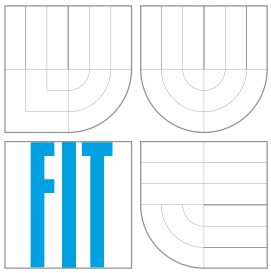
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK SLOVÁK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DISTRIBUTED RAY TRACING V ROZUMNÉM ČASE

DISTRIBUTED RAY TRACING IN REASONABLE TIME

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK SLOVÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. ADAM HEROUT, Ph.D.

BRNO 2011

Abstrakt

Práce se zabývá zobrazovací metodou distribuovaného sledování paprsku se zaměřením na optimalizace této metody. Metoda poskytuje generování velmi kvalitních a částečně realistických obrazů pomocí simulace některých vlastností světla distribucí světelných paprsků. Daní za realističnost některých efektů je i v dnešní době vysoká výpočetní náročnost. Práce rozebírá teorii a problematiku s tím spojenou. Velký prostor je pak věnovaný optimalizacím této metody jako je hledání nejbližšího průsečíku pomocí kd-stromu, kvazi náhodné generování vzorků s rychlejší konvergencí, použití instrukční sady SSE a rychlý průsečík paprsku s trojúhelníkem. Tyto optimalizace přinesly značné urychlení. V rámci práce jsou diskutované metody naimplementovány. Při implementaci se klade také důraz na praktickou použitelnost zahrnující generování pokročilejších animací a univerzální popis objektů.

Abstract

This thesis deals with the method of distributed ray tracing focusing on optimalization of this method. The method uses simulation of some attributes of light by distributing rays of lights and it produces high quality and partly realistic images. The price for realitic effects is the high computational complexity of the method. The thesis analysis the theory connected with these aspects. A large part describes optimalizations of this method, i.e. searching for the nearest triangle intersection using kd-trees, quasi random sampling with faster convergence, the use of SSE instruction set and fast ray – triangle intersection. These optimalizations brought a noticable speed – up. The thesis includes description of implementation of these techniques. The implementation itself emphasises the practical usability including generating some advanced animations and universal description of objects.

Klíčová slova

distribuované sledování paprsku, nadvzorkování, rozmazání pohybem, hloubka ostrosti, měkké stíny, matné materiály, optimalizace, kd-strom, kvazi-náhodné vzorkování, paralelní zpracování, rychlý průsečík paprsku s trojúhelníkem, SSE, animace, trojúhelníková síť

Keywords

distributed ray tracing, supersampling, mution blur, depth of field, soft shadows, matte materials, optimalization, kd-tree, quasi random sampling, parallel processing, fast ray – triangle intersection, SSE, animation, mesh

Citace

Radek Slovák: Distributed Ray Tracing v rozumném čase, diplomová práce, Brno, FIT VUT v Brně, 2011

Distributed Ray Tracing v rozumném čase

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Adama Herouta, Ph..D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Radek Slovák
1. června 2011

Poděkování

Děkuji Ing. Adamovi Heroutovi, Ph..D., vedoucímu projektu za poskytnuté cenné informace k projektu a navedení na ten správný směr. Dále děkuji Martinu Střížovi, který mi poskytl svou implementaci kd-stromu a umožnil mi tak rychlejší vývoj celé aplikace.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Metoda sledování paprsku, anglicky nazývaná ray tracing, se v počítačové grafice používá ke generování 2D zobrazení matematicky popsaných 3D scén. Je založena na zpětném "sledování" paprsků dopadajících na projekční plochu (v reálném světě např. sítnice či film fotoaparátu), díky kterým se simuluje rovinné šíření světla v prostoru.

Metoda v základní podobě je velmi zjednodušena oproti fyzikálnímu modelu a nedokáže tak ve většině případů generovat věrně působící obraz. Je to způsobeno tím že nereflektuje náhodnost a nedokonalosti reálného světa jako jsou nedokonalé povrchy objektů, plošná světla, a nedokonalosti kamer či očí. Díky tomu se pak ve výsledném obraze neprojeví nedokonalé odrazy či refrakce, měkké stíny objektů, konečná hloubka ostrosti a rozmazání scény při pohybu.

Tyto chybějící vlastnosti se pak dají simulovat pomocí rozšíření metody o distribuci náhodných paprsků a to buď v prostoru (matné povrchy, hloubka ostrosti a měkké stíny), nebo v čase (rozmazání scény). Toto rozšíření je konkrétní případ velmi obecné a při určitých parametrech na výpočet velmi náročné metody path tracing, která poměrně přesně simuluje šíření světla prostorem.

Tato rozšířená metoda, nazývaná **Distribuovaný ray tracing**, nebo též Monte Carlo ray tracing vysílá od pozorovatele ke světlu, či při odrazu světla namísto jednoho, náhodný svazek paprsků s určitým směrem a rozložením, čímž pak v důsledku částečně simuluje některá chování světla v reálném světě. Díky tomu je pak možné sledovat výše uvedené jevy. Od světla pak vysílá svazek paprsků jen v jednom zanoření, které simuluje měkké stíny. Metoda však, vzhledem ke svému omezení, nedokáže simulovat další efekty, jako jsou difrakce, kaustiky, vyzařování energie do okolí apod.

I přesto že je tato metoda dosti omezena v možnostech pokročilé simulace světla, tak je velmi náročná. Výpočetní náročnost je až tak vysoká, že jednoduchá scéna, složená pouze z malého počtu objektů, při nastavení vyšší kvality vzorkování (více paprsků simulujících danou vlastnost světla) může renderovat na dnešních procesorech i několik dnů či týdnů, což je dosti nepraktické.

Situaci, kdy požadujeme dané reálné vlastnosti v obrazu scény lze pak řešit 3 možnostmi.

Jednou z nich je pořízení speciálně upraveného hardware pro tento problém, což je velmi nákladná možnost.

Další možností je nesnažit se omezeně simulovat rovnici šíření světla přímo, ale nepřímo, např. pomocí propracovaného použití rasterizačních technik hardwarově implementovaných na 3D grafických akcelerátorech. Toto není špatná volba, která se hlavně ukazuje v nejnovějších hrách, kdy lze občas již jen těžko rozeznat hru od reality. Ovšem nikdy nám neposkytne tak do detailů fyzikálně relativně věrný obraz a k tomu vývoj a implementace

technik realistického zobrazení touto metodou jsou velmi náročné jak na čas, schopnosti, tak peníze.

Poslední možností je využít velmi přímočaré, zjednodušené simulace světla a optimalizovat ji pro vyšší rychlost renderingu. Optimalizace lze pak rozdělit do 2 kategorií.

Do 1. kategorie se řadí ty optimalizace, které zjednodušují výpočty za cenu žádné, či minimální ztráty kvality obrazu, snižují jejich složitost a zajistí, že se některé velmi často počítané úseky před počítají a pak jen již využijí jejich výsledky. Do této kategorie patří kvazi-náhodné generování čísel, zjednodušené chování závěrky kamery, kd-strom jako vyhledávací obálka scény a předpočítání poloh objektů a kamery. [18] Do 2. kategorie se řadí optimalizace samotného kódu. Sem patří použití lepšího kompilátoru, optimalizace kódu na danou hardwarovou architekturu (lepší přístup do paměti, převedení vykonávání velmi náročných matematických operací na jednodušší), a převod časově nejnáročnějších úseků programu do speciálních, výkonnějších instrukční sady (např. SSE pro práci s vektory).

Cílem diplomové práce je prostudování metody distribuovaného sledování paprsku, analýza výchozího řešení, které je mou bakalářskou prací [18]. Dále navrhnutí vhodných optimalizací, které výrazně urychlí výpočty obrázků, jejich popis a následné vygenerování poutavých scén v podobě obrázků a videa.

V následujících kapitolách vás nejdříve provedu klasickým a distribuovaným ray tracingem společně s potřebnou matematickou teorií, poté výchozí implementací a jejími vlastnostmi, pak optimalizacemi jak výpočetních, tak kódu a na závěr výsledky řešení a možnostmi do budoucna. Tato práce navazuje na semestrální projekt [19]. Přebírá část textu a rozšiřuje jak teorii, tak implementaci o další prvky.

Kapitola 2

Distribuované sledování paprsku

Tato kapitola bude představovat uvedení do problematiky sledování paprsku a použitých matematických výpočtů. Částečné čerpá z informací mé bakalářské práce [18], která se taktéž zabývá problematikou sledování paprsku, ovšem zaměřená na základní rozbor a porovnání klasické a distribuované verze a jejich implementace.

Tato práce je věnována teorii, implementaci metody distribuovaného sledování paprsku a především optimalizacím této metody.

Nejdříve vás seznámím s pojmy použitými v metodě distribuovaného sledování paprsku, pak vás provedu velmi krátkým popisem klasického i distribuovaného sledování paprsku, jejími rozdíly a rozborem použité matematické teorie.

2.1 Použité pojmy

V této části jen ve zkratce vysvětlím nutné základní pojmy a jejich význam v tomto textu:

- **Vektor** – tří prvková n-tice, směrový vektor v 3D prostoru,
- **Bod** – tří prvková n-tice, poloha v 3D prostoru, též vektor od počátku globálního souřadného systému.
- **Barva** – tří prvková n-tice, reprezentující barvu popsanou relativním RGB barevným modelem, složeným z červené, zelené a modré složky. Výsledná barva je dána aditivním smícháním těchto složek.
- **Projekční plocha** – 2D pole barev o určitém rozlišení reprezentující snímací plochu kamery.
- **Těleso (tvar)** – matematicky popsaný geometrický tvar v 3D prostoru.
- **Materiál** – množina parametrů určující barvu, matnost, odrazivost, a další parametry povrchu.
- **Objekt** – reprezentace objektu, tedy tělesa a jeho materiálu.
- **Světlo** – zdroj světelných paprsků, umístěný ve scéně. Dělí se na:
- **Bodové Světlo** – nereálné, nekonečně malé světlo definované jen polohou a barvou. Používá se v klasickém ray tracingu.
- **Plošné světlo** – reálné světlo definované navíc oproti bodovému světlu tvarem.
- **Scéna** – obsahuje soubory všech objektů a světel.
- **Kamera** – reprezentuje z reálného světa např. kameru či lidské oko. Obsahuje projekční plochu a určuje vlastnosti zobrazení projekční plochy. Definuje např. poměr stran výstupního obrazu, jeho zorné pole, vlastnosti závěrky a optiky (např. poloměr čočky). Dále je definována poloha kamery v prostoru.

2.2 Šíření světla

Pro potřeby ray tracingu budeme používat pouze geometrickou optiku, která je nejstarší částí optiky. Tato optika je založena na předpokladu, že když světlo prochází prostředím ve kterém jsou v porovnání s vlnovou délkou světla dostatečně rozměrné předměty, tak pak se vlnové vlastnosti světla projevují jen velmi málo [29].

Pro účely renderování jsou tyto vlnové vlastnosti obecně zanedbatelné, pokud se zobrazení nezaměřuje přímo na ně. Pokud se objeví potřeba efektu, způsobeného vlnovými vlastnostmi světla, lze jej simulovat pomocí jednodušších konkrétních simulačních modelů. O tom, že se v ray tracingu využívá vesměs jen geometrická optika vypovídá i název této techniky – ray tracing, neboli také sledování paprsku. Paprsek je totiž v geometrické optice důležitý pojem značící abstrakci šíření světla.

2.2.1 Paprsek

Z geometrické optiky budeme uvažovat 3 základní zákony [26], které jsou odvoditelné od Fermatova principu [22], který je dále odvozen od Huygensova principu [28].

Fermatův princip říká, že se světlo z jednoho bodu do druhého šíří tak, aby doba potřebná k přesunu po této dráze nabývala extrémní hodnoty, konkrétně minima.

Odvozené zákony jsou následující:

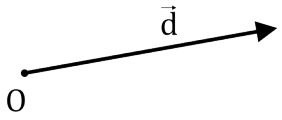
1. **Zákon přímočarého šíření světla**
   V homogenním a izotropním prostředí se světlo šíří přímočaře. Toto přímočaré šíření pak lze popsat svazkem světelných paprsků, přesněji řečeno paprsků šířících světlo.

2. **Zákon vzájemné nezávislosti**
   Pokud nebereme v potaz vlnovou optiku a zanedbáme tak ohybové jevy a interferenci světla, tak pak jsou na sobě paprsky nezávislé a neovlivňují se. Tento zákon v reálném světě z vlnových vlastností světla neplatí, ale budeme jej pro naše potřeby považovat za platný.

3. **Zákon záměnnosti chodů paprsků**
   Tento princip definuje, že pokud se paprsek šíří z jednoho bodu do bodu druhého, tak se může také šířit z druhého bodu do prvního. Tento princip se využívá na rozhraních, kdy se může změnit směr paprsků. Konkrétně odraz a lom světla na rozhraní dvou prostředí o různých hustotách (např. paprsek odražený zrcadlem zpět po stejné cestě).

Paprsek tedy budeme považovat za abstrakci šíření světla. Zároveň je základem výpočtů v ray tracingu.

Paprsek r pak definujeme jako polopřímku s počátkem v bodě O a směrem udaným jednotkovým směrovým vektorem $\vec{d}$.



Obrázek 2.1: Grafické znázornění paprsku

Matematická definice paprsku r v parametrické formě pak je:

$$r(t) = O + t\vec{d} \qquad 0 \leq t \leq \infty, \tag{2.1}$$

kde t je vzdálenost bodu $p = r(t)$ od počátku paprsku r.

Další nutnou součástí paprsku je jejich chování při dopadu na rozhraní 2 prostředí. Popis následuje v dalších podkapitolách.

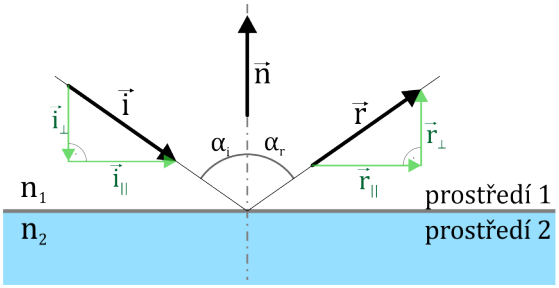
2.2.2 Chování paprsku na rozhraní dvou prostředí

Pokud paprsek dopadne na rozhraní 2 prostředí o různých hustotách, řídí se **zákonem odrazu** a **Snellovým zákonem lomu** [7]. Fyzikálně jsou tyto principy vysvětleny pomocí vlnové optiky, obecně pomocí Huygensova principu [7]. Krásná interaktivní ukázka principu tohoto zákonu je publikována Walterem Fendtem na webových stránkách [3]. Následuje popis získání odraženého, respektive lomeného paprsku v následujících podkapitolách.

2.2.3 Odraz paprsku

Pokud máme rovinné rozhraní dvou prostředí o různých indexech lomu, pak se paprsek při dopadu odrazí. Z geometrické optiky platí, že úhel dopadu paprsku α_i se rovná úhlu odraženého paprsku α_r [7]:

$$\alpha_i = \alpha_r \tag{2.2}$$



Obrázek 2.2: Odraz paprsku na rozhraní

kde $\vec{i}$ je jednotkový směrový vektor příchozího paprsku, $\vec{r}$ jednotkový směrový vektor odraženého paprsku, $\vec{n}$ jednotkový vektor normály rozhraní, $\vec{i_\perp}$ složka směrového vektoru příchozího paprsku kolmá na rozhraní, $\vec{i_\parallel}$ složka směrového vektoru příchozího paprsku rovnoběžná s rozhraním, $\vec{r_\perp}$ složka směrového vektoru odraženého paprsku kolmá na rozhraní, $\vec{r_\parallel}$ složka směrového vektoru odraženého paprsku rovnoběžná s rozhraním.

Následuje výpočet odraženého paprsku výborně popsaný Bram de Grevem v [5]. Pro obecný vektor $\vec{v}$, jeho úhel vzhledem k normále α a normálový vektor k rozhraní $\vec{n}$, potažmo také pro vektory $\vec{i}$ a $\vec{r}$ platí následující vzorce:
Složka vektoru kolmá na rozhraním:

$$\vec{v_\perp} = \frac{\vec{v} \cdot \vec{n}}{|\vec{n}|^2}\vec{n} = (\vec{v} \cdot \vec{n})\vec{n} \tag{2.3}$$

Složka vektoru rovnoběžná s rozhraním:

$$\vec{v_\parallel} = \vec{v} - \vec{v_\perp} \tag{2.4}$$

Obě tyto složky jsou na sebe automaticky kolmé a jejich skalární součin se rovná 0. Platí také Pythagorova věta:

$$|\vec{v}|^2 = |\vec{v_\parallel}|^2 - |\vec{v_\perp}|^2 \tag{2.5}$$

Z trigonometrických funkcí a vzhledem k tomu že $\vec{v}$ je jednotkový vektor platí:

$$cos\alpha = \frac{|\vec{v_\perp}|}{|\vec{v}|} = |\vec{v_\perp}| \tag{2.6}$$

$$sin\alpha = \frac{|\vec{v_\parallel}|}{|\vec{v}|} = |\vec{v_\parallel}| \tag{2.7}$$

Platí rovnoběžnost všech složek kolmých k rozhraní resp. rovnoběžných s rozhraním.

$$\vec{i_\perp} \| \vec{r_\perp} \| \vec{t_\perp} \| \vec{i_\perp} \tag{2.8}$$

$$\vec{i_{\parallel}} \| \vec{r_{\parallel}} \| \vec{t_{\parallel}} \| \vec{i_{\parallel}} \tag{2.9}$$

Nyní správným dosazováním rovnic dojdeme postupně ke vzorci počítající směr odraženého paprsku (vektory $\vec{i}$, $\vec{r}$,$\vec{n}$ jsou jednotkové). Budeme přitom vycházet z faktu, že platí rovnost úhlu dopadajícího paprsku a paprsku odraženého (2.2). Po dosazení rovnic (2.6) a (2.7) do (2.2) dostáváme:

$$|\vec{v_{\perp}}| = cos\alpha_r = cos\alpha_i = |\vec{v_{\perp}}|$$
$$|\vec{v_{\parallel}}| = sin\alpha_r = sin\alpha_i = |\vec{v_{\parallel}}|$$

Pokud jsou úhly dopadu a odrazu stejné vzhledem k normále, pak lze z (2.8) a (2.9) vyvodit, že obě složky obou vektorů jsou ve vztahu:

$$\vec{r_{\perp}} = -\vec{i_{\perp}}$$
$$\vec{r_{\parallel}} = \vec{i_{\parallel}}$$

Po dosazení do rovnice paprsku a jeho složek a následném dosazení (2.3) a (2.4) dostaneme:

$$\begin{aligned}\vec{r} &= \vec{r_{\parallel}} + \vec{r_{\perp}} \\ &= \vec{i_{\parallel}} - \vec{i_{\perp}} \\ &= [\vec{i} - (\vec{i} \cdot \vec{n})\vec{n}] - (\vec{i} \cdot \vec{n})\vec{n}\end{aligned}$$

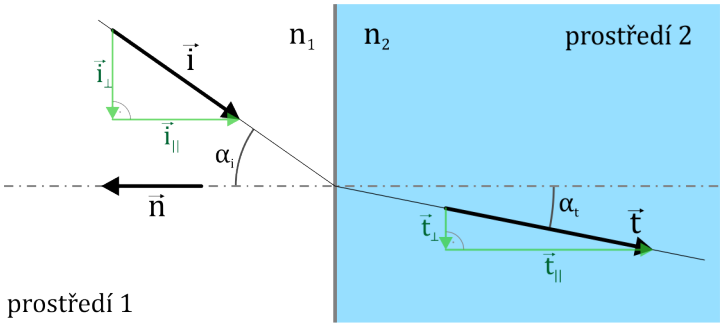
Výsledná rovnice pro odražený paprsek pak je:

$$\vec{r} = \vec{i} - 2(\vec{i} \cdot \vec{n})\vec{n} \tag{2.10}$$

2.2.4 Lom paprsku

Výpočet lomeného paprsku vychází ze Snellova zákona [31]. V závislosti na prostředí a úhlu dopadu se paprsek dále láme určitým směrem. Přičemž platí, že pokud paprsky přechází z opticky řidšího prostředí do opticky hustšího, tak se paprsek láme ke kolmici rozhraní a naopak z opticky hustšího do opticky řidšího od kolmice. Snellův zákon:

$$\frac{sin\alpha_i}{sin\alpha_t} = \frac{n_1}{n_2} \tag{2.11}$$



Obrázek 2.3: Lom paprsku

Na obrázku 2.3 značí $\vec{i}$ jednotkový směrový vektor příchozího paprsku, $\vec{t}$ jednotkový směrový vektor lomeného paprsku, $\vec{n}$ jednotkový vektor normály rozhraní, $\vec{i_\perp}$ složka směrového vektoru příchozího paprsku kolmá na rozhraní, $\vec{i_\parallel}$ složka směrového vektoru příchozího paprsku rovnoběžná s rozhraním, $\vec{t_\perp}$ složka směrového vektoru lomeného paprsku kolmá na rozhraní, $\vec{t_\parallel}$ složka směrového vektoru lomeného paprsku rovnoběžná s rozhraním.

Opět jsem čerpal publikace Bram de Grevema v [5]. Nyní uvedu nutnou podmínku pro výpočet směru nového paprsku.
Pokud platí $sin\alpha_i > \frac{n_2}{n_1}$, tak nastává situace, kdy $sin\alpha_t > 1$, což není validní výsledek. K tomuto stavu dojde ve chvíli, kdy úhel dopadu paprsku je příliš velký a zároveň výstupní prostředí o mnoho řidší než vstupní a nastane totální vnitřní odraz světla, kdy se paprsek namísto lomu odrazí. Budeme proto vycházet ze vztahu omezeného podmínkou:

$$sin\alpha_t = \frac{n_1}{n_2} sin\alpha_i \qquad sin\alpha_i \leq \frac{n_2}{n_1} \tag{2.12}$$

Nyní, když víme, že vektory $\vec{i}$, $\vec{t}$ a $\vec{n}$ jsou jednotkové, tak také víme, že velikost rovnoběžné složky s normálou $|\vec{i_\parallel}|$ (2.4) vektoru $\vec{i}$ je rovna $sin\alpha_i$. Za pomocí toho poznatku upravíme předešlou rovnici na tvar:

$$|\vec{t_\parallel}| = \frac{n_1}{n_2} |\vec{i_\parallel}|$$

Dále za pomocí výpočtu rovnoběžné složky vektoru (2.4), znalosti rovnoběžnosti vektorů (2.9) a zjednodušení výpočtu $cos\alpha$ (2.6) upravíme rovnici paralelní složky do výsledného tvaru:

$$|\vec{t_\parallel}| = \frac{n_1}{n_2} [\vec{i} - cos\alpha_i \vec{n}]$$

Nyní vypočteme kolmou složku výsledného vektoru z Pythagorovy věty (2.5):

$$\vec{t_\perp} = -\sqrt{1 - |\vec{t_\parallel}|^2} \cdot \vec{n}$$

Sečtením těchto 2 složek, úpravou rovnice a nahrazením velikosti rovnoběžného vektoru sinem úhlu získáme výsledný vztah:

$$\vec{t} = \frac{n_1}{n_2} \vec{i} - \left(\frac{n_1}{n_2} cos\alpha_i + \sqrt{1 - sin^2\alpha_t} \right) \vec{n} \qquad sin^2\alpha_t \leq 1 \tag{2.13}$$

Složky kosinu a sinu se pak vypočtou následovně:

$$cos\alpha_i = \pm \vec{i} \cdot \vec{n}, \quad sin^2\alpha_t = \left(\frac{n_1}{n_2} \right)^2 \left(1 - cos^2\alpha_i \right) \tag{2.14}$$

kde znaménko u $cos\alpha_i$ je závislé na vzájemné poloze vektorů $\vec{i}$ a $\vec{n}$
Tímto jsme elegantně rozdělili výpočet na nutnou podmínku pro další výpočet a samotný výpočet, přičemž si zároveň předpočítáváme hodnoty $cos\alpha_i$ a $sin^2\alpha_t$.

2.3 Osvětlovací model

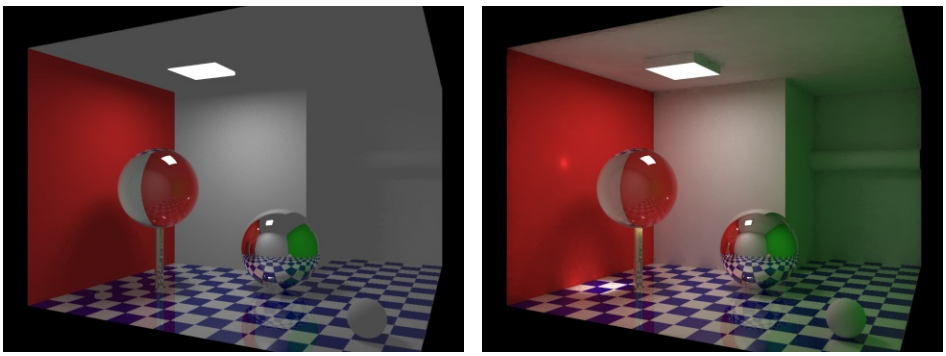
Osvětlovací model patří mezi důležité části renderování. Určuje intenzitu světla v bodech scény. Výstup osvětlovacího modelu je závislý především na poloze projekční plochy, světelných zdrojích, parametrech objektů ve scéně a scéně samotné. Modely se obecně dělí do 2 kategorií:

- lokální
- globální.

Lokální modely reflektují při výpočtu intenzity světla v daném bodu ve scéně pouze světelné zdroje, tedy přímé vyzařování a případné světelné odrazy od tohoto místa směrem k pozorovateli. Z tohoto chování lze vyčíst, že se jedná o nedokonalý osvětlovací model. Např. při renderování silně nasvícené venkovní scény se stolem může nastat situace, kdy pod tímto stolem bude dokonale černý stín, případně – při použití konstantní ambietní složky – velmi tmavá jednolitá plocha s konstantní barvou. Na druhou stranu tento typ osvětlovacího modelu má i své klady. Není tak náročný na výpočetní výkon, protože bere v potaz při výpočtu intensity světla jen velmi malou část scény – světelné zdroje (samozřejmě scéna se stále používá pro určení, zda je místo osvětlené či ne).

Naproti tomu **globální modely** při výpočtu intenzity světla berou v potaz i světlo vyzařované z okolních objektů, tedy nepřímé vyzařování. To vznikne tak, že na objekt dopadá záření z osvětlení, případně odrazem od lesklých předmětu. Toto záření objekt částečně pohltí a částečně, v závislosti na materiálu, vyzáří dále. Globální osvětlovací model je proto daleko výpočetně náročnější oproti lokálnímu modelu. Jeho přínos pro výsledný obraz spočívá především ve vyšší věrnosti osvětlení scény.

V některých případech postačí i lokální osvětlovací model. Může se jednat např. o dobře navržené osvětlení scény, případně záběr, kde místa ozářené pouze nepřímým osvětlením nejsou tolik viditelné, barvy materiálů se tolik neliší a síla osvětlení je tak velká, že rozptýlené světlo nemá takový vliv. Také distribuované efekty jako jsou matné odrazy mohou rozdíly zobrazení smazávat.



(a) Lokální osvětlovací model (b) Globální osvětlovací model

Obrázek 2.4: Rozdíly osvětlovacích modelů. Převzato v souladu s licencí ze zdroje [27].

V případě **lokálního osvětlovacího modelu** na obrázku 2.4a strop se světelným zdrojem a stěna v pravé části splývá v jednolitou plochu, což je způsobeno tím, že tyto části nejsou žádným světlem ozářeny přímo a projevuje se při zobrazení jen barva jejich konstantní ambietní složky.

Naproti tomu v případě **globálního osvětlovacího modelu** na obrázku 2.4b jsou tyto části hezky prokresleny nepřímým rozptýleným světlem. Dále lze také pozorovat, že barva zadní stěny s trubkou je ozářena zelenou barvou, což je barva, z tohoto úhlu pohledu neviditelné, stěny po pravé straně. Na zemi se také objevily kaustiky nepřímého lomeného světla procházejícího levou koulí. Rozdíly mezi těmito zobrazeními jsou poměrně výrazné, avšak pokud by koule nebyly čiré a pravá stěna neměla zelenou barvu, rozdíly v zobrazení

by byly minimální a daly by se kompenzovat přidáním pomocného zdroje světla, který by rozptýlené světlo od podlahy nahrazoval.

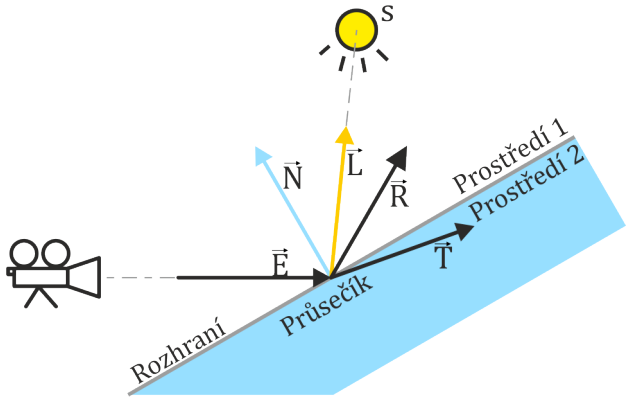
Použití lokálního osvětlovacího modelu tak může být mnohdy výhodnější protože složitost výpočtů tohoto modelu může být odhadem až řádově nižší než v případě globálního, záleží ovšem na situaci.

Dále se budeme zabývat jen lokálními osvětlovacím modelem, který je zároveň použit v této práci.

2.3.1 Phongův osvětlovací model

V této práci je jako lokální osvětlovací model použit Phongův osvětlovací model [30]. Čerpám také z [18].

Tento model byl vyvinut v roce 1973 vědcem Bui Tuong Phongem jako empirický osvětlovací model použitelný pro výpočetní použití.



Obrázek 2.5: Schéma vektorů pro výpočet Phongova lokálního osvětlovacího modelu

kde jsou všechny vektory jednotkové a mají význam:

- $\vec{E}$ – směr příchozího (dopadajícího) paprsku.
- $\vec{N}$ – směr normály, kolmé na povrch objektu v místě dopadu.
- $\vec{L}$ – směr od průsečíku ke světlu.
- $\vec{R}$ – směr odraženého paprsku.
- $\vec{T}$ – směr lomeného paprsku.

Phongův osvětlovací model se vypočítává ve chvíli kdy nějaký paprsek narazí ve scéně na překážku a je vypočten vždy pro jeden konkrétní paprsek, tedy jeho průsečík s objektem a směr tohoto příchozího paprsku. Výsledek Phongova modelu je je pak přiřazen danému příchozímu paprsku a vrácen odesílateli. Je složen z následujících součástí:

Ambientní složka – je náhražka za dopadající rozptýlené světlo z okolí. Určuje barvu materiálu, která je viditelná, pokud daný objekt není nijak osvětlen.

$$I_a = i_a k_a$$

kde I_a je výsledná barva složky, i_a koeficient síly složky a k_a značí ambientní barvu materiálu.

Difúzní složka – představuje rozptýlené světlo v daném bodě způsobené osvětlením.

$$I_d = I(\vec{L} \cdot \vec{N})k_d$$

kde I_d je výsledná barva složky, I barva světla, k_a značí difúzní barvu materiálu.

Zrcadlová složka – představuje odražené světlo přicházející ze světelného zdroje.

$$I_s = I \cdot k_s \cdot cos^{\alpha}(\varphi) = I(\vec{E} \cdot \vec{R})^{\alpha} k_s$$

kde I_s je výsledná barva složky, I barva světla, k_s značí zrcadlovou barvu materiálu, α ostrost odlesků a φ je úhel mezi $\vec{R}$ a $\vec{E}$.

Odražená resp. lomená složka – představuje světlo, které se odráží v daném bodě směrem ke odesílateli paprsku. Jedná se o rekurzivní vyslání dalších odražených resp. lomených paprsků směrem $\vec{R}$ resp. $\vec{T}$ a jejich výsledných barev I_r resp. I_t.

$$I_r = I_r \cdot k_r$$

$$I_t = I_t \cdot k_t$$

kde k_r značí koeficient s jakým se připíše k výsledné barvě barva odraženého paprsku a k_t značí koeficient s jakým se připíše k výsledné barvě barva lomeného paprsku.

Výsledný vzorec výpočtu Phongova osvětlovacího modelu pak je:

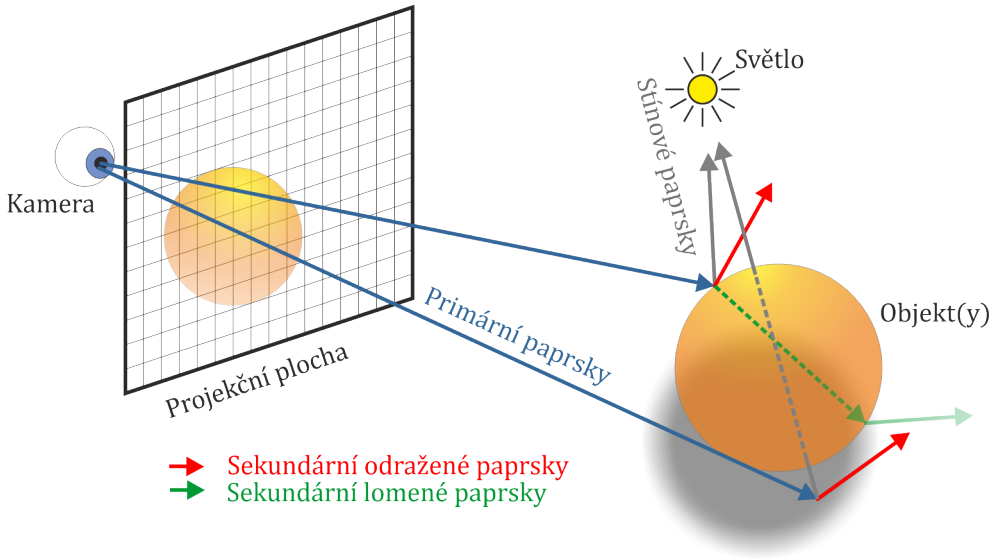
$$I = I_a + I_r + I_t \sum_{s \in S} I_d + I_s \tag{2.15}$$

kde s značí jedno konkrétní světlo z množiny světel S scény.

Všechny parametry potřebné pro výpočet osvětlovacího modelu jsou součástí materiálu který je součástí objektu.

2.4 Klasické sledování paprsku

Cílem techniky ray tracingu je získat 2D bitmapový obrazový výstup z 3D scény složené z objektů a světel pomocí simulace šíření paprsků scénou. V ray tracingu se zjednodušuje simulace šíření světla na přímočaré a je tedy omezeno na geometrickou (šíření) a jen z



Obrázek 2.6: Názorné schéma ray tracingu

části vlnovou (odrazy a lomy) optiku. Tím pádem nelze již z principu simulovat touto metodou optické jevy jako jsou disperze, ohyb. Dále se světelné zdroje omezují pouze na bodové, jinak by bylo potřeba počítat stochasticky, což se týká až distribuovaného rozšíření. Pro implementaci, se dnes používá ve většině případů pouze zpětného sledování paprsku (anglicky *back ray tracing*, v textu vždy uveden jako sledování paprsku, či ray tracing). Paprsky se vrhají zpětně z kamery přes snímací projekční plochu a sleduje se jejich šíření ve scéně. Využívá se faktu, že paprsky šířící se ze světla do projekční plochy lze zpětně spočítat a omezit se tak jen na ty paprsky které se podílí na výsledném obrazu. Díky tomu složitost výpočtu výrazně klesá.

2.4.1 Posloupnost výpočtu

Při výpočtech se paprsky rozlišují do 3 kategorií:

- **Primární paprsky** – procházející projekční plochou, mající největší vliv na zobrazení scény.
- **Sekundární paprsky** – odražené či lomené na odrazivých, či průhledných rozhraních,
- **Stínové paprsky** – používající se k výpočtu stínů, aneb pro zjištění, zda-li objekt není stíněn.

Pro každý obrazový bod scény se nejdříve z kamery přes projekční plochu vyšle primární paprsek, vypočítá se nejbližší protínaný objekt scény a daný průsečík. Zde se určí normála objektu. Pro každé světlo se vyšle 1 stínový paprsek a pokud mezi průsečíkem a světlem není žádný další objekt, tak se spočítá osvětlovací model pro průsečík. Poté se vyšlou 2 sekundární (rekurzivně odražený a lomený) paprsky, které se spočítají stejně jako primární. Jednotlivé barvy osvětlovacího modelu paprsků se váhově sečtou a výsledek se zapíše do

pixelu projekční plochy. Rekurzivní volání sekundárních paprsků je nutné omezit maximální hloubkou rekurze.

2.5 Pravděpodobnost, Statistika a numerická metoda Monte Carlo

2.5.1 Využití v distribuovaném ray tracingu

V klasickém ray tracingu se počítá jen s bodovým vzorkováním scény pomocí paprsků, kdy se pro každý pixel vyšle jen jeden do scény, při dopadu se také vyšle jen po jednom odraženém a lomeném, a při výpočtu osvětlovacího modelu pro světelné zdroje se také otestuje scéna jen na jeden stínový paprsek. Všechny tyto paprsky jsou nekonečně tenké a při dopadu na těleso tak každý z nich vzorkuje jen jeden jediný bod.

V distribuovaném ray tracingu se naproti tomu počítá s plochou. Vycházíme z faktu, že v reálném světě není nic nekonečně malé a dokonale rovné.

Např. pixel projekční plochy se již nebere jako nekonečně malý bod, ale jako čtvercová plocha. U odrazu a lomu paprsku jako abstrakce šíření světla se pak neuvažuje jen jeden dokonale odražený paprsek, ale svazek rozptýlených paprsků. Ty simulují několikrát náhodně odražené světlo v nedokonalém povrchu objektu a vycházející z průsečíku do všech míst na polokouli kolem tohoto průsečíku s určitou intenzitou v závislosti na směru dokonale odraženého resp. lomeného paprsku a nedokonalosti (matnosti) povrchu. U vzorkování světel se již počítá s tím, že neexistuje nekonečně malé bodové světlo a musí mít tedy určitou plochu se kterou se počítá. K tomu nám přibývá to že čočka kamery nemůže být nekonečně malá, protože by skrz ní neprocházelo světlo a opět nám vychází plocha, v tomto případě kruh abstrahující reálnou soustavu čoček kamery.

V distribuovaném ray tracingu se tak musíme vypořádat s problémem čím nahradit bodové zjišťování hodnoty funkce, které nám postačovalo v klasické verzi sledování paprsku. Funkce nám v tomto případě vyjadřuje výslednou barvu určitého paprsku.

Pokud chceme zjistit hodnotu určité funkce pro určitou plochu, přichází na řadu integrální počet. Integrál této funkce přes plochu nám pak vyjadřuje hodnotu všech paprsků vycházejících z určitého místa a procházející touto plochou.

Nyní nastává otázka jak vypočíst tento integrál. Pokud se zamyslíme nad průběhem funkce vyjadřující barvu paprsku ve scéně, může být a (u jakékoliv ne primitivní scény) je nespojitá. Je to dáno tím, že paprsky mohou narážet na překážky v této scéně a tak skokově měnit svou barvu. Dále je také nutno si uvědomit, že více zanořených efektů znamená také vícerozměrné integrály. Z těchto důvodů nám odpadá možnost počítat tyto integrály efektivně analyticky. Přichází na řadu numerická aproximace integrálu, konkrétně metoda Monte Carlo. K té budeme potřebovat generování náhodných vzorků na počítané ploše. Proto se také distribuovaný ray tracing občas nazývá stochastický.

Metodu Monte Carlo si popíšeme po popisu potřebných znalostí z pravděpodobnosti a statistiky. Některé principy postupů jsem čerpal z [13] a [2]. Jsou tam také dostupné podrobnější informace k této tématice.

2.5.2 Pojmy pravděpodobnosti a statistiky

Nyní si popíšeme pár potřebných znalostí z pravděpodobnosti a statistiky potřebných pro metodu Monte Carlo.

Náhodná veličina X

Je to proměnná, která může nabývat určité hodnoty x z množiny Ω. Ω označuje množinu všech hodnot, kterých může veličina X nabývat. Může se jednat o diskrétní nebo spojitou náhodnou veličinu. Pro potřeby ray tracingu bude většinou spojitá a tu budeme v následujícím textu užívat. Dále platí, že $P(\Omega) = 1$, tedy že součet pravděpodobností všech hodnot x, kterých může X nabývat je rovna jedné (u spojité veličiny to může být např $\mathbb{R}$).

Hustota rozdělení pravděpodobnosti $f(x)$

Je to funkce, která určuje rozdělení pravděpodobnosti pro jednotlivé hodnoty náhodné veličiny $x \in X$. Vždy musí platit: $\int_\Omega f(x)\,\mathrm{dx} = 1$. Pravděpodobnost že X nabývá hodnoty z intervalu $< a; b >$ pak je:

$$P\left(X \in < a; b >\right) = \int_a^b f(x)\,\mathrm{d}x. \tag{2.16}$$

Distribuční funkce $F(x)$

Distribuční funkce, pro každou reálnou hodnotu x, určuje jaká bude pravděpodobnost, že náhodná veličina X bude nabývat hodnoty menší nebo rovné x: $F(x) = P(X \leq x)$. Je neklesající a nabývá hodnot z intervalu $F(x) \in < 0; 1 >$. Distribuční funkce se vypočte z hustoty pravděpodobnosti následovně:

$$F(x) = P(X \leq c) = P\left(X \in < -\infty; x >\right) = \int_{-\infty}^{x} f(t)\,\mathrm{d}t. \tag{2.17}$$

Inverzní funkce f'

je taková, kdy vstupní parametr se stává výstupem a výstupní vstupem, např.:

$$y = f(x), \qquad x = f'(y)$$

Střední hodnota E (pro veličinu X se označuje EX)

Tato hodnota je teoretický průměr náhodné veličiny X, tedy očekávaná hodnota:

$$EX = \int_{-\infty}^{\infty} x \cdot f(x)\,\mathrm{d}x \tag{2.18}$$

Rozptyl D (pro veličinu X se označuje DX

Označuje střední hodnotu čtverce odchylky veličiny X od své střední hodnoty EX:

$$\begin{aligned} DX &= E(X - EX)^2 \\ &= E(X^2) - (EX)^2 \end{aligned} \tag{2.19}$$

Vlastnosti E a D

Předchozí vyjádření je odvozeno za pomoci následujících pravidel, která platí pro E a D:

$$E(aX - bY) = a \cdot EX - b \cdot EY$$
$$D(aX - bY) = a^2 \cdot EX - b^2 \cdot EY$$
$$E\left(\sum_i X_i\right) = \sum_i E\left(X_i\right)$$
$$D\left(\sum_i X_i\right) = \sum_i D\left(X_i\right)$$

Typy rozložení pravděpodobnosti

- **Rovnoměrné rozložení**
  U tohoto rozložení má každý ze všech možných jevů stejnou pravděpodobnost, že nastane. V ray tracingu se využívá pro distribuci paprsků na čočce, protože to odpovídá reálnému chování, kdy čočkou světlo nikde neprochází více, či méně, ale na všech místech stejné množství.

- **Normální rozložení**
  Má hustotu funkce definovanou Gaussovou křivkou. U té je obsah pod křivkou roven přesně jedné. Velmi se hodí pro simulování složitých systému, u kterých je velmi mnoho závislostí a není přesně zřejmé, která náhodná veličina určuje, že tomu bude právě takto. Hodí se např. pro distribuci odražených paprsků, protože světlo se při odrazu láme složitě povrchem a nelze přesně určit proč se lomí právě takto, ale víme jistě že to bude přibližně určitým směrem.

- **Další rozložení**
  Dále pak mohou být rozložení určené pro speciální případy. Např. rozložení pravděpodobnosti na závěrce kamery, která má tvar 5ti-úhelníku.

2.5.3 Metoda Monte Carlo

Monte Carlo je numerická (simulační) stochastická metoda, která má široké použití díky tomu, že k výpočtu potřebuje jen popis zkoumané veličiny a popis distribuční funkce s jakou tato veličina nastává. Průběh výpočtu se dá rozložit do kroků:

- Specifikace modelu řešeného problému, včetně pravděpodobnostních charakteristik, vytvoření potřebného náhodného rozložení pro simulaci. Určení vstupu a výstupu modelu.

- Mnoho jednotlivých simulací modelu s náhodným vstupem s relevantním rozložením.

- Statistické zpracování vstupů/výstupů a získání potřebné hledané hodnoty. Výsledkem této metody může být například v této práci uvažovaná střední hodnota veličiny, případně jiná charakteristika modelu.

Rychlost konvergence je daní za její univerzálnost. Metoda konverguje podle počtu vzorků u pseudo-náhodného generátoru čísel následovně:

$$O\left(\frac{1}{\sqrt{N}}\right) \tag{2.20}$$

Následuje vzorec pro použití metody Monte Carlo:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{g(X_i)}{f(X_i)} \tag{2.21}$$

kde N je počet vzorků, a $g(x)$ funkce, pro kterou zjišťujeme střední hodnotu. $f(X_i)$ je hustota pravděpodobnosti, se kterou nastává jev $g(X_i)$. Pro rovnoměrné rozložení se metoda Monte Carlo počítá následovně:

$$F_N = \frac{I_X}{N} \sum_{i=1}^{N} g(X_i) \tag{2.22}$$

kde I_x představuje interval veličiny X, dále platí, že $I_X = \frac{1}{f(x)}$, kde x je libovolné z X, protože $f(x)$ je konstantní.

Dejme tomu, že chceme zjistit střední hodnotu integrálu jednorozměrné funkce $g(x)$, tedy $\int_a^b g(x)\,\mathrm{d}x$. $X_i \in < a; b >$ je s rovnoměrným rozložením a tedy $f(X_i)$ je konstantní. Metoda Monte Carlo se pak použije následovně. Dále odvodíme z této rovnice původní rovnici rovnici integrálu, čím dokážeme, že metoda funguje:

$$\begin{aligned}
F_N &= \frac{b-a}{N} \sum_{i=1}^{N} g(X_i) \\
E[F_N] &= E\left[\frac{b-a}{N} \sum_{i=1}^{N} g(X_i)\right] \\
&= \frac{b-a}{N} \sum_{i=1}^{N} E\left[g(X_i)\right] \\
&= \frac{b-a}{N} \sum_{i=1}^{N} \int_a^b g(x) \cdot f(x)\,\mathrm{d}x \\
&= \frac{1}{N} \sum_{i=1}^{N} \int_a^b g(x)\,\mathrm{d}x \\
&= \int_a^b g(x)\,\mathrm{d}x
\end{aligned}$$

kde je nutné zmínit, že hustota $f(x)$ patřící náhodné veličině X_i musí být rovna $\frac{1}{b-a}$, protože je konstantní a zároveň $\int_a^b f(x)\,\mathrm{d}x = 1$. Jelikož se v tomto případě ve výsledku integrálu projeví konstantní hustota $f(x)$ jen jako součinitel, mohli jsme si jej dovolit v předposledním kroku výpočtu vykrátit $b - a$ s $f(x)$.

Při počítání distribuovaných efektů pak musíme spočítat integrál funkce, vyjadřující nám barvu světla. Integrovat budeme přes plochu na které potřebujeme zjistit střední hodnotu. Náhodné vzorky pak budou mít rovnoměrné rozložení.

Ještě můžeme výpočet pomocí metody Monte Carlo zjednodušit tak, že budeme používat vždy generátor náhodných čísel na intervalu $< 0; 1 >$ s tím že nám takový generátor pokrývá pomocí transformace celý prostor, který zkoumáme, např. pokud budeme vzorkovat obdélník, a vstupní náhodná veličina bude generovaná v prostoru $[0, 1]^2$ s tím, že tato

veličina vyjadřuje relativně pozici na celém obdélníku, pak hustota pravděpodobnosti pro každý prvek náhodné veličiny bude činit 1. Díky tomu pak můžeme zjednodušeně psát:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} g(X_i) \tag{2.23}$$

2.5.4 Aproximace vícedimenzionálních integrálů

Výpočty jednotlivých v sobě zanořených výpočtů integrálu pak můžeme počítat pomocí více dimenzionálního integrálu. Stačí vytvořit odpovídající více dimenzionální náhodnou veličinu společně s její hustotou a pak můžeme napsat pro výpočet 3 vnořených integrálů:

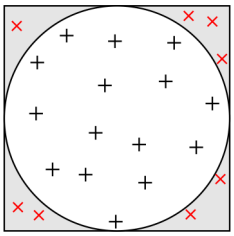
$$\int_{x_0}^{x_1} \int_{y_0}^{y_1} \int_{z_0}^{z_1} g(x, y, z) \, \mathrm{d}z \, \mathrm{d}y \, \mathrm{d}x$$

kde vzorek $X_i = (x_i, y_i, z_i)$ je generován s rovnoměrným rozložením v krychli definované dvěma rohovými body (x_0, y_0, z_0) a (x_1, y_1, z_1). Hustota $f(X_i)$ je pak konstantní, a aproximace takového integrálu pomocí metody Monte Carlo je:

$$E(g(x, y, z)) = \frac{(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)}{N} \sum_{i=1}^{N} g(X_i) \tag{2.24}$$

2.5.5 Metoda odmítnutí vzorků

Pokud se dostaneme do situace, že máme rozložení, které neodpovídá požadavkům a $P(\Omega) \neq 1$ a tvorba takového rozložení je příliš komplikovaná, případně nemožná, můžeme si pomoci metodou Odmítnutí vzorků. Jako příklad si uvedeme rovnoměrné generování vzorků například v kruhu.



Obrázek 2.7: Generování rovnoměrných vzorků v kruhu

Postup generování rovnoměrně rozmístěných vzorků v kruhu je pak následující:

- Budeme generovat vzorky ve čtverci pomocí 2 základních rovnoměrných nezávislých rozložení.
- U každého vzorku si zjistíme, jestli leží v kruhu, ve kterém chceme generovat rozložení.
- Pokud ano, vzorek použijeme.
- Pokud ne, vzorek zahodíme a generujeme další.

Takto můžeme generovat vzorky o určitém rozložení téměř pro jakýkoliv tvar. Metoda se dá úspěšně použít při generování vzorků například na čočce.

2.5.6 Importance Sampling

Importance sampling je vylepšení metody Monte Carlo. Vezměme si jako příklad, že pomocí metody Monte Carlo zjišťujeme barvu odraženého světla. K tomu použijeme rovnoměrné rozložení paprsků na polokouli. Paprsky tak mají stejný přínos pro výpočet. Ovšem jejich hodnota se zvětšujícím se úhlem od dokonale odraženého paprsku klesá podle určitého vzorce. Případný vygenerovaný paprsek, který bude blízko dokonalému, tak bude mít největší světelný přínos pro výsledek, kdežto ten který se blíží úhlu 90° téměř žádný. Přitom jejich přínos jako vzorku metody Monte Carlo je stejný.

Importance sampling je pak založen na tom, že pokud dokážeme najít přibližnou hustotu pravděpodobnosti, $p(X_i)$, která bude mít podobný průběh jako $f(X_i)$, tak metoda Monte Carlo bude konvergovat daleko rychleji. Intuitivně si popíšeme jak to funguje.

Dejme tomu, že si chceme znovu spočítat integrál nějaké funkce $\int g(x)\,\mathrm{d}x$. Pak dáme hustotu s funkcí do rovnosti s určitým parametrem c $f(x) = c \cdot g(x)$.Tento parametr se bude rovnat:

$$c = \frac{1}{\int g(x)\,\mathrm{d}x} \tag{2.25}$$

Pro takovou hustotu je třeba znát hodnotu integrálu, což je v prvé řadě ta hodnota, kterou vůbec hledáme a samozřejmě neznáme. Ovšem pro vysvětlení s tím budeme dále pracovat jako že ji známe. Pokud bychom mohli generovat s tímto rozložením vzorek, jeho hodnota by byla

$$\frac{g(X_i)}{f(X_i)} = \frac{1}{c} = \int g(x)\,\mathrm{d}x.$$

Jelikož c je v tomto případě konstanta, každý výpočet vzorku, by měl stejnou hodnotu a tím pádem rozptyl vzorků by byl 0.

Toto je samozřejmě naprostý nesmysl, protože právě hodnotu integrálu funkce $g(x)$ hledáme/neznáme a pokud bychom ji znali, bylo by zbytečné používat metodu Monte Carlo.

Ovšem tento příklad ukazuje, že pokud si zvolíme hustotu $f(x)$ podobnou funkci $g(x)$ odchylka výsledné střední hodnoty od opravdové bude nižší.

Pro podrobnější popis a vysvětlení doporučuji publikaci [14]. Z té jsem čerpal tento přístup k vysvětlení.

2.6 Efekty distribuovaného sledování paprsku

Jak jsme si již popsali v kapitole 2.5.1, tak klasický ray tracing počítá s barvou jen jednoho bodového vzorku scény. Naproti tomu metoda distribuovaného ray tracingu počítá s barvou vzorku, který není definován jedním paprskem, ale všemi paprsky a určité ploše. K získání této hodnoty tedy potřebujeme vypočíst integrál funkce barvy scény přes danou plochu. K tomu použijeme metodu Monte Carlo. Prakticky se pak výsledná barva získá zprůměrováním určitého počtu paprsků (vzorků) náhodně rozložených na dané ploše.

Výsledek takových výpočtů pak může být mírně rozostřený, na pohled "měkčí" a obecně vypadá reálněji, což je dáno také tím, že se touto metodou alespoň částečně simulují reálné vlastnosti světla. Paprsky, potažmo vzorky jsou u simulace jednotlivých jevů světla generovány pokaždé trochu jiným způsobem a budou popsány v následujících kapitolách.

Jevy, které lze distribuovaným ray tracingem simulovat jsou:

- Supersampling (každý pixel je brán jako plocha, ne jako bod).
- Motion blur (rozmazání obrazu pohybem objektů, či kamery).

- Hloubka ostrosti (anglicky depth of field, zkratkou DOF).
- Měkké stíny (anglicky soft shadows).
- Matné materiály (rozptýlené odrazy, anglicky diffuse reflection).
- Průsvitné materiály (rozptýlené refrakce, angl. diffuse refraction).

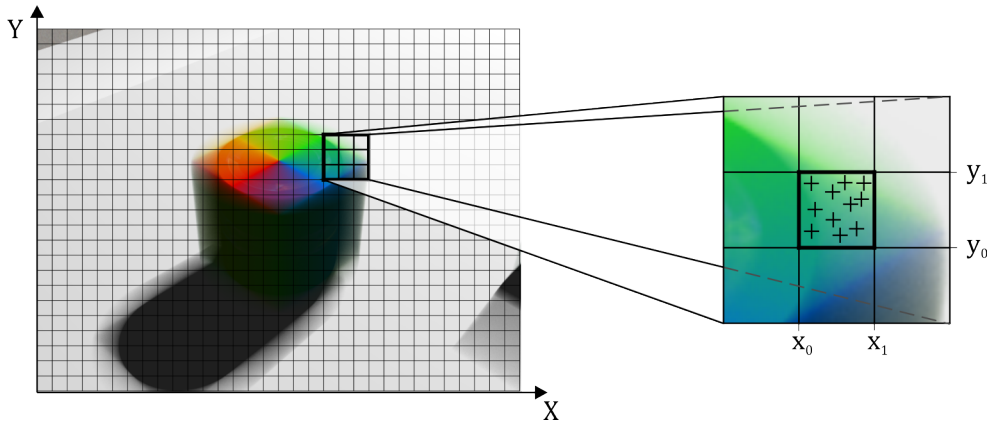
Díky těmto efektům, vypadá výsledný obraz poměrně reálně a oproti klasické metodě o mnoho lépe.

Následuje popis jednotlivý efektů distribuovaného ray tracingu.

2.6.1 Supersampling

Supersampling (česky nadvzorkování) je metoda, díky které lze z obrazu odstranit aliasing, tedy jev, kdy se na hranách objektů mohou objevit ostré přechody kopírující hrany pixelů. Tento jev může nastat v případě, že jsou pixely brány jako nekonečně malé body o nulové ploše. V takovém případě pak při vysokém kontrastu pixelů dojde k porušení Nyquistova teorému [24]. Jednotlivé pixely obrazu totiž ve skutečnosti zabírají určitou plochu.

Metoda supersamplingu pak pracuje právě na principu, že pixel je určitá plocha o jedné výsledné barvě, která je dána střední hodnotou všech vzorků na této ploše.



Obrázek 2.8: Supersampling - nadvzorkování pixelu

Řekněme, že $c(x, y)$ je funkce barvy scény pro určitou pozici na spojité projekční ploše. Pak výsledná barva pixelu C_{pixel} bude:

$$C_{pixel} = \int_{y0}^{y1} \int_{x0}^{x1} c(x, y)\, \mathrm{d}x\, \mathrm{d}y \tag{2.26}$$

Při použití metody Monte Carlo, kde budeme uvažovat rovnoměrné rozložení pravděpodobnosti to pak bude:

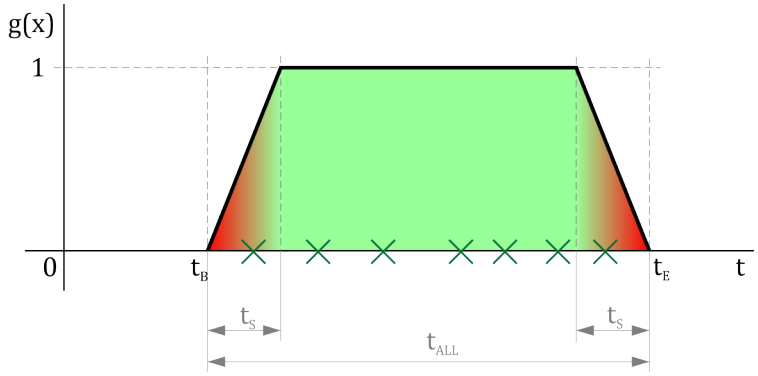
$$C_{pixel} = \frac{(y_1 - y_0) \cdot (x_1 - x_0)}{N} \sum_{i=1}^{N} c(X_i) \tag{2.27}$$

kde $X_i = (x_i, y_i)$ je vzorek generovaný náhodně s rovnoměrným rozložením v obdélníku ohraničeným přímkami rovnoběžnými s osami v místech x_0, x_1, y_0, y_1.

2.6.2 Motion Blur

Motion blur je jev, který způsobí při rychlém pohybu objektu, či kamery, rozmazání obrazu. Efekt vychází z reálných vlastností kamer, kdy je třeba určitou dobu scénu snímat s otevřenou závěrkou a umožnit tak světlu dopadat na snímací čip, popřípadě film citlivý na světlo. Během této doby se ale scéna může pohybovat a tento pohyb je na výsledný obraz zaznamenán.

Závěrka kamery se ve velmi krátkém čase otevře, poté následuje doba po kterou se snímá scéna a následně se závěrka zavírá.



Obrázek 2.9: Znázornění simulace závěrky

Kvalitně, poměrně přesně a zároveň jednoduše lze závěrku simulovat v podobě lomené úsečky o 3 segmentech. Funkce $g(t)$ popisuje průběh otevření závěrky v závislosti na čase t a zároveň tedy koeficient intensity světla pro daný čas.

Řekněme, že jíž máme vybraný pevný bod na projekční ploše a funkce $b(t)$ nám určuje barvu scény pro určitý čas. Nyní nás zajímá barva tohoto bodu pro určitý časový interval. Opět se bude jednat o výpočet integrálu, tentokrát jednorozměrného:

$$C_{blur} = \int_{t_B}^{t_E} b(t) \cdot g(t)\,\mathrm{d}t \tag{2.28}$$

Výpočet metodou Monte Carlo s rovnoměrným rozložením náhodné veličiny T na intervalu $< t_B; t_E >$ bude následovný:

$$C_{blur} = \frac{t_E - t_B}{N} \sum_{i=1}^{N} b(T_i) \cdot g(T_i) \tag{2.29}$$

Můžeme ovšem využít možnosti *Importance sampling*, kdy můžeme zjistit hustotu pravděpodobnosti $f(t)$ přímo ze znalosti průběhu otevření závěrky. Ta pak bude podobná průběhu $b(t) \cdot g(t)$ (v tomto případě totožná s $g(t)$). Samotný generátor pak můžeme sestrojit pomocí metody *Rejection sampling* kdy ze vzorků využijeme jen x-ovou souřadnici. Ovšem vzorec pro výpočet metody Monte Carlo pak musíme použít obecný, který počítá s hustotou rozložení (2.21):

$$C_{blur} = \frac{1}{N} \sum_{i=1}^{N} \frac{b(Ti)g(T_i)}{f(T_i)} \tag{2.30}$$

jelikož jsme sestrojili rozložení pravděpodobnosti s hustotou $f(t)$, která je totožná s mírou otevření závěrky $g(t)$, můžeme je vykrátit a dostaneme:

$$C_{blur} = \frac{1}{N} \sum_{i=1}^{N} b(Ti) \tag{2.31}$$

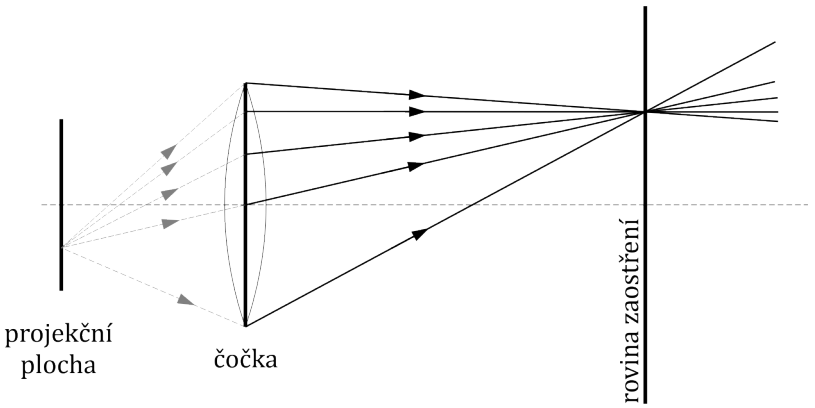
přičemž se nám rozptyl při výpočtu sníží.

2.6.3 Hloubka ostrosti (Depth Of Field)

Mějme jednoduchou kameru, která snímá obraz. Obsahuje základní 2 prvky, snímací plochu a otvor či optickou čočku, přes kterou světlo dopadá na snímací plochu. Při nekonečně malém otvoru je teoreticky (v rámci geometrické optiky) obraz zaznamenaný na snímacím čipu dokonale ostrý.

Ovšem v reálném světě není možné aby průměr čočky objektivu kamery byl nekonečně malý, protože by přes tuto čočku nemohlo procházet světlo. I pokud by čočka byla určité minimální velikosti, a procházelo by přes ní světlo, dopadalo by ho na snímací čip tak málo, že by obraz byl buď velmi nekvalitní, nebo příliš tmavý. Čím víc se průměr čočky zvětšuje, tím více dopadá na snímací čip světla, ale také se do výsledného obrazu zapojuje efekt **hloubky ostrosti**, při kterém se objekty mimo zaostřenou vzdálenost jeví jako neostré. Je to dáno tím, že se nezaostřený objekt přes čočku nepromítá přímo do roviny čipu, ale za ní, případně před ní. Takový nezaostřený bod scény se pak projevuje na čipu jako kruh.

Tento efekt může být za určitých okolností nežádoucí, ovšem většině snímků, ať již reálným, či renderovaným dodává lepší vizuální vzhled, a může působit umělecky na výsledný obraz velmi kladně. Efekt totiž nedůležité vzdálené předměty rozostřuje (rozmazává) a ty důležité, na které je zaostřena kamera, jsou pak v obraze zdůrazněné a více vyniknou jejich detaily.



Obrázek 2.10: Hloubka ostrosti

Matematicky pak budeme uvažovat tenkou čočku u několikanásobně větším poloměru, než jejím průměru. Při výpočtech pak nebudeme zjišťovat funkční hodnotu scény (barvu) pro bodový vzorek, ale hodnotu pro plochu kruhu, který reprezentuje čočku, tedy integrál funkce barvy scény přes plochu kruhu. Světelné paprsky na čočku dopadají na všech místech se stejnou pravděpodobností, proto budeme potřebovat rovnoměrné rozložení. Budeme přitom uvažovat, že již máme zvolený pevný bod projekční plochy i čas. K výpočtu

výsledné barvy jednoho vzorku čočky nám pak bude sloužit funkce $d(u, v)$, kde u, v jsou svým způsobem barycentrické souřadnice na čočce.

Výpočet výsledné barvy pak pomocí metody Monte Carlo bude následovný:

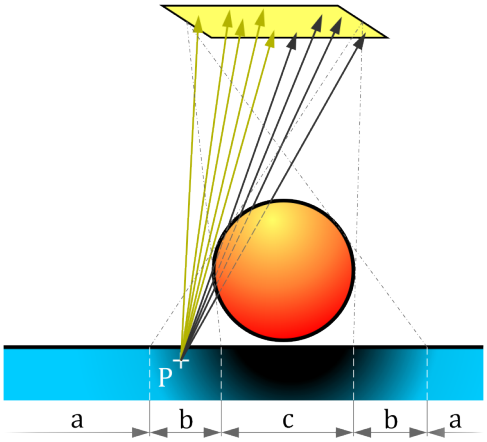
$$C_{dof} = \frac{1}{N} \sum_{i=1}^{N} d(X_i) \tag{2.32}$$

kde X je náhodná veličina z prostoru $[0, 1]^2$ a funkce d(X) rovnoměrně mapuje tento prostor na prostor kruhu a vrací výsledek pro jeden vzorek potažmo paprsek.

Další způsob je využít metody *Importance sampling* společně s *Rejection sampling* a generovat tak přímé pozice na kruhu s tím, že výsledný vzorec bude identický.

2.6.4 Měkké stíny

V klasickém ray tracingu se vyskytují pouze ostré stíny. To je dáno tím, že uvažuje jen nekonečně malá bodová světla. V reálném světě ovšem neexistuje nekonečně malé světlo a můžeme se setkat jen se světly, které mají určitý tvar a rozměr. Tento typ světel pak vrhá přes objekty měkké stíny, kdy intenzita osvětlení závisí na tom kolika procenty plochy světla je bod ozařován a kolik procent světla je ve stínu. Tato situace je zobrazena na obrázku 2.11. Oblasti a jsou ozářené celým světlem, proto jsou nejjasnější, naopak oblast c je kompletně ve stínu a je tedy velmi tmavá, kdežto oblasti b jsou ozářeny jen částečně a podle toho jaký je poměr mezi viditelnou části světla a zastíněnou pak mají intenzitu.



Obrázek 2.11: Princip měkkých stínů

Opět se jedná o výpočet integrálu funkce osvětlení $l(x)$ přes plochu světla. Metodou Monte Carlo se pak výpočet dá popsat:

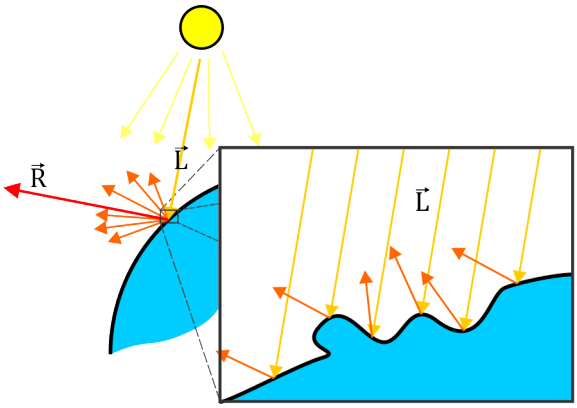
$$C_{light} = \frac{1}{N} \sum_{i=1}^{N} l(X_i) \tag{2.33}$$

kde náhodná veličina X_i rovnoměrně pokrývá prostor $[0, 1]^2$ a tento prostor je namapován na plochu světla. V případě obdélníku je toto řešení triviální, ovšem při použití kompliko-vanějšího světla může být generování rovnoměrných vzorků netriviální.

Simulovat se tento efekt v distribuovaném ray tracingu pak dá pomocí generování více paprsků s počátkem v bodě P a cílem někde na povrchu světla, s tím, že jsou tyto cíle rovnoměrně generované. Pak stačí vypočíst intenzitu světla paprsků a vypočítat z nich pomocí metody Monte Carlo průměr. Princip je velmi podobný metodě Supersamplingu.

2.6.5 Matné a průsvitné povrchy

V klasickém ray tracingu se považují všechny povrchy objektů za dokonale přesné a tedy perfektně zrcadlové. V reálném světě ovšem takové objekty nenajdeme. Některé se tomu ovšem mohou na pohled vypadat. Ovšem pod mikroskopem by tomu bylo jinak. Odražený paprsek světla se pak od povrchu odráží podle této mikrostruktury povrchu. Čím drsnější je povrch, tím více se odrazy rozptýlí. Jak by to mohlo vypadat na povrchu tělesa pak ukazuje obrázek 2.12.



Obrázek 2.12: Matné povrchy (Obrázek převzat a upraven z [18])

Stejně to funguje i s odrazem paprsku vycházejícím ze scény. Jelikož paprsek představuje abstrakci přímočarého šíření světla, lze si jej představit jako svazek rovnoběžných velmi blízkých "podpaprsků" . Každý z nich se pak při odrazu láme v závislosti na struktuře povrchu mírně jiným směrem kolem dokonale odraženého paprsku.

Tento efekt lze také počítat integrálem počítající funkci přes plochu. V tomto případě ovšem plocha nebude obdélníkového tvaru, ale polokoule se středem v bodě dopadu paprsku a směřujíc po normále ven z objektu. Polokoule může být rovnoměrně vzorkována, ovšem jednotlivé paprsky budou mít pro výsledný příspěvek rozdílné váhy. V této práci vychází z Phongova osvětlovacího modelu a je dán kosinusem úhlu mezi distribuovaným a dokonale odraženým paprskem. Kolmo odražený paprsek má pak nulový přínos pro výpočet výsledné barvy.

Z tohoto důvodu je dobré použít metodu *Importance sampling* a generovat vzorky na polokouli tak, že hustota pravděpodobnosti těchto paprsků je dána právě jejich váhou, díky tomu se váha společně s pravděpodobnostní mírou vykrátí a sníží tak celkový rozptyl výsledku. Postup jakým k tomu dojdeme bude podobný postupu u hloubky ostrosti v části 2.6.3.

Průsvitné povrchy pak fungují na stejném principu, jen se distribuují paprsky lomené místo odražených.

2.6.6 Zanoření výpočtů

Výpočty efektů jsou standardně zanořené do sebe, kdy se nejdříve začne počítat střední hodnota barvy pixelu projekční plochy, pro každý z generovaných paprsků se přitom počítá střední hodnota distribuovaných paprsků v čase, pro každý z nich se přitom počítá střední hodnota paprsků generovaných na čočce a pro každý z nich se počítá osvětlovací model scény, s tím že se generují rekurzivní odražené resp. lomené paprsky, které se dále distribuují. Obecný algoritmus by se dal popsat následovně:

```
Pro každý pixel projekční plochy:
{
  Počítej supersampling a pro každý jeho vzorek:
  {
    Počítej motion blur a pro každý jeho vzorek:
    {
      Počítej hloubku ostrosti a pro každý její vzorek:
      {
        Najdi průsečík se scénou. Počítej jeho výslednou barvu:
        {
          Pro každé světlo: Přičti barvu osvětlovacího modelu.
          Počítej matné odrazy a pro každý vzorek:
          {
            Vypočti barvu odraženého paprsku a přičti.
          }
          Přičti barvu matného odrazu.
          Počítej lom průsvitného povrchu a pro každý vzorek:
          {
            Vypočti barvu lomeného paprsku a přičti.
          }
          Přičti barvu průsvitného povrchu.
        }
        Vrať výslednou barvu osvětlovacího modelu.
      }
      Vypočti střední hodnotu hloubky ostrosti a vrať.
    }
    Vypočti střední hodnotu motion blur a vrať.
  }
  Vypočti střední hodnotu supersamplingu a zapiš jako výslednou hodnotu pixelu.
}
```

Tento výpočet je poměrně velmi náročný a také je poměrně obtížné určit počet vzorků pro jednotlivé efekty scény. Pokud ovšem spojíme výpočty jednotlivých efektů kamery do vícedimenzionálního integrálu pro jeden pixel následovně do 5ti dimenzionálního integrálu:

$$C_{pixel} = \int_{y0}^{y1} \int_{x0}^{x1} \int_{t_B}^{t_E} \int_{v_0}^{v_1} \int_{u_0}^{u_1} l(y, x, t, v, u) \cdot g(t) \, \mathrm{d}u \, \mathrm{d}v \, \mathrm{d}t \, \mathrm{d}x \, \mathrm{d}y \tag{2.34}$$

pak při použití aproximace vícedimenzionálního integrálu z kapitoly 2.5.4 můžeme výpočet zapsat pomocí metody Monte Carlo následovně:

$$E(l(y,x,t,u,v)) = \frac{1}{N}\sum_{i=1}^{N}\frac{l(X_i)}{f(X_i)} \tag{2.35}$$

kde vzorek $X_i = (y_i, x_i, t_i, v_i, u_i)$ je generován pomocí kombinace jednotlivých rozložení pro jednotlivé efekty. $l(y,x,t,v,u)$ je funkce počítající barvu paprsku scény, pro který platí, že je snímán na projekční ploše v pozici (x,y), v čase t a s pozici na čočce kamery (u,v). Popis jak se k jednotlivým rozložením přijde je v předcházejících kapitolách.

Kapitola 3

Teorie optimalizačních technik

V této kapitole se zaměřím na teoretické základy některých optimalizačních technik, které ve své práci používám. Řeč bude o hierarchickém dělení scény, kvazi náhodných sekvencích, speciální instrukční sadě SSE a rychlém průsečíku paprsku s trojúhelníkem.

3.1 Hierarchické dělení scény

Pro ray tracing, obzvláště verzi s distribuovanými efekty, je časově kritický počet výpočtů průsečíků s objekty scény, dokud se nenajde nejbližší průsečík scény.

V případě jednoduchého procházení všech objektů scény musíme pro nalezení nejbližšího průsečíku se scénou spočítat průsečík pro každý objekt, s tím, že se uchovává jen ten nejbližší. Dejme tomu, že budeme mít uzavřenou scénu (tedy pokaždé paprsek narazí na objekt) obsahující 100 trojúhelníků, tak pro nalezení průsečíku se scénou budeme muset vypočíst průsečík 100krát. Z toho 99 není dále užitečných.

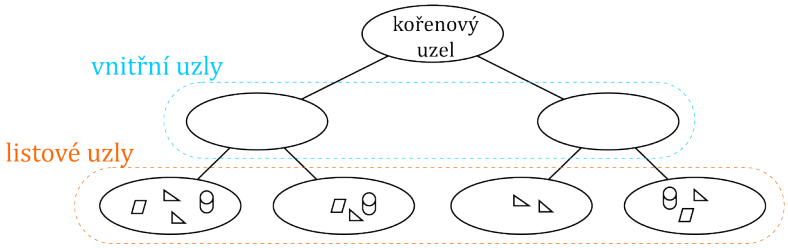
Časová složitost pro nalezení jednoho nejbližšího průsečíku pro N objektů je pak lineární $O(N)$. Na první pohled se může zdát, že to není až takový problém. Avšak v případě jen trochu detailní scény, řekněme o 10^6 trojúhelníků, se čas výpočtu scény oproti výše uvedenému stu zvýší 10000krát. Pokud výpočet předešlé scény trval dejme tomu 10s, tak nyní bude trvat 27 hodin. A to je již velmi dlouhá doba.

Navíc s rostoucím rozměrem výpočtů (distribuovaných a rekurzivních efektů) v závislosti na nastavení scény, roste počet takových průsečíků se scénou společně s dobou výpočtu scény velmi rychlým tempem.

Pro řešení tohoto problému, obecně nazývaného řešení viditelnosti, bylo vyvinuto mnoho různých algoritmů snižující časovou náročnost. Jako příklad můžeme uvést pravidelnou osově zarovnanou mříž, obalové tělesa a stromy. Ty ve své disertační práci *Heuristic Ray Shooting Algorithms* [9] porovnal a shrnul Vlastimil Havran. Zaměřil se přitom na vrhání paprsků. Jako obecně nejvhodnější, s co nejnižší časovou náročností vyšel algoritmus binárního dělení prostoru (anglicky Binary Space Partitioning - BSP), konkrétně sestavující kd-stromovou strukturu za pomocí SAH heuristiky (surface area heuristic) pro sestavení stromu. Toto řešení z praktického hlediska dosahovalo průměrné časové složitosti $O(logN)$. Což by v případě naší scény o 100 trojúhelnících zvýšilo dobu výpočtu pouze přibližně 5krát, tedy na $\pm$ 50s.

3.1.1 Binární stromy

Výsledkem metody rekurzivního binárního dělení prostoru je Binární stromová struktura, která se skládá z uzlů. Oproti obecnému stromu se liší především v tom, že umožňuje dělit prostor pouze na 2 podprostory.



Obrázek 3.1: Binární stromová struktura

Uzly binárního stromu se dělí na následující typy:

- Kořenový uzel - uzel nejvyšší úrovně ve stromové struktuře. Bývá většinou vnitřní, výjimečně může být listový, pokud algoritmus uzná za vhodné nerozdělovat scénu.
- Vnitřní - uzly, které dělí samy sebe na 2 pod-uzly. Neobsahují žádné objekty.
- Listové - uzly, které se již dále nedělí. Obsahují seznam menšího počtu objektů protínajících tento uzel.

Obecně algoritmus pak funguje tak, že uživatel na počátku přiřadí data kořenovému uzlu binárního stromu, spustí předzpracování, ve kterém algoritmus rekurzivně vytvoří stromovou strukturu dokud nejsou splněny podmínky pro ukončení dělení. Po ukončení dělení je možné použít tuto strukturu pro vyhledávání, kdy se začíná u kořenového uzlu a postupně se ve stromu prohledává, dokud nejsou data nalezena.

Dále se budeme bavit jen o stromech zpracovávajících 3D prostor. Binární stromy se dále pak rozlišují podle toho, jak se vnitřní uzly dělí na další uzly, existují následující typy:

- BSP strom (Binary space partition tree) - dělí prostor pomocí libovolně orientované plochy. Při použití složitějších geometrických tvarů může dojít k problémům při určování, zda objekt protíná obálku synovského uzlu, protože ten je definován pomocí polygonové sítě.
- kd-strom (k-dimensional tree) - dělí prostor pouze pomocí osově zarovnaných ploch. Každý uzel tak lze definovat osově zarovnaným obalovým kvádrem (voxelem), který zjednodušuje rozhodování zda objekt patří do uzlu a také je velmi jednoduché zjistit, ve kterém místě bude paprsek dělící rovinu protínat.

Mějme osu $a = \{osa | osa \in (X, Y, Z)\}$, paprsek $r : r(t) = O + t \cdot \vec{d}$, nezarovnanou rovinu $p_N : ax + by + cz + k = 0$ a zarovnanou rovinu $p_A : p_a = konst.$ s osou a. Pak rovnice vzdáleností paprsku od dělící nezarovnané, resp. zarovnané roviny jsou:

$$t = \frac{a \cdot O_x + b \cdot O_x + c \cdot O_z + k}{a \cdot d_x + b \cdot d_x + c \cdot d_x} \qquad (3.1) \qquad t = \frac{p_a - O_a}{d_a} \qquad (3.2)$$

Pro výpočet vzdálenosti od nezarovnané roviny je počet základních aritmetických operací přibližně 6krát vyšší, než u zarovnané roviny. Z čehož vyplývá také pomalejší průchod stromem.

Dále se budeme zabývat pouze kd-stromy, protože jsou pro ray tracing vhodnější [9].

3.1.2 Obálka uzlů a objektů

Pro potřeby výpočtů v kd-stromu zavedeme pro každý objekt, potažmo také pro uzly obálku v podobě osově zarovnaného kvádru, anglicky též Axis Aligned Bounding Box, neboli zkratkou $AABB$. Tak budeme případně v textu označovat takovou obálku.

Tento kvádr je možné a také vhodné definovat pouze 2 body a to dolním levým a horním pravým. Z těchto dvou bodů je pak možné vyčíst veškeré informace o tomto obalovém kvádru, jako jsou jeho hranice pro jednotlivé osy, obsah stran, objem a také, která strana je nejdelší.

Pro uzly je obálka důležitá, protože při dělení uzlu se pro všechny objekty počítá zda zasahují do obálek pod-uzlů, či ne. Tedy intersekce daného objektu s obalovým kvádrem.

3.1.3 KD-Strom

Jak již bylo řečeno výše, kd-strom dělí rekurzivně prostor osově zarovnanou rovinou. Znamená to tedy, že každý uzel kd-stromu je ohraničený voxelem. Začněme u kořenového uzlu. Ten obsahuje všechny objekty scény a je definován ohraničujícím voxelem, ve kterém se nacházejí všechny objekty. Dále tento uzel obsahuje informaci podle které osy se bude dělit (X,Y, nebo Z) a souřadnici určující místo dělení na dané ose. Tímto je definována dělící rovina, která dělí uzel na 2 synovské a zároveň je tímto pro synovské uzly definován jejich obalový voxel. Platí přitom, že součet objemů synovských uzlů se rovná objemu rodičovského uzlu. Vnitřní uzly mají naprosto shodnou funkci, jen neobsahují žádné objekty. Listové uzly pak již neobsahují dělící rovinu a jsou jen definovány obalovým voxelem a seznamem objektů, protínajících tento voxel. Listové uzly nemusí obsahovat žádné objekty.

Procesu vytvoření objektu se říká sestavení stromu a je to poměrně časově náročná operace, ovšem v porovnání s celkovou dobou renderování scény většinou naprosto zanedbatelná.

Kd-strom ovšem není vhodný pro všechny typy ray tracingů. Např. není vhodný pro ray tracing v reálném čase, protože je nutné dynamicky aktualizovat scénu a sestavení kd-stromu je příliš náročné na to aby ho bylo možné provádět v řádově sekundových intervalech. U animací, si lze pomoci různými optimalizacemi, díky kterým nemusíme strom znovu sestavovat. Tím se budeme zabývat v této práci později.

3.1.4 Sestavení kd-stromu

Sestavení je nejnáročnější operace s kd-stromem. Nejčastějším způsobem sestavení je shora dolů za pomocí rekurze. Následuje pseudokód sestavení kd-stromu, ten ilustruje sestavení stromu daleko lépe než slovní popis:

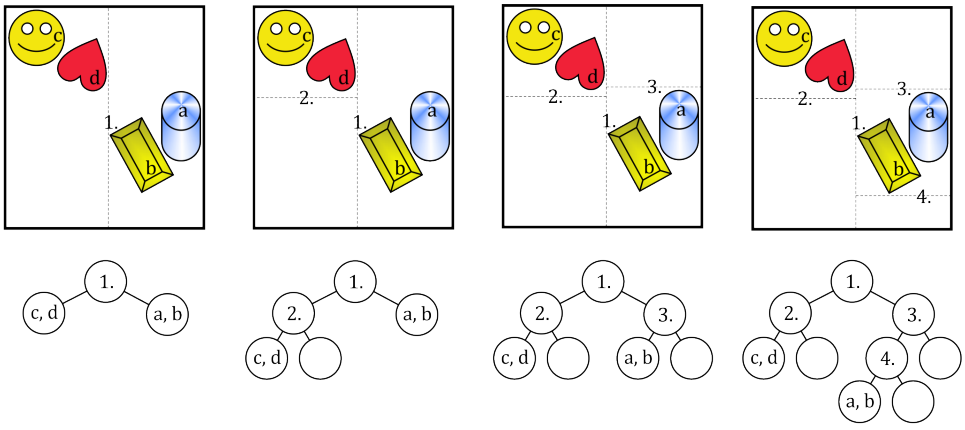
```
Uzel obsahuje informace:
        typ uzlu, seznam objektů, dělící rovinu, pravý a levý uzel

Sestav Strom (seznam objektů)
{
```

```
    Vytvoř kořenový uzel stromu.
    Nastav kořenový uzel na listový.
    Přiřaď objekty kořenovému uzlu.
    Rekurzivně rozděl uzel (kořenový uzel, 0).
}

Rekurzivně rozděl uzel(dělený uzel, zanoření)
{
    Pokud je zanoření větší než maximální,
    nebo počet objektů v děleném uzlu je menší než minimální,
        tak nedělej s uzlem nic.
    Jinak:
        Urči dělící rovinu dělenému uzlu podle jeho objektů.
        Vytvoř listové uzly: levý uzel, pravý uzel.
        Do levého uzlu přiřaď objekty děleného uzlu, které do něj zasahují.
        Do pravého uzlu přiřaď objekty děleného uzlu, které do něj zasahují.
        Převeď dělený uzel na vnitřní a smaž jeho seznam objektů.
        Přiřaď levý a pravý uzel jako synovské uzly dělenému uzlu.
        Rekurzivně rozděl uzel(levý uzel, zanoření + 1).
        Rekurzivně rozděl uzel(pravý uzel, zanoření + 1).
}
```

Vytvoření takového stromu je ilustrování níže:



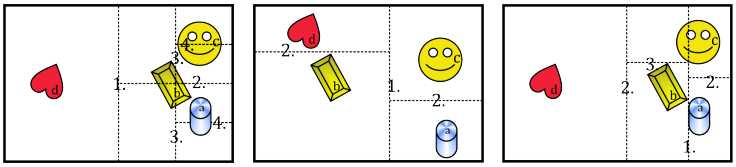
(a) krok 1. (b) krok 2. (c) krok 3. (d) krok 4.

Obrázek 3.2: Sestavení kd-stromu ukázka

Na obrázku 3.2 vidíme příklad sestavení stromu z objektů s následným grafem. Čísla značí dělící roviny, potažmo vnitřní uzly stromu obsahující tuto rovinu, písmena pak značí objekty.

3.1.5 Určení dělící roviny uzlu

Pokud se uzel dělí, tak se tak děje na základě obalového voxelu a seznamu objektů, které má přiřazeny. Existuje mnoho způsobů jak rozhodnout, kde vytvořit dělící rovinu. Některé způsoby jsou vhodnější, některé méně.



(a) Prostorový medián (b) Objektový medián (c) Geometrický střed

Obrázek 3.3: Typy dělení uzlu kd-stromu

Dělící rovina se dá určit:

- **Prostorovým mediánem**
  Nejjednodušší způsob. Uzel se pak dělí podle nejdelší strany voxelu. Místo dělení je přesně ve středu této strany. Tento algoritmus vytváří prostorově vyvážené uzly. Není příliš vhodný pro ray tracing až na případ, kdy objekty v prostoru jsou rozloženy rovnoměrně.

- **Objektovým mediánem**
  Tento způsob není až tak triviální, ale je pro ray tracing nevhodný. Vytváří vyvážený strom, kdy v každé větvi je přibližně stejný počet objektů. Tento algoritmus je vhodný pro vyhledávací stromy, ve kterých mají všechny listy stromu stejnou pravděpodobnost.

- **Geometrickým středem**
  Jde o způsob, ve kterém se nalezne geometrický střed všech objektů uzlu a podle něj se pak dělí. Tento způsob také není moc vhodný, protože objekty neseskupuje, ale spíše rozděluje.

- **Pomocí Surface Area Heuristic - SAH**
  Jde o metodu dělení scény pracující s cenovým modelem odpovídajícím na otázku "Které rozdělení uzlu vede k cenově lepšímu prostorovému rozdělení scény pro výpočet průsečíku paprsku s se scénou?" Metoda je dle [9] pro ray tracing jedna z obecně nejvhodnějších. Popíšeme si ji podrobněji v následujícím textu.

3.1.6 Heuristika SAH(The Surface Area Heuristic)

Heuristika dělení scény The Surface Area Heuristic (dále jen SAH) je model, který na základě geometrické pravděpodobnosti a ohodnocení výpočetní náročnosti jednotlivých operací při hledání nejbližšího průsečíku ve scéně je schopen "odpovědět" na otázku, která z dělících rovin je nejvýhodnější a minimalizuje tak cenu průchodu [15].

Řekněme, že jsme se při sestavování stromu dostali do nějakého uzlu U. Dále se budeme zabývat jen tímto uzlem, rozhodováním co s ním a předchozí nebudeme brát v potaz.

Uzel U můžeme jednoduše označit za listový a ponechat mu všechny jeho objekty. Případný paprsek procházející tímto uzlem by pak byl podroben testu na průsečík se všemi objekty v něm obsažených. Díky tomu pak lze zjistit zda vůbec naráží do nějakého objektu uzlu a případný nejbližší průsečík. Při tom se spotřebuje určité množství výpočetních prostředků

$$C_{LIST} = \sum_{i=1}^{N} C_I(i)$$

kde N je počet objektů v uzlu, $C_I(i)$ výpočetní cena (případně čas výpočtu) testu průsečíku paprsku s i-tým objektem.

Tato výpočetní cena může být odlišná pro různé typy objektů, případně pro různé situace paprsku a objektu. V této práci ovšem budeme předpokládat, že výpočetní cena C_I je konstantní pro všechny situace a objekty. Ve výsledku by se toto zjednodušení nemělo výrazně projevit, protože ve scéně počítáme především s trojúhelníky. Množství spotřebovaných prostředků pro uzel U pak lze vyjádřit jednodušeji a to jako

$$C_{LIST} = N \cdot C_I$$

Druhou možností je rozdělit uzel na 2 synovské uzly L a R v nějakém místě. V tom případě cena při průchodu tímto uzlem stromu a výpočtu průsečíků v synovských uzlech může být vyjádřeno následovně:

$$C_{SPLIT} = C(A, B) = C_T + p_A(N_A \cdot C_I) + p_B(N_B \cdot C_I) \tag{3.3}$$

kde C_T je cena průchodu uzlem (zjištění kterým synovským uzlem bude paprsek pokračovat a pokračování v tomto synovském uzlu, je taktéž konstantní pro všechny uzly), p_A a p_B je pravděpodobnost, že paprsek bude pokračovat jedním, resp. druhým synovským uzlem, při zachování $p_A + p_B = 1$. N_A resp. N_B je počet objektů zasahujících do uzlu A resp. B.

Pravděpodobnosti p_A a p_B jsou založené na geometrické pravděpodobnosti. Mějme uzel U s konvexní obálkou a uzel A v něm obsaženy (taktéž s konvexní obálkou) (v našem případě kvádry). Podmíněná pravděpodobnost, že paprsek, který prochází obálkou U bude procházet i obálkou A, je poměr mezi jejími povrchy:

$$p_A = P(A|U) = \frac{S_A}{S_U}$$

Podobně pak pro p_B:

$$p_B = P(B|U) = \frac{S_B}{S_U}$$

kde S_U resp. S_A resp. S_B je obsah povrchu obálky uzlu U resp. uzlu A resp. uzlu B.

Heuristika SAH se pak snaží dělit uzly tak, aby cena průchodu stromu pro náhodný paprsek do nalezení požadovaného objektu byla co nejmenší. Toho se dosáhne tak, že při vytváření stromu se při rozhodování zda uzel nechat jako listový, nebo jej rozdělit počítá s minimální cenou, kdy se porovná cena při ponechání uzlu listovým s cenami všech možných rozdělení uzlu. Pokud pak pro nějaké rozdělení platí

$$C_{SPLIT} < C_{LIST}$$

tak se uzel rozdělí a pokračuje se rekurzivně dále, jinak se ponechá uzel listovým a tato větev stromu se ukončí. Samozřejmě se přitom musí dodržovat podmínky ohledně maximálního zanoření stromu apod.

3.1.7 Hledání dělící roviny s minimální cenou

Počet pozic dělení scény je nekonečný, je tedy nutné jej omezit vhodným způsobem. Funkce $C_{SPLIT(s,a)}$, kde s značí pozici dělení na ose a, je na celém svém definičním spojitá až na místa, kde se mění počet N_A, či N_B. Protože by bylo náročné zjišťovat pro každé počítané dělení zda daný objekt (např. trojúhelník) zasahuje do dané části, využijeme $AABB$ obálek objektů, které jsme definovali dříve.

Uvažujme pouze jednu osu dělení, např. X, pracujeme tedy nyní jen s 1D prostorem na ose X. Hranice $AABB$ obálek všech objektů pak na ose X představují body ve kterých se mění počet N_A či N_B. Jsou to tedy místa, které má cenu zkoumat. Těchto míst je ovšem velmi mnoho a zjištění zda objekt protíná synovský uzel či ne je poměrně náročná operace, která se nedá zjednodušit na obálky. Pro jedno dělení se pak musí provést $2N$krát (každý objekt pro oba synovské uzly), těchto dělení je ovšem $3 \cdot 2 \cdot N$ a tedy celkový počet testů na průnik s obálkou uzlu je $12 \cdot N^2$.

Funkce možného nalezení dělící roviny s minimální cenou $SPLIT_{MIN}$ má tedy kvadratickou složitost:

$$SPLIT_{MIN} \in O(N^2) \tag{3.4}$$

Proto je tento přístup pro jakékoliv rozsáhlejší scény naprosto nevhodný. Popíšeme si nyní metody jak snížit složitost funkce $SPLIT_{MIN}$:

- **Řazením objektů** – Využívá $AABB$ obálky objektu pro určení počtu objektů v synovských uzlech. Princip je takový, že se vytvoří seznam všech bodů vhodných pro dělící roviny (tedy levý a pravý okraj každého $AABB$ tělesa). Každému prvku se připíše počet hraničních levých bodů na tomto místě a počet hraničních pravých bodů v místě. Seznam se poté seřadí podle pozice. Počet prvků v obou částech se určí iterativním průchodem pomocí diference počtu levých a pravých hraničních bodů. Časová složitost této metody je $O(N^2 \log N)$ [9].

- **Vzorkováním** – Danou dělenou osu rozdělíme na určitý počet intervalů. hranice těchto intervalů pak slouží jako dělící rovina. Pro každou dělící rovinu vypočítáme počet objektů nalevo a napravo. Metoda je rychlá s poměrně dobrými výsledky. Jen u jednodušších scén kvalita klesá.

- **Pevným výběrem osy** – Pokud počet objektů děleného uzlu přesahuje určitý vysoký limit, můžeme 3x zrychlit hledání dělící roviny díky zanedbání 2 souřadných os. Ta na které budeme hledat rovinu pak bude ta, která má ze všech tří největší rozměr. Tuto metodu je dobré použít u uzlů s opravdu velkým počtem objektů.

3.1.8 Průchod kd-stromem

Při hledání nejbližšího průsečíku se scénou nejdříve otestujeme zda vůbec paprsek protíná obálku celého stromu. Jako vstupní bod nám slouží kořenový uzel a jeho obálka. Poté rekurzivně procházíme uzly stromu a při každém kroku se řídíme pravidly:

1. Pokud je uzel vnitřní, tak při vstupu paprsku do něj máme k dispozici bod vstupu paprsku pro obálku uzlu. Podle směru paprsku a vstupního bodu zároveň zjistíme, zda paprsek protíná jen jeden synovský uzel, či oba.

(a) Pokud protíná jen jeden, tak rekurzivně procházíme tento synovský uzel stejným způsobem.

(b) Pokud paprsek protíná oba, tak nejprve rekurzivně projdeme bližším synovským uzlem (ten ve kterém se paprsek při vstupu do uzlu nachází). Pokud paprsek na nic nenarazí, projdeme druhý synovský uzel.

(c) Pokud se ani v jednom uzlu paprsek nenarazí na překážku vracíme tento neúspěch otcovskému uzlu.

2. Pokud je uzel listovým, tak projdeme seznam objektů a pro každý z nich provedeme test na průsečík. Přitom si vždy ukládáme ten nejbližší a při dalším úspěšném testu kontrolujeme, zda nový průsečík je bližší než předchozí.

   (a) Při úspěchu ukončujeme procházení stromem a vracíme průsečík s jeho parametry.

   (b) Při neúspěchu vracíme tento neúspěch otcovskému uzlu.

Rekurzivní průchod kd-stromem je také možné převést na iterativní a tím podstatně snížit režijní výpočetní nároky.

3.2 Quazi-náhodné vzorkování scény

Mezi hlavní nedostatky metody Monte Carlo u distribuovaného ray-tracingu patří pomalá konvergence výpočtu hledaného řešení. Ta je omezena horní asymptotickou složitostí $O\left(\frac{1}{\sqrt{N}}\right)$. Nedostatek je to u distribuovaného ray-tracingu hlavně proto, že při složité scéně s vysokým rozlišením a nízkém počtu vzorků, je scéna příliš zašuměná a při vysokém počtu vzorků výpočet scény trvá extrémně dlouhou dobu (řádově hodiny, dny). Je to dáno až příliš náhodným rozložením vzorků a tedy u nízkého počtu vzorků zároveň nerovnoměrným.

Tento nedostatek řeší částečně použití *low-discrepancy (LD)* sekvencí místo pseudo-náhodných čísel. *LD* nazýváme sekvenci bodů, která v určitém k-dimenzionálním prostoru vyplňuje jednotkový prostor rovnoměrněji, než sekvence pseudo-náhodných čísel. Podrobnější informace k celé této kapitole lze nalézt v článcích a knize, ze kterých jsem také čerpal v celé této kapitole: [16], [4] a [12].

3.2.1 Diskrepance sekvence

Jedná se o matematickou definici, která určuje kvalitu, či lépe řečeno rovnoměrnost dané sekvence. Mějme prvky $P = x_1, x_2, x_3, \ldots, x_n$, které jsou sekvencí k-dimenzionálního prostoru $[0,1]^k$. Dále mějme k-dimenzionální podprostor (v 3D kvádr) b, pro který platí $b \subseteq [0,1]^k$, jeden roh je umístěn v počátku a jedná se o osově zarovnaný Objem pro tento podprostor je definován jako $V(b) = \prod_{i=1}^{k} b_i$. Pak *star diskrepanci*2 definujeme jako:

$$D_N^*(B,P) = \sup_{b \in B} \left| \frac{\sharp\{x_i \in b\}}{N} - V(v) \right| \tag{3.5}$$

kde $\sharp\{x_i \in b\}$ je počet bodů v b, B představuje množinu těles. Přitom právě $\sharp\{x_i \in b\}$ představuje aproximaci tělesa b danou body P. Tato diskrepance určuje nejhorší aproximační chybu, která byla nalezena mezi tělesy množiny B. *star* je konkrétní, nejužívanější typ diskrepance, kdy se berou tělesa zarovnané s osami a jeden roh je umístěn do počátku.

Pokud bychom chtěli určitou sekvenci použít pro metodu Monte Carlo, je nutné, aby sekvence splňovala podmínku:

$$D^* \to 0 \quad pro \quad N \to \infty \tag{3.6}$$

Tato podmínka zaručuje, že sekvence pokrývá prostor rovnoměrně a je použitelná.

Diskrepance pro kvalitní pseudo náhodné sekvence činí [11]:

$$D^*_{PSEUDO\_RANDOM}(x_n) = O\left(\frac{log(logN)}{\sqrt{N}}\right) \tag{3.7}$$

Z této diskrepance je pak odvozena rychlost konvergence metody Monte Carlo, kdy se .

3.2.2 Quazi-náhodné sekvence

Základem, pro tvorbu *LD* sekvencí je funkce anglicky zvaná *radical inverse*. Mějme nějaké celé číslo n, to se pak dá popsat v soustavě o základu b sekvencí cifer $d_1, d_2, \ldots, d_m$:

$$n = \sum_{i=1}^{\infty} a_i b^{i-1} \tag{3.8}$$

Pak funkce *radical inverse* se dá popsat následovně:

$$\Phi_b(n) = 0.d_1 d_2 d_3 \ldots d_m \tag{3.9}$$

Van Der Corputova sekvence je pak výstupem této funkce pro soustavu $b = 2$:

n	n_b (b=2)	*radical inverse*	Van der Corput
1	1	0,1	0,5
2	10	0,01	0,25
3	11	0,11	0,75
4	100	0,001	0,125
5	101	0,101	0,625
6	110	0,011	0,375
7	111	0,111	0,875

Tabulka 3.1: Výpočet Van der Corputovy sekvence

Mezi nejznámější LD sekvence patří: Haltonova, Faureho, Sobolova, Hammerslyova a další. V této práci se dále budeme zabývat pouze Sobolovou sekvencí (tedy částečně Haltonovy, protože první dimenze jsou shodné). Konkrétně pak jejími pouze prvními dvěma dimenzemi. Haltonova sekvence má totiž dobré vlastnosti jen pro pár prvočísel a ve vyšších dimenzích je až příliš pravidelná a k tomu nerovnoměrná to ve velmi znatelné podobě.

Haltonova sekvence

Haltonova sekvence je k-dimenzionální sekvence v jednotkovém prostoru $[0, 1]^k$ a má následující vlastnosti:

- Sekvence první dimenze tvoří Van der Corputova sekvence o bázi 2.

- Sekvence druhé dimenze tvoří Van der Corputova sekvence o bázi 3.
- Další dimenze tvoří vždy sekvence o bázi k-tého prvočísla (tedy 5,7,11,13,17...).
- Velkou její výhodou je, že nemusíme dopředu znát počet potřebných prvků.
- Další obrovskou výhodou je její na pohled ne tak výrazná pravidelnost a velmi nízká diskrepance:

$$D_N^*(x_n) = O\left(\frac{(logN)^2}{N}\right) \tag{3.10}$$

  která zároveň určuje rychlost konvergence metody Monte Carlo.

Sobolova sekvence

Je k-dimenzionální sekvencí.

- První dimenze je identická s Van der Corputovou sekvencí o bázi 2.
- Další dimenze pak tvoříme pomocí permutace první dimenze.

Důležitá je pro nás druhá dimenze vyjádřena permutací první dimenze. Zjednodušeně řečeno se pro jednotlivé vzorky vezme binární podoba cifer a provede se nad nimi a ciframi posunutými o jeden bit vpravo exkluzivní součet. Jinak se postupuje úplně stejně jako u Haltonovy sekvence. Tento postup je velmi dobře popsán společně s teorií této sekvence v [10].

Permutace sekvencí

Velkou nevýhodou kvazi-náhodných sekvencí je to, že jsou deterministické a tedy při každém novém generování celé sekvence jsou její hodnoty stále stejné. Díky tomu mohou a budou v obraze znatelné artefakty, protože se prakticky používá určitý počet vzorků stále dokola. Tento zjevný handicap lze odstranit permutací jednotlivých cifer generovaného vzorku. Matematicky se pak dá popsat *permutovaná radical inverse* funkce následovně:

$$\Psi_b(n) = 0.p(d_1)p(d_2)p(d_3)\dots p(d_m) \tag{3.11}$$

kde funkce $p(d_j)$ označuje cifru která se po permutaci bude nacházet na pozici j. Je nutné podotknout, že aby měla sekvence stále stejné vlastnosti co se týče kvality, je nutné, aby permutace byla shodná pro celou jednu generovanou sekvenci.

Pokud budeme permutovat sekvenci o základu 2, můžeme místo permutace jednotlivých cifer zvolit rychlejší způsob zakódování pomocí náhodného binární sekvence o stejné délce a její exkluzivní součet (XOR) s vzorky této sekvence. Odpadá tak poměrně složité permutování jednotlivých cifer, přičemž výsledky jsou téměř stejné.

Sekvence(0,2)

Sekvence(0,2) je speciální typ sekvence, kde pro první dimenzi se generují vzorky pomocí Haltonovy permutované sekvence o základu 2 a druhou dimenzi generujeme pomocí permutované Sobolovy sekvence druhé dimenze. Obě sekvence jsou tedy postaveny nad binárními čísly. Tento typ 2D sekvence má mnoho dobrých vlastností, které jej předurčují k použití v ray tracingu. Má o něco málo horší rovnoměrnost, než Haltonovy sekvence o základu 2 a 3, ale nabízí přitom jiné, dobře využitelné vlastnosti.

Jednou z nich je že, tato 2D sekvence pro $2^n, \quad n \geq 1$ vzorků splňuje následující: Pokud rozdělíme prostor pravidelně v jedné, resp. druhé ose tak, aby počet úseků byl vždy 2^k resp. 2^l a celkový počet rozdělení 2D prostoru je maximálně roven počtu vzorků sekvence, má tato sekvence stejný počet vzorků ve všech těchto částech.

Vzorky jsou pak tedy rovnoměrně rozděleny v úsecích a zároveň jsou dostatečně "náhodné". Nevýhodu pak lze vidět v tom, že je nutné dodržet násobek dvou pro počet vzorků, jinak nebude sekvence kvalitní.

Tento typ sekvence má i další kladné vlastnosti. Pro podrobnější informace doporučuji výbornou knihu [16].

3.3 Instrukční sada SSE

Zkratka SIMD (Single Instruction, Multiple Data) znamená v češtině "zpracování více dat jednou instrukcí'. Jde typ architektury procesorů, určené především pro zpracování velmi náročných výpočtů. SSE pak znamená *Streaming SIMD Extension*. Jedná se o rozšíření instrukční sady procesorů x86 a x64 o vektorové zpracování dat, namísto skalárního. Tato instrukční sada umožňuje zpracování takovýchto dat jak s reálnými, tak s celými čísly. V závislosti na použití. K dnešnímu dni existuje 5. revize tohoto rozšíření, která již obsahuje přibližně 460 instrukcí pro zpracování dat.

3.3.1 Intrinsiky

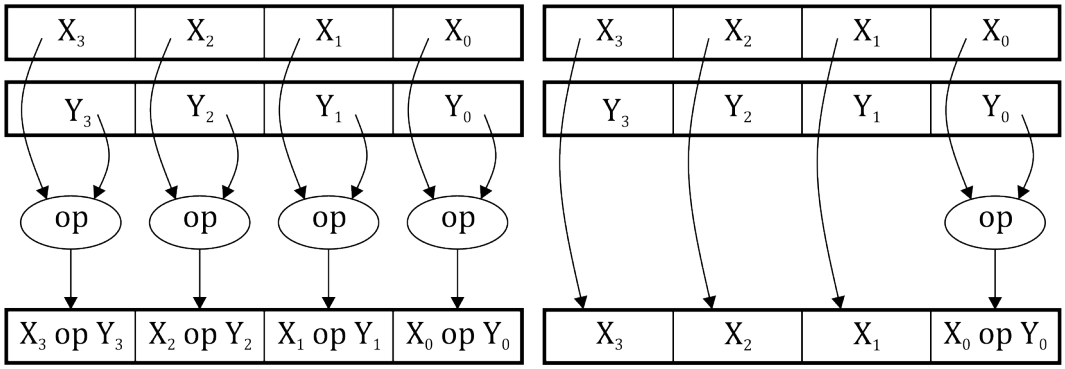
Při programování v C++ se mohou přímo používat instrukce procesoru pomocí speciálního inline assembler bloku vloženého do programu. Tento způsob ovšem nezaručuje dostatečnou kompatibilitu napříč kompilátory a neumožňuje použití na 64 bitových architekturách. Je proto vhodnější použít intrinsiky (angl. Intrinsic function), což jsou funkce, které jsou na rozdíl od ostatních implementovány přímo do kompilátoru, který tak umožňuje případnou další optimalizaci kódu oproti inline assembleru. Toho se užívá také u rozšíření SSE.

3.3.2 Jednotka zpracování

Jednotka určena pro zpracování SSE instrukcí používá k výpočtům 128-bitové registry nazývané XMM. V 32-bitovém režimu procesoru jich je dostupných 8 (registry XMM0 až XMM7), v 64-bitovém režimu je navíc dostupných dalších 8 jednotek (XMM8-XMM15). Dále je dostupný kontrolní/řídící registr MXCSR, registry pro obecné použití a adresní prostor procesoru. Tato jednotka je zároveň jednotkou pro využití MMX instrukcí, které SSE rozšiřuje. Kompilátory umožňující použití intrinsiků SSE instrukcí většinou definují 128bitový datový typ, s možností přístupu k jednotlivým složkám pro jednodušší práci s vektory.

Pro registry XMM existuje omezení, kdy se XMM registr nesmí použít k adresování operandů, ale jen k přímým výpočtům a načítání/ukládání dat z/do paměti.

V jednom registru pak může být buď 16 bajtů/znaků, nebo 8 16-bitových celých čísel, nebo 4 32-bitová celá/reálná čísla, nebo 2 64-bitové celá/reálná čísla. Vždy je to 128 bitů, tedy šířka registru XMM. Dále se budeme zabývat jen čtyřmi 32bitovými reálnými čísly. Pro každý z těchto datových typů jsou pak dostupné určité instrukce pro práci s ním. Výpočty probíhají v zásadě dvojího druhu. Buď čistě vektorový (např. vektorový součet, instrukce končící *PS*), nebo skalární (např. přičtení hodnoty k jedné složce vektoru, instrukce končící *SS*):



(a) Vektorová operace (b) Skalární operace

Obrázek 3.4: Typy operací SSE jednotky

kde *op* představuje operaci s vektory a X_i resp. Y_i představují složky vektorů.

Mezi základní operace, kterými jednotka SSE disponuje patří: Načtení vektorových dat z paměti/registru do paměti/registru (MOVAPS, MOVUPS), načtení skalární hodnoty z paměti/registru do paměti/registru (MOVSS), vektorový resp. skalární součet (ADDPS resp. ADDSS), rozdíl (SUBPS, SUBSS), součin (MULPS, MULSS), podíl vektorů (DIVPS, DIVSS) a převrácená hodnota (RCPPS, RCPSS). Mezi pokročilé operace pak patří např. skalární součin vektorů (DPPS), odmocnina (SQRTPS, SQRTSS) a načtení znamének (MOVMSKPS).

3.3.3 Zarovnání dat v paměti

Při načítání dat z paměti do registru XMM, případně obráceně, se implicitně počítá s tím, že data jsou zarovnány v paměti na adrese násobku 16, tedy po 16 bajtech (128bitech). Je to z toho důvodu, že načítání adresně zarovnaných dat je velmi rychlé. V případě nezarovnaných dat v paměti pak při pokusu načíst tyto data dojde k chybě programu z důvodu neoprávněného přístupu k paměti. Instrukce zajišťující toto rychlé načtení dat je MOVAPS.

Paměť lze ovšem také načítat z paměti do registru XMM (či obráceně) i z nezarovnané paměti. K tomu slouží instrukce MOVUPS. Tato operace je ale několikrát časově náročnější.

Pokud chceme ale využít rychlejší variantu a používat tak načítání z/do zarovnané paměti, je třeba pro všechny data/struktury využívající, nebo komponující 128bitovou datovou strukturu zařídit toto zarovnaní v paměti. Při alokování dat na zásobníků lze toho jednoduše docílit díky speciálnímu parametru kompilátoru (např. u C++ kompilátoru GCC), který kompilátoru řekne, aby dané data vytvořil na zásobníku se zarovnáním.

Horší situace nastává u dat alokovaných na hromadě. Tam je již třeba mít k dispozici alokátor umožňující 16 bajtové zarovnání a pokud není k dispozici, nezbývá, než si jej naimplementovat ručně. U všech tříd (např. u jazyka C++) pak je potřeba zařídit, aby pracovaly s alokátorem zarovnané paměti. Problém také může nastat u rozšiřujících knihoven (např. STL knihoven u C++ jazyka), kdy je potřeba třídy těchto knihoven upravit tak, aby také pracovaly se 16 bajtově zarovnanou pamětí. Samozřejmě se bavíme jen o případě, že chceme danou třídu využít pro zpracování 128bitového datového typu (např. *vector* z STL knihovny C++).

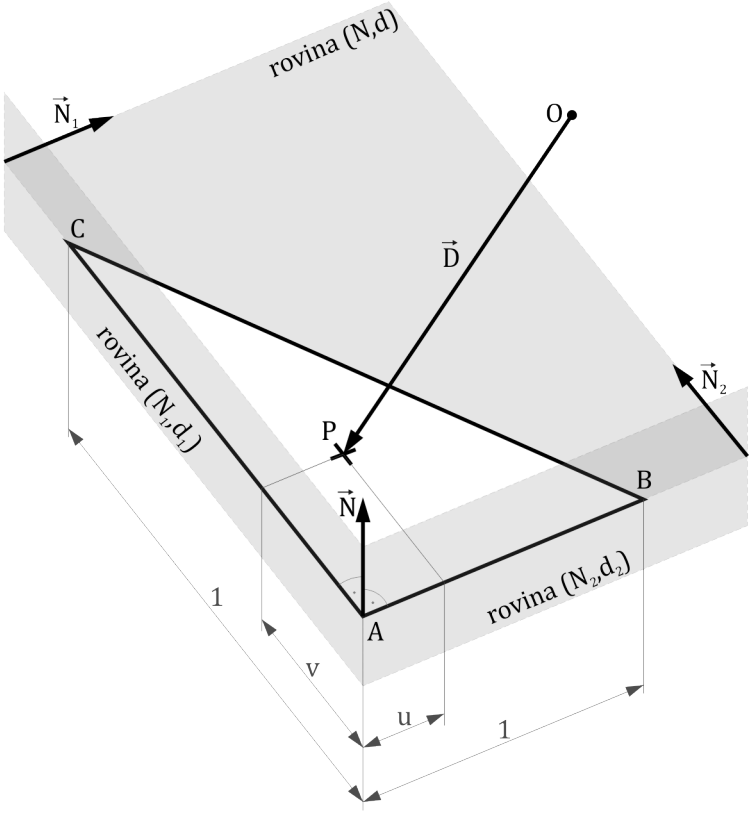
3.4 Rychlý průsečík paprsku s trojúhelníkem

Jednou z časově nejnáročnějších a zároveň často vykonávanou pasáží ray tracingu je test průsečíku s trojúhelníkem. Proto jsem se pokusil nalézt rychlejší alternativu ke stávajícímu řešení. Současné řešení je postaveno na již poměrně dobře optimalizované metodě popsané Ingo Waldem v jeho disertační práci [21]. Ta je založena na předpočítaných informacích o trojúhelníku a testování případného průsečíku na barycentrické souřadnice trojúhelníku, podle kterých lze zjistit, zda bod leží v trojúhelníku, či ne.

S příchodem nových procesorů a nových instrukčních sad je možné tento výpočet optimalizovat ještě o něco více. Příkladem toho může být metoda popsána Jiřím Havlem a Adamem Heroutem v článku [8]. Ta je založena na principech Waldově [21] a Shevtsovově [17] metodách a přidává k nim optimalizaci pomocí SIMD SSE4.1 instrukcí procesoru a odlišný postup výpočtu, při kterém se některé informace předpočítávají tak aby samotný výpočet byl co nejméně náročný. Toho je např. dosaženo pouze jedním potřebným dělením, které může být ve srovnání s násobením a sčítáním několikrát časově náročnější. Navíc je toto dělení ve výsledném kódu odloženo až do chvíle kdy je jisté, že průsečík s trojúhelníkem nastal. Navíc může být nahrazeno výpočtem opačné hodnoty ($\frac{1}{hodnota}$), která se dá dále mírně výkonově vylepšit.

Metoda je založena na výpočtu bodu pomocí barycentrických souřadnic u, v a 3 rovinami vymezujících trojúhelník rovinou, ve které trojúhelník leží $(\vec{N}, d)$ a dvěma dalšími, kolmými na první, které vymezují odvěsny trojúhelníku , $(\vec{N}_1, d_1)$ a $(\vec{N}_2, d_2)$. Některé informace se předpočítají a nemusí se tak počítat znovu při každém dalším testu na průsečík.

3.4.1 Matematický postup



Obrázek 3.5: popisek

Bod P ležící v trojúhelníku pomocí je vyjádřen pomocí barycentrických souřadnic a bodů trojúhelníku následovně:

$$P = A + u \cdot (B - A) + v \cdot (C - A) \tag{3.12}$$

kde barycentrické souřadnice musí splňovat následující podmínky aby bod ležel v trojúhelníku:

$$\begin{aligned} u &\geq 0 \\ v &\geq 0 \\ (u + v) &\leq 0 \end{aligned} \tag{3.13}$$

Pokud dáme do rovnosti mírně upravenou tuto rovnici společně s rovnicí výpočtu bodu na paprsku (2.1) dostaneme rovnici vyjadřující průsečík paprsku s trojúhelníkem:

$$O + t \cdot \vec{d} = (1 - u - v) \cdot A + u \cdot B + v \cdot C \tag{3.14}$$

musí samozřejmě také platit podmínka pro paprsek $0 \leq t < \infty$ aby byl průsečík pro ray tracing platný.

Předpočítané informace, které slouží pro další výpočty jsou právě parametry třech vymezujících rovin v podobě normálového vektoru a parametru d:

$$\begin{aligned} \vec{N} &= \vec{AB} \times \vec{AC}, & d &= -\vec{N} \cdot A \\ \vec{N_1} &= \frac{\vec{AC} \times \vec{N}}{|N|^2}, & d_1 &= -\vec{N_1} \cdot A \\ \vec{N_2} &= \frac{\vec{N} \times \vec{AB}}{|N|^2}, & d_2 &= -\vec{N_2} \cdot A \end{aligned} \tag{3.15}$$

kde $\vec{N}$ musí být takto vypočten a použit jako jmenovatel, aby výpočet barycentrických souřadnic byl správný. Údaje o průsečíku se pak dají vypočíst následovně:

$$\begin{aligned} t &= \frac{\vec{N} \cdot O + d}{\vec{N} \cdot \vec{D}} \\ P &= O + t \cdot \vec{D} \\ u &= \vec{N_1} \cdot P + d_1 \\ v &= \vec{N_2} \cdot P + d_2 \end{aligned} \tag{3.16}$$

Nyní můžeme odsunout dělení hodnotou $\vec{N} \cdot \vec{D}$ na později až po testování zda leží průsečík v trojúhelníku. Toho docílíme tak, že rovnice (3.16) vynásobíme pomocí det čímž nám vzniknou nové dočasné zjednodušené rovnice. Tento postup je založen na Shevtsovově metodě [17]:

$$\begin{aligned} det &= \vec{D} \cdot \vec{N} \\ t' &= d - (O \cdot \vec{N}) \\ P' &= det \cdot O + t' \cdot \vec{D} \\ u' &= P' \cdot \vec{N_1} + det \cdot d_1 \\ v' &= P' \cdot \vec{N_2} + det \cdot d_2 \end{aligned}$$

Po vypočtení těchto dočasných hodnot můžeme otestovat zda leží průsečík v trojúhelníku ABC díky následujícím podmínkám znamének dočasných hodnot t', u', v'. Tyto podmínky jsou obdobou podmínek (3.13).

$$\begin{aligned} sgn(t') &= sgn(det - t') \\ sgn(u') &= sgn(det - u') \\ sgn(v') &= sgn(det - u' - v') \end{aligned} \tag{3.17}$$

Pokud průsečík leží v trojúhelníku, můžeme vypočíst výsledné hodnoty t, u, v čímž dostaneme potřebné informace v podobě vzdálenosti průsečíku od počátku paprsku a jeho polohy v trojúhelníku vyjádřené barycentrickými souřadnicemi:

$$\begin{aligned} t &= \frac{1}{det} \cdot t' \\ t &= \frac{1}{det} \cdot u' \\ v &= \frac{1}{det} \cdot v' \end{aligned} \tag{3.18}$$

Tento výpočet lze převést do instrukční sady SSE4.1, výsledný kód je pak prezentován v článku Havla a Herouta [8] a postup vytvoření podobného kódu v článku Shevtsova [17].

Kapitola 4

Analýza výchozího řešení a možné optimalizace

Nejdříve se podíváme na výchozí program, ze kterého jsem vycházel, jeho možnosti a implementaci. Poté ve zkratce popíšu použitý nástroj pro analýzu programu, díky kterému je možné změřit a vyhodnotit nejnáročnější části zkoumané implementace projektu. Po vyhodnocení výstupních dat z analyzátoru následuje popis předpokládaných slabých míst a návrh optimalizací a úprav, které by mohly vést k podstatnému zrychlení. Popis těchto vylepšení bude následovat v dalších podkapitolách.

4.1 Výchozí řešení

Tato práce navazuje na mou bakalářskou práci [18] a tedy i program, který byl k ní navržen a implementován. V této části se zaměřím na schopnosti výchozího řešení, jeho návrh a implementaci.

Schopnosti

Nyní popíšu bodově co obsahuje a umožňuje simulovat samotné původní řešení:

- Phongův osvětlovací model.
- Podporuje bodová a plošná světla (pouze obdélníky).
- Odrazy paprsků v rekurzivním volání.
- Pohyblivé úseky kamery definované Beziérovými křivkami, bez předpočítávání poloh.
  - Při potřebě získat polohu a nastavení kamery se musí z Beziérových křivek zjistit poloha, směr a nastavení kamery.
- Scénou, potažmo, při hledání nejbližšího průsečíku se prochází seznam těchto objektů iterativně.
- Podporuje základní typy těles a jejich jednoduché kompozice (koule, roviny, plochy trojúhelníky, válce, obdélníky, kvádry).
- Jednoduché pohyblivé objekty (pouze u koulí).

  - Pohyb dán pomocí silových impulsů a následného vzorkování rozdílových kvaternionů celé fyzikální simulace, při které se vzorky ukládají do seznamu pro každý pohyblivý objekt.
  - Při hledání potřebné polohy se pak seznam celý projde a pro daný čas se najdou dva nejbližší vzorky, z nich se pomocí lineární interpolace vytvoří výsledný a teprve podle něj se vypočte poloha vybraného objektu.
  - objekty mají pevné, nenastavitelné fyzikální vlastnosti.

- Umožňuje načítání scény a jejího nastavení ze souboru ve vlastním formátu.

- Efekty distribuovaného ray tracingu (Motion blur, hloubka ostrosti, měkké stíny, matné materiály, průsvitné materiály).

  - Zanořené volání jednotlivých efektů. Odpovídá vnořenému řešení metody Monte Carlo.

- Motion blur a simulace závěrky kamery:

  - Simulace závěrky kamery s úseky otevírání a zavírání.
  - Vzorkování pomocí metody odmítnutí (generování vzorků v obdélníku vybírání jen těch co jsou ve vnořeném 5ti úhelníku).
  - Ve vzorkovaných časech se spouští výpočet hloubky ostrosti.

- Hloubku ostrosti a simulace čočky kamery.

  - Simulace čočky pomocí generování paprsků z kolmo umístěného kruhu vůči paprsku.
  - Generování paprsku pomocí 2 úhlu (úhel od referenčního paprsku, úhel v kruhu. (rovnoměrné a normální rozložení).
  - Matematicky nesprávná implementace (kruh má mít normálu ve směrovém vektoru kamery, ne paprsku).

- Matné a průsvitné povrchy.

  - Generování rekurzivních paprsků pomocí normálního rozložení kolem dokonale odraženého/lomeného paprsku.
  - Při "inspirování se" pro vývoj jiného projektu jsme zjistili, že lomené paprsky jsou chybně implementovány a lámou se od normály špatným směrem.

- Měkké stíny

  - Rovnoměrné vzorkování plošného světla (pouze obdélník) metodou Monte Carlo. Výsledná barva se získá z paprsku mířícího přesně doprostřed a poměru dopadajících paprsků vůči celkovému počtu.
  - Opět dostačující, ovšem ne úplně správná metoda.

4.1.1 Popis implementace

Implementace ray traceru byla kompletně napsána v C++ za pomoci standardních knihoven, STL knihoven a knihovny pro fyzikální simulaci ODE (Open Dynamics Engine). Program je napsán konkrétně ve standardu C99 pro překladač GCC (G++) což ho umožňuje přenášet mezi platformami, pro které existuje překladač podporující tento standard.

Program je psán v maximální možné míře objektovým návrhem, který usnadňuje orientaci v projektu a umožňuje snazší rozšiřitelnost, než by tomu bylo v případě jazyka C. Přitom zajišťuje dostatečnou efektivitu kódu. Celá implementace je navržena jako sada služeb a podpůrných komponovaných prvků.

Většina služeb je potřeba v celém programu jen v jedné instanci a proto jsou implementovány za pomoci globálního návrhového vzoru Meyersového singletonu. Ten je zároveň při dobrém návrhu chrání proti chybnému přístupu, aniž by u nich bylo třeba složitého komponování. Podpůrné prvky jsou pak implementovány pomocí standardních komponovaných tříd. Tento přístup zjednodušil návrh celého programu, správu komponent a zabezpečuje také před hůře odhalitelnými vnořenými chybami.

Mezi základní stavební prvky složitějších komponent patří:

- CVector implementující 3D vektory a operace s nimi.
- CRay jako abstrakci paprsků.
- CColor jako abstrakce barvy/světla.
- UV třída představující uv souřadnice textur.

Mezi používané globální služby se řadí generování náhodných čísel (CGen), projekční plocha (CColorBuf) umožňující uložit obrázek na disk, logování programu (CLog), jednotné nastavení (CSettings) ray tracingu a parsování scény (CParser) pro načítání ze souboru.

Hlavní služba (CEngine), pak zajišťuje samotný rendering - výpočet distribuovaného ray tracingu, povely pro scénu a další podstatné kroky. Potřebuje na vstup scénu (CobjectGroup), seznam světel (ClightSet) a seznam kamer (CCameras). CObjectGroup je seznam všech objektů odvozených od obecného objektu (třídy CObject). Obsahuje metody na procházení a vyhledávání nejbližších objektů ve scéně a jejich správu. Objekt (CObject) abstrahuje fyzikálně interaktivní geometrické tělesa a práci s nimi. Každý objekt má přiřazený materiál (CMaterial), seznam hybných impulsů a potřebná rozhraní pro přístup k nim. Konkrétní výpočty, jako jsou výpočet průsečíku s paprskem, získání UV souřadnic z bodu na povrchu apod. vztahující se pouze k danému typu objektu (např. koule, kvádr, válec) jsou implementovány ve specializacích objektu. Materiály jsou koncipovány podobně. Každý materiál obsahuje jednu či více informací o vlastnostech materiálu (CPhong) nutných pro výpočet osvětlovacího modelu. Seznam všech světel (ClightSet) obsahuje jednotlivé světla (CLight) pro scénu. Ty jsou třeba pro výpočet osvětlovacího modelu ve scéně. Světla jsou také specializována na bodové (ClightPoint) a plošné obdélníkové (ClightRect). Seznam kamer (CCameras) zajišťuje vybrání té správné kamery pro určitý čas a její správné nastavení (poloha, otočení). CCamera je abstrakce kamery, obsahuje trasu pohybu a cíl záběru po Beziérových křivkách. Obsahuje pokročilé mechanismy kontroly nastavení.

4.2 Analýza

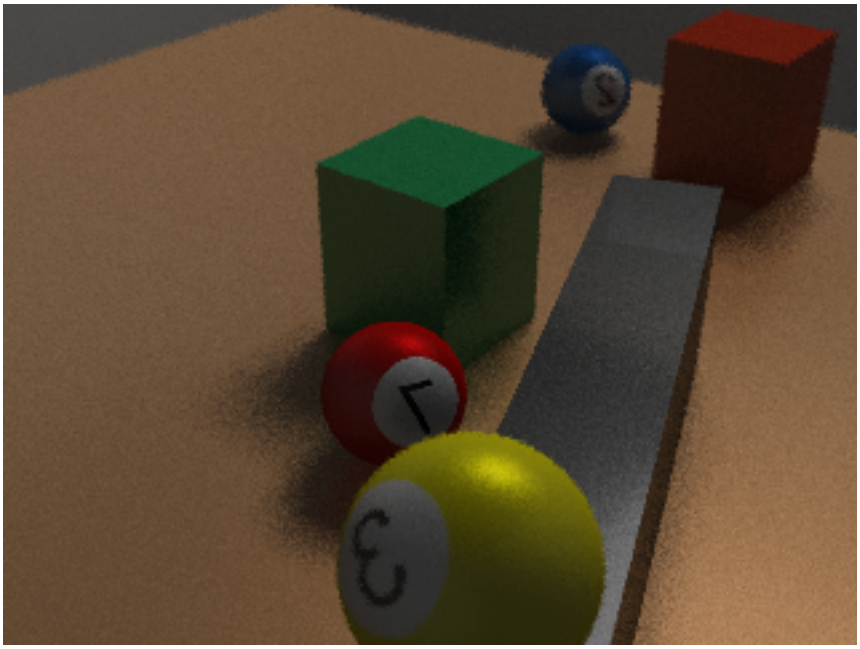
Pro analýzu programu bylo nutné nejdříve vytvořit určitou scénu, na které bude dobře pozorovatelné, které části programu budou nejpomalejší.

Tato scéna se složena z:

- 100 trojúhelníků,
- několika koulí,
- několika ploch,
- 1 plošným obdélníkovým světlem

a byla renderovaná s nastavením:

- rozlišení obrazu 320 × 240
- hloubka ostrosti 6 vzorků,
- měkké stíny 3 vzorky,
- matné materiály 3 vzorky,
- rekurze 1 zanoření



Obrázek 4.1: Testovací scéna

Scéna obsahuje všechny efekty až na motion blur a mohla by při testování určit představu, které části programu bude třeba optimalizovat.

4.2.1 Vliv kompilátoru

V této analýze jsem zjišťoval jaký vliv má na výpočet kompilátor, kdy jsem nejdříve hledal vhodné kandidáty pro porovnání s tím že bylo nutné, aby kompilátor byl co nejvíce aktuální

a volně dostupný. Z nejznámějších tomuto vyhovují kompilátory GNU Compile Collection (GCC) a Microsoft Visual C++. Jako další možný se jeví kompilátor od společnosti Intelu, ten ale není volně dostupný a proto jsem jej nepoužil.

Poté jsem program zkompiloval pro oba zmíněné kompilátory vícekrát s různým nastavením a následně nechal v každém z nich vyrenderovat scénu popsanou v předchozí kapitole.

Vyzkoušel jsem jaký vliv má změna přesnosti čísel s pohyblivou řadovou čárkou a také vliv nastavení ohledně optimalizací rychlosti kompilátorů. Konkrétně O2 – optimalizace pro rychlost a velikost a O3 (Ox u VS C++) - optimalizace pro maximální rychlost na úkor velikosti. [1] Výsledky měření jsou v následující tabulce:

320x240, 1 světlo, vzorky: DOF 6, stíny 3, matnost 3, rekurze 1				
plošné světlo	přesnost čísel	nastavení optimalizace	čas renderování scény (min.)	
			GCC (G++) v4.41	Visual C++ 2008
ne	float	O3 (Ox)	0:45	0:38
		O2	1:08	0:39
	double	O3 (Ox)	0:52	0:35
		O2	1:22	0:35
ano	float	O3 (Ox)	1:29	1:14
		O2	2:14	1:14
	double	O3 (Ox)	1:43	1:08
		O2	2:44	1:07

Tabulka 4.1: Doby výpočtů testovací scény na různých kompilátorech

Z výsledků vyplývá, že kompilátor má určitý vliv na výkon. Ovšem cena použití kompilátoru od Microsoftu by byla příliš vysoká, tedy ztráta určité míry přenositelnosti, v porovnání s přínosem o něco málo rychlejšího běhu programu. Navíc toto měření proběhlo na počátku, před implementací optimalizací a v tu dobu nebylo jisté, zda by změna kompilátoru měla ve výsledku až takový přínos. Proto jsem od ní časem upustil a vrátil se k přenositelnému GCC.

4.2.2 Profilování programu

Profilování je metoda sloužící k podrobné dynamické analýze programu. Díky profilerům, jak se profilovací nástroje nazývají, je možné provést analýzu kódu programu a z výsledků pak zjistit velmi užitečné informace, jako je např. procentuální časové zastoupení jednotlivých úseků kódu (například funkcí) při běhu programu. Díky této analýze, je pak možno určit časově nejnáročnější části a ty se pak pokusit optimalizovat. [2]

Při kompilaci pomocí kompilátoru GCC je možné předat parametr *-pg*, který způsobí, že se program přeloží tak, aby pak sám "sbíral" při svém běhu tyto profilovací data.

Data se po každém skončení běhu programu uloží do binárního souboru s příponou *out*. Pro člověka jsou samy o sobě informačně nepoužitelné a je nutné si z nich informace o běhu

[1] Tyto parametry se mohou implementačně i obsahem konkrétních optimalizací lišit, proto mohou být výsledky zavádějící. V jiných, než uvedených kompilátorech nemusí být dostupné.

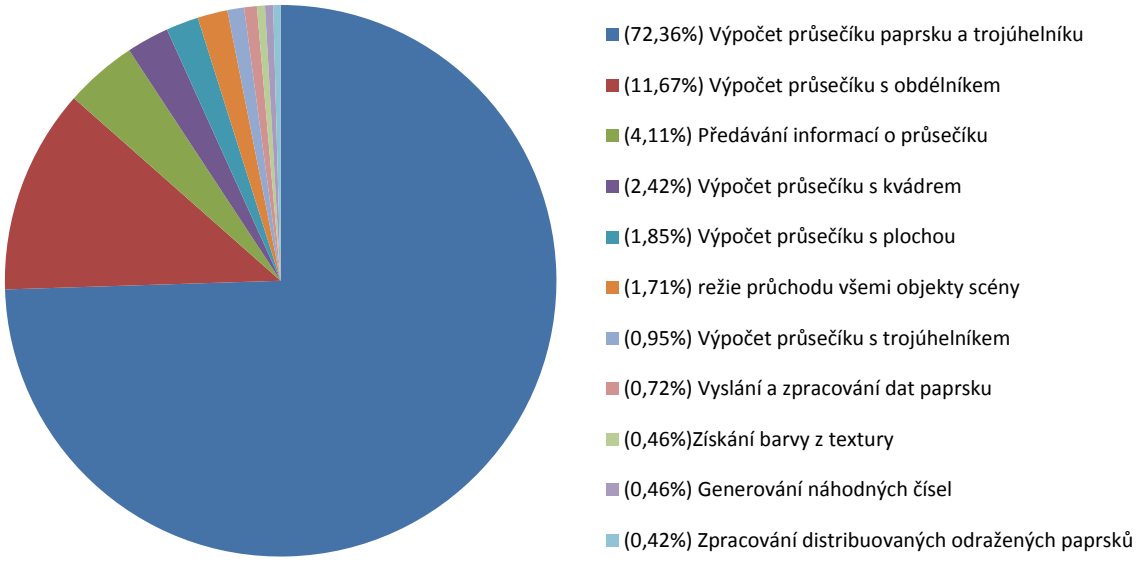
[2] Tato kapitola se bude zabývat jen profilovacími nástroji GCC kompilátoru.

nechat vypracovat pomocí dalších nástrojů. Jeden z nich je dodávaný přímo s kompilátorem GCC a jmenuje se The GNU Profiler. Tento nástroj je dodávaný ve formě programu *gprof*. Informace pak lze získat jeho spuštěním, kdy se mu předají parametry: Název analyzovaného programu, název souboru s profilovacími daty a další volitelné, které určují jaký bude statistický výstup. Statistické výsledky se pak uloží do souboru *profile.txt*. V něm lze nalézt podle předaných parametrů například časy běhů jednotlivých funkcí.

Toto jsem provedl pro běh programu ray traceru s testovací scénou a nechal vygenerovat statistické data.

Pro ověření jsem program zkompiloval také pod Visual C++ překladačem a analýzu provedl pomocí programu CodeAnalyst od společnosti AMD. Výsledky byly téměř identické a rozdíly byly v rámci chyby měření/odlišnosti kompilátoru. I přesto mohou být data zkreslené optimalizacemi, kdy se například řádkově vložené funkce nemusí jevit samostatně, ale jako součást jiné funkce. Pro hrubou představu to ovšem stačí.

Provedl jsem analýzu, ve kterých částech programu se tráví nejvíce času. Analýza dopadla následovně:



Obrázek 4.2: Výstup analýzy programu

Je nutné vzít na vědomí, že některé funkce se díky optimalizacím překladače nemusí při profilování objevit, i když se v nich tráví mnoho času samo o sobě velmi náročné.

4.3 Analýza "pomalosti" řešení a návrh možných řešení

Z výsledků profilování vyplývá, že se celých 72% času program počítá průsečíky s trojúhelníky, což předurčuje tuto funkci jako kandidáta číslo 1 k optimalizaci. K tomu nám poslouží **rychlý test průsečíku paprsku s trojúhelníkem**.

V tuto chvíli je dobré se zamyslet proč se téměř 95% času tráví výpočtem průsečíků s objekty. Pro každý paprsek, pro který se hledá nejbližší průsečík ve scéně se pokaždé znovu iterativně prochází všechny objekty scény a jednoduchým porovnáváním se hledá nejbližší průsečík. Časová náročnost takového prohledávání s počtem objektů bude minimálně lineárně stoupat. Ve skutečnosti může vzhledem k režii, při správě dat v paměti, stoupat

daleko rychleji. U větších scén to bude znamenat, že se prakticky celý čas jen vyhledávají průsečíky se scénou a ostatní výpočty budou dělat jen malý zlomek z celkové doby výpočtů. Takový růst času renderingu v závislosti na složitosti scény je pro rychlé a hlavně prakticky využitelné renderování nepoužitelné. Je tedy nutné provést optimalizaci procházení scény při hledání nejbližšího průsečíku. Jako ideální se z momentálně známých metod vyhledávání v 3D prostoru jeví **hierarchické dělení scény pomocí kd-stromu** . Ten by mohl díky svým vlastnostem přinést podstatné snížení časové složitosti vyhledávání nejbližšího průsečíku.

Z grafu také vyplývá, že předávání výsledku intersekce v podobě komplexní třídy CHit je časově dosti náročné na to, že se jedná jen o pouhé předávání výsledků. To bude způsobeno tím že se vždy předávají informace o průsečíku, i když se žádný nenajde. Ve značné části případů se tak předávají tyto informace zbytečně. Bude tak určitě přínosné, pokud budou **předávány pouze výsledky, když dojde k průsečíku**.

Také při hledání nejbližšího průsečíku se s každým testem vypočtou informace o průsečíku a ty se předají, teprve potom se zjišťuje, jestli nový průsečík je blíže, než současný nejbližší. Tento **test na blízkost průsečíku** je možné provádět již během výpočtu průsečíku, kdy např. u trojúhelníků se nejprve najde jak je vzdálená rovina ve které leží a až poté se zjistí, jestli průsečík leží v trojúhelníku a případně se vypočtou další informace o průsečíku.

Intersekce obdélníku, kvádru a obecně všech objektů složených z jednodušších zabírá příliš mnoho prostředků vzhledem k tomu, že se jedná jen o objekty, které volají testování trojúhelníků, případně jiných komponovaných objektů, na průsečíky. Např. kvádr volá testování pro 6 obdélníků a ty dále pro 2 trojúhelníky. Bude tedy určitě vhodné toto **zbytečné zanořování objektů odstranit**. Jednou z nejlepších cest pak je implementace obecné **sítě trojúhelníků** jako univerzální reprezentace všech možných objektů. Tímto také odstraníme zanoření komponovaných objektů.

To můžeme pojat obecněji. Pokud je potřeba při testu na průsečík paprsku a nějakého objektu potřeba jen geometrických znalostí o objektu a přitom jsou v objektu informace o materiálu, animaci, případně více takových objektů zanořených do sebe, bude výhodné odstranit tuto časově náročnou kompozici a **oddělit tvar od ostatních vlastností objektu**. Zmenší to tak velikost struktury potřebné pro výpočet průsečíku, urychlí vyhledávání nejbližšího průsečíku, tak také zjednoduší rozhraní a umožní další optimalizace týkající se animací.

Více informací se z časové analýzy programu nedovíme a musíme intuitivně hledat další místa k možné optimalizaci sami.

Již jen z principu fungování metody Monte Carlo u distribuovaného ray tracingu vyplývá, že se počítá s velkým množstvím paprsků. Díky tomu také přibylo velké množství vektorových výpočtů, protože na nich je většina kódu postavena. Nabízí se proto myšlenka urychlit tyto základní **operace s vektory**. V dnešní době podporují moderní procesory čím dál tím více nových instrukčních sad, které se zaměřují na oblasti, kde je třeba zpracovat velké množství multimediálních dat přes identický typ výpočtů (např. konverze videa, počítání transformací v 3D grafice, apod.). K těmto účelům byla např. vyvinuta rozšíření instrukční sady o SIMD [3] **instrukce SSE**, které již nepracují se skalární jednou hodnotou, ale n-ticí hodnot, konkrétně se 4D vektory. Umožňují nad nimi provádět základní aritmetické operace i pokročilejší výpočty.

Velké množství paprsků je také potřeba hlavně pro to, aby scéna nebyla tolik zašuměná

[3] Single instruction multiple data – více dat jednou instrukcí

a tedy aby výsledek nebyl tak rozdílný od ideálního. V tomto směru je velmi nepříznivá rychlost konvergence metody Monte Carlo s pseudonáhodným generátorem čísel. Tady se nabízí možnost použít pro generování náhodných vzorků, **kvazi-náhodné sekvence**, se kterými konverguje Monte Carlo daleko rychleji, i když za cenu určitých artefaktů při menším počtu vzorků pro daný efekt. I přes tyto nevýhody je tato metoda velmi vhodná, protože výsledný obraz neposuzuje stroj, ale člověk a ten vnímá kvalitu obrazu více subjektivně. V tomto směru kvazi-náhodné sekvence představují krok vpřed, protože člověk vnímá v obraze daleko více výrazný šum, než určité, z blízka pozorovatelné artefakty v podobě pravidelných vzorů a mírně skokových přechodů.

Také by bylo dobré počet těchto vzorků omezit pokud například snímáme velkou jednolitou plochu. K tomu nám může posloužit **adaptivní vzorkování**. Existuje mnoho metod. Jednou z nich, nutno poznamenat, že v současné době asi nejpokročilejší je metoda Multidimenzionálního adaptivního samplování [6]. Ta je ovšem velmi náročná a rozsáhlá jak teorií, tak implementací a vyžaduje také poměrně komplikovanou rekonstrukci obrazu. Rozhodl jsem se proto vyzkoušet jednoduché klasické adaptivní samplování podle kontrastu vzorků vůči střední hodnotě. Pro výpočet kontrastu lze použít různé metody, zvolil jsem výpočet Weberova a Michelsonova kontrastu a k tomu možnost místo kontrastu použít jednoduchou diferenci vzorků.

Program navíc není vůbec optimalizován pro více-jádrové výpočetní jednotky. Přitom již dnes se průmysl ubírá směrem, kdy se výpočetní výkon CPU dále příliš nenavyšuje rychlostí, ale zvyšováním počtu jejich jader. K tomu se velmi hodí OpenMP, což je multiplatformní API zajišťující podporu **vícevláknového zpracování** programu a tedy i zpracování na více jádrech procesoru. To vše za pomocí maker preprocesoru. V případě ray tracingu bude nutné kód mírně upravit, ovšem nebude se jednat o až tak velké zásahy do kódu. Spíše půjde o zajištění nezávislých dat. To bude nutné otestovat a ujistit se, že k takovému ovlivňování nebude docházet.

Program je také závislý na fyzikální knihovně ODE, což může mít negativní dopad na výkon programu, kdy každý objekt musí vlastnit také informace o svých fyzikálních vlastnostech. S tím nám přibývá potřeba navrhnout určitý **animační systém**, ve kterém můžeme zadávat objektům jejich polohu, trajektorii a natočení v prostoru. Ten je možné založit na "snímcích" popsaných Beziérovými kubikami a časem. Kubikami pak lze popsat jak samotnou pozici, tak další 2 vektory určující směr a "nebe" pro objekt.

V případě animací objektů a kamer také může být velmi neefektivní pro každý Motion Blur paprsek vypočítávat pozici a natočení z popisu jejich trajektorie. Určitě lepší bude **předpočítat polohu a natočení objektů a kamer** určitým navzorkováním pro interval ve kterém se generuje snímek a ostatní polohy zahodit. Ušetří nám to poměrně náročnou operaci a také nám tím odpadá potřeba vyhledávání správného snímku objektu/kamery, pokud dokážeme z času vygenerovat index v seznamu přímo. Také nám to pomůže u hierarchického dělení scény, protože se tím omezí obalová schránka objektů jen na nejnutnější prostor a ne na celou animaci, která ve snímku stejně není zachycena. Obálky objektů tak budou ve scéně zabírat menší prostor.

S tím také souvisí, že se složitějšími objekty v podobě trojúhelníkových sítí by bylo velmi náročné pro každý časový okamžik transformovat každý z těchto trojúhelníků na novou pozici. K vyřešení tohoto problému nám mohou pomoci **vnořené kd-stromy pro animované objekty**, kde každý animovaný objekt bude mít vlastní pod strom s tělesy a transformovat pak budeme jen jeho obálku. Pokud navíc budeme místo seznamu poloh a natočení objektu uchovávat raději seznam transformací pro parsek, úplně se vyhneme úpravě objektů, což nám zároveň umožní zachovat scénu konstantní. Díky tomu pak při vícevlák-

novém zpracování nebude potřeba pro každé vlákno vytvářet samostatnou scénu a bude možné použít sdílenou paměť pro hledání nejbližšího průsečíku. To nám může výkonnostně velmi pomoci, protože cache procesu nebude tak vytížená.

Také se jeví jako použitelné řešení zvolit **z idealizovanou závěrku**, která se nekonečně rychle otevírá a zavírá. Umožní nám to vypustit odmítání nevhodných vzorků a také výpočet zda se vzorek nachází pod křivkou závěrky či ne.

U generování paprsků u efektu hloubky ostrosti pak také lze u "snímku" kamery (předpočítaná pozice a směr kamery s možností vypočíst vzorek pro určitý pixel a posunutí na čočce kamery) optimalizovat předpočítané hodnoty, tak aby pak již bylo možné **vygenerovat paprsek kamery** tak, aby byl již posunutý jak v místě odkud vychází (na čočce), tak také na projekční ploše. Toho můžeme dosáhnout tak, že projekční plochu umístíme do polohy hloubky ostrosti. Velikost této plochy pak vynásobíme velikostí hloubky ostrosti a v jednom směru poměrem stran obrazu.

U generování odražených distribuovaných paprsků pak bude vhodné implementovat základní **Importance sampling**.

U stínových paprsků pak experimentálně zkusím implementovat vzorkování pouze jednoho světla na jeden výpočet osvětlovacího modelu.

Také by bylo dobré **některé vnořené výpočty sloučit** a odstranit tak režii způsobenou vnořeným voláním. Například distribuované výpočty projekční plochy a časových okamžiků lze díky metodě Monte Carlo sloučit místo vzorkování 2D a následného 1D vzorkování lze vzorkovat rovnou 3D vzorek s tím že matematicky to bude stále validní.

4.4 Seznam vhodných optimalizací a rozšíření určených k implementaci

V předchozích částech jsem provedl profilování výchozího řešení. Na základě toho pak provedl analýzu a navrhl možné optimalizace. Nyní všechny tyto optimalizace shrnu do bodů seznamu. Jsou to zároveň optimalizace, které jsou cílem nové implementace metody distribuovaného sledování paprsku. V další kapitole pak bude následovat popis těchto rozšíření či optimalizací. Některé budou ovšem popsány až v části zabývající se implementací, protože jsou založeny na čistě implementačních záležitostech.

Optimalizace a další rozšíření použité pro tuto práci jsou následující:

- Obecná trojúhelníková síť s možností "vyhlazování" povrchu pomocí interpolace normál jednotlivých bodů.
- Vektorové výpočty s pomocí instrukční sady SSE.
- Rychlý test průsečíku paprsku s trojúhelníkem implementovaný za pomocí SSE instrukcí.
- Výpočet a předávání všech informací o průsečíku pouze v případě, že došlo k bližšímu průsečíku.
- Hierarchické dělení scény pomocí kd-stromu s heuristikou SAH.
- Rozdělení objektu na souhrnné vlastnosti objektu a geometrický tvar. Ten pak bude sloužit pro zjišťování nejbližšího průsečíku.

- Animační systém s předpočítáváním transformací objektů a kamer pouze pro daný snímek. Omezení počtu transformací objekty obsahujícími vlastní kd-strom s trojúhelníkovou sítí.
- Vzorkování pomocí kvazi-náhodných sekvencí.
- Adaptivní vzorkování projekční plochy.
- Vícevláknové zpracování.
- Zjednodušené generování paprsků kamery.
- Možnost vzorkovat jen jedno světlo na jeden průsečík.

Kapitola 5

Řešení a implementace

V minulé kapitole jsme si shrnuli slabá místa výchozího řešení. V této kapitole se pak zaměřím na řešení optimalizačních technik, které jsou použity ve výsledné implementaci. Dále také na strukturu scény, animační systém a generování distribuovaných paprsků. V této kapitole budou popsány implementačně nezávislé techniky. Některé části budou vycházet z původní implementace na kterou aplikujeme nové, lepší postupy.

5.1 Struktura a správa scény

V ray tracingu se dají algoritmy rozdělit na 2 hlavní oblasti. Jedna z nich zajišťuje výpočty osvětlovacího modelu, směry paprsků, odrazy apod. Ta druhá se pak stará o informace pro tyto algoritmy, tedy vlastnosti objektů a světla scény, místa průsečíků paprsků s objekty a další.

V této části bude řeč právě o scéně, její funkčnosti a optimalizacích.

Scéna nám slouží především pro nalezení nejbližšího průsečíku a získávání informací o tomto průsečíku, tedy objektu, na kterém byl průsečík nalezen a jeho vlastnostech, jako je materiál, tvar povrchu a další.

Všechna světla a objekty patří do scény, která je spravuje. Některé objekty mohou být také animované.

- **Světla**, protože jich ve scénách nebývá mnoho, jsou obsaženy v jednoduchém seznamu, který zajišťuje jejich dostatečnou správu. Každé světlo je pak definované barvou a intenzitou, plošná světla pak také tvarem (v této práci jen obdélníková). Tvar světla je zároveň objekt, na který mohou dopadat paprsky.
- **Objekty**, potažmo jejich **tvary** jsou pro velké množství a velmi časté používání obsaženy v kd-stromu.
- **Kamera(y)** přímo do scény nepatří, ale zajišťují pozici a natočení projekční plochy, nastavení doby snímání, délky závěrky, zaostření apod.

5.1.1 Oddělení vlastností objektů a geometrických tvarů a odstranění komponování objektů

Jedním z nedostatků při prohledávání scény v původní implementaci je to, že každé primitivum, či těleso je popsáno jako samostatný objekt se všemi náležitostmi. Obsahuje tak

veškeré potřebné informace jak pro výpočet průsečíku s paprskem, tak také ostatní vlastnosti, jako materiál pro výpočet osvětlovacího modelu a další.

Pokud se pak prochází scéna a testuje se objekt na průsečík, musí se z paměti do cache procesoru nahrát pro každé primitivum celá jeho struktura a tedy i všechny tyto informace o něm. To je ovšem neefektivní, protože informace pro osvětlovací model potřebujeme jen pro nejbližší průsečík scény a představují tak zbytečnou režii neustálého nahrávání informací o objektech, které vůbec nepotřebujeme.

Pokud navíc vezmeme v úvahu, že v původní implementaci objekt většinou obsahuje vnořené objekty (např. kvádr 6 ploch, které jsou složeny dále ze 2 trojúhelníků), pak výpočetní režie takové správy objektů a předávání výsledků intersekce a informací o objektech může představovat nezanedbatelnou část celkového výpočtu generovaného obrazu, což dokazuje i analýza původního řešení 4.2.

Proto jsem se rozhodl pro nové uspořádání informací o objektech scény. Nadále již objekty **neobsahují svůj tvar** (těleso či primitivum). Tvar objektu je nyní reprezentován jedním, či větším počtem základních primitiv (koule a trojúhelníky).

Objekt má samozřejmě stále ke svému tvaru přístup v podobě seznamu těchto primitiv a je v jeho kompetenci je vytvářet, transformovat a dále s nimi manipulovat. Samotný Objekt pak představuje především vlastnosti jako je materiál, pozice tělesa ve scéně, případné transformace a další potřebné vlastnosti. Všechny primitiva objektu pak znají svého vlastníka - objekt.

Pokud pak bude třeba vytvořit například objekt kvádru, nebude se komponovat z dalších objektů, ale bude již jen obsahovat své vlastnosti a 12 základních primitiv (trojúhelníků).

Díky tomuto oddělení již nebude třeba při vyhledávání průsečíku používat kompletní objekty, nýbrž pouze primitiva těchto objektů.

Díky tomu bude možné efektivně používat kD-Strom na vyhledávání, protože ten je koncipován spíše na používání podobně složitých předmětů. Jelikož scény budou převážně tvořeny trojúhelníky, je tento předpoklad naplněn.

Toto řešení a propojení přináší i řadu dalších výhod. Budou popsány v dalších kapitolách. Patří mezi ně například to, že umožňuje učinit všechny tvary scény neměnnými po celou dobu výpočtu a případné transformace primitiv řešit pomocí pohyblivé obálky objektu. Také nám to usnadní implementaci vícevláknového zpracování. O tom ale až později.

Nyní nám jde o to, že toto rozdělení odstraňuje zbytečnou režii při hledání průsečíku a vracení výsledků.

Popis objektů, tvarů a jejich vlastností bude následovat v dalším textu.

5.1.2 Vracení výsledků, jen v případě bližšího průsečíku

Z analýzy také vyplynulo, že režie způsobena neustálým výpočtem výsledků průsečíku i když je již zřejmé, že průsečík je dále než současný nejbližší je poměrně vysoká. Z tohoto důvodu se při výpočtu průsečíku s objektem pokud možno nejdříve zjišťuje jestli je případný průsečík blíže než prozatímní nejbližší. Pokud ano, teprve poté se vypočtou zbylé informace (normála v místě průsečíku, UV souřadnice odkaz na objekt a tvar) a výsledky jsou předány. Pokud případný průsečík je dále, než současný nejbližší, pak se jen ukončí výpočet a žádné výsledky kromě záporné odpovědi se nepředávají. Zdánlivě primitivní úprava ovšem značně ulehčí nárokům na výpočetní výkon. Především v případě optimalizovaném testu na průsečík trojúhelníku. Který odsouvá nejnáročnější výpočtu až po tomto testu.

5.1.3 Geometrický tvar

Geometrický tvar, slouží pro popis grafických primitiv jako je koule, či trojúhelník. V této práci používám pro jednoduchost jen tyto 2 typy. Trojúhelníky lze navíc díky jejich univerzálnosti popsat téměř jakoukoliv scénu. Mnoho těchto tvarů pak popisuje celé těleso reprezentující objekt.

Primitivum má jen pár vlastností/schopností. Jsou to:

- Relativní pozici vůči svému vlastníkovi - objektu .
- Popis tvaru, potřebný pro výpočet průsečíku s paprskem. U koule je to střed a poloměr, u trojúhelníku předpočítané údaje o třech rovinách definujících tento trojúhelník.
- Odkaz na svého vlastníka, tedy objekt.

5.1.4 Objekt a jeho vlastnosti

Obsahuje následující vlastnosti/schopnosti:

- Informace o pozici a natočení ve scéně.
- Informace o materiálu povrchu.
- Seznam všech primitiv,popisující těleso objektu.
- Popis trajektorie polohy a natočení v čase pomocí Beziérových kubik.
- V případě animovaného objektu, objekt obsahuje vlastní kD-Strom do kterého jsou vložena všechna primitiva objektu.
- Obsahuje obálku tělesem *AABB*. Tato obálka obepíná všechny primitiva objektu pro všechny možná posunutí a natočení definované v trajektorii. Popis bude následovat později.

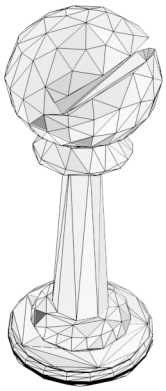
Objekt tedy především sdružuje primitiva a určuje jejich trajektorii. Jeho vlastnosti jsou nám pak užitečné při výpočtu osvětlovacího modelu, ale také při potřebě zajistit rotaci primitiv v čase.

Obálkou pro pohybující se tělesa se budeme zabývat později.

5.1.5 Obecný popis tvaru objektu pomocí trojúhelníkové sítě

Obecná trojúhelníková síť s možností "vyhlazování" povrchu pomocí interpolace normál jednotlivých bodů

Jak jsem již bylo uvedeno dříve, pro vyšší efektivitu je třeba zbavit se zanoření jednotlivých objektů. Toho jsme již docílili oddělením vlastností objektu a tvarů jemu přiřazených. Můžeme tak mít například typ objektu kvádr, který určíme 3 body a výškou a z tohoto popisu jsme schopni vytvořit těleso složené z 12 trojúhelníků. Pro vyšší univerzálnost budeme používat obecný popis tělesa pomocí trojúhelníkovou síť (v angličtině *mesh*).



Obrázek 5.1: Trojúhelníková síť (mesh) - šachová figurka.

Takové těleso se pak dá definovat seznamem vrcholů V a množinou trojúhelníkových ploch P.

Každý vrchol $v \in V$ je pak definován jako $v = (x, y, z, \vec{n})$, kde x, y, z určují polohu bodu v 3D prostoru a $\vec{n}$ je vektor určující normálu povrchu tělesa v daném bodě.

Trojúhelníková plocha $p \in P$ se pak dá definovat třemi vrcholy V_i, V_j, V_k, $p = (V_i, V_j, V_k)$, kde i, j, k značí pořadí (index) určitého vrcholu v seznamu V. Pro trojúhelníkovou síť platí, že každý vrchol musí být součástí jednoho či více trojúhelníků.

Pokud je vrchol součástí jen jednoho trojúhelníku, jeho normála $\vec{n}$ se určí podle normály daného trojúhelníku. Pokud však vrchol patří více trojúhelníkům, pak se normála určí součtem normál všech přiřazených trojúhelníků a normalizováním na jednotkovou velikost. Tyto normály nám mohou posloužit pro opticky vyhlazený povrch tělesa.

Tímto způsobem jsme schopni vytvořit objekt trojúhelníkové sítě, kdy objektu předáme popis tělesa pomocí seznamu vrcholů a seznamu trojic indexů vrcholů a na základě těchto informací pak vytvoříme geometrická primitiva - trojúhelníky. Výhodou popisu je značná univerzálnost, protože tímto způsobem jsme schopni popsat téměř jakýkoliv předmět. Výhodou také je, že předmět můžeme modelovat v 3D grafickém editoru (např. Blender) a mesh z něj exportovat do námi požadovaného formátu a ten poté použít při načítání.

Velkým negativem mesh tělesa je, že pokud popisujeme předmět nehranatého tvaru, a síť tohoto předmětu nemá dostatečný počet trojúhelníků, těleso pak ve výsledném obrazu vypadá jakoby obsahovalo velké množství hran při hranách trojúhelníků sítě. Tento problém se dá opticky, bez navýšení počtu trojúhelníků (a tedy při nenavyšování výpočetní náročnosti) řešit pomocí interpolace normály pro osvětlovací model.

5.1.6 Vyhlazený povrch pomocí interpolace normál

Při tvorbě *mesh* objektu máme na výběr, zda-li bude těleso "vyhlazené" či ne. Pokud vytvoříme "nevyhlazený" objekt tak se při výpočtu normály průsečíku pro osvětlovací model použije normála trojúhelníku. Opticky bude tedy možné pozorovat na hranách sítě ostré přechody osvětlovacího modelu. Pokud jsme sítí popsali hranatý předmět, pak je vše v pořádku. Pokud jsme ovšem sítí popsali předmět neobsahující hrany a síť nebude dostatečně jemná, bude těleso ve scéně vypadat hranatě a bude obsahovat ostré přechody osvětlovacího modelu. Výpočetně je to sice správně, ale nutí nás to k použití velmi jemné sítě, který může prodloužit dobu výpočtu scény.

Z tohoto důvodu je výhodnější použít hrubší trojúhelníkovou a interpolaci normál povrchu. Zjednodušeně řečeno je opticky tvar tělesa stále stejný, ale osvětlovací model je počítán tak, jakoby objekt byl vyhlazený.

(a) při použití normál trojúhelníků

(b) při použití interpolace normál vrcholů trojúhelníků

Obrázek 5.2: Trojúhelníková síť (mesh) a vliv "vyhlazení" na výsledné zobrazení

Na obrázku 5.2 vidíme identický mesh objekt. Ovšem v případě použití vyhlazování normál pomocí interpolace i v případě použití velmi hrubého modelu dosahuje výstup daleko lepších výsledků. Samozřejmě to platí pouze v případě objektů neobsahující ostré hrany.

"Vyhlazená"normála se dá pak získat z barycentrických UV souřadnic průsečíku v trojúhelníku následujícím vzorcem. Vycházíme přitom z popisu obrázku 3.5 za předpokladu, že normály vrcholů A, B, C jsou vypočteny výše uvedeným způsobem pro trojúhelníkovou síť.

$$\vec{N_P} = (1 - u - v) \cdot \vec{N_A} + u \cdot \vec{N_B} + v \cdot \vec{N_C} \tag{5.1}$$

V případě, že vrcholy A, B, C jsou součástí pouze jednoho trojúhelníku, pak výsledná normála bude shodná s normálou tohoto trojúhelníku.

5.1.7 Použití kd-stromu pro vyhledávání nejbližšího průsečíku

Ve výchozí implementaci byly objekty sdružené pouze v jednoduchém seznamu. To je ovšem nevhodné a s narůstající složitostí scény velmi pomalé řešení. Proto použijeme o mnoho rychlejší kd-strom. Jeho primárním cílem je hledání nejbližšího průsečíku ve scéně. Kd-strom pak představuje objektovou část scény a obsahuje v sobě veškeré primitiva objektů. Ty hierarchicky dělí do uzlů. Hierarchické dělení scény jsme si obecně popsali již v kapitole 3.1. Tam lze také nalézt podrobnější informace.

V případě animovaných objektů bude platit výjimka a do stromu nebudou přidány primitiva objektu, nýbrž přímo celý objekt. Takovému objektu pak bude inicializován jeho vlastní kD-Strom, ve kterém budou obsaženy všechny jeho primitiva. Objektu pak bude přiřazena $AABB$ obálka, která bude sjednocením všech obálek vnitřního kD-Stromu, kterých může nabývat při jeho pohybu scénou. Toto si podrobněji popíšeme v kapitole 5.2.2. Na první pohled se toto rozdělení scény může zdát jako nesmyslné, ovšem při správném použití a optimalizaci se jedná o efektivní řešení u animované scény, kdy kD-Strom není zbytečně degenerován velkými obalovými schránkami všech pohyblivých primitiv scény.

5.2 Animační systém

Animační systém je důležitou součástí distribuovaného ray tracingu umožňující simulaci efektu Motion Blur, který je způsoben pohybem objektů ve scéně. Tento pohyb je potřeba nějakým způsobem popsat a zajistit, aby se objekty správně transformovaly dle potřeby v určitém čase. S tím je spojeno mnoho výkonnostních problémů, které je třeba vyřešit. Bez řešení těchto problémů by byl výpočet obrazu neefektivní a se složitější a delší animací opravdu velmi pomalý.

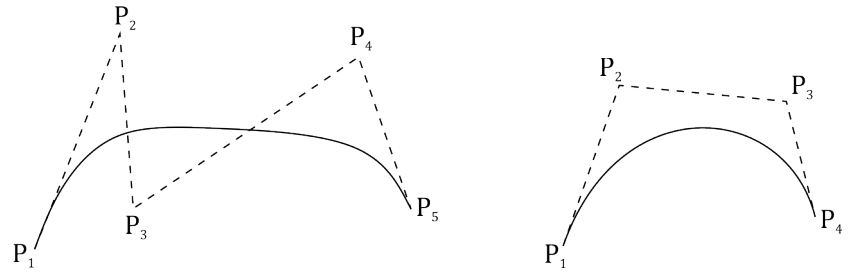
Co se týče animací, je třeba vzít hlavně v úvahu, že námi navrhovaný ray tracer pracuje "offline", tedy že nepracuje v reálném čase a doba renderování scény bude o mnoho delší než časový průběh snímku. Také bude vždy pro jedno spuštění renderovat právě jeden obraz. Na základě těchto 2 hlavních předpokladů budeme navrhovat, přizpůsobovat a optimalizovat celý animační systém.

V této části si popíšeme jednotlivé části animačního systému použitého v této práci a řešení výkonnostních komplikací s ním spojených.

Začneme tím, že si popíšeme jak jsou v této práci animace definovány, poté si popíšeme jak jsou animace zakomponovány do 2-úrovňového kd-stromu scény, Pak si povíme jak můžeme ponechat scénu konstantní pomocí převodu transformací objektu na transformaci paprsku. Dále si popíšeme, jak lze zefektivnit získávání transformací, pomocí navzorkování transformací pro určitý časový úsek a zakončíme tuto sekci tím jak efektivně a rychle potřebnou transformaci pro určitý čas získat.

5.2.1 Definice animací pomocí Beziérových kubik

Díky tomu, že námi navrhovaný ray tracer pracuje offline, budeme potřebovat určitý systém pro popis animací, protože ty nebudou interaktivní. Ve výchozí implementaci byl použit fyzikální systém, který animoval scénu na základě silových impulsů. Tento systém byl vhodný jen jako demonstrační pro určitou scénu. Pro offline ray tracer je totiž spíše nevhodný, protože pomocí něj nemůže člověk přesně definovat animace ve scéně. Z tohoto důvodu jsem se rozhodl navrhnout a implementovat vlastní animační systém založený na obecných Beziérových křivkách v případě kamer a Beziérových kubikách v případě objektů.



(a) Obecná Beziérova křivka (na obrázku řádu n=4)

(b) Kubická Beziérova křivka (řád vždy n=3)

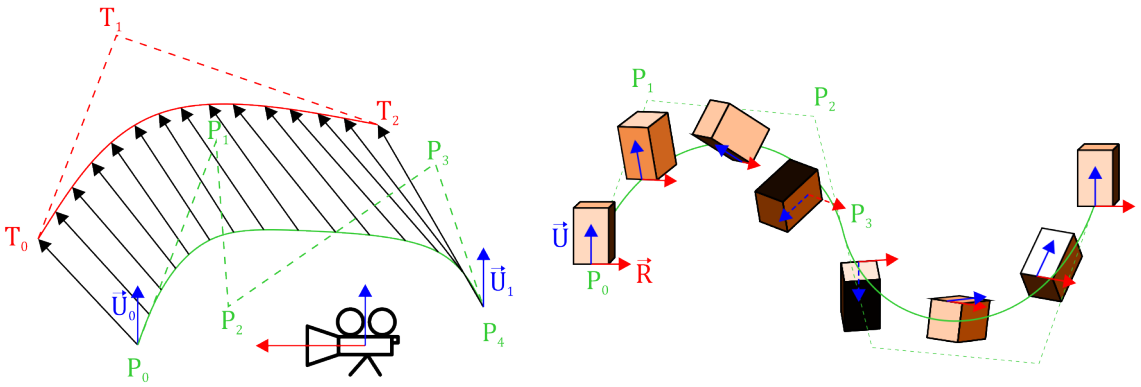
Obrázek 5.3: Beziérovy křivky

Poloha C na obecné Beziérově křivce n-tého stupně na intervalu $t \in < 0; 1 >$ se pak vypočte z následujícího vzorce [1]:

$$C(t) = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} P_i \tag{5.2}$$

kde P_i je řídící bod Beziérovy křivky.

Jiný způsob definice u kamer je především z důvodu odlišné definice a jiných potřeb animací kamery a objektů. U objektů potřebujeme pro definici polohy a natočení v prostoru 1 polohový bod a 2 vektory určující natočení v prostoru. Kdežto u kamery potřebujeme především určit polohu a cíl kamery a vektor "nebe", tedy směr jakoby vzhůru od směru pohledu kamery.



(a) Kamera (b) Objekt

Obrázek 5.4: Obvyklý pohyb po Beziérových křivkách pro kameru a objekt

Na obrázku 5.4 $P_i, i \in \mathbb{N}^0$ značí řídící bod Beziérovy křivky pozice kamery či objektu, $U_i, i \in \mathbb{N}^0$ značí řídící bod Beziérovy křivky vektoru "nebe" kamery či objektu, $R_i, i \in \mathbb{N}^0$ značí řídící bod Beziérovy křivky "pravého" vektoru objektu. $T_i, i \in \mathbb{N}^0$ značí řídící bod Beziérovy křivky cíle kamery.

Animační segment kamery (tedy určité klapky) kamery je pak definován vlastnostmi:

- Seznamem řídících bodů pozice P.
- Seznamem cílů kamery T.
- Počátečním a koncovým vektorem "nebe" U.
- Počáteční a koncový úhel výhledu FOV.
- Počáteční a koncová hloubka ostrosti DOF.
- Počáteční a koncový čas scény pro segment.
- Délka závěrky $BLUR$.

Pohyb a natočení objektu je definován seznamem tzv. "klíčových snímků" objektu.

Klíčový snímek je přitom definován jako půlka Beziérovy kubiky předcházející a následující. Tedy:

- Časem.
- Polohou, vektorem "nebe" a "pravým" vektorem daného objektu pro:
  - Předposlední řídící bod předcházející kubiky.

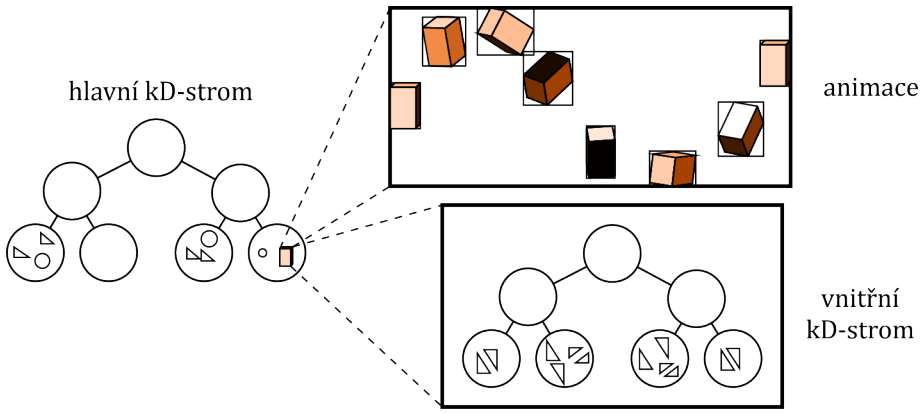
  - Poslední řídící bod předcházející kubiky, který je zároveň roven prvnímu řídícímu bodu následující kubiky.
  - Druhý řídící bod následující kubiky.

přičemž je systém navržen tak, aby nebylo potřeba udávat jednotlivé složky v případě že se nezměnily a také není nutné udávat neklíčové řídící body, pak se jedná o jednoduchou lineární interpolaci vektorů.

Díky tomuto návrhu je popis animace poměrně intuitivní a prakticky použitelný i bez speciálního vizuálního editoru.

5.2.2 Zakomponování animací do 2-úrovňového kd-stromu

Je třeba vzít v úvahu, že animace musíme zakomponovat do struktury scény, tedy kd-stromu. To nám přináší problém v tom, že v případě animace složitého objektu o několika tisících trojúhelnících by bylo třeba pro všechny tyto primitiva zvětšit obálku na prostor, ve kterém se v animaci pohybuje. Takové řešení by bylo ovšem nadmíru neefektivní, protože by narušovalo hlavní výhodu použití kD-Stromů a to, co možná nejlepší rozdělení scény podle pravděpodobnosti zásahu paprsku a tím pádem také nižší časovou složitost. Scéna by se tak v nejhorším případě (kdy by všechny trojúhelníky zasahovaly na své trajektorii do prostoru celé scény) stala opět jakýmsi seznamem, který se musí projít celý a časová náročnost by s počtem objektů opět stoupala lineárně.



Obrázek 5.5: Princip dvou úrovňového kd-stromu

Pro tento problém jsem navrhl použití 2-úrovňového kD-Stromu a to následovně:

- Primitiva všech nepohyblivých objektů jsou standardně nahrána do kd-stromu.
- Všechny animované objekty se vloží do kD stromu jakoby se jednalo o geometrické primitivum.
  - Obálka takového animovaného objektu je tvořena sjednocením obálek všech primitiv daného objektu pro všechny časy scény.
  - Všechny tyto primitiva animovaného objektu jsou vložena do nového kd-stromu, který patří pouze danému animovanému objektu.
- Při dopadu paprsku na animovaný objekt se:

– Vypočte transformace dle zadaných parametrů animace.
– Pomocí této transformace se transformuje vnitřní kd-strom na danou polohu.
– Provede se test intersekce paprsku s tímto vnitřním kd-stromem s tím že se pokračuje totožným prohledáváním jako u hlavního kd-stromu.

Toto na první pohled zvláštní použití kd-stromu přináší na druhý pohled navýšení režijních výpočetních nákladů. Bude třeba vypočíst transformaci objektu pro zadaný čas, transformovat kd-strom a znovu začít procházet kd-strom. Tyto operace ve skutečnosti se nezdají být až tak náročné (možná až na transformaci kd-stromu, pokud se dělá neefektivním způsobem). V distribuovaném ray tracingu bude ale počet těchto operací tak velký, že nároky na toto zpracování budou opravdu vysoké. Stačí si jenom uvědomit, že všechny tyto operace se budou provádět pro každý *motion blur* paprsek.

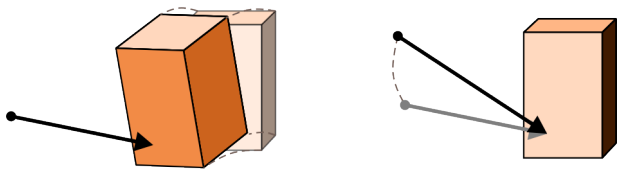
Každopádně toto vylepšení přináší řešení problému co s mnoha primitivy pohybujícího se objektu. Ty jsou umístěny ve vnitřním kd-stromu a dále se manipuluje jen s ním.

I když vypadá použití takové struktury málo výhodné a nepraktické, tak po dalších optimalizacích začne být více než použitelné a hlavně velmi rychlé.

Jednou z těchto optimalizací bude netransformovat celý vnitřní kd-strom, ale pouze paprsek dopadající na obálku animovaného objektu. Další optimalizací bude předpočítání transformací vnitřního kd-stromu namísto zjišťování "na místě". Další na řadě bude optimalizace omezení animací pouze pro daný počítaný snímek a optimalizace objektu za určitých podmínek. Poslední optimalizací pak bude velmi rychlý přístup k požadované transformaci, bez zbytečného vyhledávání v poli.

5.2.3 Převedení pohybu objektů na transformaci dopadajícího paprsku

Pokud získáme transformaci objektu pro určitý čas, není nutné transformovat celý vnitřní kd-strom pomocí rotace R a poté posunu T, nýbrž stačí transformovat pouze dopadající paprsek nejdříve posunem $-T$ a poté rotací $-R$. Operace jsou to, co se týče výsledku, identické. Je ovšem třeba dodržet správné pořadí transformací.



(a) Transformace kd-stromu (b) Transformace paprsku

Obrázek 5.6: Transformace animovaného objektu. Můžeme buď složitě transformovat kd-strom se všemi vnitřními uzly, nebo jednoduše jen paprsek dopadající na obálku animovaného objektu.

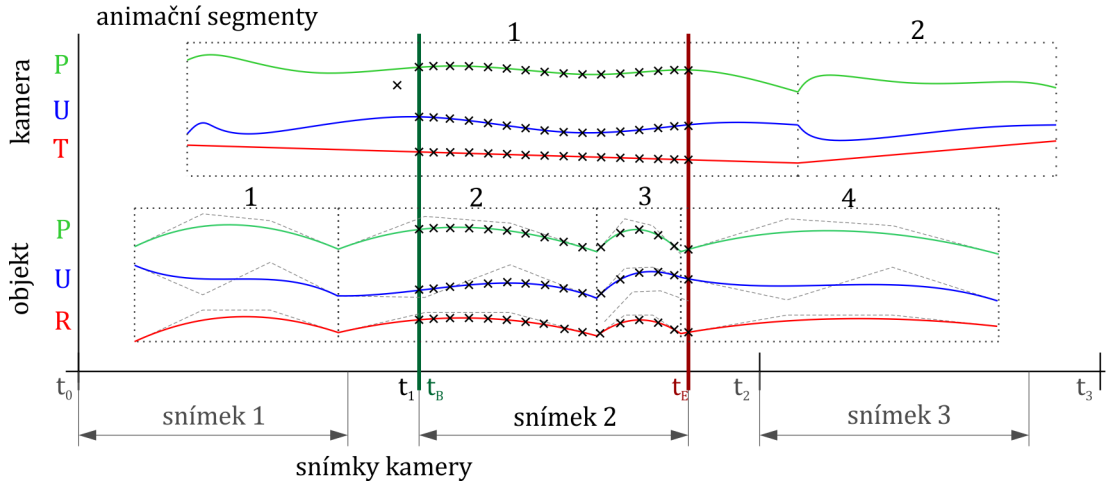
Také je třeba po zjištění výsledku testu průsečíku rotovat dopředu R normálový vektor průsečíku. Pokud bychom předávali jako výsledek také bod průsečíku a nejenom vzdálenost od počátku paprsku, bylo by nutné transformovat také tento bod a to posloupností rotace R a posunu T.

Díky této optimalizaci odpadne časově velmi náročné transformování kd-stromu. Také nám to přináší další velmi pozitivní a později velmi potřebnou vlastnost. A to sice, že celá scéna se tímto stává konstantní. Díky tomu bude o mnoho jednodušší implementovat vícevláknové zpracování.

5.2.4 Předpočítávání transformací objektů a kamer

Animaci tedy máme definovanou pomocí více Beziérových křivek v závislosti na tom zda-li se jedná o objekt či kameru 5.4.

Pro každý vzorek efektu motion blur pak potřebujeme získat transformaci objektu/kamery v prostoru. Tento výpočet je ovšem dosti náročný, protože je u objektu třeba nalézt správný animační segment, vypočíst z Beziérových křivek vektory, z nich pak vytvořit transformace na 3D Beziérově křivce. U kamery pak navíc vypočíst další potřebná nastavení. Přitom je tento výpočet pro určitý čas vždy stejný.



Obrázek 5.7: Vzorkování transformací

Všechny tyto údaje si tedy můžeme předpočítat dopředu a to tak, že si je navzorkujeme pro daný časový úsek snímku kamery kamery s dostatečným rozlišením (např. 2 vzorky na 1ms). Tyto animační vzorky vlastností kamery a objektů pak již nemusíme během každého požadavku na transformaci objetku znovu počítat.

5.2.5 Optimalizace objektů scény dle aktuálních podmínek

Pokud objekt neobsahuje žádné animační prvky, jsou automaticky přidány do kd-stromu scény jeho geometrická primitiva. Pokud obsahuje animaci, pak se do kd-stromu vloží jako objekt s rozšířenou obálkou tak, aby ta obepínala objekt ve všech jeho pozicích a polohách.

Je však zbytečné, aby obálka obsahovala celou animaci objektu, protože naše řešení počítá vždy jen jeden snímek. To můžeme vyřešit tak, že pro všechny animační vzorky objektu před samotným výpočtem obrazu vypočteme obálku a provedeme sjednocení s obálkou s animovaného objektu v kd-stromu. Animovaný objekt pak bude zabírat jen minimum místa a jeho obálka nebude o tolik větší oproti nepohyblivé verzi. Díky tomu podstatně zvýšíme kvalitu kd-stromu).

Dále, pokud je objekt animovaný, ale jeho animace začíná až po konci generovaného snímku, nebo končí před začátkem tohoto snímku, je pak pro daný snímek považován za statický. Tím se ušetří další výpočetní nároky.

5.2.6 Rychlé získávání transformace pro určitý čas

Pokud paprsek narazí v kd-stromu na animovaný objekt, potřebujeme nalézt a aplikovat transformaci na paprsek v čase. Pokud je animace navzorkována dostatečně jemně, není již nutné provádět lineární interpolaci a můžeme použít rovnou nejbližší animační vzorek aniž by se to nějak projevilo na výsledném obrazu.

Pro nalezení toho správného vzorku musíme část pole projít. Tento průchod s testováním na čas také stojí určité výpočetní prostředky a díky tomu, že je hledání průsečíku nejčastější operace, tak poměrně mnoho.

Tomuto procházení se můžeme vyhnout. Provedeme to tak, že převedeme čas počátku snímku t_B a čas konce snímku t_E na normalizované hodnoty 0 a 1. Tedy interně převedeme interval generovaného snímku $< t_B; t_E >$ na normalizovaný interval $< 0; 1 >$.

Díky tomu, že máme pro kameru a pro každý animovaný objekt připraven seznam s n animačními vzorky, které jsou jen v úseku $< t_B; t_E >$ a jsou v pravidelných odstupech, můžeme poměrově rovnou vypočíst index $k \in (1; n)$ požadované animačního vzorku v seznamu pro čas $t \in < 0; 1 >$:

$$k \doteq t \cdot n \tag{5.3}$$

kde $\doteq$ označuje celočíselné zaokrouhlování k nejbližšímu celému číslu.

Díky tomuto kroku již máme velmi optimalizované procházení animované scény v 2-úrovňovém kd-stromu. Nejdříve jsme navzorkovali transformace animovaných objektů pouze pro dobu daného snímku, poté jsme ty, které se pro daný snímek nepohybují označili za statické, dále jsme místo času scény začali používat normalizovaný čas snímku a na závěr vypočítáváme jednoduše index animačního vzorku pomocí.

Díky všem těmto krokům se výpočtu kolem animace objektů redukují jen na načtení transformace z pole a následné aplikace na paprsek a značně tak zredukovali režijní výpočetní nároky pro efekt motion blur.

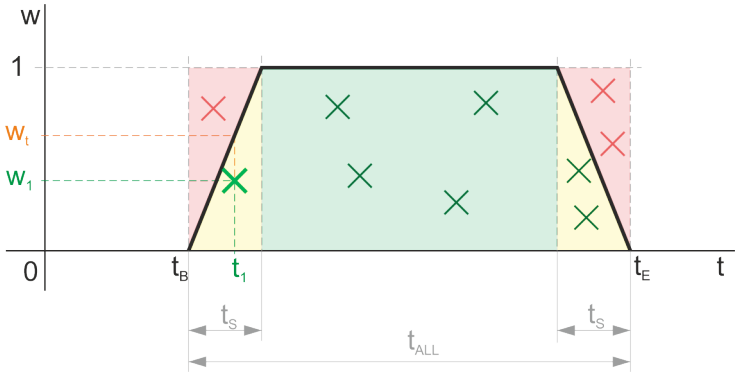
5.3 Distribuované výpočty - generování paprsků

V této části si popíšeme postup výpočtů pro jednotlivé distribuované efekty. Budou zde uvedeny i optimalizace s tím spojené.

Jako první se vzorkuje projekční plocha. Tu v distribuovaném ray tracingu vzorkujeme metodou supersampling. Jak ji vzorkovat jsme si popsali v části 2.6.1. Výpočty dalších efektů jsou popsány dále.

5.3.1 Motion blur

Cílem výpočtů motion blur efektu je simulovat závěrku kamery. Jednoduchou, avšak poměrně reálnou závěrku lze simulovat v podobě lomené úsečky o 3 segmentech, které znázorňují jakou měrou je závěrka otevřená. Simulovat taková závěrka by se dala pomocí rovnoměrného vzorkování plochy a následného použití t složky jako generovaného času. Toho se dá dosáhnout díky použití metody rejection sampling.



Obrázek 5.8: Znázornění simulace závěrky

kde t_B je počátek otevírání závěrky, t_E čas úplného uzavření závěrky, t_S doba otevírání/zavírání závěrky, t_1 náhodný čas vzorku, w_1 druhý rozměr náhodného vzorku pro metodu Rejection sampling a w_t limitní hodnota druhého rozměru vzorku do kdy je ještě vzorek platný.

Toto vzorkování sice není moc náročné, ale pořád představuje určité výpočetní nároky. Navíc se jedná o velmi častou operaci. Proto při výpočtu časového vzorku budeme vycházet z předpokladu dokonalé závěrky kamery, tedy takové, která se v nekonečně krátkém čase otevře a po snímání v nekonečně krátkém čase také zavře. Tento typ závěrky sice neexistuje, protože by musela závěrka mít nekonečně velké zrychlení, ale cílem dnešní kamerové techniky určitě není tento jev vyzdvihovat. Navíc výsledný obraz se téměř neliší proti tomu generovanému s reálnou závěrkou.

Navíc je možné rovnou generovat relativní čas $t \in < 0; 1 >$. Díky tomu se vyhneme převodu na relativní čas, který potřebujeme pro zjištění indexu animační transformace objektu/kamery v seznamu transformací (5.3).

Obrázek 5.9: Simulace závěrky

Postup generování vzorků se pak omezuje na rovnoměrné generování času t_1 v intervalu $< 0; 1 >$ což je i základní interval většiny rovnoměrných generátorů.

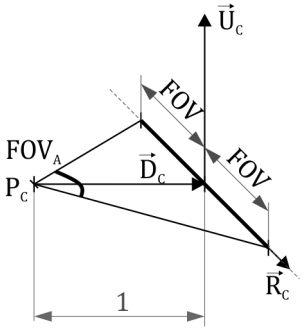
Použití metody pak vypadá následovně pro n vzorků:

- Pro určitý vzorek na projekční ploše budeme generovat časové vzorky.
- Vygenerujeme n náhodných časů $t \in < 0; 1 >$
- Pro každý z nich vypočteme barvu scény (můžeme buď rovnou snímat scénu, nebo dále čas použít pro efekt hloubky ostrosti)
- Výsledné vzorky zprůměrujeme.

5.3.2 Generování paprsků kamery a hloubka ostrosti

O generování paprsků efektu hloubky ostrosti, potažmo i normálních paprsků se stará kamera. Kameru C si můžeme definovat pozicí P_C, "pravým" vektorem $\vec{R_C}$, vektorem "nebe" $\vec{U_C}$, směrovým vektorem $\vec{D_C}$ (cíl kamery se vypočte jako $T_C = \vec{D_C} \cdot DOF$), hloubkou ostrosti DOF, poloměrem čočky c a lineárním zorným polem FOV (lze převést na úhel výhledu FOV_A). Všechny vektory kamery jsou jednotkové.

Zorné pole nám definuje prostor který je kamerou viditelný. lineární zorné pole se definuje jako poměr poloviny viditelné úsečky vůči vzdálenosti od pozorovatele Zorným polem se také ovlivňuje přiblížení



Obrázek 5.10: Zorné pole (FOV)

Zorné pole se z úhlu výhledu vypočte následovně:

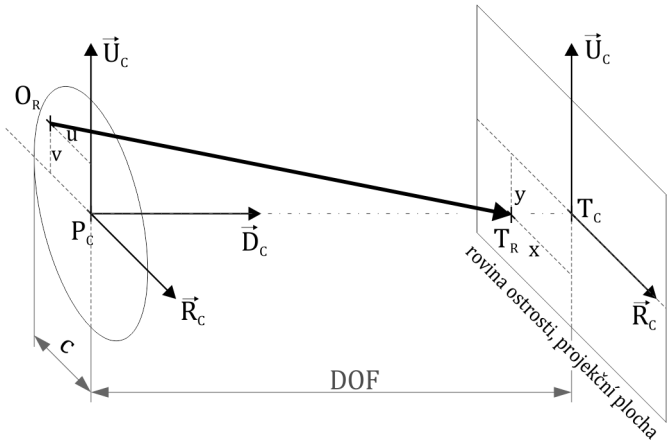
$$FOV = \tan(FOV_A) \cdot d \tag{5.4}$$

kde d je vzdálenost pozorovatele od místa, kde se měří zorné pole (v našem případě $d = 1$). Úhel výhledu se pak dá ze zorného pole přibližně vypočíst pomocí vzorce:

$$FOV_A \approx \frac{360}{2\pi} \cdot \frac{FOV}{d} \quad [^\circ] \tag{5.5}$$

Efekt hloubky ostrosti se dá simulovat při generování paprsku kamery použitím kruhu rozptylu, kdy se již nemusí počítat lom čočkou, ale stačí nám jen poznatek, že bod v prostoru je ostře pozorovatelný pouze pokud leží v rovině rovnoběžné s projekční plochou, na kterou je kamera zaostřena. Pak nám stačí generovat výchozí body $\vec{O_R}$ paprsků v tomto kruhu rozptylu s tím že jejich cíl bude stále stejný (zaostřený bod v prostoru).

V této práci používám optimalizovanou verzi výpočtu, kdy se projekční plocha umístí do roviny hloubky ostrosti. Standardně je vzdálenost kamery od projekční plochy rovna 1. Nyní po zarovnání s hloubkou ostrosti bude vzdálenost DOF. Proto se musí při výpočtu toto zohlednit a koeficient FOV při každém použití vynásobit touto novou vzdáleností.



Obrázek 5.11: Generování paprsků kamery

Pro výpočet výsledné barvy efektu hloubky ostrosti potřebujeme znát jako vstup polohu na projekční ploše (x, y) a počet vzorků n. Obecný postup je následovný:

- Máme výslednou barvu C (zatím čistě černá).
- Vygeneruj n paprsků. Pro každý paprsek R:
  - Vygeneruj náhodnou pozici (u, v) v jednotkovém kruhu (rovnoměrně).
  - Vypočti paprsek R.
  - Vypočti pro paprsek R barvu scény a přičti k výsledné barvě.
- Výslednou barvu poděl počtem vzorků ($C = \frac{C}{n}$)

Pro generování jednoho paprsku hloubky ostrosti R pro pozici (x, y) na projekční ploše a pozicí (u, v) na kruhu rozptylu pak postupujeme:

- Zjistíme počátek paprsku O_R:

$$O_R = P_C + u \cdot c \cdot \vec{R} + v \cdot c \cdot \vec{U} \tag{5.6}$$

- Vypočteme relativní pozici na projekční ploše $\vec{diff}$:

$$\vec{diff} = ratio \cdot x \cdot DOF \cdot FOV \cdot \vec{R_C} + \cdot y \cdot DOF \cdot FOV \cdot \vec{U_C} \tag{5.7}$$

- Z toho následně vypočteme cíl kamery paprsku T_R:

$$T_R = P_C + DOF \cdot \vec{D_C} + \vec{diff} \tag{5.8}$$

- Směrový vektor paprsku D_R pak je:

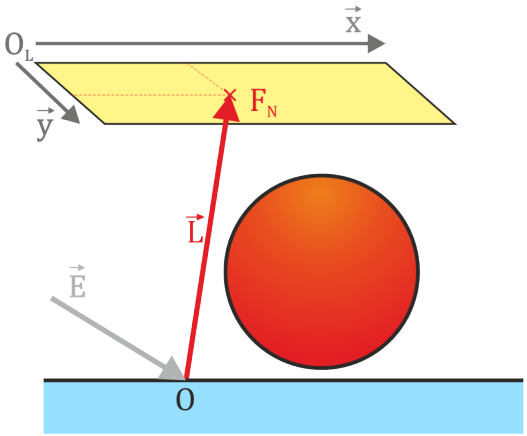
$$D_R = T_R - O_R \tag{5.9}$$

Tento postup se pak dá ještě optimalizovat tak, že se substituuje T_R a O_R a při výpočtu D_R se nám vykrátí některé počty.

5.3.3 Měkké stíny

V distribuovaném ray tracingu se paprsku efektu měkké stíny generují tak, že se rovnoměrně snímá určitá plocha a vypočítá se průměrná hodnota vzorků.

V této práci používám ze světelných zdrojů použitelných pro měkké stíny pouze obdélníkové světla.



Obrázek 5.12: Generování paprsku měkkých stínů

Pro výpočet směru světelného paprsku $\vec{L}$ se pak postupuje následovně:

- Vygeneruj rovnoměrně náhodnou souřadnici $(u, v) \in [0, 1] \times [0, 1]$
- Vypočti pozici v obdélníku F_N:

$$F_N = O_L + u \cdot \vec{x} + v \cdot \vec{y} \tag{5.10}$$

- Vypočti směrový výsledný směrový vektor $\vec{L}$:

$$\vec{L} = F_N - O \tag{5.11}$$

Možnost vzorkovat jen jedno světlo na jeden průsečík

– Pokud máme ve scéně mnoho světel, můžeme pro bod ve scéně namísto výpočtu několika vzorků osvětlovacího modelu pro každé světlo vypočíst jej jen určitý počet vzorků a pro každý z nich použít jedno náhodné světlo pro jedno a vynásobit výslednou barvu počtem světel. Toto řešení je založeno na to, že pokud bychom měli nekonečně vzorků

5.3.4 Sloučení výpočtu efektů

Výpočet zanořených efektů distribuovaného ray tracingu můžeme sloučit do jedné funkce přičemž nemusíme používat vnořené cykly. To nám umožní mírně redukovat režijní náklady. Například tak můžeme sjednotit supersampling, motion blur a hloubku ostrosti. Tím nám vznikne 5 dimenzionální funkce, kdy na vstupu máme pozici na projekční ploše, čas, a uv souřadnice na čočce kamery.

5.3.5 Adaptivní vzorkování

Může se stát, že při renderování budou na určitých místech ve scéně jednolité plochy, např. stěny. Pokud pak budeme mít nastavený velký počet vzorků pro projekční plochu, bude v těchto místech scéna zbytečně převzorkována.

Může použít adaptivního vzorkování pro jednotlivé pixely projekční plochy. To pak funguje tak, že se po určitém počtu vzorků vypočte střední hodnota I_E, a zjistí se kontrast jednotlivých vzorků I proti této střední hodnotě I_E. Pokud přesáhne kontrast C alespoň jednoho vzorku určitou danou mez, tak pak budeme pokračovat dále ve vzorkování, jinak vzorkování předčasně ukončíme a snížíme tím tak výpočetní náročnost.

Existuje více způsobů jak určit kontrast dvou bodů. Zvolil jsem 3 následující, ze kterých se pak dá vybírat. Informace jsem čerpal ze zdroje [25].

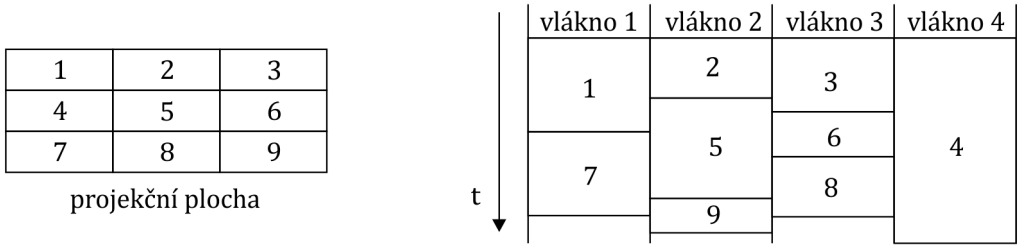
- **Weberův kontrast** - používá se pro určení kontrastu mezi malým prostorem o intenzitě I a velkou jednolitou plochou o intenzitě I_E. Pro tmavá místa i menší změna intenzity znamená poměrně vysoký kontrast. Naproti tomu u světlých míst je kontrast poměrově nižší.
$$C = \frac{I - I_E}{I_E} \tag{5.12}$$
- **Michelsonův kontrast** - používá se pro určení kontrastu mezi určitými místy, přičemž ty jsou zasazeny do členitého prostoru, kde jsou jejich barvy zastoupeny přibližně stejně. Pro tmavá místa již kontrast není tak velký jako u Weberova kontrastu, ale při vyšších intenzitách je kontrast velmi malý.
$$C = \frac{|(I - I_E)|}{I + I_E} \tag{5.13}$$
- **Rozdíl** - představuje absolutní hodnotu rozdílu oproti střední hodnotě. Výhoda tohoto řešení je, že při reprezentaci barvy jako $[0, 1] \times [0, 1] \times [0, 1]$ nejsou světlá místa více zašuměná oproti tmavým.
$$C = |(I - I_E)| \tag{5.14}$$

Adaptivní samplování sebou přináší i jisté režijní náklady. Ty sice nejsou vysoké, ovšem v případě použití u několika zanořených výpočtů by se mohlo stát, že by se těchto režijních informací pro adaptivní samplování muselo počítat třeba i několik desítek miliónů. Navíc je vhodné pro nižší náročnost renderingu zvolit vyšší počet vzorků pro obraz (kde je počet pixelů a také vzorků předem znám), než pro zanořené efekty. Adaptivní samplování by tak u zanořených efektů (např. odrazů/lomů, světel, a možná také u hloubky ostrosti) většiny renderovaných scén postrádalo smysl.

5.3.6 Vícevláknové zpracování

Výkon počítačů se dnes již významně nezvyšuje pro 1 jádro procesoru, ale za to se zvyšuje počet jader v procesorech. V roce 2011 tak patří k středně výkonným procesorům 4 jádrové procesory, umožňující zpracování až 8 vláken, tedy 2 vlákna na jádro. Z tohoto důvodu je nutné pro plné využití výkonu procesoru úlohu ray tracingu paralelizovat.

To provedeme tak, že zavedeme více vláknové zpracování renderingu, kdy se výsledný obraz renderuje po určitých sekcích v jednotlivých vláknech.



(a) Rozdělení plochy na sekce pro vláknové zpracování (b) Vícevláknové zpracování

Obrázek 5.13: popisek

Pro možnost paralelního zpracování je ovšem nutné dodržet určité zásady:

- Nezapisovat do stejné paměti z více vláken. Toto je nutná podmínka, která zaručuje, že paměť bude validní. Kdyby totiž mohlo více vláken zapisovat do stejné paměti a zároveň by z ní čerpaly informace, ve výsledném strojovém kódu na časové ose by instrukce pracující z touto pamětí byly proházené. S největší pravděpodobností by nastala situace, kdy jedno vlákno by si načetlo paměťové místo, druhé by mezitím toto místo přepsalo a první pak změnilo hodnotu a zapsalo změnu do paměti. První vlákno by tak nerespektovalo změny způsobené druhým vláknem. To by pak vedlo k chybnému a nepředvídatelnému chování.

- Pokud je třeba paměti, ze které se simultánně čte a zapisuje z více vláken (např. informace o průběhu renderování), je nutné zavést řízení vláken, kdy jen jedno vlákno může v jednom čase přistupovat k dané paměti. Tedy je nutné zavést atomičnost operací.

- Zajistit, aby další vlákna nepracovaly s pamětí, která je platná jen pro jedno vlákno.

Díky optimalizaci animací a scény, kdy jsme zavedli transformace paprsku namísto objektů pro určitý čas jsme tímto učinili scénu konstantní po dobu výpočtu. Díky tomu není nutné scénu kopírovat pro jednotlivé vlákna a ty pak ke scéně mohou přistupovat validně paralelně. Máme tak elegantně vyřešen problém animací scény i pro více vláken.

Jelikož si generátory pseudo-náhodných čísel ukládají vnitřní stav a tedy mění určitou paměť pro výpočet dalšího čísla, je nutné pro každé vlákno vytvořit novou instanci generátoru. Pokud bychom tak neučinili, generátor by již neprodukoval dané rozložení.

Jediná část, která může potřebovat sdílenou paměť s řízeným přístupem může být informace o tom jaká část scény je již vygenerovaná. Tady používáme jednoduchý čítač počtu již vyrenderovaných pixelů. K paměti, kde se nachází jeho vnitřní stav pak budeme přistupovat tak, že pokud vlákno začne používat danou paměť označí ji jako používanou, přičemž při každém přístupu se kontroluje, jestli již danou paměť jiné vlákno nepoužívá.

Další část společná pro všechny vlákna je výsledný generovaný obraz. Každé vlákno ovšem přistupuje pouze k určité sekci a proto problém nevalidního přístupu nenastává.

Scénu můžeme rozdělit na určitý počet sekcí. Vláknové zpracování je pak řešeno tak, že každé vlákno dostane přiřazen určitý počet sekcí (např. jen 1), tu zpracuje a zažádá si o další. Pokud sekce dojdou, vlákno svou činnost ukončí.

5.4 Návrh programu a jeho implementace

V této části si popíšeme implementačně závislé optimalizace a samotnou implementaci programu, který zpracovává metodou distribuovaného ray tracingu určitou scénu. Nejdříve si popíšeme některé další problémy, které se vztahují k optimalizacím, dále v čem jsem implementoval celý program, jaké nástroje jsem k tomu použil. Následně si lehce popíšeme implementaci programu. Také uvedu jak se program ovládá.

5.4.1 Vícevláknové zpracování pomocí OpenMP

Vícevláknové zpracování tak jak je popsáno v 5.3.6 je řešeno samotném programu pomocí soustavy direktiv překladače OpenMP [23]. OpenMP je standard pro programování počítačů se sdílenou pamětí. Následovat bude jak jsem řešil jednotlivé úkony pomocí této sady direktiv.

Vícevláknové zpracování sekcí projekční plochy

Pro paralelizaci jsem použil vícevláknové řízené zpracování. Tomu určíme kolik vláken se pro výpočet má použít, určí se počet sekcí a počet naráz přidělených sekcí jednomu vláknu. Pak je jednotlivým vláknům přidělován daný počet sekcí a ty se zpracovávají. Ve chvíli, kdy se počet zbývajících sekcí zmenší pod určitý počet, řízené zpracování způsobí, že se bude přidělovat menší počet sekcí jednomu vláknu. Díky tomu se částečně vyhneme situaci, kdy na konci vykonávání zůstane pracovat jen jedno vlákno. Kód v C++ OpenMP pro řízené paralelní zpracování pak přibližně vypadá následovně:

```
*scene; // scéna se světly a objekty
*camera // kamera pro generování paprsků
&stopWatch // stopky a také zobrazovač průběhu renderingu

// direktiva zajišťující paralelní zpracování s určitým počtem vláken
#pragma omp parallel num_threads(počet_vláken)
{
    // direktiva, která paralelně rozdělí cyklus
    // do vláken pomocí řízeného zpracování
    #pragma omp for schedule(guided, počet_sekcí_na_vlákno)
    for (int i = 0; i < celkový_počet_sekcí; i++)
    {
        // měřič postupu renderování, který vypisuje
        // průběžné výsledky pouze v prvním vlákně
        StopWatch* pStopWatch = 0;
        if(omp_get_thread_num() == 0){
           pStopWatch = &stopWatch;
        }
        // Inicializace jedné sekce pro renderování
        RenderArea rArea = RenderAreaCreator::In().GetRenderArea(i);
        // Generátor náhodných vzorků
        Sampler * sampler = Samples::In().GetNewSampler();
        // Metoda distribovaného ray tracingu
        Engine engine = Engine(sampler);
```

```
            // samotné generování dané sekce
            engine.DistributedRender(*scene,*camera, rArea, pStopWatch);
        }
    }
```

Zajištění řízeného přístupu k datům

Atomické zpracování jednodušší a kratší části kódu se řeší pomocí direktivy `omp atomic`. Ta zajistí, že daný blok bude vždy vykonávaný pouze v jednom vláknu. Příklad pro použití čítače hotových pixelů projekční plochy je následovný:

```
#pragma omp atomic
citac += prirustek_hotovych_pixelu;
```

5.4.2 Vektorové operace pomocí SSE instrukcí

Jak jsme si popsali v kapitole 3.3 můžeme nahradit skalární výpočty vektorových operací pomocí přímých vektorových výpočtů na podporovaných procesorech, jako třeba výpočet skalárního součinu. To může přinést značné zrychlení. Konkrétně u operace vektorový součin je třeba pro dva 3D vektory U a V provést výpočet $U_x*V_x+U_y*V_y+U_z*V_z$, což představuje 5 aritmetických operací a tedy v podobě programu také 5 instrukcí. Po převedení této operace do SSE instrukcí pak bude představovat ve výsledném kódu pouze jednu instrukci. Toto zrychlení nemusí být právě 6×. Některé instrukce se totiž mohou vykonávat déle než jiné a také práce s SSE jednotkami přináší určité režijní náklady. Dále budeme uvažovat jazyk C++.

Pro zakomponování SSE intrinsiků do výpočtů je nutné pro začátek připravit strukturu dat pro danou třídu Vector. To by se dalo zajistit jednoduchým nahrazením 4 skalárních čísel jednou proměnnou typu `__m128`. Tím bychom ale zároveň přišli o možnost přímého přístupu k těmto skalárním složkám vektoru. Situace se dá vyřešit použitím sjednocení proměnných (union). Na místo původních složek vektorů:

```
float x, y, z;
```

Lze použít řešení:

```
union {
    __m128 m128;
    struct {float x, y, z, w;};
    float raw[4];
};
```

ve kterém je zachován přímý přístup ke složkám vektoru a zároveň je možné využití datového typu pro SSE `__m128` bez zbytečné režie. Nyní již stačí jen nahradit jednotlivé výpočty SSE ekvivalenty v daných funkcích. Jako příklad uvedu náhradu funkce skalárního součinu 2 vektorů. Původní kód:

```
inline friend float Dot(const VectorFPU & vInA, const VectorFPU & vInB)
{
    return vInA.x *vInB.x + vInA.y *vInB.y + vInA.z *vInB.z;
}
```

a nový kód s použitím SSE intrisik:

```
friend float Dot(const VectorSSE41 & vInA, const VectorSSE41 & vInB)
{
    const __m128 tmp = _mm_dp_ps(vInA.m128, vInB.m128, 0x71);
    return ((float *)&tmp)[0];
}
```

Následuje výběr nejdůležitějších intrinsiků instrukcí ze sady SSE4.1 pro použití ve výpočtech ray tracingu (před znaménkem rovná se je uvedený návratová hodnota):

- `_mm_setzero_ps()` - vynulování vektoru.
- `_mm_set_ps(float, float, float, float)` - nastavení ze 4 skalárních čísel.
- `_mm_add_ps(__m128, __m128)` - vektorový součet 4 čísel.
- `_mm_sub_ps(__m128, __m128)` - vektorový rozdíl 4 čísel.
- `_mm_mul_ps(__m128, __m128)` - vektor součinů složek vektoru.
- `_mm_sqrt_ps(__m128)` - vektor odmocnin složek.
- `__mm_dp_ps(__m128, __m128, MASK)` - vektor skalárních součinů s maskou.
- `__mm_shuffle_ps(__m128, __m128, MASK)` - načtení vektoru ze 2 vektorů dle určující bitové masky.

Zarovnání v paměti

Pro použité SSE instrukce je nutné, aby data byly v paměti zarovnána na na násobek 16 bajtů. Pro data alokovaná na zásobníku se toto řeší tak, že pro každou třídu objektů využívající datový typ `__m128`, případně jiné třídy tento datový typ používající se použije při deklaraci speciální parametr kompilátoru `__attribute__((aligned(16)))` (pouze u GCC, u jiných kompilátorů se může lišit), který určí, že daná struktura bude v paměti zarovnávat na 16 bajtů (příklad pro GCC):

```
class VectorSSE41  {.. } __attribute__((aligned(16)));
```

U dat dynamicky alokovaných na hromadě je nutno přetížit operátor new a alokovat zarovnanou paměť. Je mnoho způsobů jak toto provést, jako nejlepší a nejméně pracný je dle mého názoru použití makra, které dané třídě přetíží operátory new, delete, s tím že se úprava alokace pro tyto třídy děje na jednom místě a omezí se tak výskyt nepříjemných chyb a zbytečná redundance kódu:

```
#define DYNAMIC_ALIGNMENT(T) \
    void* operator new(size_t size) { \
        return aligned_malloc(size, __alignof(T)); \
    } \
    void* operator new(size_t size, int alignment) { \
        return aligned_malloc(size, alignment); \
    } \
    void* operator new[](size_t size) { \
        return aligned_malloc(size, __alignof(T)); \
```

```
} \
void* operator new[](size_t size, int alignment) { \
    return aligned_malloc(size, alignment); \
} \
void operator delete(void* p) { \
    aligned_free(p); \
} \
void operator delete[](void* p) { \
    aligned_free(p); \
}
```

kde `aligned_malloc` je funkce pro alokaci zarovnaného bloku paměti. Tímto makrem se pro každou třídu, ve které je použito se přetíží operátor new a delete tak aby pracovaly se zarovnanou pamětí.

5.4.3 Nástroje a jazyky použité při vývoji

Projekt byl vyvíjen především v prostředí Windows. Jako vývojové prostředí programu sloužilo Netbeans, chvíli také Visual Studio. Dokumentace byla psána v prostředí Eclipse. Pro tvorbu tabulek a grafů sloužil Excel, ilustrace pak byly tvořeny z velké části ve vektorovém editoru Corel Draw. Pro tvorbu scén sloužil Blender a také textový editor PsPad.

Pro psaní textu byl zvolen balík maker LaTeX. Pro vývoj programu pak programovací jazyk C++ s použitím pouze základních knihoven, knihoven STL a knihovny lodepng sloužící pro načítání a ukládání obrázků ve formátu PNG. Dále jsem se svolením Martina Stříže upravil a použil implementaci kd-stromu prezentovanou v [20]. Implementace byla generalizována pomocí šablony a částečně upravena v některých pasážích pro vyšší výkon, a aby lépe "pasovala" do mého řešení.

Jako kompilátor jazyka byl zvolen pro svou podporu moderních technologií a norem GCC.

Jazyk C++ byl zvolen především pro své kompromisní vlastnosti, kdy poskytuje velmi vysoký výkon kompilovaných aplikací a zároveň objektové programování, které značně vylepšuje udržitelnost a organizaci kódu.

5.4.4 Návrh programu

Program je navržen jako konzolová aplikace jako vstup pak slouží pár parametrů, mezi nimi je i soubor scény, ve kterém jsou pak nastaveny další důležité parametry. Program byl navrhován tak, aby zvládal následující funkcionalitu:

- Zvládal načítat soubor scény bez nutnosti kompilace programu.
- Umožňoval pokročilé nastavení scény a animací.
- Měl schopnost načítání textur objektů.
- Výsledky ukládal do formátu PNG.
- Byl přenositelný.
- Nepotřeboval GUI, díky čemuž ho lze pouštět i na serverech pouze pomocí příkazové řádky.

- Dal se k programu napsat skript, který bude spouštět program tak, že se postupně budou generovat obrazy určité animace.
- Umožňoval vícevláknové zpracování na více jádrových procesorech.
- Upřednostňoval výkon před kompatibilitou, jedná se především o podporu pouze na procesorech s instrukční sadou SSE3, nebo SSE4.1 (v závislosti na kompilaci).

Program jsem se snažil psát co nejvíce přehledně to šlo, aby tak dokumentoval sám, sebe, protože dle mého názoru není lepšího komentáře, než správně zvolených názvů funkcí a proměnných. Komentáře ovšem tímto způsobem úplně nahradit nejdou a jsou proto použity tam, kde je to pro vysvětlení třeba.

5.4.5 Popis tříd a vykonávání programu

Následuje popis některých důležitých tříd programu. Popis vychází z popisu uvedeného v kapitole 4.1.1

Oproti původní implementaci již nepoužívám v takové míře globálně přístupné služby, protože při podpoře vícevláknového zpracování již není možné zpracovávat většinu věcí globálně. Program je tedy navrhnut čistě objektově a používám v něm mnoho komponovaných tříd.

Mezi základní stavební prvky složitějších komponent stejně jako u výchozí implementace patří:

- Vector implementující 3D vektory a operace s nimi.
- Ray jako abstrakci paprsků.
- Color jako abstrakce barvy/světla.
- UV třída představující uv souřadnice textur.
- SFrame značící relativní rotaci v prostoru.
- Vertex pro vyjádření bodu v prostoru s dalšími vlastnostmi (normála)

Značnou část funkcionality jsem již popsal v kapitolách zabývající se řešením. Následuje tedy nástin toho jak program postupuje při výpočtech.

Při spuštění programu se nejdříve načtou parametry z příkazové řádky. Následně se ze souboru scény načtou pomocí třídy pro načítání (Parser) všechny potřebné informace. Pro každý objekt (Object) se načtou jak všechny potřebné informace o daném typu objektu, tak seznam animačních transformací (BezierKeyFrames). Dále se načte seznam světel (LightsSet) Obsahující všechny jak bodová (LightPoint) tak obdélníková světla (LightRectangle). Také se načte nastavení scény (Samples), všechny materiály (MatPhong, MatTextured). Při načítání se všechny objekty přímo vytvářejí. Všechny objekty a světla se pak umisťují do třídy zajištující provedení optimalizace scény (SceneLoader). Dále se načte kamera (Camera) se všemi segmenty (CamSegment). Pro tu se pak vytvoří seznam vzorků transformací pro celý snímek a nastaví se další informace pro rendering z vlastností kamery. Následně generátor scény (SceneLoader) provede optimalizaci jednotlivých objektů na základě informaci o právě generovaném snímku. Každý objekt pak optimalizuje tak, že mu předgeneruje veškeré animační transformace. Dále vytvoří seznam neurčitých objektů (VAbsObjects) do

kterého pro statické objekty nahrává přímo jejich geometrické tvary a animační objekty normálně se všemi transformacemi. Tento seznam poté vloží do kd-stromu a ten inicializuje. Jako výsledek pak vrátí scénu (Scene), která obsahuje kd-strom a seznam světel.

Následně se paralelně spustí výpočet obrazu pomocí třídy Engine. Ta v sobě zahrnuje veškeré výpočty osvětlovacího modelu, odražených paprsků a distribuovaných efektů. Na svůj vstup dostane generátor náhodných vzorků (Sampler), scénu, kameru, a stopky (StopWatch), které měří průběh generování. Následně se začne vzorkovat projekční plocha kdy se pomocí kamery vygenerují potřebné paprsky a ty se předávají kd-stromu, který pak vyhledává nejbližší průsečík (Intersection). Pro něj se pak vypočte osvětlovací model a vrhají se další rekurzivní paprsky. Vyhledávání průsečíku ve scéně probíhá pomocí kd-stromu (KdTree). Po výpočtu výsledné barvy (Color) určitého pixelu se tato hodnota zapíše do obrazu (Film). Po skončení všech výpočtů se tento obraz uloží na disk.

Pro podrobnější informace doporučuji shlédnout zdrojové kódy a diagram tříd. Tyto informace jsou přiloženy na dodaném CD.

5.4.6 Popis a ovládání programu

Aplikace, kterou jsem pojmenoval *slo-RT*, se pouští z příkazové řádky. Jedno puštění programu generuje právě jeden obraz. Přebírá z příkazové řádky následující parametry:

- `--frame <číslo>, -f` - určuje číslo snímku, který se bude generovat,
- `--scene <název_souboru>, -s` - soubor se specifikací scény,
- `--prefix <řetězec>, -pre` - předpona výsledného názu souboru pro generovaný obraz,
- `--indir <relativní_cesta>, -id` - cesta ke složce, která obsahuje soubor specifikovaný v parametru, `--scene`
- `--outdit <relativní_cesta>, -od` - výstupní složka, kam bude uložen výsledný obraz,
- `--info, -i` - tento parametr přidá do názvu výsledného souboru informace o nastavení parametrů scény a času renderingu. Výborně tak slouží pro testovací účely.

Soubor specifikovaný parametrem `--scene` pak obsahuje veškeré nastavení scény. Část popisu nastavení scény (celý popis by byl příliš rozsáhlý) se nachází v příloze.

Implementace programu se podařila v mírně větším rozsahu, než jaký byl původně záměr. O tom svědčí také textový rozsah této práce. Program je schopný generovat kvalitní výstupy na základě vstupní scény a poradí si i s animovanými scénami. Program byl intenzivně testován a opravován při generování poměrně povedené animace a neobsahuje tak nejspíše žádné chyby běhu programu. Také se s tímto při návrhu počítalo a díky tomu, že je program psán defenzivním stylem, se spíše během testování vypínal z důvodu nesplnění kritérií, než samotnými chybami kódu a sám vypisoval chyby, kde nastaly, a čeho se přibližně může týkat. Díky tomu bylo ladění programu o mnoho snadnější.

Kapitola 6

Dosažené výsledky

V této kapitole nejdříve popíšu výsledky některých optimalizačních technik a následně zhodnotím některé obrazové výstupy.

Všechny výstupy jsou generovány pomocí implementovaného programu. Testovací výpočty snímků pro měření probíhaly na stroji Lenovo Thinkpad R61 s procesorem Intel Core2duo T8100 2,1GHz se 2GB RAM. Další výstupy, především animace, která je přiložená na CD, probíhaly i na dalších strojích, buď desktopových systémech, či noteboocích.

6.1 Přínos optimalizačních technik

V této části si popíšeme přínos některých optimalizačních technik. Pokud není uvedeno jinak vychází se z testovací scény z analýzy programu 4.2.

Je nutné podotknout, že přínos SSE zpracování a optimalizace scény nevychází z finální verze. To je způsobeno tím, že s postupnými optimalizacemi a vylepšeními se postupně vyvíjel i konfigurační soubor scény a scéna se stala postupně nekompatibilní úplně (dočasně byly z implementace odstraněny koule). Testy jsou ale i tak přínosné a proto je uvádím. Výsledky výsledného programu budou uvedeny v dalších subkapitolách.

6.1.1 Přínos zpracování pomocí SSE

Pro jednotlivé operace s vektorem jsem provedl syntetické měření pomocí smyčky, která dokola počítá nějaké dané výpočty. Výsledek je dále ovlivněn kolem umístěných FPU výpočtů, které mohou přínos omezit a simulují případné reálné nasazení v projektu.

U některých operací typu součet, rozdíl, násobení, skalární součin byl efekt zrychlení i několikanásobný, což je způsobeno tím, že místo několika instrukcí (u skalárního součinu 5 výpočtů + další pro načítání každého operandu) lze provést výpočet pomocí jedné instrukce (případně další 3 pro načtení a uložení dat). Daleko zajímavější bylo porovnání u složitějších výpočtů, jako je normalizace vektoru, kdy se musí použít inverze čísla (dělení 1/x). Tento výpočet lze provést více způsoby. Jedním z nich je výpočet pomocí standardních SSE instrukcí SQRTPS DIVPS. Další možnost je nahrazení inverze čísla pomocí méně přesné aproximace inverze čísla RCPPS. Tato se ovšem pro výpočty nehodí, protože při nich dochází k nepřesnostem, které by se mohly později projevit. Tuto nepřesnost však lze odstranit provedením 2 kroků Newtonovy metody pro zpřesnění. Díky ní je již výsledek přesný. Následuje tabulka rychlosti výpočtů pro operace $x = x + y$ a

normalizaci vektoru:

100 000 000 pokusů	FPU	SSE	SSE (rcp)	SSE (rcp+newton)
	Čas [s]			
normalizace	4,696	3,096	2,750	3,333
Operace +=	4,297	1,474	x	x

Tabulka 6.1: Test zrychlení operací pomocí SSE instrukcí

Z těchto výsledků je znatelné že jednoduché výpočty jsou pomocí SSE instrukcí výrazně urychleny. U normalizace, jakožto složitějšího výpočtu, je situace sporná a ve výsledku nepřináší žádné zrychlení. Varianta SSE (RCP - odhad inverzního čísla) je nepoužitelná, protože výsledky nejsou dostatečně přesné a u operace jako je normalizace by to mohlo mít vliv na řadu výpočtů, ve kterých by mohlo docházet k nepříjemným a těžko odhalitelným chybám.

Po kompletní implementaci jsem otestoval rychlost reálného renderingu na testovací scéně:

320x240, 1 světlo, vzorky: DOF 6, stíny 3, matnost 3, rekurze 1			
plošné světlo	čas renderingu [min] - MS VC++ 2008		zrychlení
	FPU Vektory	SSE Vektory	
ne	0:45	0:32	28%
ano	1:22	1:04	22%

Tabulka 6.2: Přínos SSE instrukcí při generování jednoduché scény

Z výsledků je patrné, že tato optimalizace přinesla citelné zrychlení. V případě širší optimalizace (omezení výpočtů FPU) by výsledek mohl být ještě lepší.

6.1.2 Celkový přínos některých optimalizací vyhledávání nejbližšího průsečíku

Následuje porovnání původního řešení s některými optimalizacemi nového řešení. Do tohoto porovnání je zahrnuto zjednodušení objektu (objekt a jeho tvar), včasné porovnávání vzdálenosti průsečíků a vyhledávání pomocí kd-stromu.

Díky převodu objektů na vlastnosti a tvar bylo nutné odstranit z výpočtů možnost geometrických rovin. Ty byly nahrazeny 2 trojúhelníky. Jejich počet v testovací scéně stoupl z 102 na 114. Následuje tabulka časů renderingu (již s SSE vektory):

	320x240, 1 světlo, vzorky: DOF 6, stíny 3, matnost 3, rekurze 1, 1 animovaný objekt			
	před	s optimalizacemi		
		bez kd-stromu	kd-strom	2 úrovňový kd-strom
čas	64	34	7	11
zrychlení [%]	-	47	89	83

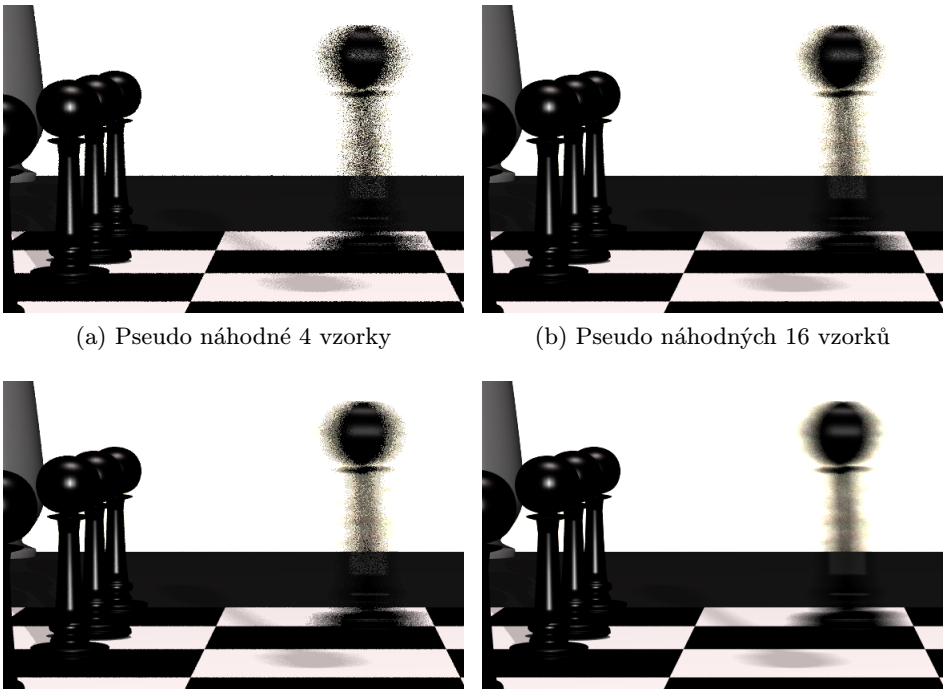
Tabulka 6.3: Přínos optimalizací scény na jednoduché scéně

Přínos kvazi-náhodného generování vzorků

Při použití kvazi náhodného generování vzorků by měla stoupnout rychlost konvergence výpočtů. To jsem si ověřil na testovacích scénách vyzdvihující efekty pro motion blur, hloubku ostrosti, matné odrazy a měkké stíny. MSE značí průměrnou kvadratickou chybu pixelů obrazu od ideálního výsledku.

Jelikož výsledky měření byly téměř identické, uvedu jen některé. Po výsledcích si shrneme co nám říkají.

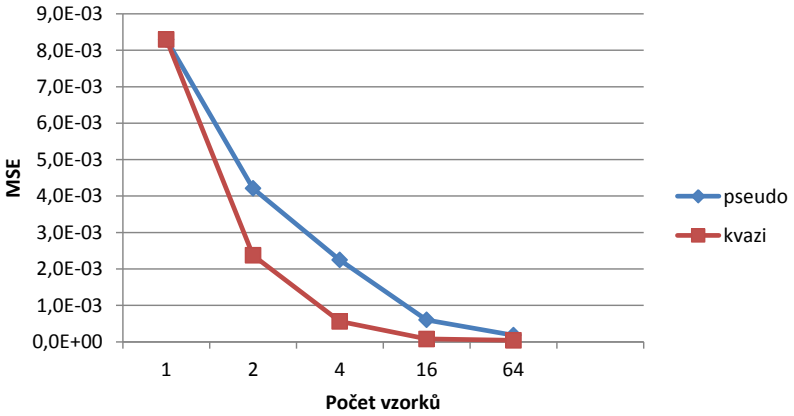
Motion blur kombinovaného se supersamplingem



(a) Pseudo náhodné 4 vzorky (b) Pseudo náhodných 16 vzorků

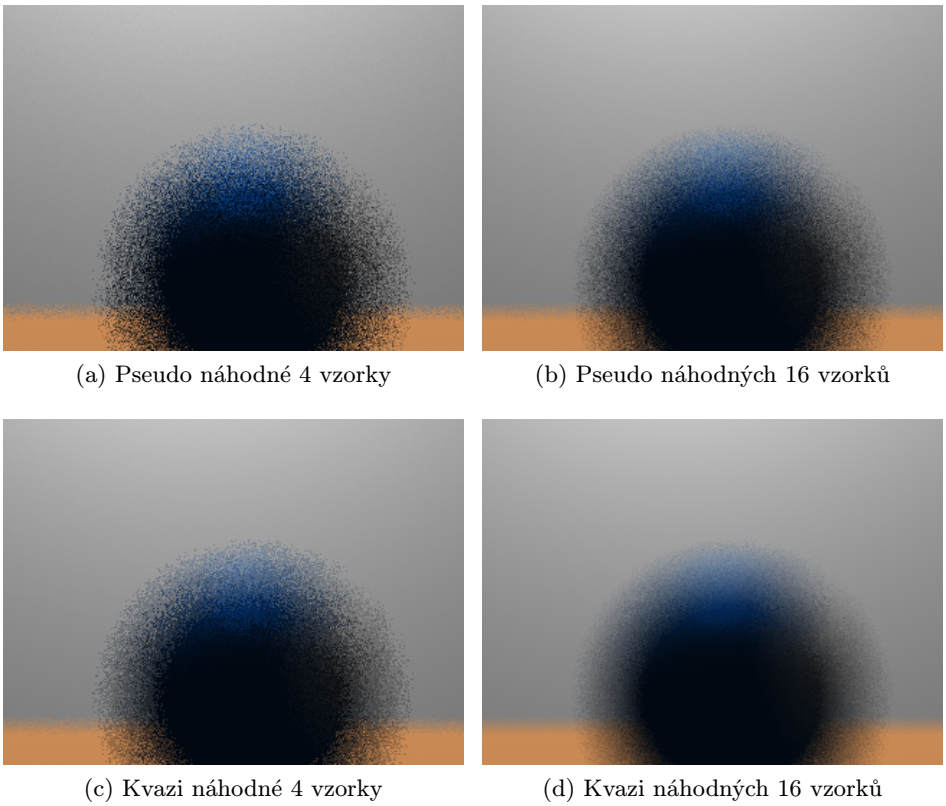
(c) Kvazi náhodné 4 vzorky (d) Kvazi náhodných 16 vzorků

Obrázek 6.1: Kvalita efektu motion blur kombinovaného se supersamplingem projekční plochy v závislosti na počtu vzorků



Obrázek 6.2: Střední kvadratická chyba pixelů obrazu efektu motion blur kombinovaného se supersamplingem projekční plochy v závislosti počtu vzorků

Hloubka ostrosti



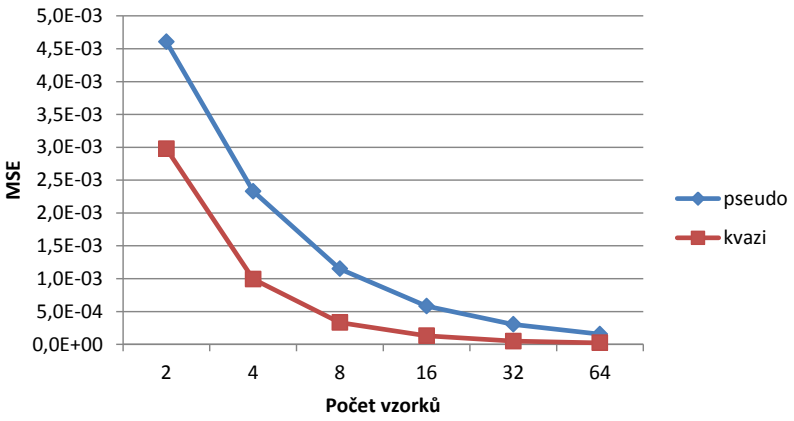
(a) Pseudo náhodné 4 vzorky

(b) Pseudo náhodných 16 vzorků

(c) Kvazi náhodné 4 vzorky

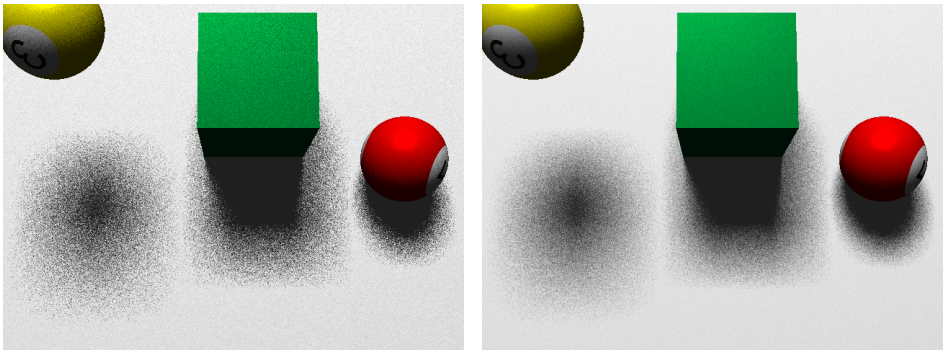
(d) Kvazi náhodných 16 vzorků

Obrázek 6.3: Kvalita efektu hloubky ostrosti v závislosti na počtu vzorků



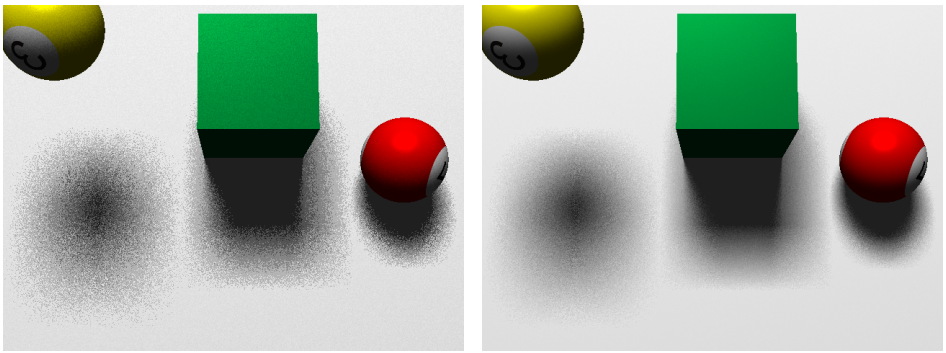
Obrázek 6.4: Střední kvadratická chyba pixelů obrazu efektu hloubky ostrosti v závislosti počtu vzorků

Měkké stíny



(a) Pseudo náhodné 4 vzorky

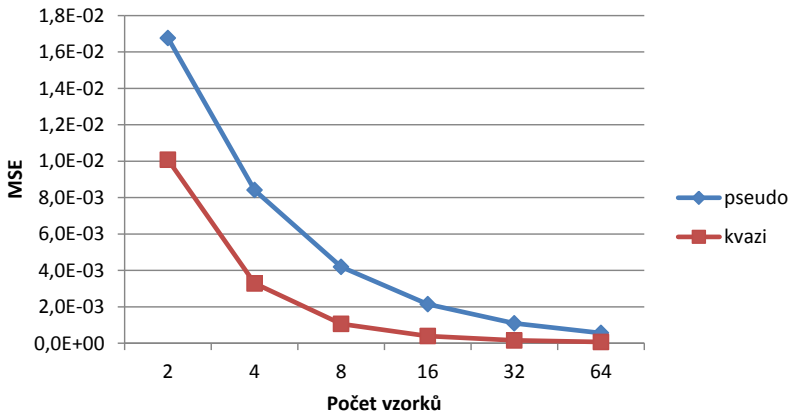
(b) Pseudo náhodných 16 vzorků



(c) Kvazi náhodné 4 vzorky

(d) Kvazi náhodných 16 vzorků

Obrázek 6.5: Kvalita efektu měkkých stínů v závislosti na počtu vzorků



Obrázek 6.6: Střední kvadratická chyba pixelů obrazu s efektem měkkých stínů v závislosti počtu vzorků

Pro matné odrazy a lomy je průběh grafu velmi podobný a pro úsporu místa nebudu výsledky uvádět.

Z grafů vyplývá, že scéna vzorkovaná kvazi náhodnými sekvencemi opravdu konverguje rychleji a to přibližně 2×.

Z grafických výstupů to jde zřetelně vidět a scéna vzorkovaná pouze 4 kvazi náhodnými vzorky vypadá téměř jako výstup vzorkovaný 16 pseudo náhodnými vzorky.

Z obrázku nejdou vidět žádné artefakty a proto je vhodné používat právě kvazi náhodné vzorkování místo pseudo-náhodného.

6.1.3 Adaptivní vzorkování

Nyní se podíváme na závislost kvality výstupu potažmo prahu kontrastu a doby výpočtu scény.

(a) Bez adaptivního vzorkování (188s)

(b) S adaptivním vzorkováním - max. kontrast 0.2 (tedy 20%) (92s)

Obrázek 6.7: Kvalita výstupu scény adaptivního samplování pro základní počet vzorků 16 a snížený 4

práh kontrastu [0-1]	0,5	0,2	0,1	0,05	0,01	0
MSE	5,73E-04	5,73E-04	3,39E-04	2,95E-04	2,54E-04	0,00E+00
čas [s]	65	92	124	140	182	188

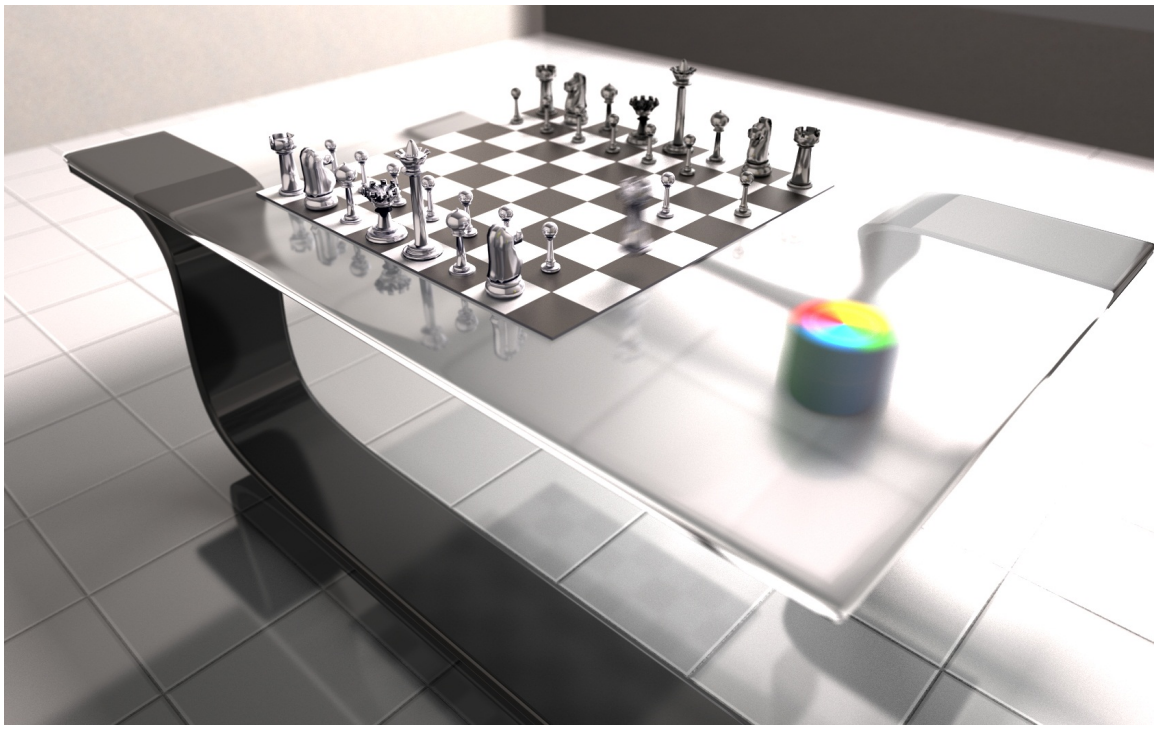
Tabulka 6.4: Závislost kontrastu, kvality a doby výpočtu adaptivního samplování

Adaptivní samplování je výborná technika, kterou se může snížit doba výpočtu scény, aniž by se výrazně snížila obrazová kvalita výstupu. Na obrázku s adaptivním samplováním lze pozorovat úbytek detailů jen místy, ne na celé ploše. Ovšem zvolený práh kontrastu je zde příliš velký, což je i úmysl, aby šel rozdíl vidět. Pokud bychom například nastavili prahový kontrast na hodnotu 5% (0.05) výsledek by měl 2× nižší chybu, ale za cenu o 50% vyšší výpočetní náročnosti.

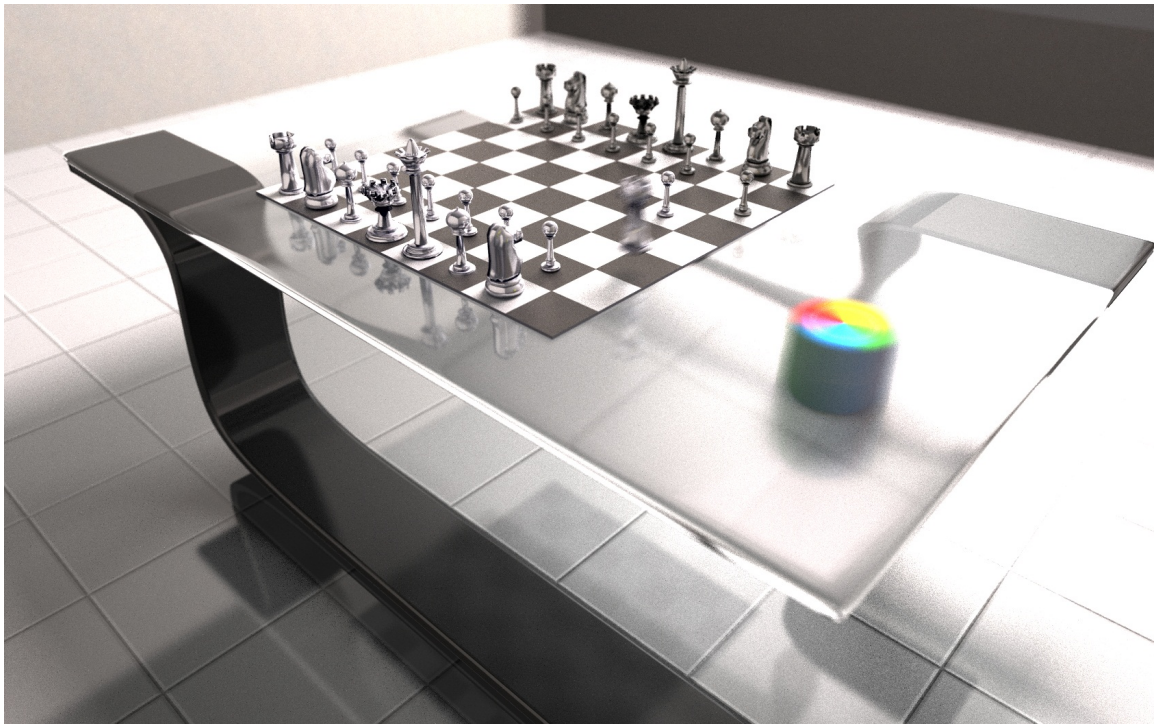
6.2 Kvalitní výstupy, doba renderingu

Nyní si ukážeme některé kvalitní výstupy s časem jejich renderingu.

Jako první se podíváme na rozdíl v kvalitě dvou stejných obrázků ovšem s různým nastavením.



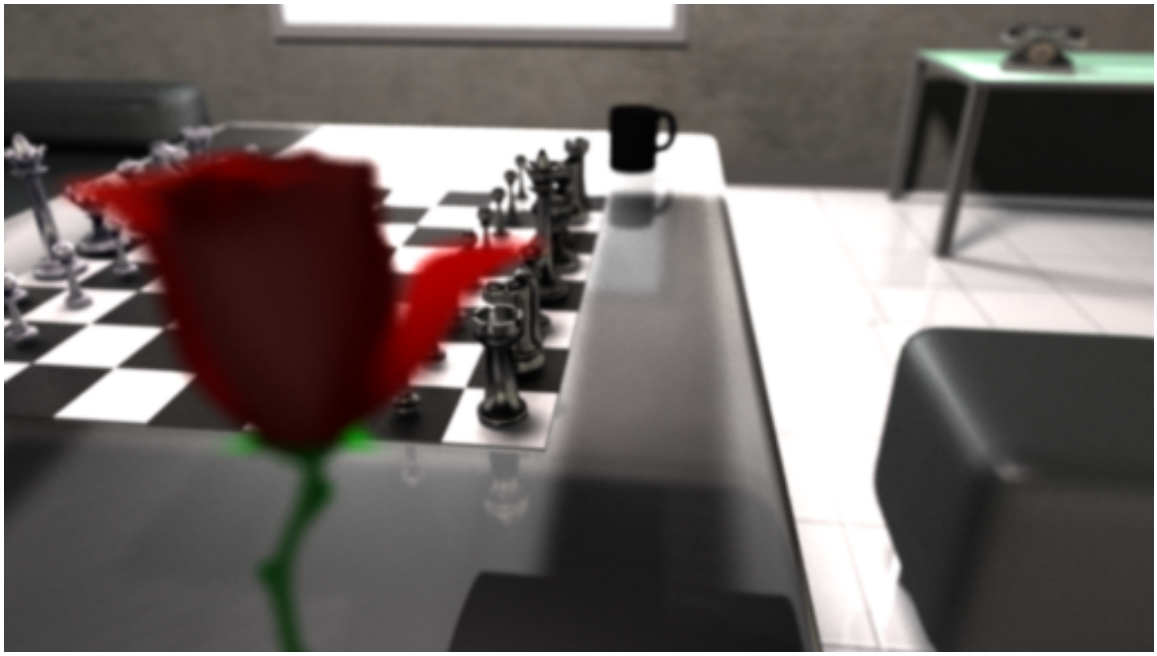
Obrázek 6.8: Scéna stolu - 32-256 vzorků projekční plochy (a MB), 2 vzorky hloubky ostrosti, 1 vzorek měkkých stínů, a 2 vzorky pro matné povrchy (odrazy i lomy). Doba renderingu 6 h. 16 min, rozlišení 1680 × 1050



Obrázek 6.9: Scéna stolu - 32-256 vzorků projekční plochy (a MB), 2 vzorky hloubky ostrosti, 1 vzorek měkkých stínů, a 2 vzorky pro matné povrchy (odrazy i lomy). Doba renderingu 28 min. 29 s, rozlišení 1680 × 1050

V této scéně vynikají všechny efekty distribuovaného ray tracingu. Na pozadí se výrazněji projevuje hloubka ostrosti, která dodává výstupům výrazné změkčení a nádech realističnosti. U šachové figurky a rotující barevné kostky se krásně projevuje efekt motion blur, který také vypadá velmi realisticky a přidává snímku určitý rozměr, kdy lze na první pohled poznat, že se nejedná o statickou scénu. Na stole pak lze pozorovat, že sklo není úplně čiré, ale mírně zamlžené, což dělá stůl nedokonale hladkým, a tedy také reálným. Na šachovnici jsou hezky vyobrazeny měkké stíny, které dodávají snímku prosvětlenost a reálný vzhled. Cenou za kvalitní výstup je ovšem vysoká časová náročnost. 6 hodin je opravdu mnoho času. Na druhou stranu je obraz naprosto bez šumu. Pokud se pak podíváme na stejnou scénu s méně vzorky, lze pozorovat určitou míru šumu, ale ne tak výraznou. Čas renderingu je pak přibližně 12× kratší.

Obrázek 6.10: Scéna zaostřené růže - 8-64 vzorků projekční plochy (a MB), 2 vzorky hloubky ostrosti, 1 vzorek měkkých stínů, a 1 vzorek pro matné povrchy (odrazy i lomy). Doba renderingu 5 min. 25 s, rozlišení 1024 × 576



Obrázek 6.11: Scéna zaostřené růže - 8-64 vzorků projekční plochy (a MB), 2 vzorky hloubky ostrosti, 1 vzorek měkkých stínů, a 1 vzorek pro matné povrchy (odrazy i lomy). Doba renderingu 20 min. 10 s, rozlišení 1024 × 576

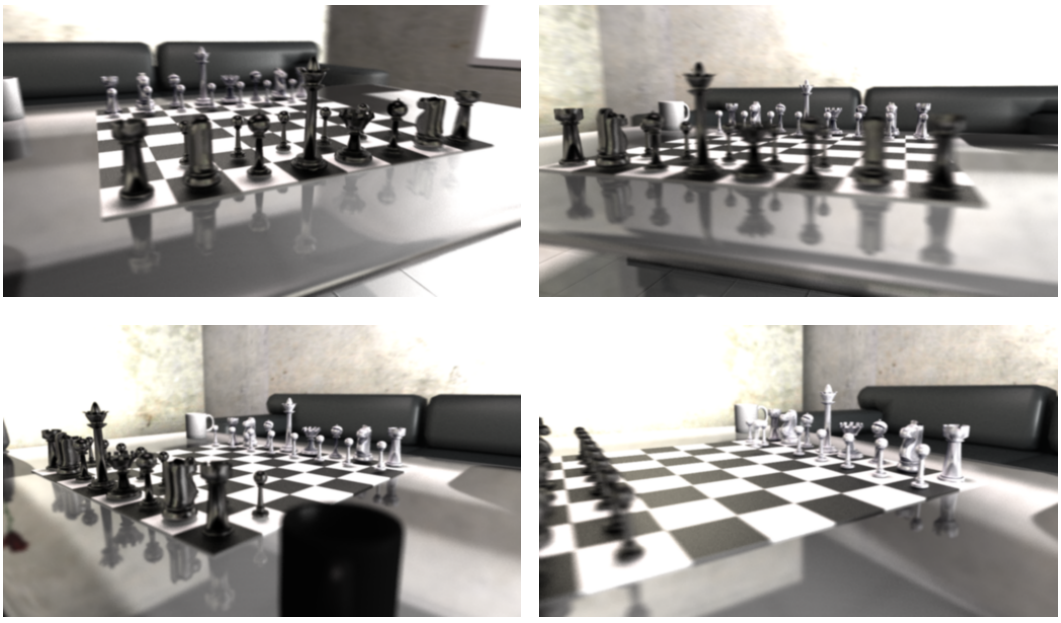
Scéna s růží zabrala relativně mnoho času z důvodu rekurze distribuovaných odražených paprsků v květu, které mají pomoci reálnějšímu zobrazení světla dopadajícího na růži.

Jako ideální se vzhledem k zanoření efektů jeví generovat co největší počet vzorků projekční plochy a malý počet vzorků měkkých stínů a rekurzivních paprsků. Výstupy jsou pak kvalitní a nestoupá tolik časová náročnost na rendering oproti navyšování počtu vzorků světel a odražených paprsků. Navíc se zvýšený počet vzorků projeví v celé scéně, kdežto složky osvětlovacího modelu jen v částech, kde se nějaké paprsky odráží/lomí resp. dopadají světelné paprsky.

6.2.1 Animace

V přiloženém CD je také uložené krátké video animace nejkratšího herního šach matu. Na tomto videu jsem si ověřil, že implementace je i prakticky použitelná. K dispozici jsem měl výpočetní výkon odpovídající přibližně 10 dvoujádrovým procesorům. Na nich jsem 32 sekundové video ve slušné kvalitě (s mírně nižším rozlišením než je poloviční HD) vyrenderoval za přibližně 2-3 dny výpočtů. Na první pohled se to může zdát hodně, avšak při počtu 25 snímků za sekundu to dělá 800 snímků a ty už zaberou nějakou dobu na renderování. Snímky uvedené výše pochází právě z této animace, nebo z této scény vychází.

Jako příklad uvádím další 2 snímky.



Obrázek 6.12: Snímky z animace přiložené na CD

Časy výpočtů nejsou vůbec přemrštěné a umožňují tak i občasné praktické renderování. Samozřejmě, že nedosahují časů těch nejlépe optimalizovaných ray tracerů. To by ovšem ani nebylo v silách jednoho studenta. Implementace však poskytuje velmi slušný výkon při dobré kvalitě výstupů.

Kapitola 7

Závěr

Distribuovaný ray tracing je jedna z nejpokročilejších metod zobrazování, která je ještě prakticky použitelná na dnešních dostupných výpočetních prostředcích. Metoda simuluje reálné vlastnosti světla a i díky tomu výstupy této metody působí na první pohled měkce a reálně. I když má metoda jisté nedostatky, výstupy z ní určitě dokážou na první pohled zaujmout. Vývoj navíc pokračuje dále, výkon a počet jader procesorů se zvyšuje a výpočty je již možné provádět na grafických kartách. Pokud to půjde podobným tempem ještě několik let, je více než pravděpodobné, že se tato metoda postupně uchytí i v real-time počítačové grafice. První pokusy již jsou na světě a i když zatím nedosahují ani zlomku poměru kvality a výkonu akcelerované 3D grafiky, díky právě 3D akcelerátorům se postupně alespoň částečně pro některé efekty již používají.

Cíl této práce byl od počátku jasný, pokusit se navrhnout a implementovat optimalizovaný off-line ray tracer s distribuovanými efekty, který bude slušně použitelný i pro praktičtější účely než testování. V této práci jsem jedno takové řešení popsal a implementoval.

Nastudoval jsem a popsal teorii distribuovaného sledování paprsků zahrnující značné množství matematického základu. Dále jsem popsal dostupné a použitelné optimalizační techniky. Mezi ně patří vyhledávání průsečíku paprsku se scénou pomocí kd-stromu, optimalizace vektorových výpočtů pomocí SSE instrukcí, zvýšení rychlosti konvergence metody pomocí kvazi-náhodných sekvencí a rychlý průsečík paprsku s trojúhelníkem.

Dále jsem vycházel z výchozí implementace započaté v mé bakalářské práci [18] a navrhl a jak a kde by se daly optimalizace použít k tomu jsem přidal další, které se logicky vybízely. Patří k ním optimalizace animačního systému, či paralelní zpracování na více jádrových procesorech pomocí OpenMP. Optimalizací bylo ovšem ještě více, ale nebyly tak významné, a na konec zapadaly do návrhu programu jako jeho nepostradatelné součásti, bez kterých by ani optimalizační techniky uvedené výše nemusely tolik pomoci k lepším výsledkům.

Všechny tyto optimalizace jsem poté implementoval do programu. K tomu přidal v této práci ne moc popsané praktické vlastnosti, které tento ray tracer činí praktičtějším. Patří sem například export trojúhelníkové sítě i s uv mapovacími souřadnicemi, který se beze změn dá připojit do scény a umožňuje tak pohodlné navrhování scén například ve zdarma dostupném 3D modelovacím nástroji Blender.

Následují výsledky implementace a jednotlivých optimalizací, kdy jsem se snažil objektivně pomocí měření zhodnotit jaký přínos mají optimalizace. Všechny optimalizace plní svou funkci na výbornou. Nemělo by smysl popisovat přínos všech technik zvlášť v krátkém závěru, proto jako příklad uvedu, že v HD rozlišení s malou mírou šumu lze komplikované scény renderovat přibližně za 1-3 hodiny. To si myslím je dobrý výsledek.

Daní za rozumný čas výpočtu je minimální požadovaná podpora instrukcí procesoru, kdy pro běh je třeba, aby procesor zvládal SSE instrukce 4.1. Těsně před odevzdáním jsem ovšem ještě stihl vytvořit odnož implementace podporující i starší SSE3.

Program jsem implementoval v jazyce C++ a kompiloval pomocí multiplatformního překladače GCC což zajišťuje i kompatibilitu kódu napříč platformami.

Jako další možný směr vylepšení, kterým by se dalo pokračovat je například více dimenzionální adaptivní samplování [6], které se jeví v současné době jako nejlepší metoda pro snížení šumu v obraze při použití malého množství vzorků. Další směr možného vývoje vidím v adaptaci metody na dnešní velmi výkonné grafické karty, které mají i několikrát vyšší hrubý výkon oproti současným procesorům.

Dále už pak lze metodu rozvíjet spíše směrem k vyšší funkcionalitě, jako přidání dalších typů osvětlovacích modelů, zakomponování nepřímého osvětlovacího modelu, tam kde se projeví na výsledku (např. kaustiky), či rovnou zobecnění na path tracing.

Také jsem vytvořil plakát znázorňující dosáhnuté výsledky.

Práce na tomto projektu mě naučila spoustu nových věcí. Pročetl jsem poměrně velké množství různých vědeckých prací a materiálů v anglickém jazyce a získal tak další cenné informace o pokročilých realistických zobrazovacích metodách. Také jsem se díky této práci dozvěděl o různých, pro mě dříve neznámých metodách jako je například kvazi náhodné vzorkování. Naučil jsem se také využívat speciální instrukční sady procesoru a o mnoho lépe programovat.

Výsledky měření ukazují, že se mi podařilo implementovat slušně výkonný ray tracer, který je schopný generovat pokročilé scény v rozumném čase. Také subjektivní dojem, který je v počítačové grafice dle mého názoru neméně důležitou částí mi říká, že se se cíle povedlo naplnit nad má očekávání, které nebyly nízké. Důkaz toho může být vygenerované video přiložené na CD.

Literatura

[1] Alexander, L. *Výuka počítačové grafiky cestou WWW.* Vysoké učení technické v Brně, Fakulta elektrotechniky a informatiky, 2000. Diplomová práce.

[2] Břetislav Fajmon, I. R. *Matematika 3.* [b.m.]: Fakulta Elektrotechniky a Komunikačních Technologií, Vysoké Učení Technické v Brně, 2005.

[3] Fendt, W. *Odraz a lom vlnění (Huygensův princip)* [online]. 1998, 2005 [cit. 27. dubna 2011]. Interaktivní aplikace. Dostupné z WWW: <http://www.walter-fendt.de/ph14cz/huygenspr_cz.htm>.

[4] Gregor, L. Ověření ocenění opcí metodou Quasi-Monte-Carlo. In *5. mezinárodní konference Finanční řízení podniku a finančních institucí* [online]. Září 2005 [cit. 15. března 2011]. Dostupné z WWW: <http://www.ekf.vsb.cz/miranda2/export/sites-root/ekf/konference/cs/okruhy/frpfi/rocnik-2005/prispevky/dokumenty/Gregor.pdf>.

[5] Greve, B. de. *Reflections and Refractions in Ray Tracing* [online]. 2004 [cit. 27. dubna 2011]. Dostupné z WWW: <http://www.flipcode.com/archives/reflection_transmission.pdf>.

[6] Hachisuka, T., Jarosz, W., Weistroffer, R. P. et al. Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH).* Srpen 2008, roč. 27.

[7] Halliday, D., Resnick, R. a Walker, J. *Fyzika. Vysokoškolské učebnice obecné fyziky. Část 4: Elektromagnetické vlny – optika – relativita.* Brno a Praha: VUTIUM a Prometheus, 2000. ISBN 80-214-1868-0.

[8] Havel, J. a Herout, A. Yet Faster Ray-Triangle Intersection (Using SSE4). *IEEE Transactions on Visualization and Computer Graphics.* 2010, roč. 2010, č. 3. S. 434–438. Dostupné z WWW: <http://www.fit.vutbr.cz/research/view_pub.php?id=8985>. ISSN 1077-2626.

[9] Havran, V. *Heuristic Ray Shooting Algorithms.* Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000. Ph.D. Thesis. Dostupné z WWW: <http://www.cgg.cvut.cz/~havran/phdthesis.html>.

[10] Joe, S. a Kuo, F. Y. *Notes on generating Sobol' sequences* [online]. srpen 2008 [cit. 15. května 2011]. Dostupné z WWW: <http://web.maths.unsw.edu.au/~kuo/sobol/joe-kuo-notes.pdf>.

[11] Ken Seng Tan, P. P. B. Applications of randomized low discrepancy sequences to the valuation of complex securities. *Journal of Economic Dynamics & Control* [online]. 2000, č. 24 [cit. 12. května 2011]. S. 1747–1782. Dostupné z WWW: <https://editorialexpress.com/jrust/econ625/boyle.pdf>.

[12] Marko, M. *Quasi-Monte Carlo* [online]. březen 2011 [cit. 16. dubna 2011]. Dostupné z WWW: <http://cgg.mff.cuni.cz/~jaroslav/teaching/npgr031/05-stgi-qmc%20-%20poznamky%20-%20matej%20marko.pdf>.

[13] Matt Pharr, G. H. *Physically Based Rendering: From Theory to Implementation.* 2. vyd. [b.m.]: Morgan KaufMann, 2010. Chapter 13: Monte Carlo Integration I: Basic Concepts, s. 637–675. ISBN 978-0-12-375079-2.

[14] Matt Pharr, G. H. *Physically Based Rendering: From Theory to Implementation.* 2. vyd. [b.m.]: Morgan KaufMann, 2010. Chapter 14: Monte Carlo Integration II: Improving Efficiency, s. 679–734. ISBN 978-0-12-375079-2.

[15] Matt Pharr, G. H. *Physically Based Rendering: From Theory to Implementation.* 2. vyd. [b.m.]: Morgan KaufMann, 2010. Chapter 04: Primitives and Intersection Acceleration, s. 183–255. ISBN 978-0-12-375079-2.

[16] Matt Pharr, G. H. *Physically Based Rendering: From Theory to Implementation.* 2. vyd. [b.m.]: Morgan KaufMann, 2010. Chapter 07: Sampling And Reconstruction, s. 323–417. ISBN 978-0-12-375079-2.

[17] Shevtsov, M., Soupikov, A. a Kapustin, A. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. In *Procedings of GraphiCon'2007.* Nizhniy Novgorod, Russia: [b.n.], 2007. S. 33–38. Dostupné z WWW: <http://www.graphicon.ru/2007/proceedings/Papers/Paper_46.pdf>.

[18] Slovák, R. *Distribuované sledování paprsku.* Brno: FIT VUT v Brně, 2008. Bakalářská práce.

[19] Slovák, R. *Distributed ray tracing v rozumném čase.* FIT VUT v Brně, 2010. Semestrální projekt.

[20] Stříž, M. *Rychlý průsečík paprsku se scénou.* Brno: FIT VUT v Brně, 2008. Bakalářská práce.

[21] Wald, I. *Realtime Ray Tracing and Interactive Global Illumination.* Computer Graphics Group, Saarland University, 2004. Disertační práce. Dostupné z WWW: <http://www.sci.utah.edu/~wald/PhD/wald_phd.pdf>.

[22] Wikipedia. *Fermatův princip* [online]. 2010, naposledy editována 16. 5. 2010 [cit. 23. dubna 2011]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Fermatův_princip>.

[23] Wikipedia. *OpenMP* [online]. 2010, naposledy editována 24. 12. 2010 [cit. 27. května 2011]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/OpenMP>.

[24] Wikipedia. *Shannonův teorém* [online]. 2010, Stránka byla naposledy editována 11. 11. 2010 [cit. 20. května 2011]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Shannon%C5%AFv_teor%C3%A9m>.

[25] Wikipedia. *Contrast (Vision)* [online]. 2011, last modified on 3 May 2011 [cit. 22. května 2011]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Contrast_%28vision%29>.

[26] Wikipedia. *Geometrická optika* [online]. 2011, naposledy editována 31. 1. 2011 [cit. 23. dubna 2011]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Geometrická_optika>.

[27] Wikipedia. *Global Illumination* [online]. 2011, last modified on 20 January 2011 [cit. 7. května 2011]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Global_illumination>.

[28] Wikipedia. *Huygensův princip* [online]. 2011, naposledy editována 11. 4. 2011 [cit. 23. dubna 2011]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Huygensův_princip>.

[29] Wikipedia. *Optika* [online]. 2011, naposledy editována 13. 3. 2011 [cit. 23. dubna 2011]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Optika>.

[30] Wikipedia. *Phong shading* [online]. 2011, last modified on 20 April 2011 [cit. 7. května 2011]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Phong_shading>.

[31] Wikipedia. *Snellův zákon* [online]. 2011, naposledy editována 25. 3. 2011 [cit. 27. dubna 2011]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Snellův_zákon>.

Seznam příloh

Obsah CD

CD obsahuje soubor readme.txt a následující adresářovou strukturu:

- bin - složka s binárními soubory
- dokumentace - složka s dokumentací
- zdrojove_kody - složka se zdrojovými kódy
- resources - složka obsahující binární soubory, skripty pro spuštění a vše potřebné pro renderování. Obsahuje dále složky:
  - outputs - složka s výstupy
  - scenes - složka se scénami, texturami a modely objektů.

Manuál

Část této sekce bude shodná s mou bakalářskou prací. A to z důvodu, že se používá velmi podobný formát pro popis scény.

Co je ray tracer?

Ray tracer je program sloužící ke renderování velmi realistických, či nereálných obrazů na základě matematického popisu scény. Následuje popis použití a seznam implementovaných vlastností.

Použití

Program se spouští z příkazové řádky. Umožňuje předat parametry:

`--frame <číslo>, -f` - číslo snímku, který se bude generovat

`--scene <název_souboru>, -s` - soubor se specifikací scény.

`--prefix <řetězec>, -pre` - předpona výsledného názvu souboru pro generovaný obraz.

`--indir <relativní_cesta>, -id` - cesta ke složce, která obsahuje soubor specifikovaný v parametru `--scene`

`--outdit <relativní_cesta>, -od` - výstupní složka, kam bude uložen výsledný obraz.

`--info, -i` - tento parametr přidá do názvu výsledného souboru informace o nastavení parametrů scény a času renderingu. Výborně tak slouží pro testovací účely.

Příklady použití:

```
raytracer

raytracer --frame 5

raytracer --scene animace.txt

raytracer -f 15 -s animace.txt --info

raytracer  -f 15 -s animace.txt -id input/scenes -od output/animace
```

Nastavení scény

Pro nastavení scény se uplatňují následující pravidla:

- mezery slouží jako oddělovač hodnot
- závorky { a } slouží pro definici určité sekce (prvku)
- číslo se zadává s desetinnou tečkou a ne podle českých norem čárkou
- textové názvy nesmí obsahovat mezery a složené závorky
- scéna je popsána prvky (např. Kamera, objekt, nastavení, materiál).
- prvky jsou určené typem, jménem a atributy
  - Atributy jsou opět prvky

Používají se 3 druhy popisu jednotlivých prvku:

- S typem, jménem a atributy ve složených závorkách
  `<typ_prvku> <jméno_prvku> { <atributy_prvku>... }`
- Základní prvek se jménem a atributy ve složených závorkách
  `<jméno_prvku> { <atributy prvku>... }`
- Základní prvek se jménem a atributem
  `<jméno_prvku> <atribut>`

Základní typy prvků použité jako atributy

Následují 3 nejzákladnější prvky, které jsou v nastavení použity.

- Název souboru (symbolicky v textu dále jen `file`)
  `filename <název_souboru>`
- Reálné číslo (symbolicky v textu dále jen `real`)
  `<název_atributu> <číslo>`
- Vektor (symbolicky v textu dále jen `vector`)
  `<název_atributu> { <číslo> <číslo> <číslo> }`

Příklad konfigurace scény s vysvětlivkami

Nejlepší a zároveň nejrychlejším způsobem popisu to jak popsat scénu je samotný popis scény s komentáři a vysvětlivkami. Příklad následuje:

```
# komentář do konce řádku
/*
víceřádkový komentář
*/

# Nastavení, které načítá hodnoty pozic v prostoru s měřítkem
# tato volba slouží pro velké scény, kdy dochází k problémům
# se zaokrouhlením.
Scaling zvetseni { 0.2 }


# Prvek nastavení parametrů scény
settings SCENA1 {
        # Typ generování vzorků
#   generatorType PSEUDO
    generatorType QUASI

        # Typ vzorkování světel
#   lightSamplingType SAMPLE_ONE_AT_TIME
    lightSamplingType SAMPLE_ALL_AT_TIME

        # Nastavení jak se bude počítat kontrast pro adaptivní vzorkování
    contrastType    DIFFERENCE
#   contrastType    WEBER
#   contrastType    MICHELSON

        # Počet vzorků projekční plochy a MB
    imageSamples 32
        # Min. počet vzorků projekční plochy,
        # slouží pro adaptivní samplování
    imageMinSamples 8
        # velikost kontrastu
    imageContrastLimit 0.03     # 0-1  (0-100%)
```

```
        # počet vzorků hloubky ostrosti
    dofSamples 2
        # počet vzorků měkkých stínů
    softShadowSamples 1
        # počet vzorků distribuovaných odrazů
    diffuseReflectionSamples 1
        # počet vzorků distribuovaných lomů
    diffuseRefractionSamples 1
        # rekurze procházení scénou
    recursion  3
        # počet sekcí pro paralelní zpracování
    sectionsHorizontal 8
    sectionsVertical 8
    parallelThreads 8       # počet vláken
    sectionsPerThread 2     # počet sekcí na vlákno

}

# nastavení kamery
Camera camera {
    fps 25              # snímků za s

    #nastavení obrazového výstupu
    Film platno {
        width 1024
        height 576
        prefix __anim
        output PNG_IMAGE
    }

    # segment kamery, může jich být více
    CamSegment O1T12 {
            # čas snímku ve výsledném videu
        videoStart 0    videoEnd 2
            # čas scény namapovaný na čas segmentu kamery ve videu
        sceneStart 0    sceneEnd 0

        addPosition { 50 -50 35 }   #pozice kamery (možno více)
        addTarget { 260 205 7 }     #cíl kamery (možno více)

            # vektor nebe kamery na počátku a konci segmentu
        skyStart { 0 0 1 }  skyEnd { 0 0 1 }

            # lineární zorné pole a hloubka ostrosti
        fovStart 0.05  fovEnd 0.05   dofStart 425  dofEnd 375
            # délka závěrky a šířka čočky kamery
        blurDuration 0.04   lensRadiusStart 0.5  lensRadiusEnd 0.6
    }
```

```
}

light_rectangle plošné světlo {
    color { 0.65 0.6 0.6 }      #barva
    vertex0 { 0 0 180 }         # body určující obdélník
    vertex1 { 50 0 180 }
    vertex2 { 50 50 180 }
}

light_point     bodové světlo {
    color { 1 1 1 }
    position { -40 0  100 }
}

nastavení osvětlovacího modelu (povrchu materiálu)
phong_info sklo {
    # jednotlivé parametry značí barvu { čevená zelená modrá }
    ambient { 0 0 0 }     # barva nepřímého rozptylového osvětlení
    diffuse { .1 .1 .1 }  # barva přímého rozptýleného světla
    specular { .7 .7 .7 } # barva odlesků
    shininess 20          # ostrost odlesků
    reflectance 0.3       # intenzita odražených paprsků
    diffuse_reflection 0.05 # rozptyl odražených paprsků
    refraction 0.9        # intenzita lomeného paprsku
    diffuse_refraction 0.08 # rozptyl lomených paprsků
    refraction_index 1.5  # index lomu
}

# materiál definovaný pouze nastavením osvětlovacího modelu
material_phong sklo {
    phong_info sklo   # odkaz na phong pomocí názvu
}

# definice textury
texture Koberec {
    filename textures/Carpet1.png
}

# definice materiálu, který má navíc texturu
material_texture Koberec {
    phong_info sklo
    texture Koberec # textový odkaz na definovanou texturu
    scale_U 4       # škálování rozměru textury
    scale_V 4
}

#definice rovinné plochy o velikosti strany 1000
plane zdi_podlaha {
```

```
    normal { 0 0 1 }        # normálový vektor
    position { 0 0 -49.99 }
    material podlaha }      # přiřazený materiál (odkaz na něj)

# definice trojúhelníkové sítě
mesh pincl {
                        # název souborů s definici sítě
    filename meshes/_pincl.slo
    smooth_normals 1    # vyhlazování normál povrchu
    material WH         # odkaz na materiál

    # definice beziérových animačních klíčů (možno více)
    addKframe {
        time 0.5 # čas scény
        # předcházející řídící bod beziérovy kubiky
        # pokud nějaká část není uvedena,
        # automaticky se pokusí doplnit zpředcházejících informací
        pre {
                pivot { 15 16  2.5 } # pozice
                up { -0.5 0 1 }
                right { 1 0 0 }
            }
        # kíčový bod beziérovy kubiky
        key {   pivot { 08 16  0   }    up { 0   0 1 } }
        # řídící bod nadcházející beziérovy kubiky
        nex {   pivot { 06 16  2   }    up { 0.5 0 1 } }
    }

}
```