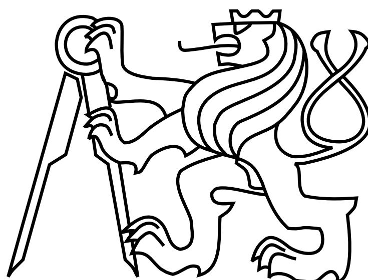
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studiijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová práce

Jednoduchá knihovna pro vrhání paprsků

Bc. Radek Loucký

Vedoucí práce: Ing. Jiří Bittner, Ph.D.

Studijní program: Otevřená informatika

Obor: Počítačová grafika

10. května 2014

Poděkování

Chtěl bych poděkovat vedoucímu své práce Ing. Jiřímu Bittnerovi, Ph.D. za jeho odborné rady a vedení mé diplomové práce. Dále bych chtěl poděkovat své rodině za podporu, kterou mi projevovali po celou dobu mého studia a své přítelkyni za trpělivost a porozumění.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 12. 5. 2014 ....

Abstract

Ray tracing is a technique of generating an imagine of synthetic scene by casting rays through pixels in an image plane. This technique produces one of the most realistic imagines of synthetic scenes. As the computing power of modern CPUs is growing every year, it's possible to synthesize scenes several times faster than it was possible in the past. This makes the global illumination methods more and more popular for synthesis both static and dynamic scenes. It leads to need of creation a library, which would be usefull for the ray casting procedure. Such library should have very simple API for ray casting and obtaining information about ray hits. Naïve approach of generating an imagine of synthetic scene by casting rays is very computationally demanding as we need to cast ray through every pixel in an image plane and test intersection of this ray with all scene triangles. This approach is more less inapplicable for scene even with tens or hundreds of triangles. This leads us to the second requirement. Such library should use some acceleration structure, which speeds up the ray casting procedure to nearly real time computation.

Goal of my thesis is to design and implement such library. The main emphasis was placed on simplicity of API, multithreading, usage of single instruction multiple data (SIMD) instructions and bounding volume hierarchy as acceleration structure, which is one of the most popular accelerating data structure used for ray casting. The library should also be easily extendable with more acceleration data structures. The library can also be useful for learning and teaching purposes.

The first part of the thesis deals with a theoretical introduction to the problem of image synthesis using global illumination methods, namely the ray tracing method and path tracing. In this part the algorithms and acceleration structures are described in detail. The second part of the thesis discusses the implementation of the library and the measurement results on both static and dynamic scenes.

Abstrakt

Metody syntézy obrazu pomocí sledování paprsků produkují jedny z nejrealističtějších obrazů syntetických scén. Díky stále rostoucímu výpočetnímu výkonu dnešních procesorů, je možné vytvářet fotorealistické scény v několikanásobně kratším čase, než tomu bylo v minulosti. To činí tyto metody globálního osvětlování stále populárnějšími pro syntézu jak statickým, tak animovaných scén. S tím vzniká také potřeba na vytvoření knihovny, kterou bude možné využít při implementaci metod sledujících paprsky. Tato knihovna by měla mít jednoduché veřejné rozhraní, pomocí kterého by mělo být možné vrhat paprsky do scény a získávat zpět informace o jejich průchodu scénou. Naivní algoritmus sledování cesty paprsku je výpočetně náročný, protože pro každý pixel a každý paprsek je nutné otestovat průsečík tohoto paprsku se všemi objekty ve scéně. I přes poměrně jednoduché zadání je toto naivní řešení z časového hlediska téměř nepoužitelné pro scény již o několika desítkách až stovkách objektů. Proto je potřeba, aby knihovna implementovala akcelerační struktury, které celý proces výrazně urychlují a spolu s využitím možností moderních procesů dovolují syntézu některých scén v reálném čase.

Má diplomová práce se tvorbou takové knihovny zabývá. Hlavní důraz při její tvorbě byl kladen na jednoduchost veřejného rozhraní, využití více vláknového zpracování paprsků, využití single instruction multiple data (SIMD) instrukcí a použití hierarchie obálek jako jedné z nejpopulárnějších akceleračních struktur pro sledování paprsků. Knihovna bude navíc snadno rozšiřitelná o další akcelerační struktury. Její využití je možné také pro výukové účely, při seznamování se s metodami využívajícími vrhání paprsků.

První část diplomové práce se zabývá teoretickým úvodem do problému syntézy obrazu pomocí metod globálního osvětlování a to konkrétně metodou vrhání paprsků a sledování cesty. Popsány jsou zde již existující řešení a v detailu poté mnou použité algoritmy a akcelerační struktury. Druhá část diplomové práce se zabývá samotnou implementací knihovny a výsledky měření na statických i dynamických scénách.

Obsah

1 Metody sledování paprsku **3**
- 1.1 Definice základních pojmů . . . 3
  - 1.1.1 Zobrazovací rovnice . . . 3
  - 1.1.2 Metody Monte Carlo . . . 4
  - 1.1.3 Dvousměrová odrazová distribuční funkce . . . 4
  - 1.1.4 Formální notace transportu světla . . . 5
  - 1.1.5 Světelné zdroje . . . 6
  - 1.1.6 Světelné jevy . . . 6
  - 1.1.7 Phongův osvětlovací model . . . 8
  - 1.1.8 Lokální zobrazovací metody . . . 9
- 1.2 Globální zobrazovací metody . . . 9
  - 1.2.1 Rekurzivní sledování paprsku . . . 9
  - 1.2.2 Sledování cesty . . . 11
- 1.3 Urychlení výpočtů . . . 13
  - 1.3.1 Paralelizace . . . 13
  - 1.3.2 Hierarchie obálek . . . 13
    - 1.3.2.1 Stavba hierarchie obálek . . . 14
    - 1.3.2.2 Traverzace hierarchie obálek . . . 16
- 1.4 Sledování paprsků a dynamické scény . . . 17

2 SIMD architektura **19**
- 2.1 SSE . . . 20
- 2.2 AVX . . . 21
- 2.3 Organizace dat . . . 21
- 2.4 Volání instrukcí . . . 21
- 2.5 Využití v metodách sledování paprsků . . . 22

3 Implementace **25**
- 3.1 Existující implementace . . . 25
- 3.2 Návrh rozhraní knihovny . . . 26
- 3.3 Implementace incidenčních operací . . . 29
  - 3.3.1 Průsečík paprsku s trojúhelníkem . . . 29
    - 3.3.1.1 Möllerův algoritmus . . . 29
    - 3.3.1.2 Waldův algoritmus . . . 30
    - 3.3.1.3 Havel/Heroutův algoritmus . . . 31

3.3.2 Průsečík paprsku s obalovým tělesem . . . 33
3.4 Testovací aplikace . . . 34
3.4.1 Konfigurační soubor . . . 34
3.4.2 Syntéza pomocí sledování paprsku . . . 35
3.4.3 Syntéza pomocí sledování cesty . . . 36

4 Výsledky **39**
4.1 Statické scény . . . 39
4.1.1 Parametry testovaných statických scén . . . 40
4.1.2 Porovnání postavených hierarchií obálek . . . 41
4.1.3 Porovnání standardní implementace proti SIMD . . . 42
4.1.4 Vliv více vláknového zpracování . . . 44
4.1.5 Profiling aplikace . . . 44
4.2 Porovnání lokální a globální osvětlovací metody . . . 45
4.3 Dynamické scény . . . 47
4.4 Porovnání naimplementovaných algoritmů . . . 53

5 Závěr **55**
5.1 Cíle diplomové práce . . . 55
5.2 Možná rozšíření . . . 55

Literatura **57**

A Porovnání standardní a SSE implementace **59**

B Instalační a uživatelská příručka **61**

C Obsah přiloženého CD **63**

Seznam obrázků

1.1 Difuzní, lesklý a čistě zrcadlový odraz světla . . . 5
1.2 Příklad použití notace transportu světla . . . 6
1.3 Nepřímý difuzní odraz . . . 7
1.4 Výpočet difuzní složky . . . 8
1.5 Výpočet spekulární složky . . . 9
1.6 Typy paprsků . . . 10
1.7 Vliv počtu paprsků procházejících jedním pixelem . . . 12
1.8 Ilustrace hierarchie obálek . . . 14
1.9 Možnosti dělení těles při stavbě hierarchie obálek [13] . . . 15
1.10 Úprava dvou obálek po změně souřadnic těles [13] . . . 18

2.1 Výpočet absolutní hodnoty . . . 19
2.2 Organizace dat vhodná pro SIMD [12] . . . 22
2.3 Ukázka SSE4 intrinsik . . . 22
2.4 Ukázka AVX intrinsik . . . 22

3.1 Datové struktury Vertex a Triangle . . . 26
3.2 Datové struktury Ray a Hit . . . 27
3.3 Výčtové typy . . . 27
3.4 Datová struktura Raycaster . . . 28
3.5 Posunutí trojúhelníku a zarovnání směru paprsku [8] . . . 30
3.6 Roviny použité pro výpočet průsečíku paprsku s trojúhelníkem [5] . . . 32
3.7 Znázornění bodů, kde paprsek protíná hraniční roviny obálky . . . 34
3.8 Aktivity diagram sledování paprsků . . . 36
3.9 Ukázka uživatelského rozhraní aplikace . . . 37
3.10 Aktivity diagram sledování cesty . . . 37

4.1 Obrázky testovaných statických scén . . . 41
4.2 Výsledky profilingu hierarchie obálek . . . 45
4.3 Porovnání sledování paprsku a sledování cesty . . . 46
4.4 Porovnání vzorkování difuzních odrazů . . . 46
4.5 Obrázky testovaných dynamických scén . . . 47
4.6 Scéna běžce s nově postavenou hierarchií pro každý snímek . . . 50
4.7 Scéna běžce s úpravami původní hierarchie . . . 50
4.8 Scéna Toasters s nově postavenou hierarchií pro každý snímek . . . 51
4.9 Scéna Toasters s úpravami původní hierarchie . . . 51

4.10 Scéna lesní víly s nově postavenou hierarchií pro každý snímek 52
4.11 Scéna lesní víly s úpravami původní hierarchie 52
4.12 Průměrný počet traverzačních kroků na paprsek u obou metod 52
4.13 Porovnání rychlosti algoritmů pro výpočet průsečíku paprsek/trojúhelník . . . 53

A.1 Standardní implementace Havel/Heroutova algoritmu 59
A.2 SSE implementace Havel/Heroutova algoritmu 60

Seznam tabulek

4.1 Parametry testovaných statických scén 41
4.2 Parametry testovaných statických scén 42
4.3 Traverzace primárních a stínových paprsků - hodnoty jsou uvedené v milionech paprsků za vteřinu 43
4.4 Průměrný počet traverzací na primární paprsek (vnitřních uzlů + listů) . . . 43
4.5 Vliv vícevláknového zpracování na rychlost traverzace primárních paprsků . . 44
4.6 Parametry testovaných dynamických scén 48
4.7 Shrnutí výsledků dynamických scén - všechny hodnoty jsou průměrem ze všech snímků, traverzace měřena na primárních paprscích 49

Úvod

Jedním z hlavních cílů počítačové grafiky je vytvořit co nejrealističtěji vypadající obraz syntetické scény. Takto vytvořený obrazy mají široké spektrum použití od filmového průmyslu, fyzikálních simulací, herního průmyslu, vizualizace virtuálních prototypů nových výrobků, architektonických staveb, simulace osvětlení interiérů a mnoho dalších. V několika posledních letech dosáhla počítačová grafika takové úrovně, kdy již v některých případech není prakticky možné rozlišit skutečnou fotografii od obrazu vytvořeného na počítači. Díky vzrůstajícímu výkonu moderních procesorů je navíc možné některé jednoduché scény vykreslovat v reálném čase. Existuje velké množství metod, kterými je takový fotorealistický obraz možné vytvořit. Já se v mé diplomové práci soustředím na dvě tyto metody, a to algoritmus sledování paprsku a algoritmus sledování cesty.

Cílem mé práce je vytvořit knihovnu, kterou bude možné využít v aplikaci používající metody vrhání paprsků. Aby bylo možné vrhat paprsky dostatečně rychle a efektivně, musí knihovna implementovat optimalizační metody a datové struktury jako jsou hierarchie obálek, vícevláknové zpracování a SIMD instrukční sady dnešních procesorů.

Diplomová práce je strukturována následujícím způsobem. První kapitola obsahuje úvod do syntézy obrazu pomocí metod sledování paprsku. Jsou v ní definovány základní pojmy, které jsou v práci dále používány. V této kapitole je také představa datová struktura hierarchie obálek, která je ve výsledné knihovně implementována. Druhá kapitola se zabývá SIMD instrukčními sadami SSE4 a AVX, a jejich využitím v metodách sledování paprsku. Poté následují dvě kapitoly popisující implementační detaily a testování. Knihovna byla testována na třech statických a třech dynamických scénách, které by měly potvrdit předpoklady vycházející z teoretického úvodu v prvních dvou kapitolách. V poslední kapitole jsou shrnuty výsledky testování a diskutován přínos vzniklé aplikace s jejím použitím a možnými rozšířeními.

Kapitola 1

Metody sledování paprsku

V počítačové grafice se metodou sledování paprsku rozumí generování obrazu pomocí paprsků, které procházejí scénou. Paprsky je možné vrhat buď směrem od kamery do scény, což není příliš podobné tomu, jak pracuje fotoaparát nebo lidské oko, nebo je možné paprsky vrhat od světel do scény a sledovat jejich cestu až do kamery. Motivace pro sledování paprsků od kamery je zřejmá. Většina paprsků vyzářených ze zdroje světla do scény se ke kameře vůbec nedostane. Ve scéně ještě navíc dochází k odrazům paprsků, výpočtu osvětlení a dalším operacím potřebným k získání věrně vypadajícího obrazu.

1.1 Definice základních pojmů

Předtím než postupně rozeberu metody sledování paprsků implementované v mé diplomové práci, rád bych v této podkapitole připomněl základní pojmy, na které se budu odvolávat dále v textu nejen této kapitoly.

1.1.1 Zobrazovací rovnice

Zobrazovací rovnice popisuje rozložení světla ve scéně a udává pro každý bod jeho radianci. Radiance představuje počet fotonů přicházejících či vyzářených v určitém směru za jednotku času a procházející průmětem diferenciální plošky, která je kolmá na tento směr [11]. Řešením zobrazovací rovnice je ustálený stav ve scéně, protože kritériem zobrazovací rovnice je zákon zachování energie, který říká, že vyzářená radiance musí být někde odražena či absorbována. Rovnice má následující tvar [6]

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_\Omega f(x,\vec{\omega},\vec{\omega_i}) L_o(x,-\vec{\omega_i}) cos\theta \mathrm{d}\vec{\omega_i}$$

$L_o(x,\vec{\omega})$ představuje celkovou radianci opouštějící bod ve směru ω
$L_e(x,\vec{\omega})$ představuje radianci, která je bodem vyzařována ve směru ω (platí pro světelné zdroje)
$f(x,\vec{\omega},\vec{\omega_i})$ je dvousměrná odrazová distribuční funkce v bodě x ze směru $\vec{\omega}$ do $\vec{\omega_i}$
θ je úhel sevřený normálovým vektorem a směrem ω_i
Ω je hemisféra ve směru normály přes kterou integrujeme

Analytické řešení této rovnice je možné pouze pro nejjednodušší případy. Pro všechny ostatní případy není možné tuto rovnici analyticky vyřešit, neboť hodnota radiance v každém bodě scény závisí na vyzářené a odražené radianci z okolních ploch. Osvětlovací metody představují aproximaci řešení.

1.1.2 Metody Monte Carlo

K vyřešení (nebo alespoň k přiblížení se řešení) zobrazovací rovnice u složitějších scén potřebujeme využít jednu z variant numerické integrace. Tou nejčastěji používanou v počítačové grafice je metoda Monte Carlo. Jedná se o stochastickou metodu používanou pro odhad hodnoty integrálu a jedním z typických příkladů jejího použití je právě zobrazovací rovnice. Pomocí metody Monte Carlo chceme určit hodnotu integrálu

$$I = \int_{\omega} f(x)\,\mathrm{d}x$$

Metoda Monte Carlo provede odhad hodnoty integrálu na základě řady vzorků, jejichž hodnoty ve výsledku zprůměruje. Tyto vzorky jsou generovány náhodně, což již vyplývá z toho, že se jedná o stochastickou metodu. S větším počtem vzorků se dostáváme blíže a blíže k přesné hodnotě integrálu, která je na druhou stranu vyvážena delším výpočetním časem. Hlavní nevýhodou metod Monte Carlo je jejich pomalá konvergence, kdy pro zmenšení chyby o polovinu potřebujeme vygenerovat čtyřikrát více vzorků. Další nepříjemnou vlastností je šum, který do výsledku vnáší nevhodně vybrané vzorky. Právě na vybírání správných vzorků se soustředí optimalizační techniky, které jsem použil také ve své práci.

První z metod se nazývá ruská ruleta [1]. Princip ruské rulety spočívá v přenesení větší váhy na vzorky tam, kde je příspěvek integrované funkce velký a snížit pocet vzorků tam, kde je naopak příspěvek malý. Ke každému vzorku můžeme vygenerovat náhodné číslo od nuly do jedné, které nám udává pravděpodobnost p. Vzorky poté vydělíme pravděpodobností p, čímž zvětšíme jejich příspěvek dle jejich pravděpodobnosti, ale jejich střední hodnota se tím pádem nezmění.

Druhou metodou optimalizace je vzorkování podle důležitosti. Tato metoda využívá toho, že některé části funkce přispívají k odhadu výsledku více, než ostatní. Typicky jsou to místa s významnými nebo rychlými změnami průběhu funkce. Jinými slovy tedy soustřeďujeme více vzorků do důležitých míst integrované funkce. Z toho důvodu je nutné, aby vzorkovací funkce měla hustotu pravděpodobnosti velice podobnou integrované funkci.

Popis obou těchto metod bude dále doplněn o příklady jejich použití u popisu algoritmu sledování cesty, který využívá metody Monte Carlo včetně obou optimalizačních metod.

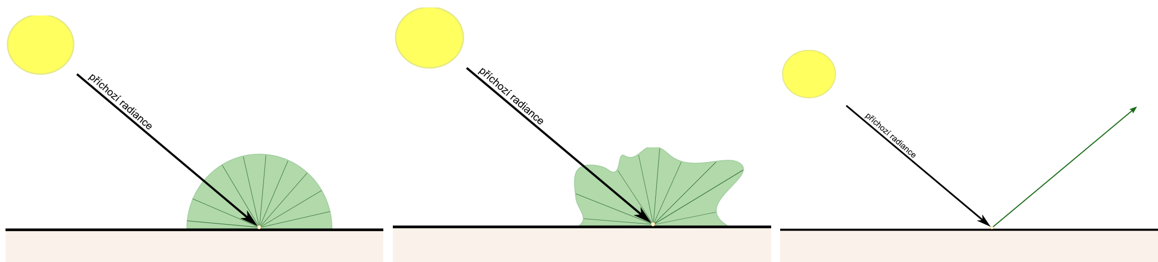
1.1.3 Dvousměrová odrazová distribuční funkce

Většina světla, které dopadá do lidského oka, je světlo odražené od povrchů objektů, které nás obklopují. Dvousměrová odrazová distribuční funkce (z anglického Bidirectional Reflectance Distribution Function - BRDF) nám poskytuje nástroj k tomu, jak popsat schopnost materiálu odrážet nebo absorbovat dopadající světlo.

Matematický popis dvousměrné odrazové distribuční funkce vypadá následovně [11]

$$f_r(x, \vec{\omega_r}, \vec{\omega_i}) = \frac{dL_r(x, \vec{\omega_r})}{dL_i(x, \vec{\omega_i})(\vec{\omega_i}.\vec{n})d\vec{\omega_i}}$$

kde $\omega_{i,r}$ představuje dopadající, resp. odražený směr světla v bodě x. BRDF definuje poměr odražené radiance $dL_r(x, \vec{\omega_r})$ ke vstupní diferenciální radianci $dL_i(x, \vec{\omega_i})$ promítnutné na kolmou plochu. Důležitými vlastnostmi BRDF je, že nikdy není záporná, a že v daném bodě zůstává stejná, i když zaměníme směr dopadu a odrazu světelného paprsku. Nejběžnější způsob reprezentace BRDF v počítačové grafice je empirický model.



Obrázek 1.1: Difuzní, lesklý a čistě zrcadlový odraz světla

1.1.4 Formální notace transportu světla

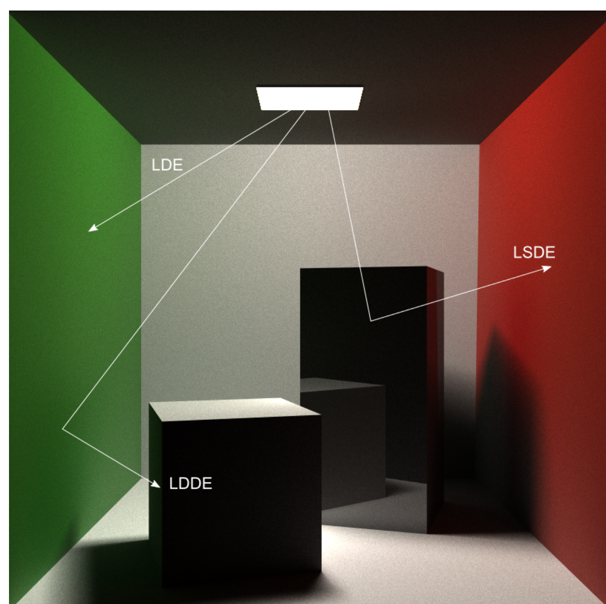
Abych mohl v dalších částech mé práce snáze popsat cesty, kterými se šíří světlo ve scéně, je potřeba zavést notaci, která toto šíření bude popisovat. Rozhodl jsem se držet notace užívané ve většině odborných článků a publikací. Každá cesta je ohraničena zdrojem světla a kamerou (okem pozorovatele)

- E - oko, kamera
- L - zdroj světla

Každý průsečík paprsku se scénou lze popsat pomocí jednoho z následujících tří symbolů

- D - dopad a odraz z difuzního povrchu
- S - dopad a odraz ze zrcadlového povrchu
- G - dopad a odraz z lesklého povrchu

Rozdíl mezi zrcadlovým a lesklým povrchem je ten, že zrcadlový povrch nemá žádnou difuzní složku. V dalších kapitolách bude tato notace používána také ve formě regulárních výrazů, k přesnému popisu jednotlivých algoritmů.



Obrázek 1.2: Příklad použití notace transportu světla

1.1.5 Světelné zdroje

Některé reálné světelné zdroje by byly z výpočetního hlediska velice náročné na korektní výpočet jejich příspěvku k osvětlení scény. Proto se v počítačové grafice zavádí několik typů světelných zdrojů, pomocí kterých se snažíme navodit co nejpřirozenější vjem, a které jsou dostatečně jednoduché na použití.

Bodové světlo je určené svou pozicí, barvou a světlo vyzařuje do prostoru rovnoměrně do všech směrů. Můžeme si ho představit jako jediný bod v prostoru, který se většinou do výsledného obrazu většinou nijak explicitně nevykresluje. Použitím bodového světla dosáhneme pouze ostrých stínů.

Plošné světelné zdroje se používají u sofistikovanějších metod sledování paprsků, protože tyto zdroje je potřeba simulovat určitým počtem vzorků. Je možné s nimi dosáhnout měkkých stínů, ale na druhou stranu je jejich použití výpočetně náročnější, protože zpravidla potřebujeme použít více vzorků než v případě bodových světel. Dalším světelným zdrojem ve scéně mohou být samotná tělesa, jejichž materiál světlo sám vyzařuje.

1.1.6 Světelné jevy

Aby byl získaný obraz co nejvěrnější skutečnosti, je potřeba, aby zvolená metoda syntézy obrazu simulovala co nejvěrněji světelné jevy, ke kterým dochází ve skutečném světě. Mezi ty jevy, které příspívají k celkovému vjemu nejvíce, patří stíny, přímý a nepřímý difuzní odraz světla a přenos barvy, spekulární odraz světla, lom světla, kaustiky a hloubka ostrosti.

Stíny poskytují velice důležitou informaci o tom, jak jsou objekty rozmístěny v prostoru a jaká je pozice světel. Stíny můžeme rozdělit do dvou kategorií a to na měkké stíny a tvrdé stíny. Tvrdé stíny se vyskytují u bodových světelných zdrojů. Z žádného místa na tvrdém

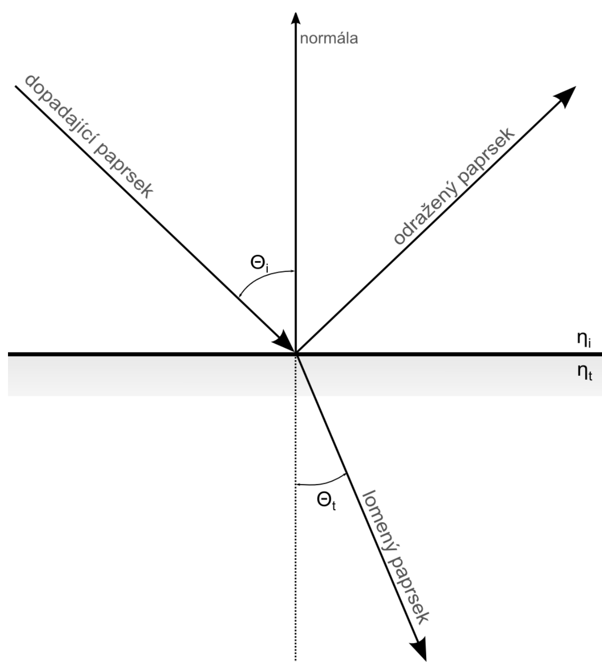
stínu není zdroj světla viditelný. Měkké stíny označují polostín, ze kterého je zdroj světla částečně vidět a zároveň je částečně zakrytý.

Přímý difuzní a spekulární odraz světla jsou popsány dále v podkapitole věnované Phongovu osvětlovacímu modelu. Nepřímý difuzní odraz vzniká v okamžiku, kdy odražené světlo osvětlí objekt nacházející se v blízkosti původního tělesa. Tento efekt dobře demostruje obrázek 1.2, na kterém je zřetelně vidět, jak červená stěna ovlivňuje barvu zadní zdi, která se nachází v její blízkosti.

Při syntéze počítačové scény musíme počítat i s případem, kdy scéna obsahuje objekty z průhledných materiálů, například ze skla. U takových materiálů dochází k lomu světla (refrakci), protože světlo se dostane na rozhraní dvou prostředí s různou optickou hustotou. Světelný paprsek je rozdělen na dvě části - na část odraženou zpět a na část lomenou, která pokračuje v průchodu objektem. Úhel, pod kterým bude lomený paprsek pokračovat v průchodu objektem, lze vypočítat ze Snellova zákona lomu

$$\frac{\sin \omega_i}{\sin \omega_t} = \frac{\eta_t}{\eta_i}$$

kde ω_i představuje úhel dopadu, ω_t úhel lomu a $\eta_{t,i}$ představuje absolutní index lomu prostředí, ve kterém se šíří dopadající, resp. lomený paprsek.



Obrázek 1.3: Nepřímý difuzní odraz

1.1.7 Phongův osvětlovací model

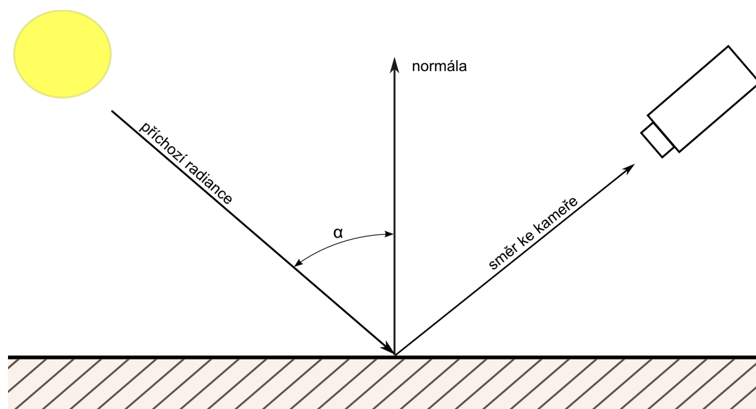
Při použití globálních i lokálních osvětlovacích metod potřebujeme nějaký empirický model pro výpočet odraženého světla z povrchu tělesa. V mé diplomové práci jsem zvolil Phongův osvětlovací model [10], který je dostatečně efektivní, snadný na implementaci a nejspíše i vůbec nejpoužívanějším osvětlovacím modelem v počítačové grafice. Výsledný odraz světla se skládá ze tří složek: ambientní, difuzní a spekulární.

Ambientní složka představuje všesměrové světlo, které na těleso dopadá rovnoměrně a ze všech směrů se stejnou intenzitou. Jedná se tedy o zjednodušený výpočet příspěvku ostatních těles ve scéně. Při použití ambientního osvětlení nebudou odvrácené plochy černé, ale budou mít barvu ambientní složky.

Difuzní odraz rozptyluje dopadající radianci do všech směrů rovnoměrně. Tento odraz tedy nezáleží na pozici pozorovatele, ale je ve všech úhlech stejný. Difuznímu odrazu obecně říkáme barva povrchu. Výpočet difuzního osvětlení je ve Phongově osvětlovacím modelu vyjádřen následovně

$$c_{dif} = (max[cos(\alpha), 0]) * c_l * c_m$$

kde α je dána jako úhel, který svírá dopadající paprsek s normálou a lze je spočítat jako skalární součin obou vektorů. Proměnná c_l představuje difuzní složku světla a c_m difuzní složku materiálu v místě dopadu. Ze vzorce je tedy vidět, že difuzní složka se mění s kosinem úhlu dopadu, který je maximální při dopadu kolmo na plochu.

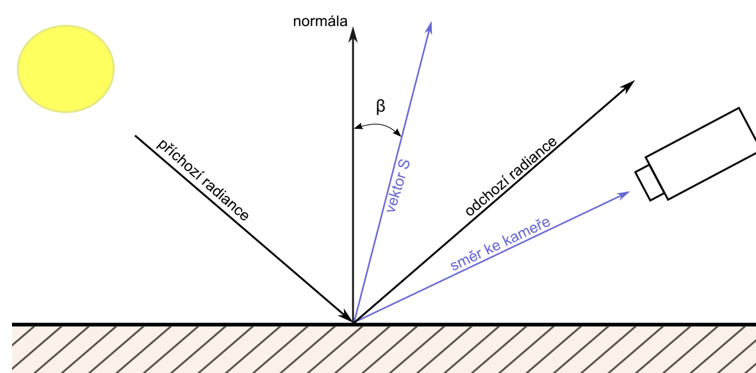


Obrázek 1.4: Výpočet difuzní složky

Spekulární (zrcadlový) odraz odráží dopadající radianci pod stejným úhlem, pod kterým na povrch dopadá. Tento typ odrazu je příčinou odlesků na površích těles. Koeficientem odrazivosti je navíc možné určit množství odražené radiance v jednotlivých směrech. Výpočet zrcadlového osvětlení je vyjádřen následujícím vzorcem

$$c_{spec} = (max[cos(\beta), 0])^e * c_l * c_m$$

Výpočet spekulárního osvětlení je dobře vidět na obrázku č. 1.5. Jak je z obrázku patrné, $cos(\beta)$ je maximální v momentě, kdy se světlo odráží přímo do oka pozorovatele. Čím větší je konstanta e ve vzorci (angl. shininess), tím je spekulární odraz ostřejší.



Obrázek 1.5: Výpočet spekulární složky

1.1.8 Lokální zobrazovací metody

Algoritmy syntézy obrazu můžeme rozdělit do dvou základních kategorií, na lokální a globální. Pojem lokální znamená, že osvětlení objektu ve scéně je dáno pouze vlastnostmi jeho materiálu a světlem příchozím z přímých světelných zdrojů. Pomocí notace zavedené v předchozí podkapitole bychom lokální metody mohli popsat jako $E(D|G)L$. To znamená, že tyto metody počítají s jedním difuzním nebo lesklým odrazem. V pokročilejších implementacích mohou podporovat i zrcadlové odrazy do určité hloubky. Jak je z notace patrné, osvětlení jednoho tělesa je závislé pouze na jeho materiálu a poloze vůči světlům. Výsledné obrazy produkované lokálními zobrazovacími metodami nepatří k těm fotorealistickým, protože nedokáží simulovat měkké stíny, nepřímý difuzní přenos barvy, nebo efekty vznikající při průchodu světla skleněnými nebo průhlednými objekty. Typickým příkladem lokální zobrazovací metody je sledování paprsku.

1.2 Globální zobrazovací metody

Vylepšením lokálních zobrazovacích metod vznikají globální zobrazovací metody, které již počítají i s nepřímým osvětlením ve scéně. Tyto metody přistupují ke všem povrchům jako ke zdrojům světla. Jako příklad pro globální zobrazovací metodu nám může posloužit metoda sledování cesty. V průběhu výpočtu osvětlení pomocí sledování cesty se nejdříve u každého tělesa vyhodnotí jeho přímé osvětlení a poté nepřímé příspěvky od ostatních objektů ve scéně. Výsledné obrazy působí daleko realističtěji než obrazy získané pomocí lokálních zobrazovacích metod, ale na druhou stranu jsou globální zobrazovací metody výpočetně náročnější a vyžadují množství dalších optimalizací, aby je bylo možné použít v aplikacích běžících v reálném čase. Použitím globálních zobrazovacích metod je možné simulovat všechny jevy zmíněné v předcházející podkapitole.

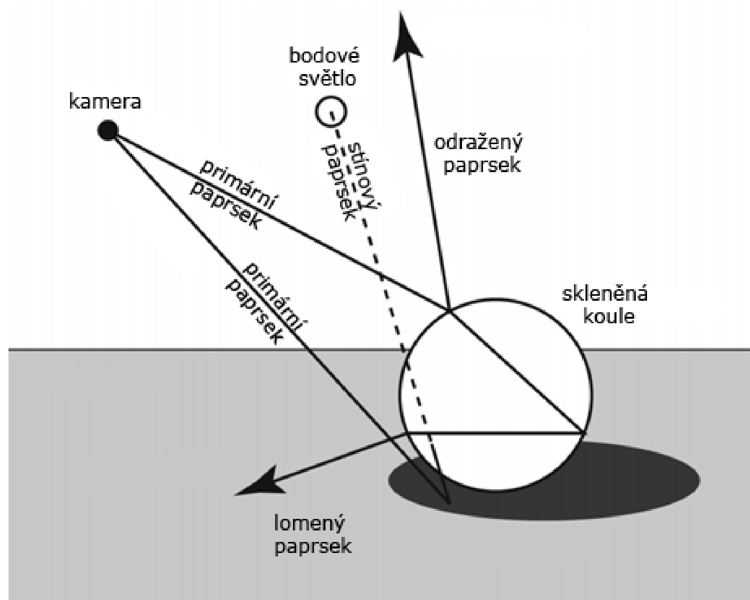
1.2.1 Rekurzivní sledování paprsku

První z nich je metoda sledování paprsku. Každým pixelem na obrazovce vedeme paprsek, který má počátek v kameře a jeho směr je dán právě pixelem, kterým má paprsek procházet.

Tento paprsek se nazývá primární a můžeme ho matematicky popsat následovně:

$$R(t) = O + t * D$$

kde O představuje počátek paprsku, D jeho směr a t je parametr udávající polohu na paprsku. Můžeme specifikovat parametr t_{max}, který představuje maximální možnou vzdálenost průsečíku paprsku se scénou. Použijeme ho hlavně při vrhání primárních paprsků, kdy porovnáváme několik nalezených průsečíků, protože nás v tomto případě zajímá ten nejbližší. Pokud paprsek žádné těleso nezasáhne, pixel bude mít buď barvu pozadí, nebo je možné například použít barvu získanou z textury. Pokud paprsek nějaké těleso zasáhne, vedeme z tohoto místa další paprsek směrem ke světlu. Tyto paprsky nazýváme stínové. V případě stínových paprsků nám stačí vědět, že se mezi počátkem paprsku a světlem nachází nějaký objekt. Pokud se tento průsečík nachází blíže, než je vzdálenost od počátku paprsku ke světlu, je bod, představující počátek paprsku, zastíněn. Výsledkem tohoto algoritmu je tedy obrázek scény s difuzním osvětlením a ostrými stíny. Pokud by algoritmus sledování paprsku v tomto bodě skončil, jednalo by se o lokální zobrazovací metodu.



Obrázek 1.6: Typy paprsků

Turner Whitted (po něm tak vznikl anglický název Whitted ray tracing) ve svém článku [20] přidal ještě třetí druh paprsků, sekundární, a představil algoritmus rekurzivního sledování paprsků. Kompletní algoritmus se tedy skládá z již zmíněných primární paprsků, stínových paprsků a sekundárních paprsků. Sekundární paprsky se vrhají v případě, že primární paprsek zasáhl těleso s lesklým povrchem nebo průhledným materiálem. Sekundární paprsek má jako svůj počátek místo zásahu primárního paprsku a jeho směr je dán odrazem od povrchu tělesa. Hloubku rekurze můžeme řídit proměnnou, která zamezí vytváření dalších sekundárních paprsků po překročení definované hloubky rekurze. Na obrázku č. 1.6 jsou vidět

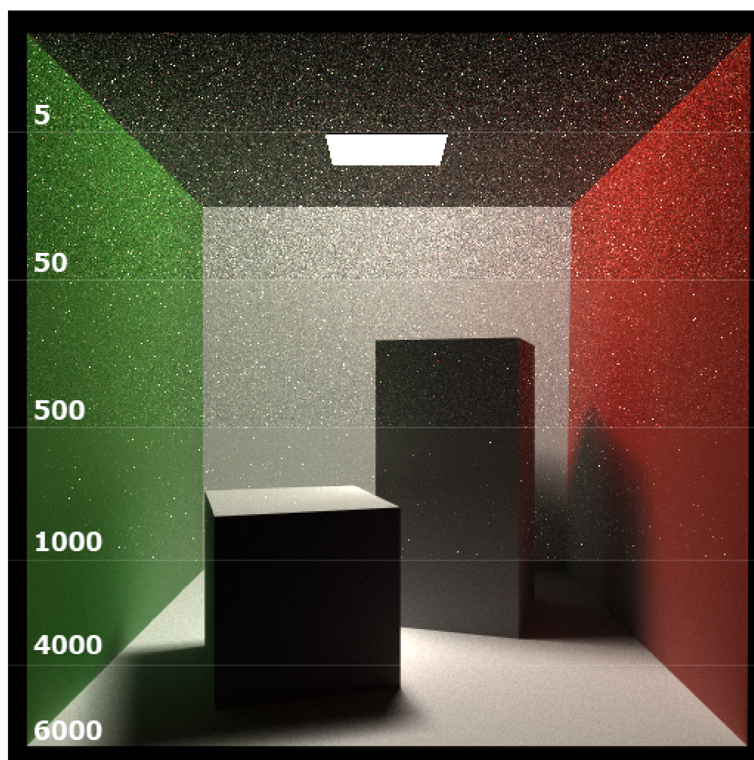
všechny typy paprsků, které mohou při rekurzivním sledování paprsků vzniknout. Metoda sledování paprsku v této verzi ještě není schopna vypočítat všechny jevy jako jsou kaustiky, difuzně-difuzní odrazy a plošné zdroje světla. K tomu nám slouží až tzv. distribuované sledování paprsku [3].

Hlavní rozdíl distribuovaného sledování paprsku oproti klasickému algoritmu je v tom, že při klasickém algoritmu se vrhne jeden stínový paprsek do každého bodového zdroje světla, takže jasně víme, jestli je daný bod zastíněn. Při distribuovaném sledování paprsku se vrhne n sekundárních paprsků po hemisféře a spočítá se příspěvek přímého a nepřímého osvětlení. Tato metoda je velmi výpočetně náročná metoda, protože z každých n paprsků je poté nutné vygenerovat dalších n v následném kroku rekurze, což nám dává exponenciální složitost. Méně výpočetně náročné řešení s podobnými výsledky nám nabízí algoritmus sledování cesty (path tracing).

1.2.2 Sledování cesty

Sledování cesty, stejně jako distribuované sledování paprsku, patří k stochastickým zobrazovacím metodám. Jak bylo popsáno v podkapitole věnované těmto metodám, Monte Carlo se snaží odhadnout hodnotu integrálu na základě odhadů, jejichž výsledek se zprůměruje. V syntéze obrazu pomocí algoritmu sledování cesty jsou těmito odhady tisíce až deseti tisíce primárních paprsků, které vrháme každým pixelem na obrazovce. Zprůměrování všech získaných hodnot poté dostaneme barvu samotného pixelu. Kdybychom vedli paprsek vždy středem pixelu, dostávali bychom pro každý paprsek velice podobné hodnoty a výsledek by nepůsobil realisticky. Z toho důvodu je potřeba definovat rozložení paprsků pro každý pixel. Například si můžeme vygenerovat dostatečný počet vzorků s hodnotou od -0.5 do 0.5, které přičítáme ke středu pixelu, a teprve tyto body určují směr paprsku. Pro generování vzorků můžeme vybírat z velkého množství způsobů - od stochastického, až po sofistikovanější metody vzorkování jako je roztřesené (jittered) vzorkování nebo n-věží (n-rooks). V každém průsečíku primárního paprsku se scénou se vypočítá jeho přímé osvětlení včetně zastínění. Sledování cesty dokáže pracovat i s plošnými zdroji světla, takže při vrhání stínového paprsku můžeme vyslat paprsek buď k jednomu světlu (pokud je plošné, musíme jeho polohu určit pomocí několika vzorků na jeho povrchu), nebo k více světlům a postupně jejich příspěvky vážit např. jejich velikostí, nebo intenzitou.

Po nalezení průsečíku primárního paprsku se scény a výpočtu přímého osvětlení se určí směr dalšího odrazu. V algoritmu sledování paprsku se sekundární paprsky vytvářely pouze při zrcadlovém odrazu, zatímco při sledování cesty se sekundární paprsky vytvářejí pokaždé. Při tvorbě sekundárního paprsku se rozhodujeme na základě toho, zda je zasažený povrch difuzní, nebo obsahuje také spekulární složku. V případě pouze difuzního odrazu můžeme náhodně vybrat směr v polokouli ve směru normály povrchu. Pokud se rozhodneme pro zrcadlový odraz, musíme si spočítat směr, do kterého se přicházející paprsek odrazí. V tomto místě můžeme aplikovat navrhovaná vylepšení metody Monte Carlo, které byly zmíněné výše. Počet odrazů ve scéně můžeme buď pevně definovat konstantou, nebo využít ruskou ruletu. Ruskou ruletu můžeme využít také pro určení dalšího odrazu paprsku na površích s difuzní i spekulární složkou. S pravděpobností p bude odraz difuzní, s pravděpodobností 1-p bude spekulární. Vzorkování podle důležitosti můžeme aplikovat na vybírání směru odraženého paprsku. V nejjednodušším případě vybereme náhodně jakýkoliv bod, který leží



Obrázek 1.7: Vliv počtu paprsků procházejících jedním pixelem

na polokouli ve směru normály povrchu. Pokud chceme využít vzorkování podle důležitosti, můžeme použít vzorkování vážené kosinem, které nám vzorky umístí blíže k normále. Při spekulárním odrazu můžeme vzorky umísťovat blíže k úhlu ideálního odrazu dopadajícího paprsku. Praktický vliv těchto optimalizací bude rozebrán v kapitole věnované testování.

Hlavní problém metody sledování cesty je to, že odražené paprsky jsou náhodné, což do výsledného obrázku vnáší šum. Dle [11] nemá na výslednou kvalitu takový vliv zvýšení počtu odrazů, ale spíše počet paprsků, které se vrhnou pro každý pixel. Mé testování toto potvrzuje. Pro většinu scén tak postačuje pět až deset odrazů. Naopak pro dosažení velmi kvalitních výsledků je potřeba každým pixelem vrhnout tisíc až několik desítek tisíc primárních paprsků, jejichž průměrem se poté získá výsledná barva pixelu.

Celá metoda je poměrně výpočetně náročná, ale jejím výsledkem jsou simulace téměr všech jevů, které se odehrávají i ve skutečném světě, jako například nepřímé difuzními odrazy, kaustiky, přenos barvy difuzním odrazem a potlačení aliasingu, který vzniká při metodě rekurzivního sledování paprsku. Zavedenou notací tranportu světla bychom mohli sledování cesty popsat jako $E[(D|G|S) + (D|G)]L$.

1.3 Urychlení výpočtů

Navzdory poměrně jednoduchému zadání – najít nejbližší průsečík polopřímky (paprsku) s množinou objektů – je hlavní nevýhodou metod sledování paprsku rychlost výpočtu. Naivní algoritmus prochází všechny objekty po jednom a testuje, zda právě tento objekt je paprskem protnut a zda je tento průsečík nejbližší. Jeho složitost je tedy $O(n)$ pro n objektů. Tento algoritmus je ovšem dostačující pouze pro opravdu minimalistické scény skládající se z maximálně desítek trojúhelníku. Berme v úvahu, že musíme počítat průsečík se všemi objekty pro každý pixel a pokud nám jde i o stínové a sekundární paprsky, tento postup sledování paprsku je prakticky nepoužitelný. Z výše popsaných algoritmů vyplývá, že způsoby, kterými lze urychlit výpočet obrazu pomocí metod sledování paprsků, jsou následující:

- Snížení počtu vrhaných paprsků
- Paralelizace procedury vrhání paprsků
- Sledování více paprsků naráz
  - více o této optimalizaci v samostatné kapitole věnované SIMD architektuře
- Snížení počtu incidenčních operací
  - více o této optimalizaci v podkapitole věnované akceleračním strukturám
- Urychlení výpočtů incidenčních operací
  - o existují více a méně rychlé algoritmy pro výpočet incidenčních operací, kdy u těch rychlejších je většinou rychlost vykoupena většími paměťovými nároky

V podkapitolách níže jsou zmíněné nejčastěji používané způsoby urychlení sledování paprsků s tím, že jejich dopad na celkovou dobu výpočtu je možné najít níže v kapitole věnované testování.

1.3.1 Paralelizace

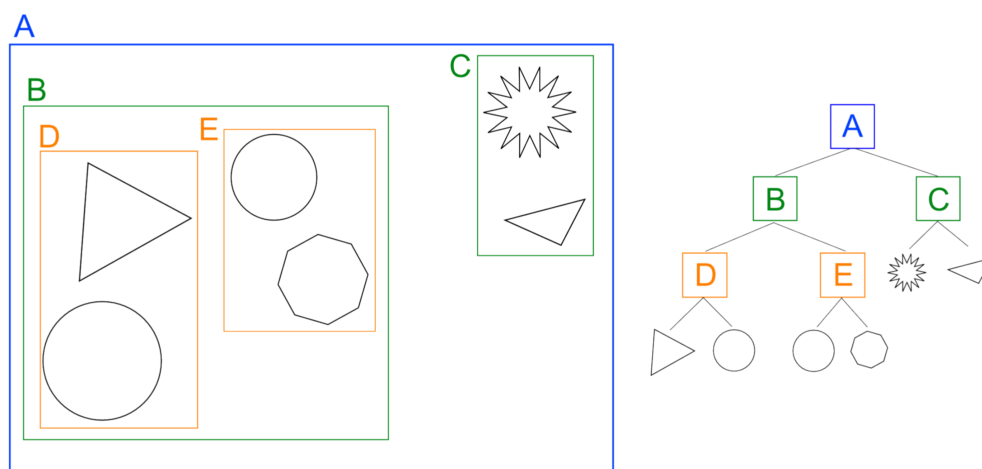
Jednou z velkých výhod metody sledování paprsků je to, že celý algoritmus se dá velice snadno paralelizovat a to z toho důvodu, že scény jsou tvořeny statickými daty, která jsou aplikaci všechna známá ještě před vrháním paprsků. Navíc algoritmus k těmto datům pouze přistupuje, nemodifikuje je. Této vlastnosti jsem využil i já a více o mé implementaci paralelního vrhání paprsků je v kapitole implementace.

1.3.2 Hierarchie obálek

K tomu, abychom nemuseli v každém pixelu hledat průsečíky s každým objektem ve scéně, jsou určeny akcelerační struktury. Pod tímto názvem si můžeme představit skupinu datových struktur, které podle nějakého klíče dělí buď prostor nebo objekty do menších skupin. Tyto datové struktury jsou schopné traverzovat desítky až stovky milionů paprsků za vteřinu, čímž výrazně urychlují celý proces syntézy obrazu pomocí metod sledování paprsků.

Hierarchie obálek patří k vůbec nejpoužívanějším datovým strukturám, které se při vrhání paprsků používají. Jedná se o stromovou datovou strukturu, která jednotlivé objekty obaluje tělesy, které se jednoduše testují na průsečík s paprskem. Těmito tělesy mohou být osově zarovnané kvádry, koule, nebo mnohostěny. Kay a Kajiya [6] definují následující doporučení, které vedou k postavení kvalitních hierarchií:

- Podstromy by měly obsahovat objekty, které se scéně nacházejí blízko u sebe a čím hlouběji jsme ve stromu, tím blíže by se objekty měly nacházet
- Objem každého uzlu by měl být minimální
- Suma objemů všech uzlů by měla být minimální
- Větší pozornost by se měla věnovat uzlům, které se nacházejí blíže ke kořenu, protože tyto části stromu jsou traverzovány nejčastěji a vhodným rozdělením lze z traverzace odstranit velké množství objektů
- Čas strávený stavbou hierarchie by měl být menší než čas ušetřený jejím použitím



Obrázek 1.8: Ilustrace hierarchie obálek

Výsledná binární stromová datová struktura se skládá z vnitřních uzlů, které odkazují na levého a pravého potomka. V listech této struktury jsou obsaženy samotné trojúhelníky.

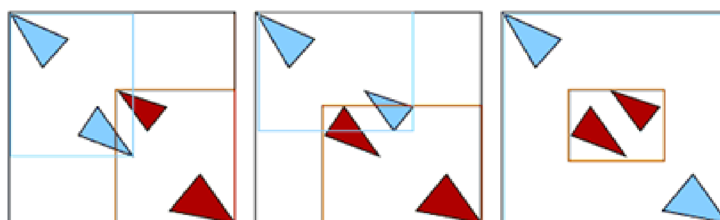
1.3.2.1 Stavba hierarchie obálek

Stavba stromové datové struktury nad scénou může probíhat odspoda nahoru, odshora dolů nebo postupným vkládáním. Při stavbě odshora dolů se v prvním kroku vytvoří kořen stromu, vezmou se všechny objekty ve scéně, vytvoří se nad nimi obalové těleso a vybraným způsobem se rozdělí scéna na dvě části, ze kterých se vytvoří potomci rodiče. Nad oběma potomky se poté rekurzivně spustí naprosto stejný postup. V nejjednodušším případě se rekurze zastaví, když uzel obsahuje méně objektů, než je námi určený minimální počet objektů v listu.

Při stavbě odspoda nahoru se začíná od listů. Již jsme tedy museli rozhodnout, jak budou objekty v listech uloženy a postupně z nich jejich slučováním stavíme celý strom. Tato metoda poskytuje kvalitněji postavenou hierarchii, ale je implementačně více náročná.

Pokud postupujeme postupným vkládáním objektů, nemusíme znát všechny objekty předem. S každým novým objektem vybereme nejlepší místo, kam ho vložit s tím, že se snažíme minimalizovat hloubku stromu.

V předchozím odstavci jsem přeskočil popis toho, jak se objekty ve vnitřním uzlu rozdělí do jeho potomků. Na obrázku č. 1.9 je vidět, že již čtyři trojúhelníky je možné rozdělit několika způsoby. Nejjednoduššími metodami na takové dělení je dělení pomocí prostorového mediánu a mediánu těles.



Obrázek 1.9: Možnosti dělení těles při stavbě hierarchie obálek [13]

Při dělení prostorovým mediánem se vybere jedna osa, ve které se trojúhelníky podle jejich středů rozdělí na dvě poloviny. Osy můžeme pravidelně střídat, nebo vzít tu, kde je největší rozdíl souřadnic. Při dělení mediánem objektů postupujeme obdobně, ale trojúhelníky dělíme podle mediánu objektů v některé z os. Tyto dva způsoby jsou jednoduché na implementaci, ale neposkytují tak kvalitní dělení, jako metoda nazývaná SAH (Surface Area Heuristic).

SAH zavádí cenovou funkci, která každému dělení přiřadí cenu. Vzorec pro výpočet ceny každého rozdělení je následující:

$$C = A_{left} * (\frac{N_{left}}{N}) + A_{right} * (\frac{N_{right}}{N})$$

kde A_{side} je plocha levé/pravé obálky, která vznikne rozdělením, N_{side} je počet trojúhelníků v potenciálně nově vzniklém podstromu a N je celkový počet trojúhelníků, které se právě dělí.

Bylo by velice výpočetně náročné otestovat všechna možná rozdělení trojúhelníků, takže nejčastěji používaný postup je vybrat osu s největším rozdílem souřadnic a na té vybrat pevně daný počet poloh dělící roviny. Stačí si tedy pouze pamatovat polohu a cenu nejvýhodnějšího dělení.

SAH nejenže dokáže stanovit cenu každého dělení, ale dokáže také určit cenu (kvalitu) konečného stromu podle vzorce:

$$C = \frac{1}{S_{root}} * ((\sum_{i=1}^{l} S_{leaf} * c_{intersection} * count_{leaf}) + (\sum_{j=1}^{n} S_{node} * c_{traverse}))$$

kde S_{root} je plocha kořene stromu, l je počet listů stromu, n je počet vnitřních uzlů a konstanty $c_{intersection}$ a $c_{traverse}$ představují cenu (náročnost) výpočtu průsečíku paprsku s trojúhelníkem, respektive paprsku s obalovým tělesem. Výpočet lze provést traverzací celého stromu po dokončení jeho stavby.

Tento výpočet dává jasné měřítko kvality postavené struktury. Dle článku [12] poskytuje hierarchie postavená pomocí SAH až 2x rychlejší traverzaci než hierarchie postavená pomocí dělení prostorovým mediánem a až 6x rychlejší než hierarchie postavená pomocí dělení mediánem objektů. V mé implementaci jsem použil oba výše zmíněné vzorce, takže v sekci výsledky je možné najít ceny mnou postavených hierarchií.

I strukturu postavenou pomocí SAH je ještě možné dále optimalizovat. Jedna z navrhovaných metod optimalizace [2] spočívá v dodatečné úpravě již postavené hierarchie obálek. Tento algoritmus obdrží na vstupu již postavenou hierarchii, postupných průchodem vybere vnitřní uzly, které jsou vhodné k optimalizaci, najde pro ně vhodnější pozici v hierarchii a vloží je na ni. Nalezení vnitřního uzlu vhodného k optimalizaci probíhá pomocí několika měřítek, které každému vnitřnímu uzlu přiřadí hodnotu, která říká, kolik volného prostoru (bez objektů) je v tomto uzlu. Druhé měření určí, zda potomci vnitřního uzlu nemají výrazně rozdílné velikosti. V článku je uveden příklad, kdy jeden potom obsahuje celý terén a druhý potomek pouze objekt nacházející se na něm. K těmto dvěma měřením je ještě připočtena plocha samotného uzlu, aby větší uzly byly optimalizovány dříve. Na základě těchto tří vlastností se poté uzly uspořádají do prioritní fronty a spustí se proces optimalizace. Zpravidla se neoptimalizují všechny uzly, ale pouze určité procento z nich. Proces optimalizace je možné zastavit po určitém čase, po zpracování určitého množství uzlů, nebo po klesnutí ceny stromu pod požadovanou úroveň. Dle testování provedeného v článku popsaná optimalizace sníží cenu hierarchie postavené pomocí SAH o 8 - 20%. Tato optimalizace ale není předmětem mé práce, ve které se soustředím na jiné oblasti metod sledování paprsků.

1.3.2.2 Traverzace hierarchie obálek

Nyní již víme, jak se hierarchie obálek staví, ještě je potřeba zmínit, jak se taková hierarchie traverzuje. Jako první představím algoritmus pro traverzaci jednotlivých paprsků, který následně doplním o algoritmy použitelné pro traverzování celých svazků paprsků. Každý paprsek nejdřív otestujeme s kořenovým uzlem stromu. Pokud paprsek nezasáhne obalové těleso celé scény, víme, že nemusíme traverzovat dále, protože paprsek nezasáhne nic. To je velká úspora, protože místo testu s n objekty jsme provedli test pouze s jedním objektem a to obalovým tělesem. Pokud paprsek obalové těleso zasáhne, projdeme postupně jeho dva potomky. V momentě, kdy se dostaneme do listu, musíme provést test průsečíku paprsku se všemi tělesy, které se v listu nacházejí. V tomto místě se dostává v dilematu, jaký je ideální počet objektů v listech. Čím více necháme objektů v listech, tím bude častěji docházet k testu paprsek/objekt, ale strom nebude tak hluboký. Na druhou stranu, čím méně

objektů bude v listech, tím bude strom hlubší, jeho stavba zabere více času, ale v listech již nebude docházet k tolika incidenčním operacím. Namísto toho bude docházet k většímu počtu testů paprsek/obalové těleso. Má implementace využívá zásobník, na který odkládá uzly připravené k traverzaci. V prvním kroku se na zásobník uloží kořen stromu. Pokud se jedná o vnitřní uzel, otestuji oba jeho potomky na průsečík s aktuálním paprskem a přidám na zásobník uzly, které paprsek protíná. Pokud se jedná o list, hledám nejbližší průsečík aktuálního paprsku s objekty uloženými v listu.

V tomto odstavci bude popsána pouze traverzace svazků paprsků s tím, že rozsáhlejší popis jejich tvorby, použití a přínosů, se nachází níže v kapitole věnované SIMD instrukcím. Na tomto místě bude dostačující, pokud zmíním, že paprsky nemusíme traverzovat pouze jednotlivě, ale můžeme z několika paprsků vytvořit svazek (v anglicky psané literatuře se většinou používá slovo packet), který traverzuje hierarchii najednou. Důvod k takové úpravě je zřejmý - primární paprsky jsou vysoce koherentní, takže sousední paprsky v hierarchii traverzují velice podobně. Tři nejpoužívanější algoritmy, které zmiňuje doporučená literatura se nazývají Masked Traversal, Ranged Traversal a Partition Traversal [9].

Algoritmus Masked traversal používá pole logických hodnot, kterými maskuje neaktivní paprsky. Jedná se o vůbec první a nejjednodušíí algoritmus pro traverzování svazků paprsků. V každém interním uzlu jsou otestovány všechny paprsky na průsečík paprsku s obalovým tělesem a pokud alespoň jeden z nich toto obalové těleso zasáhne, všechny paprsky sestupují. Neaktivní paprsky jsou při traverzaci postupně maskovány. V porovnání s Ranged Traversal a Partition Traversal je tento algoritmus výkonově srovnatelný pouze do velikosti svazků 4x4 a pouze pro scény s nepříliš komplikovanou geometrií (do 20 tisíc trojúhelníků).

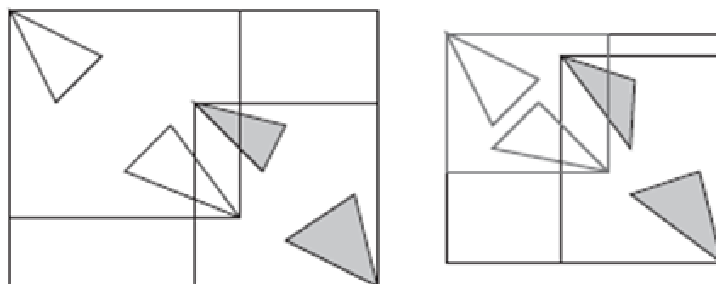
Algoritmus Ranged Traversal poskytuje ve většině testovaných případů nejlepší výsledky pro svazky do velikosti 16x16 paprsků. Ranged Traversal si ukládá hodnotu prvního aktivního paprsku ve svazku, díky čemuž se vyhne spoustě testů na průsečík paprsku s obalovým tělesem, ke kterým docházelo při použití Masked Traversal. Při průchodu vnitřními uzly máme uložen index prvního aktivního paprsku a v momentě, kdy se dostaneme až do listu hierarchie, zjistíme index posledního paprsku, který zasahuje obalové těleso tohoto listu. V listu testujeme na průsečík paprsku s trojúhelníkem pouze paprsky mezi první a posledním aktivním paprskem.

Algoritmus Partition Traversal je nejrobustnější algoritmus ze všech tří zmíněných. Jeho hlavní výhody se projeví v traverzování velkých (32x32) a nekoherentního svazku paprsků, například stínových nebo sekundárních paprsků. Algoritmus si spolu s polem paprsků udržuje ještě pole indexů. Místo sledování indexu prvního aktivního indexu, Partition Traversal sleduje index prvního neaktivního paprsku. Pole indexů je rozděleno na dvě části, které rozděluje právě index prvního neaktivního paprsku. Všechny paprsky před tímto indexem jsou aktivní, všechny za ním jsou neaktivní.

1.4 Sledování paprsků a dynamické scény

Dlouhou dobu byly metody sledování paprsků považovány za použitelné pouze pro statické scény. Tento názor se však poslední dobou mění, hlavně kvůli stále vzrůstajícímu výkonu výpočetních jednotek.

Dnešní aplikace používající metody sledování paprsku k zobrazení dynamických či animovaných scén musejí být velice optimalizované, aby bylo možné dosáhnout velkého počtu vykreslených snímků za vteřinu u náročnějších scén. Jedním z triků, který se používá v aplikacích, kde může uživatel přímo v reálném čase měnit pozici kamery, je postupné zlepšování obrazu. Po přemístění kamery do nového místa obsahuje obraz šum, ale jak je postupně vrháno více a více paprsků, dochází k zlepšení kvality výsledného obrazu.



Obrázek 1.10: Úprava dvou obálek po změně souřadnic těles [13]

Vykreslování animovaných scén, u kterých předem známe polohy jednotlivých objektů, se dá provést ještě rychleji. Nejjednodušším řešením by bylo postavit pokaždé novou akcelerační strukturu pro každý vykreslovaný snímek. Ani v dnešní době však není možné postavit hierarchii obálek pro každý snímek zvlášť a udržet si vysoký počet vykreslených snímků za vteřinu. Další možností, která nás jistě napadne je před vytvoření a uložení hierarchií pro jednotlivé snímky a poté je jenom načítat. To se již ovšem nedá považovat za opravdu interaktivní přístup. Další možností je projít všechny snímky a poté vybrat tu hierarchii, která je pro všechny snímky nejlepší.

Metoda, která má nejlepší poměr náročnosti implementace a výsledného rychlosti vykreslování je metoda postupné úpravy hierarchie z předešlého snímku. V této metodě neměníme strukturu původního stromu, ale pouze zvětšujeme nebo zmenšujeme obalová tělesa. Při přestavbě stromu tedy pouze procházíme listy, porovnáme polohu objektů a předchozím a novém snímku a odpovídajícím způsobem zmenšíme nebo zvětšíme obalová tělesa. Takto postupujeme od listů až ke kořeni. Na obrázku č. 1.10 je vidět úprava dvou obalových těles po přesunutí trojúhelníků.

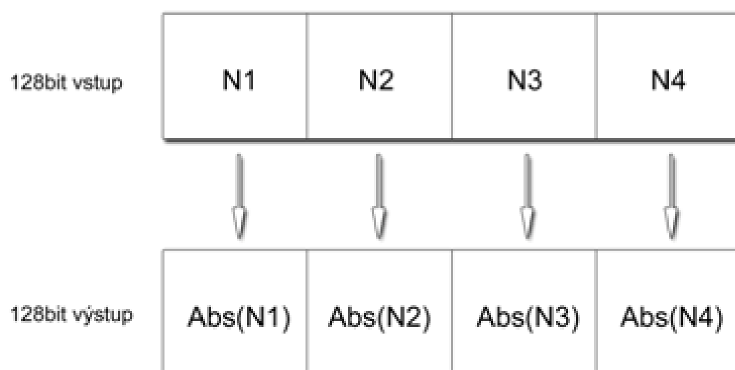
Přestavba hierarchie obálek pomocí této metody je implementačně náročnější a dostačující pro většinu animovaných scén. Problémem mohou být scény, ve kterých se například vyskytuje postava, která má v úvodním snímku ruce za hlavou, takže se pravděpodobně nacházejí ve stejném obalovém tělese jako hlava. Kdyby se na dalších snímcích ruce od hlavy oddalovaly, byly by stále s hlavou v jednom obalovém tělese, i když by bylo lepší je již oddělit. Jedním ze způsobů jak toto vyřešit je sledovat cenu každé nově postavené hierarchie a pokud přesáhne určitou mez, můžeme hierarchii kompletně přestavět od začátku.

Kapitola 2

SIMD architektura

Všechny výpočty v mé knihovně probíhají na procesu, takže je potřeba využít co nejvíce možností, které moderní procesory poskytují pro urychlení výpočtu operací s čísly s plovoucí desetinou čárkou. Za účelem zvýšení výkonu při počítání s čísly s plovoucí desetinou čárkou jsou moderní procesy vybaveny tzv. single instruction multiple data (dále jen SIMD) instrukcemi, které umožňují provést jednu instrukci nad celou množinou dat. Naproti tomu, při tradičním zpracování (SISD – single instruction single data) dat dochází k volání jedné instrukce nad každým skalárem. Jednotky pracující se SIMD instrukcemi se nacházejí fyzicky v jádře procesoru, je možné provádět tolik SIMD instrukcí paralelně, kolik máme k dispozici fyzických jader.

Na následujícím příkladě si můžeme ukázat hlavní výhodu SIMD oproti standardnímu zpracování dat. Pokud chceme spočítat absolutní hodnotu čtyř čísel uložených v poli, můžeme na to použít jednu SIMD instrukci, namísto používání jedné instrukce na každý prvek pole.



Obrázek 2.1: Výpočet absolutní hodnoty

Další rozdíl mezi SISD a SIMD zpracováním dat je v tom, že SISD využívá paralelismus na úrovni instrukcí, zatímco SIMD využívá paralelismus na úrovni dat. Instrukční paralelismus znamená, že je možné využít více instrukcí na jeden proud dat. Datový paralelismus znamená, že je možné využít jednu instrukci na více dat zároveň.

Data: Pole velikosti n
Result: Pole s vypočítanými absolutními hodnotami každého prvku

1 **for** $i \leftarrow 0$ **to** $(n-1)$ **do**
2 načti i do registru
vypočítej absolutní hodnotu
zapiš výsledek
3 **end**

Algorithm 1: Klasický přístup počítání absolutní hodnoty prvků v poli

Data: Pole velikosti n
Result: Pole s vypočítanými absolutními hodnotami každého prvku

1 **for** $i \leftarrow 0$ **to** $(n-4)$ **do**
2 načti 4 prvky do SSE registru
vypočti 4 absolutní hodnoty v jedné operaci
zapiš výsledek
3 **end**

Algorithm 2: SSE přístup počítání absolutní hodnoty prvků v poli

SIMD instrukce je v dnešní době možné nalézt ve větší či menší míře na většině vyráběných procesů, ať už jsou to procesory společnosti Intel, která svoji instrukční sadu nazývá SSE (Streaming SIMD Extensions), nebo procesory společnosti AMD, kde se tato technologie nazývá 3DNow! V dnešní době již ovšem i procesory společnosti AMD podporují instrukční sadu SSE. K těmto operacím jsou na CPU k dispozici zvláštní registry o velikosti 128 bitů až do 512 bitů, do kterých je tedy možné nahrát 4-16 čísel s plovoucí desetinou čárkou.

2.1 SSE

SSE v první verzi nabízelo 8 registrů o velikosti 128 bitů, které měly označení XMM0 až XMM7. V první verzi tyto registry mohly obsahovat čtyři 32 bitová čísla s plovoucí desetinnou čárkou (single precision). Ve verzi SSE2 již bylo možné do registrů ukládat navíc následující datové typy:

- čtyři 32 bitová čísla s plovoucí desetinnou čárkou (single-precision)
- dvě 64 bitová čísla s plovoucí desetinnou čárkou (double-precision)
- dvě 64 bitová celá čísla
- čtyři 32 bitová celá čísla
- osm 16 bitových krátkých celých čísel
- šestnáct 8bitových bytů nebo znaků

Dalším velké vylepšení přichází v SSE4, které přidává podporu operace skalárního součinu, která je velice používaná právě v počítačové grafice.

2.2 AVX

Instrukční sada Advanced Vector Extensions (AVX) se dá označit za přímého nástupce SSE. První procesory využívající tuto novou instrukční sadu se dostaly na trh v prvním čtvrtletí roku 2011 a byly to procesory s kódovým označením Sandy Bridge u společnosti Intel a Bulldozer u společnosti AMD. Mezi hlavní novinky patřilo zvětšení registrů ze 128 na 256 bitů a zavedení nedestruktivních operací. Pokud v SSE nějaká funkce vyžadovala dva operandy, byla provedena jako A = A+B, takže zdrojový operand A byl nahrazen výsledkem. AVX zavádí nedestruktivní operandy, které pracují jako C = A+B. AVX-512 poté přichází s dalším rozšířením registrů až na 512 bitů. Pokud za běhu programu dochází k přepínání mezi SSE a AVX instrukcemi, tato činnost má dopad na celkový výkon aplikace, což ovšem neplatí pro přepínání mezi AVX a AVX-512. Je důležité zmínit, že AVX instrukce není možné používat na starších operačních systémech jako například Windows XP nebo Windows Vista, i když je samotný procesor podporuje. V operačních systémech postavených na linuxovém jádře, je možné využívat AVX od verze 2.6.3.

2.3 Organizace dat

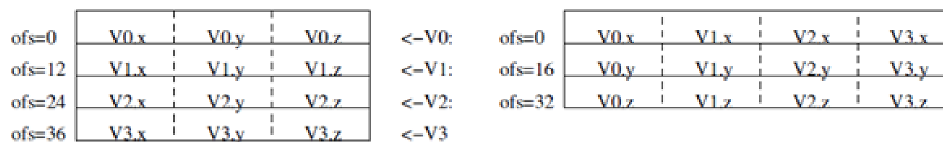
V této kapitole budu uvažovat jednotlivé registry o velikosti 128 bitů, to znamená, že do jednoho registru je možné uložit 4 čísla s plovoucí desetinnou čárkou (single precision).

Jedním z důvodů, proč je čtyřnásobné zrychlení pouze teoretické je fakt, že data je potřeba organizovat trochu odlišně, než je obvyklé při standardním přístupu. Na obrázku č. 2.2 je vlevo uvedeno standardní uložení (angl. Array of Structures) čtyř vektorů se třemi souřadnicemi tak, jak by ho nejspíše zvolil každý, kdo nemá se SIMD zkušenosti. Na pravé straně je vidět uložení (angl. Structure of Arrays), které již plně vyhovuje požadavkům SIMD. Tato úprava většinou vyžaduje změnu přemýšlení nad tvorbou programu a manuální přepsání již fungujících skalárních algoritmů. Na příkladu zmíněném výše, kde chceme spočítat absolutní hodnotu čtyř prvků, by pravděpodobně úprava dat pro SIMD zpracování a poté jejich úprava zpět do zpracovatelné podoby zabrala více času, než kolik bychom ušetřili použitím SIMD.

Navíc ne všechny algoritmy jsou vhodné pro vektorizaci. Další věc, kterou je potřeba mít na paměti je fakt, že data musejí být nejen správně uložena, ale také správně zarovnána. Operace nad nezarovnanými daty může vést k poklesu výkonu nebo také čtení v nealokované paměti. Tento požadavek byl již v AVX odstraněn, ale práce s nezarovnanými daty vede ke snížení výkonu, takže i v AVX je doporučené používat zarovnání (na 16 bytů při použití 128 bitových registrů a na 32 bytů při použití 256 bitových registrů). Zarovnání provedeme v jazyce C++ použitím speciální předpony __declspec(align(32)) nebo __declspec(align(16)) pro zarovnání na 16 bytů.

2.4 Volání instrukcí

Kód můžeme vektorizovat třemi různými způsoby. Může nám ho sám zvektorizovat kompiler, můžeme psát přímo kód v assembleru nebo využít tzv. intrinsik (angl. intrinsic). Některé moderní kompilery jako například GNU gcc umožňují nastavit možnost automatické vektorizace. Experimenty však ukazují, že i když jsou data uložena v SIMD přívětivé podobě,



Obrázek 2.2: Organizace dat vhodná pro SIMD [12]

přínos je naprosto minimální [12]. Druhá možnost, psaní kódu v assembleru, je velice časově náročná.

Jako nejvýhodnější se mi jeví možnost používání intrinsik. Jedná se o funkce, které umožňují přístup k mnoha instrukčním sadám jako SSE, AVX a další, bez nutnosti psát kód v assembleru. Jejich volání připomíná standardní funkce v jazyce C++, do kterého je možné je libovolně vkládat. Při použití také není nutné se starat o to, do kterého fyzického registru budou data vložena. Na obrázku č. 2.3 je příklad použití dvou SSE4 intrinsik. Intrinsika _mm_set1_ps načte do 128 bitového registru proměnnou n_u a intrinsika _mm_mul_ps provede standardní násobení a uložení výsledku. Na dalším obrázku č. 2.4 je vidět použití dvou AVX intrinsik na načtení hodnot do 256 bitového registru a na násobení dvou registrů. AVX instristiky jsou velice podobné těm z sady SSE4, liší se pouze v malých detailech.

```
const __m128 n_u = _mm_set1_ps(this->nu);
const __m128 ounu = _mm_mul_ps(ray.origin.t[ku], n_u);
```

Obrázek 2.3: Ukázka SSE4 intrinsik

```
const __m256 n_d = _mm256_set1_ps(this->nd);
const __m256 ounu = _mm256_mul_ps(ray.origin.t[ku], n_u);
```

Obrázek 2.4: Ukázka AVX intrinsik

2.5 Využití v metodách sledování paprsků

Použití SIMD instrukcí je tedy přímo podmínkou k tomu, abychom naplno dokázali využít možnosti dnešních procesorů. Jak bude vidět v kapitole věnované výsledkům, nejvíce výpočetně náročnou operací při sledování paprsku je výpočet jeho průsečíku s jinými objekty ve scéně – typický trojúhelníky, koulemi a obalovými tělesy. Dle výše napsaného je možné počítat incidenční operace čtyř a více paprsků s jedním tělesem, nebo také jednoho paprsku se čtyřmi a více tělesy.

První přístup využívá koherence primární paprsků, které mají stejný počátek a velice podobný směr, takže velice pravděpodobně budou protínat i stejná tělesa ve scéně. To nám umožňuje shlukovat čtveřice paprsků dohromady a traverzovat s nimi scénu jako, kdyby se jednalo o jediný paprsek. Teoreticky je možné dosáhnout až čtyřnásobného zrychlení pro

traverzaci primárních paprsků, ale také až čtyřnásobné zrychlení pro stínové paprsky, které mají různý počátek, ale stejný směr. Sekundární paprsky již nemají stejný počátek, ani stejný směr, takže efektivita použití SIMD se u nich liší scénu od scény [9]. Použití SIMD instrukcí je vhodné hlavně v případě použití algoritmu zpětného sledování paprsku, kde jsou jak primární, tak stínové paprsky, velice koherentní. Při syntéze obrazu pomocí sledování cest by se SIMD instrukce daly použít pouze pro primární paprsky, protože u tohoto algoritmu dochází k velkému množství odrazů ve scéně, které rozbijí koherenci sousedních paprsků.

Druhý přístup (testování průsečíku jednoho paprsku se čtyřmi trojúhelníky najednou) již není tak častý. Efektivita tohoto přístupu se projeví v případě, že paprsek protíná velké množství trojúhelníků. Při vrhání paprsků se většinou používá akcelerační struktura, která usnadňuje traverzaci scénou. Pokud je touto strukturou hierarchie obálek, velice často se stává, že v listech se nachází méně trojúhelníků, než čtyři.

Teoreticky je tedy při použití SSE instrukcí se 128 bitovými registry možné zrychlit původní algoritmus až 4x, což se ale v praxi většinou neděje, právě kvůli potřebě upravit vstupní data a práci s nimi. Jak bude vidět později v sekci věnované výsledkům, zrychlení závisí také na struktuře scény.

Kapitola 3

Implementace

Při návrhu knihovny jsem bral v potaz hlavně možné způsoby použití takové knihovny. Prvním případem použití je vrhání jednotlivých paprsků do scény a vyhodnocení informace o zásahu. Občas chceme vrhnout pouze jednotlivé paprsky, protože vrhání ve svazcích a využití SIMD instrukcí se nám již nevyplatí. Druhým případem použití je vrhání více paprsků zároveň, kdy si uživatel například vytvoří všechny primární paprsky, které nechá vrhnout do scény a poté vyhodnotí zásahy celé skupiny najednou. Jelikož má knihovna umožňovat použití SIMD instrukcí, bylo by dobré, kdyby si uživatel mohl vybrat, zda chce vrhat paprsky standardně, nebo ve skupině za využití SIMD instrukcí. Většina moderních CPU již nějaký typ SIMD instrukcí podporuje, ale pořád je možné nalézt scény, ve kterých je vrhání jednotlivých paprsků rychlejší, než vrhání paketů paprsků (viz dále v kapitole věnované testování). Jako třetí použití vidím ve využití dvou výše zmíněných případů a použití podpory více jader moderních procesorů.

3.1 Existující implementace

Samozřejmě má knihovna nebude první svého druhu. Na internetu je možné nalézt velké množství projektů knihoven na vrhání paprsků, které se zaměřují na různé oblasti v syntéze obrazu pomocí metod sledování paprsku. Spousta projektů již nepokračuje ve vývoji, ale rád bych zde zmínil několik stále aktivních projektů, které jsem shledal zajímavými.

První knihovna je vyvíjena v samotných Intel Labs a jmenuje se Embree [15]. Jedná se o knihovnu napsanou v jazyce C++, maximálně využívající současné procesory společnosti Intel s instrukčními sadami SSE4 a AVX2. Je speciálně navržena pro Monte Carlo metody, u kterých je většina paprsků nekoherentních. Jako svoji akcelerační strukturu používá binární hierarchii obálek. Její rozhraní umožňuje vrhat také svazky paprsků o velikostech 1, 4, 8 a 16 paprsků. Licence umožňuje použití knihovny jako open source (Apache 2.0 licence), její zdrojové kódy i veřejné rozhraní jsou volně dostupné, stejně jako sada výukových materiálů dostupných přímo na domovských stránkách knihovny.

Jako druhý projekt bych rád zmínil projekt s názvem Mitsuba physically based renderer [19], který, dle mého názoru, nemá na poli otevřených knihoven momentálně konkurenci. Jedná se o knihovnu vyvíjenou post graduálním studentem Curyšské univerzity Jakobem Wenzelem. Mezi její hlavní schopnosti patří vysoká optimalizace pro moderní procesory,

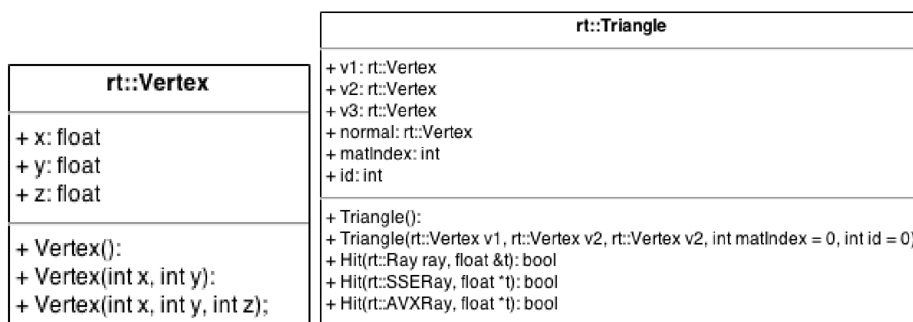
podpora jak lokálních, tak globálních osvětlovacích technik, implementace radiozity, fotonových map, zastínění prostředím, vykreslování kouře a jiných čistě objemových dat a mnoho dalšího. Na domovské stránce celého projektu je možné si stáhnout dokumentaci, zdrojové kódy celé knihovny a okolo stovky zásuvných modulů, které implementují jednotlivé funkcionality. Knihovna je velmi robustní, takže je potřeba před používáním strávit nějaký čas čtením téměř tří set stránkové dokumentace. Knihovnu je možné používat pod operačními systémy Microsoft Windows, Mac OS X, Linux (distribuce Ubuntu, Debian, Fedora, Arch Linux). Mitsuba renderer je stále ve vývoji. Nové verze se objevují každé 3-4 měsíce.

3.2 Návrh rozhraní knihovny

Rozhraní knihovny (dále jen API) by mělo podporovat všechny tři operace zmíněné v úvodu této kapitoly a umožňovat vrhání paprsků co nejjednodušším způsobem. Uživateli tedy k úspěšnému vrhnutí a vyhodnocení paprsku stačí pět tříd, které představují celé rozhraní. Všechny datové objekty knihovny se nacházejí ve jmenném prostoru *rt*.

Jakýkoliv až třírozměrný bod nebo vektor je v knihovně reprezentován třídou Vertex (diagram na obrázku č. 3.1). Knihovna poskytuje mnoho funkcí, které je možno nad objekty třídy Vertex volat. Například vektorový součin, skalární součin, normalizaci, zjištění délky vektoru a další.

Aktuálně jsou všechny objekty v knihovně definovány jako trojúhelníky (diagram na obrázku č. 3.1). Trojúhelník je dán svými třemi vrcholy. Dále si v každém objektu třídy Triangle uchováváme i předpočítanou normálu. U trojúhelníku si potřebujeme také pamatovat jeho identifikátor, který se používá při animovaných scénách, abychom byli schopni rozlišit pohyby jednotlivých trojúhelníků mezi snímky. Poslední veřejnou proměnnou je index do knihovny s materiály. Knihovna na vrhání paprsků má být co nejjednodušší, takže v sobě neudržuje žádnou informaci o materiálu tělesa, pouze index do knihovny materiálů. Třída Triangle obsahuje ještě sedm předpočítaných proměnných, které se používají pro urychlení výpočtu průsečíku trojúhelníku s balíkem paprsků pomocí Waldova algoritmu.

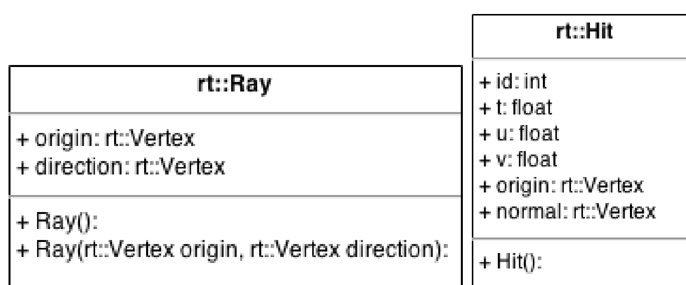


Obrázek 3.1: Datové struktury Vertex a Triangle

Pro vrhání paprsků je samozřejmě potřebná samotná třída reprezentující paprsek (diagram na obrázku č. 3.3). Paprsek je dán bodem určujícím jeho počátek, kdy pro primární paprsky je tímto bodem střed kamery, a směrem, který je pro primární paprsky bod na

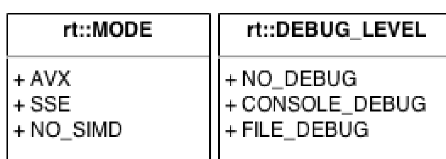
promítací rovině představující jeden pixel na výsledném obrázku. Rozhodl jsem se neposkytovat uživateli k dispozici datové struktury používané pro vrhání paprsků pomocí SIMD instrukcí, které jsou lehce odlišné od standardní struktury paprsku, protože data do těchto struktur převedu přímo v knihovně po zavolání příslušné metody. Uživatel je tak odstíněn od používání rozdílných struktur pro SIMD a standardní vrhání paprsků.

Dalším rozhodnutím bylo oddělit informaci o zásahu od třídy Ray a vytvořit pro ni samostatnou třídu Hit (diagram na obrázku č. 3.3). V jiných implementacích je možné vidět tyto dvě třídy sloučené do jedné. Třída Hit obsahuje informaci o bodě zásahu, směru normály v tomto bodě a čísle objektu, který byl paprskem zasažen. Při proceduře vrhání paprsků by mělo platit, že k jednomu paprsku se váže jeden objekt třídy Hit. V objektu jsou poté uloženy ještě barycentrické souřadnice použitelné při mapování textury na objekt. Stejně jako s objekty třídy Ray je nutné i u třídy Hit používat lehce odlišné struktury reprezentující zásah u vrhání paprsků s využitím SIMD instrukcí. I v tomto případě je uživatel od tohoto problému odstíněn.



Obrázek 3.2: Datové struktury Ray a Hit

Tím máme definovány základní entity, nad kterými už můžeme začít vrhat paprsky. K této proceduře slouží třída Raycaster. Celkový třídní diagram třídy Raycaster je vidět na obrázku č. 3.4. Tato třída poskytuje metody na vrhání jedno paprsku a skupiny paprsků. Před samotným vrháním paprsků je ovšem potřeba knihovnu nainicializovat. K inicializaci knihovny, ale také k nastavení způsobu vrhání paprsků, se používají výčtové typy DEBUG_LEVEL a MODE. Výčtový typ DEBUG_LEVEL slouží k nastavení způsobů informování uživatele o charakteristice postavené datové struktury (hloubka, počet listů, délka stavby, průměrný počet traverzací na paprsek) a času potřebnému k vrhnutí paprsků. Uživatel si může vybrat, zda nechce být informován vůbec, nebo chce zobrazovat informace do konzole, případně do souboru BVH.csv. Knihovna ukládá informace do CSV souborů ve formě vhodné k dalšímu zpracování.



Obrázek 3.3: Výčtové typy

Při zavolání metody Initialize proběhne stavba hierarchie obálek nad scénou předanou v parametru této metody. Při vykreslování dynamických scén je možné zavolat metodu Initialize pro každý snímek, čímž dojde k přestavbě hierarchie z předchozího snímku (volání). Knihovna sama pozná, že se jedná o další snímek a má hierarchii pouze aktualizovat. Mezi jednotlivými snímky není možné aktualizovat polohu pouze jednotlivých trojúhelníků, vždy je potřeba předat jako parametr celou scénu.

```
rt::Raycaster
+ RayCaster() :
+ Initialize(rt::DEBUG_LEVEL debug, std::vector<rt::Triangle*> triangles, int trianglesInLeaf = 10) : void
+ CastRay(rt::Ray &ray, rt::Hit &hit) : bool
+ CastRays(rt::MODE mode, rt::PACKET_SIZE size, std::vector<rt::Ray> &rays, std::vector<rt::Hit> &hits, int threads = 1) : bool
```

Obrázek 3.4: Datová struktura Raycaster

V závislosti na nastavení parametru MODE se určí způsob vrhání paprsků. K vrhání paprsků pomocí SSE4 a AVX slouží třída SSERaycaster, respektive AVXRaycaster. Tyto třídy se liší v použitých datových strukturách, kdy pro SSE instrukce jsou všechna data organizována po čtveřicích, zatímco pro AVX instrukce jsou všechna data po osmicích. Jinak obě tyto třídy používají stejné algoritmy pro výpočet průsečíku paprsku s trojúhelníkem a obalovým tělesem. V článku [12], který jsem dostal k prostudování v souvislosti se zadáním diplomové práce, je jako vhodný algoritmus pro SIMD implementaci testování průsečíku paprsku s trojúhelníkem, navrhován Waldův algoritmus [12]. Alternativou k tomuto algoritmu by mohl být Havel/Heroutův algoritmus [5], který podle autorů umožňuje dokonce rychlejší výpočet průsečíku než Waldův, ale potřebuje mít u každého trojúhelníku předpočítány o tři neceločíselné proměnné více. Jako vhodný algoritmu pro testování průsečíku paprsku s kvádrem je uváděn algoritmus Kay-Kajiya [6]. Další algoritmus, který jsem se rozhodl otestovat, je Möllerův algoritmus [8], který patří k nejrychlejším algoritmům pro výpočet průsečíku paprsku s trojúhelníkem bez dalších předpočítaných hodnot pro každý trojúhelník. Všechny čtyři zmíněné algoritmy jsou detailně popsány v následující podkapitole a naměřené výsledky jejich rychlosti jsou diskutovány v kapitole věnované testování.

Uživatel se nemusí starat o to, zda data správně zarovná do struktur požadovaných pro SIMD zpracování. Pokud uživatel nastaví vrhání paprsků pomocí SIMD, knihovna vezme všechny paprsky ze vstupního vektoru a rozdělí je do čtveřic nebo osmic. Pokud vstupní počet paprsků není dělitelný čtyřmi, přebývající paprsky se vrhnou samostatně. Kromě paprsků je potřeba předělat ještě strukturu reprezentující zásah paprsku. Po vrhnutí paprsků je musí knihovna opět vrátit do původních struktur. Právě tyto dvě operace převedení do nových struktur a vrácení zpět, způsobují to, že SSE4 zpracování nebude 4x rychlejší, ale zrychlení bude o něco menší. Při využití více vláknového zpracování se inicializuje počet jader, které uživatel chce využít a na každé jádro se pošle ke zpracování poměrná část z vstupního balíku paprsků.

Obě tyto třídy využívají pro urychlení vrhání paprsků hierarchii obálek. Ta je postavena rekurzivně metodou SAH. Pro svojí diplomovou práci jsem jako základ použil implementaci z aplikace Minimax [18], kterou jsem dále rozšířil podle potřeby. Tato implementace využívá setříděný vektor trojúhelníků, kde nám v každém listu stačí znát pouze index prvního a

poslední prvku v tomto vektoru a všechno ostatní, co se nachází mezi těmito indexy, patří do právě procházeného listu. Každý uzel stromu obsahuje obalový obdélník (zadaný pomocí minimálních a maximálních souřadnic v každé ose) a informaci o ose, ve které proběhlo dělení. Vnitřní uzly navíc obsahují ukazatele na levý a pravý podstrom. Jak již bylo zmíněno výše, listy navíc obsahují ještě index prvního a posledního trojúhelníku. Při volání metody Initialize je možné v parametru nastavit, jaký je maximální počet trojúhelníků v listech.

3.3 Implementace incidenčních operací

Jedněmi z nejdůležitějších algoritmů v knihovně vrhající paprsky jsou incidenční algoritmy. Tyto algoritmy musejí pracovat naprosto spolehlivě a být schopny rychle rozhodnout o průsečíku paprsku s trojúhelníkem nebo obalovým tělesem a to nejen v počítačové grafice, ale také v mnoha dalších odvětvích jako například fyzikální simulace, výpočet kolizí objektů a dalších. V této podkapitole budou popsány čtyři algoritmy použité v mé diplomové práci.

3.3.1 Průsečík paprsku s trojúhelníkem

3.3.1.1 Möllerův algoritmus

Möllerův algoritmus [8] z roku 1997 popsaný Tomasem Möllerem and Benem Trumborem představuje jeden z nejrychlejších způsobů určení průsečíku paprsku s trojúhelníkem bez nutnosti předpočítání jakýchkoliv dat - vystačí pouze tři body na trojúhelníku. Dle samotných autorů dosahuje úspora v paměti pro trojúhelníkové sítě 25-50%. Bod na trojúhelníku $T(u, v)$ je dán jako:

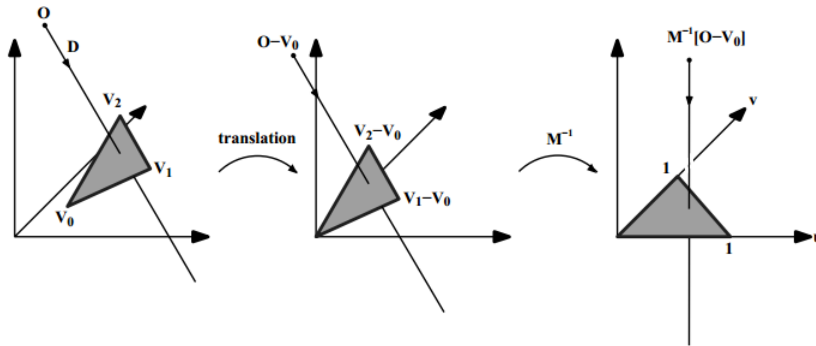
$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2,$$

kde u,v jsou barycentrické souřadnice, pro které musí platit $u \geq 0$, $v \geq 0$ a $u + v \leq 1$. Vypočítání průsečíku paprsku $R(t)$ s trojúhelníkem $T(u, v)$ je možné zapsat jako $R(t) = T(u, v)$, z čehož plyne:

$$\begin{gathered} O + t * D = (1 - u - v)V_0 + uV_1 + vV_2 \\ [-D, V_1 - V_0, V_2 - V0] * [t\ u\ v]' = O - V_0, \end{gathered}$$

což si můžeme představit jako posunutí trojúhelníku do počátku souřadné soustavy, znormalizování v osách y a z a zarovnání směru paprsku s osou x. Celý proces ilustruje obrázek č. 3.5 (matice $M = [-D, V_1 - V_0, V_2 - V_0]$ z předchozí rovnice). Vypočítané t, u a v musí splňovat podmínky $u \geq 0$, $v \geq 0$ a $u + v \leq 1$, aby paprsek protínal trojúhelník. Pokud ještě nahradíme $E1 = V_1 - V_0$,$E2 = V_2 - V_0$ a $T = O - V_0$ můžeme řešení vyjádřit jako

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{| - D, E_1, E_2|} \begin{bmatrix} |T, E_1, E_2| \\ | - D, T, E_2| \\ | - D, E_1, T| \end{bmatrix}$$



Obrázek 3.5: Posunutí trojúhelníku a zarovnání směru paprsku [8]

Z lineární algebry víme, že $|A, B, C| = -(A \times C) * B = -(C \times B) * A$. Finální úprava za použití pravidla z předchozí věty tedy vypadá

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2).E_1} \begin{bmatrix} (T \times E_1).E_2 \\ (D \times E_2).T \\ (T \times E_1).D \end{bmatrix}$$

Výsledky mé implementace Möllerova algoritmu [8] a porovnání rychlosti s ostatními algoritmy je možné najít v kapitole věnované testování.

3.3.1.2 Waldův algoritmus

Jelikož implementace Möllerova algoritmu v SIMD verzi nepatří mezi ty nejrychlejší [12], rozhodl jsem se pro výpočet průsečíku čtyř paprsků s jedním trojúhelníkem zvolit Waldův algoritmus [12], který je nejvíce doporučován pro SIMD zpracování. Waldův algoritmus vyžaduje mít pro každý trojúhelník předpočítány koeficienty roviny, ve které tento trojúhelník leží, což ho činí více paměťově náročným než Möllerův algoritmus. Také tento algoritmus používá barycentrické souřadnice. V prvním kroku Waldova algoritmu se spočítá vzdálenost papsku od roviny trojúhelníka. To lze provést pokud známe normálu ($N = (B-A) \times (C-A)$) a vrchol trojúhelníka A, $t_{plane} = -\frac{(O-A).N}{D.N}$. Tím zjistíme, zda paprsek není s rovinou rovnoběžný, zda se nachází ve menší vzdálenosti než dosud nejbližší nalezených průsečík, nebo jestli nezačíná až za touto rovinou. Pokud nastane nějaká ze zmíněných situací, nemusíme pokračovat dál. Nyní máme zaručeno, že paprsek protíná rovinu trojúhelníka, ale ještě nevím, zda se tento bod nachází v trojúhelníku. V tomto kroku Ingo Wald představuje novou variaci na standardní test barycentrických souřadnic. Jeho hlavní myšlenkou je projekce trojúhelníka na jednu z rovin X/Y nebo Z a provádění všech výpočtů ve 2D. Z důvodu výpočetní stability by měla být projekce provedena na tu rovinu, kde je rozdíl souřadnic trojúhelníka největší. Například při zobrazení na rovinu XY hledáme koeficienty α, β, γ v rovnici

$$H' = \alpha A' + \beta B' + \gamma C'$$

kde A', B', C' jsou projekce bodů trojúhelníka a H' je projekce průsečíku paprsku s rovinou trojúhelníka. Substitucí $\alpha = 1 - \beta - \gamma$ dostáváme

$$\beta(B' - A') + \gamma(C' - A') = H' - A'$$

což umíme vyřešit například za použití Hornerova schématu. Abychom mohli vyhodnotit zásah paprsku s trojúhelníkem jako kladný, musí pro výsledné koeficienty α, β, γ musí platit, že jsou v intervalu od nuly do jedné a jejich součet je roven jedné. Podrobný popis tohoto algoritmu s odvozením celého postupu je detailně popsán v jeho disertační práci [12].

Z profilingu SSE implementace Waldova algoritmu je patrné, že nejnáročnější na výpočet je operace dělení a určení, zda se zjištěný parametr t nachází blíže než aktuální nejbližší průsečík.

3.3.1.3 Havel/Heroutův algoritmus

Poslední algoritmus na hledání průsečíku paprsku s trojúhelníkem, který jsem v mé diplomové práci implementoval, pochází od Ing. Jana Havla a Doc. Ing. Adama Herouta, Ph.D. z Vysokého učení technického v Brně. Dle jejich článku [5] je jich algoritmus dokonce rychlejší než Waldův algoritmus, ze kterého vychází. U každého trojúhelníka je potřeba předpočítat jeho rovinu $(\vec{N}, d)$ a ostatní roviny $(\vec{N_1}, d_1)$,$(\vec{N_2}, d_2)$. Používá tedy o tři desetinná čísla více pro každý trojúhelník, než Wald. Vektor $\vec{N}$ je normála trojúhelníka a $d = -A.\vec{N}$ je koeficient z obecné rovnice roviny $xN_x + yN_y + zN_z + d = 0$. Vektory $\vec{N_1}, \vec{N_2}$ a koeficienty d_1, d_2 získáme analogicky.

$$\vec{N_1} = \frac{\vec{AC} \times \vec{N}}{|\vec{N}|^2}, d_1 = -\vec{N_1}.A,$$

$$\vec{N_2} = \frac{\vec{N} \times \vec{AB}}{|\vec{N}|^2}, d_2 = -\vec{N_2}.A.$$

Barycentrické souřadnice průsečíku jsou

$$P = O + t\vec{D},$$

$$u = \vec{N_1}.P + d_1,$$

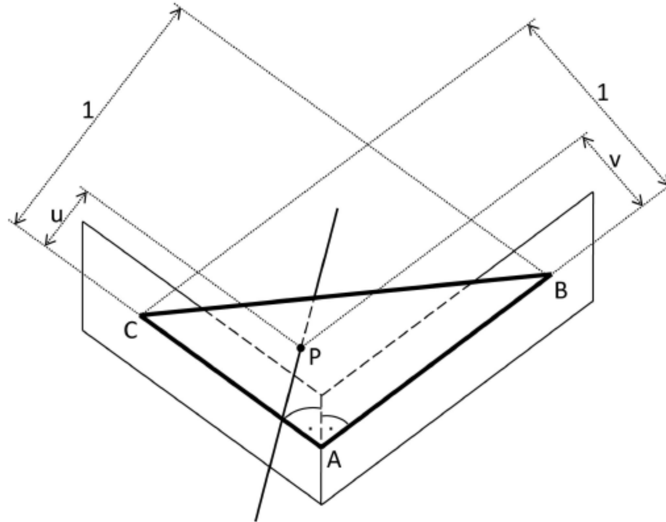
$$v = \vec{N_2}.P + d_2.$$

Samotný výpočet průsečíku společně se získáním barycentrických souřadnic probíhá následovně

$$det = \vec{D}.\vec{N},$$

$$t' = d - (O.\vec{N}),$$

$$P' = det.O + t'.\vec{D},$$



Obrázek 3.6: Roviny použité pro výpočet průsečíku paprsku s trojúhelníkem [5]

$$u' = P'.\vec{N_1} + det.d_1,$$
$$v' = P'.\vec{N_2} + det.d_2,$$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{det} \begin{bmatrix} t' \\ u' \\ v' \end{bmatrix}$$

Abychom mohli prohlásit, že paprsek protíná trojúhelník, je ještě potřeba ověřit následující trojici podmínek

$$sign(t') = sign(det.t_{max} - t'),$$
$$sign(u') = sign(det - u'),$$
$$sign(v') = sign(det - u' - v').$$

Všechny tři výše uvedené algoritmy jsou v mé diplomové práci implementovány a porovnání jejich rychlosti se nachází v kapitole testování. Implementace Havel/Heroutova algoritmu ve standardní i SSE verzi se nachází v příloze A diplomové práce. Po vyhodnocení výsledků implementace těchto tří algoritmů jsem se rozhodl, že pro test průsečíku paprsku s trojúhelníkem použiji Möllerův algoritmus, který nevyužívá žádné předpočítané hodnoty, což ho činí méně paměťově náročným oproti ostatním algoritmům. Z toho samého důvodu jsem pro SIMD implementaci zvolil Waldův algoritmus, který potřebuje o tři neceločíselné předpočítané hodnoty pro každý trojúhelník méně, než Havel/Heroutův algoritmus a dosahuje podobných výsledků. Vyšší paměťové nároky Havel/Heroutova algoritmu by se mohly negativně projevit při vykreslování velkých scén čítajících několik milionů trojúhelníků.

3.3.2 Průsečík paprsku s obalovým tělesem

V mé diplomové práci používám jako obalová tělesa osově zarovnané kvádry. Výpočet průsečíků takového obalového tělesa s paprskem je poměrně jednoduchý. Na druhou stranu se nejedná o nejtěsnější obalová tělesa, ale v případě obalových těles si vždy musíme vybírat mezi výpočetní náročností a těsností obálky. K výpočtu průsečíku obálky s paprskem musíme vědět, že parametrická rovnice přímky má tvar:

$$P = A + t * u;\ t \epsilon R,\ u \neq o,$$

kde A je bod přímky, u je směrový vektor a proměnná t se nazývá parametr. Paprsek můžeme popsat jako:

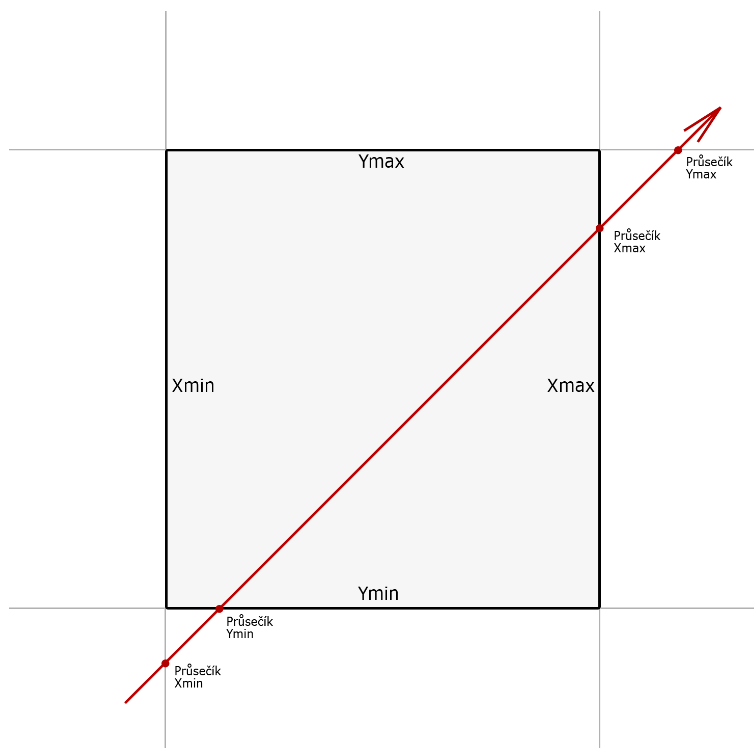
$$R(t) = O + t * D,$$

kde O představuje počátek paprsku, D jeho směr a t je parametr udávající polohu na paprsku. Osově zarovnaný kvádr můžeme uložit pomocí dvou tří prvkových polí, které obsahují maximální a minimální souřadnice v každé ose. Tyto krajní souřadnice definují množinu čar, z nichž některé jsou na sebe kolmé a jiné rovnoběžné. Abychom získali bod, ve kterém paprsek protíná jednu z hraničních přímek (například minimální souřadnici v ose x), budeme vycházet z rovnice:

$$x_{min} = (Box_{minX} - O_x)/D_x$$

Obdobně vypočteme všechny body, kde paprsek protíná hraniční přímky obalového kvádru (obrázek č. 3.7). Ještě ovšem nevíme, zda dané body leží na povrchu kvádru, nebo ne. Toto vede na několik podmínek, které musíme v každé ose otestovat.

Z profilingu SSE4 implementace výše zmíněného algoritmu na výpočet průsečíku paketu paprsků (velikosti 2x2) s obálkou vyplynulo, že jednou z nejnáročnějších operací v této implementaci je překopírování hodnot z SSE4 registru do dynamického pole (intrinsika __mm_store_ps), se kterým se pracuje dále.



Obrázek 3.7: Znázornění bodů, kde paprsek protíná hraniční roviny obálky

3.4 Testovací aplikace

Pro demonstraci použití knihovny byla vytvořena aplikace, která umožňuje syntézu obrazu jak statických, tak dynamických scén za použití algoritmů sledování paprsku a sledování cesty. Jedná se o konzolovou aplikaci napsanou v jazyce C++, konfigurovatelnou pomocí konfiguračního souboru. Jejím výstupem jsou obrázky ve formátu TGA, pro dynamické scény je to sekvence obrázků.

3.4.1 Konfigurační soubor

Parametry, které je možné nakonfigurovat, jsou následující:

- width – šířka výsledného obrázku
- height - výška výsledného obrázku
- samples – počet vzorků na jeden pixel (využití při algoritmu sledování cesty)
- threads – počet jader, které může aplikace využít při vrhání paprsků
- bucketSize - maximální počet trojúhelníků, které je možné uložit v listu hierarchie obálek

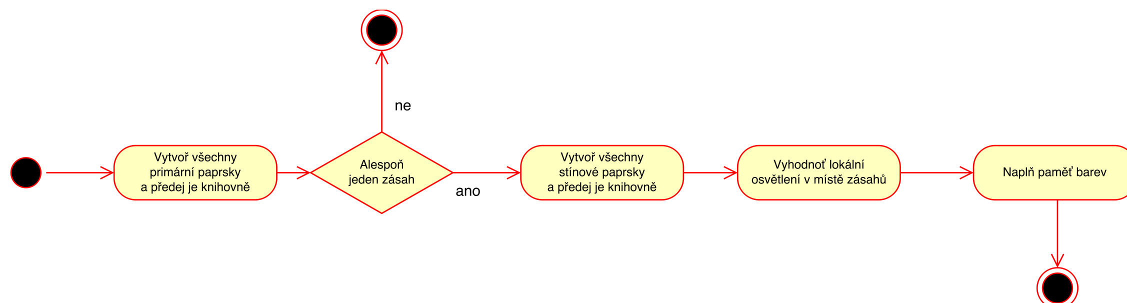
- sceneFile - cesta k souboru se scénou
- cameraPosition - pozice kamery ve světových souřadnicích
- cameraLookAt - bod ve světových souřadnicích, na který je kamera zamířena
- primaryMode - mód vrhání primárních paprsků (SSE, AVX, NonSIMD)
- shadowMode - mód vrhání stínových paprsků (SSE, AVX, NonSIMD)
- lightPosition – pozice bodového světelného zdroje (pokud se ve scéně nějaký nachází)
- lightColor - barva světla v RGB složkách
- method - metoda syntézy obrazu (pathtracing, raytracing)
- outputFile - jméno výstupního souboru
- firstFrame - číslo prvního snímku v animaci
- lastFrame - číslo posledního snímku v animaci

Každý parametr začíná předponou, která určuje, o jaký datový typ se jedná (f = desetinná čísla, i = celá čísla, c = řetězec znaků, v = vrchol o třech souřadnicích, rgb = barva definovaná v prostoru RGB). Tato předpona se poté již nepoužívá při získávání parametrů z konfiguračního souboru. Aplikace pracuje s grafickými modely ve formátu OBJ. Jedná se o otevřený formát popisu scény, ve kterém je definována pozice každého vrcholu, UV souřadnice textur, normály a plochy definované jako list vrcholů, které tyto plochy tvoří. Materiál jednotlivých vrcholů/ploch je definován v samostatném souboru s příponou MTL. Z jednoho OBJ souboru je možné odkazovat na více MTL souborů, které každému materiálu přiřadí jméno, pomocí kterého je možné materiál použít v OBJ souboru. Existuje spousta otevřených knihoven, které je možné použít pro načítání OBJ souborů. Já jsem se rozhodl použít knihovnu Kixor [16], která umožňuje snadné načítání jak geometrie, tak materiálů. V konfiguračním souboru je možné psát i komentáře. Stačí, aby řádek začínal symbolem # a bude automaticky vyhodnocen jako komentář.

3.4.2 Syntéza pomocí sledování paprsku

Syntéza statické scény pomocí sledování paprsku z pohledu aplikace probíhá následovně. Po načtení scény z OBJ souboru se inicializuje třída sloužící k vrhání paprsků. Tato inicializace vytvoří datovou sktukturu hierarchie obálek a nastaví stav knihovny pro vrhání paprsků (způsob vrhání paprsků, počet jader, výpis informací do konzole nebo souboru a další). Poté se vytvoří všechny primární paprsky, které se vrhnou do scény všechny najednou (dle konfigurace se vybere třída na vrhání paprsků a počet jader). Následně proběhne vyhodnocení. Ke všem primárním paprskům, které zasáhly nějaké těleso ve scéně, se vytvoří stínové paprsky a opět se vrhnou všechny najednou. Můžeme se opět rozhodnout, jestli využijeme SIMD zpracování, nebo standardní. V případě malého počtu primárních zásahu je lepší využít standardní zpracování, protože čas strávený úpravou dat pro SIMD zpracování by byl větší než benefit, který by nám toto zpracování přineslo. Poté již můžeme vyhodnotit

zásahy stínových paprsků a v případě nezastínění vypočítat lokální osvětlení. K tomu se využije Phongův osvětlovací model. Tento algoritmus tedy provádí syntézu scény pouze pomocí lokálního osvětlení.



Obrázek 3.8: Aktivity diagram sledování paprsků

Při syntéze dynamických scén je algoritmu obdobný až na to, že již neprobíhá inicializace třídy sloužící k vrhání paprsků, ale pouze přestavba hierarchie obálek. Uživatel nemusí volat žádnou metodu, která by tuto přestavbu spustila, ale knihovna sama pozná, kdy se jedná o následující snímek. Toto rozhodování probíhá na základě toho, jestli již v minulosti byla nějaká scéna vykreslena. Benefit využití SIMD bude největší, pokud uživatel vloží koherentní paprsky do vektoru vedle sebe. Pokud bychom chtěli postavit hierarchii obálek pro každou scénu zvlášť, stačí pro každý snímek zvonu zavolat inicializační metodu.

3.4.3 Syntéza pomocí sledování cesty

Syntéza pomocí sledování cesty se v několika bodech liší od sledování paprsku. První odlišnost přichází již v generování primárních paprsků. Při sledování cesty si vygenerujeme několik primárních paprsků pro každý pixel. Toto můžeme udělat i při sledování paprsku, protože nám to pomůže odstranit aliasing a zubaté hrany při generování obrázků s menším rozlišením. U generování paprsků procházejících jedním pixelem využijeme vzorkování. Můžeme si vygenerovat náhodné souřadnice x a y v intervalu od mínus do plus jedné poloviny a přičíst je k souřadnicím středu aktuálně generovaného pixelu. Každý sloupec pixelů na výsledném obrázku je zpracováván na jednom jádře, které po ukončení své práce začne zpracovávat další sloupec. I když C++ od verze 11 poskytuje k vícevláknovému zpracování třídu thread, která je součástí jménného prostoru std, rozhodl jsem se pro vícevláknové zpracování použít OpenMP. Jedná se o nadstavbu jazyka C++, která umožňuje snadnou paralelizaci částí programu. Pro použití direktiv OpenMP je potřeba přeložit program s patřičnými přepínači (pokud používáme Visual Studio, jedná se o přepínač /openmp).

Každý primární paprsek je nejprve vyhodnocen na jeho průsečík se scénou. Pokud paprsek scénu zasahuje, pomocí ruské rulety se určí, jak bude tento paprsek dále vyhodnocen. Vstupním parametrem ruské rulety v mé aplikaci je maximální hodnota difuzní nebo spekulární komponenty (v intervalu od nuly do jedné) materiálu, na který dopadl paprsek. Tato maximální hodnota je poté porovnána s náhodně vygenerovaným číslem. Pokud je toto náhodně vygenerované číslo menší, než maximální vstupní hodnota, sledování cesty paprsku je ukončeno. Je možné ruskou ruletu úplně přeskočit a sledovat paprsek pouze do určitého

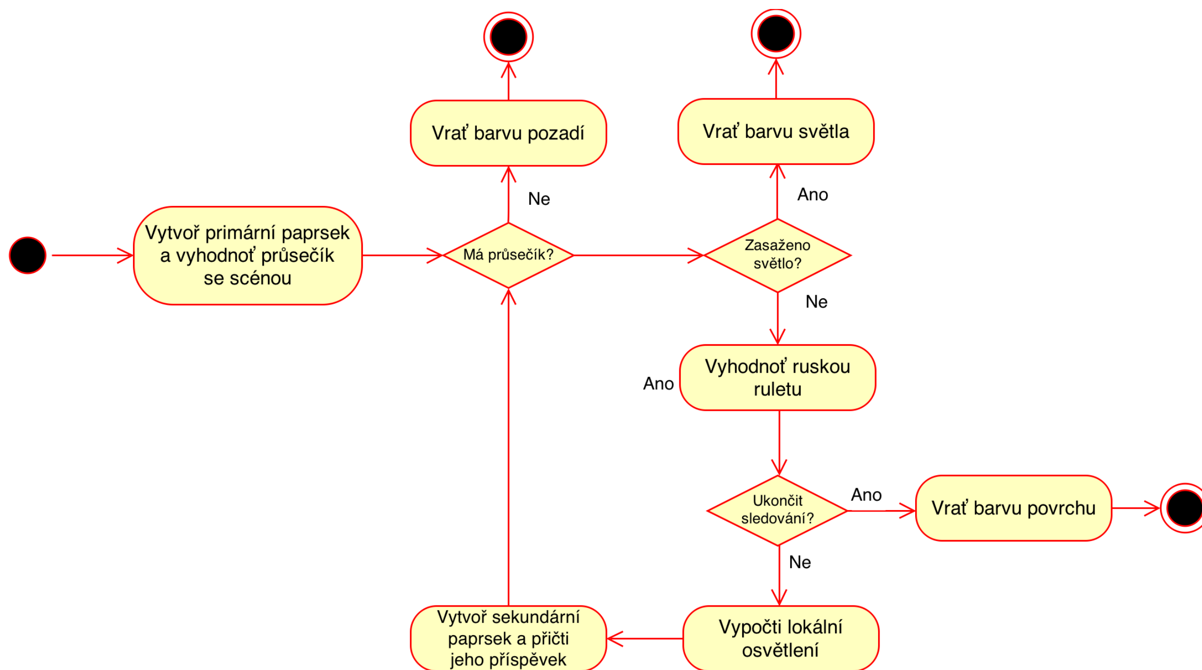
pevně daného maximálního počtu odrazů. V případě, že paprsek je dále vyhodnocován, určí se v místě zásahu lokální osvětlení a vytvoří se nový paprsek, který vychází z místa zásahu primárního paprsku. Směr tohoto odraženého paprsku závisí na materiálu povrchu. Pokud se paprsek odráží z difuzního povrchu, teoreticky je možné vybrat libovolný směr na polokouli ve směru normály povrchu. Má aplikace využívá kosínem vážené vzorkování, které umísťuje směr odraženého paprsku více k normále zasažené plochy, kde předpokládáme větší přínos globálního osvětlení od okolních předmětů. Při odrazu ze spekulárního povrchu aplikace provádí ideální zrcadlový odraz.

Jelikož je algoritmus sledování cesty poměrně časově náročný, aplikace umožňuje sledovat aktuální procentuální vyjádření již zpracovaných pixelů.

```
Initialization phase started
----------------------------
Parsing configuration file: DONE!
Parsing scene OBJ file: DONE!
Number of triangles: 36

Starting rendering
----------------------------
Pathtracing scene ...
Progress:                63%
```

Obrázek 3.9: Ukázka uživatelského rozhraní aplikace



Obrázek 3.10: Aktivity diagram sledování cesty

Kapitola 4

Výsledky

Tato kapitola má za úkol popsat výsledky testování knihovny na vrhání paprsků. Kapitola testování je rozdělena na dvě části – statické scény a dynamické scény. Knihovnu jsem se rozhodl otestovat na třech statických scénách a také na třech dynamických scénách, jejichž náhledy se nacházejí na obrázcích níže v příslušných kapitolách. Testování probíhalo na dvou sestavách:
Pracovní název: PC32

- Operační systém: Windows 7 Professional SP1, 32 bitový operační systém
- Procesor: Inter(R) Core(TM) i5-2520M CPU @ 2.50GHz (2 jádra, 2 vlákna)
- Paměť RAM: 4 GB DDR3 @ 1333MHz
- Kompilátor: Visual Studio 2013

Pracovní název: PC64

- Operační systém: Windows 7 Professional SP1, 64 bitový operační systém
- Procesor: Inter(R) Core(TM) i5-3350P CPU @ 3.30GHz (4 jádra, 4 vlákna)
- Paměť RAM: 8 GB DDR3 @ 1600MHz
- Kompilátor: Visual Studio 2013

Kompilátor byl nastaven na optimalizaci rychlosti (přepínač /O2) a na použití AVX (přepínač /arch:AVX) respektive SSE4 (přepínač /arch:SSE) instrukcí. U standardní implementace nevyužívající AVX nebo SSE4 nebyl tento přepínač nastaven. Každá scéna byla vykreslena třikrát a jako konečný výsledek se vzala nejlepší naměřená hodnota.

4.1 Statické scény

Kapitola o testování statických scén, vykreslených pomocí algoritmu sledování paprsku, je rozdělena do následujících částí:

- Porovnání různé stavby hierarchie obálek pro všechny scény a vliv na její cenu a rychlost traverzace
- Porovnání traverzace standardním algoritmem po jednom paprsku s traverzací využívajících SIMD instrukce
- Porovnání traverzace na jednom jádře a za využití více jader procesoru
- Porovnání standardních algoritmů pro výpočet průsečíku paprsku s trojúhelníkem

Při testování statických scén, vykreslených pomocí algoritmu sledování cesty, jsem se zaměřil hlavně na porovnání výsledných obrázků získaných metodou globálního a lokálního zobrazení, času, který byl potřeba pro vykreslení a přínos optimalizačních metod ruské rulety a vzorkování podle důležitosti.

K jednotlivým bodům můžeme formulovat předpoklady, které vycházejí z teorie popsané v první kapitole. Maximální zrychlení, které jsme schopní dosáhnout při použití 128 bitových registrů, SSE4 instrukcí a naprosto totožných algoritmů, je čtyřnásobné. Vzhledem k režii, která je potřeba pro upravení dat do přijatelné podoby pro SSE4, můžeme očekávat celkové zrychlení na úrovni 2,5-3,5x původního času vykreslování. Zajímavé bude také sledovat zrychlení u různých typů scén. Dá se očekávat, že pro řidčeji zaplněné scény nebude celkový přínos tak výrazný, naopak pro velice husté scény by měl být rozdíl v časech vykreslení znatelný. Podobný předpoklad můžeme formulovat i pro porovnání SSE4 a AVX. Dle referenčního článku [4] by mělo být vykreslení za použití AVX o 50% rychlejší než za použití SSE4 instrukcí.

Hierarchie obálek je vždy postavena pomocí SAH, ale s různým počtem objektů v listech. Z toho, co víme z kapitoly o hierarchii obálek, můžeme usoudit, že stavba s menším počtem objektů v listech by měla trvat déle, ale výsledná struktura by měla poskytovat rychlejší traverzaci (záleží ovšem také na struktuře scény a volbě konstant $c_{intersection}$ a $c_{traverse}$ v cenové funkci).

Testování statických scén je zaměřené spíše na hrubou výpočetní sílu vyjádřenou v milionech vrhnutých paprsků za vteřinu. Tato veličina nám dává přehled o tom, jak rychle jsme schopni danou scénu vykreslit, i když přesně neznáme parametry obrázku. Počet stínových a sekundárních paprsků nejsme schopni předem přesně odhadnout.

4.1.1 Parametry testovaných statických scén

Statické scény pocházejí z úložiště grafických modelů McGuire [17]. V tabulce č. 4.1 jsou vidět parametry testovaných statických scén. Všechny byly otestovány v rozlišení 2048x2048 pixelů, odtud tedy 4194304 primárních paprsků na každou scénu. Počet stínových paprsků se liší v závislosti na tom, jaké je nastavení kamery. Conference Room je model skutečné zasedací místnosti. Scéna je často používaná v různých článcích zabývajících se vrháním paprsků. Happy Buddha představuje středně složitou scénu s jedním objektem obsahujícím více než milion troúhelníků. Nejkomplexnější scénou se nazývá San Miguel. Jedná se o dům s téměř osmi miliony trojúhelníků s velice detailní geometrií květin, stromů a různých architektonických prvků. Referenční obrázky testovaných scén, pocházející přímo z testování, jsou na obrázku č. 4.1.

Obrázek 4.1: Obrázky testovaných statických scén

Jméno	Počet trojúhelníků	Primárních paprsků	Stínových paprsků
Conference Room	331 179	4 194 304	4 194 251
Happy Buddha	1 087 474	4 194 304	940 289
San Miguel	7 880 512	4 194 304	4 194 304

Tabulka 4.1: Parametry testovaných statických scén

4.1.2 Porovnání postavených hierarchií obálek

Tabulka č. 4.2 obsahuje charakteristiky hierarchií obálek, nad kterými probíhala traverzace paprsků. Čas stavby všech hierarchií uvedených v tabulce č. 4.2 byl změřen na testovací sestavě PC64. Na sestavě PC32 probíhala stavba hierarchie pro scénu Conference Room o 10% pomaleji, pro scénu Happy Buddha o 30% pomaleji a pro scénu San Miguel o 50% pomaleji. Sloupec $\triangle$ *v listu* udává maximální počet trojúhelníků, které se mohou nacházet v listu hierarchie. Pro stanovení ceny hierarchie byly použity konstanty $c_{intersection} = 2$ a $c_{traverse} = 3$. Konstanty jsou takto zvoleny záměrně, aby bylo možné porovnat kvalitu postavených hierarchií s referenčním článkem [2], který poskytuje referenční hodnoty cen a délku stavby pro scény Conference Room a Happy Buddha, které jsem použil i já ve své práci. Nejkvalitnější hierarchie, jakou jsem pro scénu Conference Room dokázal postavit, měla cenu 152.615 a její stavba zabrala 2.229 vteřiny. V referenčním článku se autorům podařilo bez optimalizace postavit hierarchii o 17% kvalitnější, ovšem potřebovali k tomu o polovinu více času. U druhé scény Happy Buddha jsem dosáhl nejlepší ceny 161.707, což znamená o 2% lepší výsledek než v referenčním článku při téměř poloviční době stavby. Všechny výše zmíněné hodnoty z článku jsou před optimalizací, které se článek hlavně týká. Způsob optimalizace navrhovaný autory dokáže ještě dále kvalitu hierarchie vylepšit. Tuto metodu jsem však do své práce neimplementoval, takže porovnávám pouze s hodnotami počáteční stavby.

Z porovnání dvou testovaných hierarchií vychází, že stavba hierarchie v mé diplomové práci probíhá rychleji než v referenčním článku a s podobnými výsledky.

Scéna	△ v listu	Stavba [s]	Cena	Hloubka	Listů
Conference Room	5	2.229	152.615	49	93 838
	10	1.338	159.566	47	47 969
	15	0.98	171.161	45	30 865
Happy Buddha	5	8.654	161.707	44	364 341
	10	5.12	175.293	41	174 029
	15	3.836	194.759	39	113 684
San Miguel	20	28.312	298.27	62	657 044
	30	23.718	331.654	60	449 482
	40	20.724	410.47	60	335 733

Tabulka 4.2: Parametry testovaných statických scén

4.1.3 Porovnání standardní implementace proti SIMD

Jako první jsem se rozhodl porovnat traverzaci primárních paprsků při použití SIMD a bez použití SIMD. Výsledky měření se nacházejí v tabulce č. 4.3. Všechny hodnoty v tabulce udávají počet miliony paprsků, které je možné traverzovat za vteřinu. Sloupec *Standard* udává rychlost traverzace paprsků, které je možné traverzovat samostatně, sloupec *SSE* udává rychlost traverzace čtveřic paprsků pomocí SSE4 a obdobně v sloupci *AVX* za využití AVX instrukcí. Měření byla provedena na konfiguraci PC64 a paprsky byly vrhány na čtyřech jádrech, takže uvedené výsledky jsou nejlepší dosažené. Přínos vrhání paprsků na více vláknech bude diskutován v další podkapitole.

Z výsledků traverzace primárních paprsků je patrné, že SIMD zpracování opravdu přináší zlepšení výkonu. U scény Conference Room je traverzace za použití SSE4 průměrně 2,87x rychlejší než standardní traverzace. Happy Buddha je případ scény, ve které 77% primárních paprsků vůbec nezasahuje objekt scény, takže jejich traverzace končí hned v kořenu hierarchie obálek. Právě proto u této scény přináší SSE4 pouze 1,98x rychlejší traverzaci primárních paprsků. Výsledky poslední scény, San Miguel, jsou velice podobné výsledkům Conference Room. Traverzace za použití SSE4 je 2,54x rychlejší než standardní algoritmus.

Zajímavé je také porovnání SSE4 a AVX instrukcí. Teoreticky by měla být traverzace pomocí AVX až 1,5x rychlejší. K tomuto teoretickému předpokladu jsem se přiblížil u testovaných scén Conference Hall a San Miguel. U scény Conference Hall byla traverzace 1,43x rychlejší a u San Miguel dokonce 1,54x. V těchto scénách se začala projevovat výhoda AVX, kde dochází k traverzaci 8 paprsků zároveň. Naměřené výsledky je možné porovnat s referenčním článkem [13], ve kterém je uveden výsledek vykreslení scény Conference Room pro svazky velikosti 2x2 paprsků. Na procesoru AMD Opteron @2.6 GHz dosáhli rychlosti traverzace 1,89 milionu paprsků za vteřinu. V článku bohužel není uvedeno, jestli traverzace probíhala na jednom, či více jádrech. V mé implementaci jsem dosáhl rychlosti traverzace 1,996 milionu paprsků za vteřinu na jednom jádře a testovací konfiguraci PC64.

Traverzace stínových paprsků je též zobrazena v tabulce č. 4.3. Zatímco u primárních paprsků byl jejich počet stejně pro každou scénu, u stínových paprsků z výsledků pozorujeme, že čím více stínových paprsků vyhodnocujeme, tím větší je přínos SIMD. Zrychlení za použití SIMD instrukcí je téměř stejné jako u primárních paprsků. Je to dáno tím, že stínové paprsky stále vykazují vysokou koherenci, protože mají společný směr.

		Traverzace [miliony paprsků za vteřinu]					
		Standard		**SSE4**		**AVX**	
Scéna	**△ v listu**	**Primární**	**Stínové**	**Primární**	**Stínové**	**Primární**	**Stínové**
Conference room	5	0.69	0.74	1.98	1.98	2.87	2.82
	10	0.7	0.77	1.99	2.10	2.85	2.76
	15	0.7	0.74	1.98	2.06	2.77	2.8
Happy Buddha	5	1.91	0.88	3.62	1.6	4.04	2.14
	10	1.82	0.89	3.59	1.59	4.03	2.06
	15	1.73	0.94	3.59	1.59	3.93	2.04
San Miguel	20	0.32	0.17	0.82	0.51	1.21	0.74
	30	0.31	0.16	0.75	0.49	1.15	0.7
	40	0.27	0.15	0.7	0.44	1.06	0.66

Tabulka 4.3: Traverzace primárních a stínových paprsků - hodnoty jsou uvedené v milionech paprsků za vteřinu

		Průměrný počet traverzací na primární paprsek		
Scéna	**△ v listu**	**Standard]**	**SSE**	**AVX**
Conference Room	5	48 (43+5)	52 (47+6)	55 (48+7)
	10	45 (41+4)	49 (44+5)	50 (45+5)
	15	43 (39+4)	47 (42+5)	48 (43+5)
Happy Buddha	5	15 (4+1)	19 (17+2)	23 (20+3)
	10	14 (13+1)	16 (15+1)	19 (17+2)
	15	13 (12+1)	15 (14+1)	18 (16+2)
San Miguel	20	98 (93+5)	134 (122+12)	138 (125+13)
	30	96 (91+5)	131 (119+12)	133 (121+12)
	40	94 (89+5)	127 (115+12)	129 (117+12)

Tabulka 4.4: Průměrný počet traverzací na primární paprsek (vnitřních uzlů + listů)

Při traverzaci většího množství paprsků zároveň dochází k tomu, že neustále dokola testujeme i paprsky, jejichž traverzace by byla při standardní implementaci již ukončena. Tento rozdíl je vidět v tabulce č. 4.4, která udává průměrný počet traverzačních kroků, které musí každý primární paprsek vykonat, než je vyhodnocen. Pro SSE4 udává průměrný počet na čtveřici paprsků, pro AVX je to průměrný počet na osmici. První číslo v závorce je průměrný počet traverzovaných vnitřních uzlů, druhé číslo je průměrný počet traverzovaných listů.

Při použití SIMD instrukcí traverzujeme průměrně o 21% více uzlů (vnitřních i listů), než při standardní implementaci. Není přitom velký rozdíl, jestli používáme SSE4 nebo AVX. Téměř podobné výsledky sledujeme i u sekundárních paprsků.

Na závěr této podkapitoly bych ještě rád porovnal výsledky dosažené na konfiguracích PC64 a PC32. Stavba hierarchie obálek na konfigurace PC64 byla průměrně o 18% rychlejší než na konfiguraci PC32. Rozdíl v rychlosti traverzace mezi oběma sestavami byl ještě markantnější, hlavně kvůli tomu, že sestava PC32 disponuje pouze dvěma fyzickými jádry

		Počet jader				
Scéna	**△ v listu**	**1**	**2**	**3**	**4**	**Přínos**
Conference Room	5	1.71	1.91	2.1	2.87	67%
Happy Buddha	5	2.23	3.06	3.18	4.04	81%
San Miguel	20	0.67	0.88	1.09	1.21	80%

Tabulka 4.5: Vliv vícevláknového zpracování na rychlost traverzace primárních paprsků

procesoru, oproti čtyřem jádrům sestavy PC64. U scény Conference Room byla traverzace primárních paprsků při použití AVX instrukcí průměrně o 95% rychlejší, než při stejném nastavení na sestavě PC32. Na scéně Happy Buddha nebyl rozdíl tak patrný, a to hlavně z toho důvodu, že zde nedochází průměrně k tolika traverzacím na paprsek. Rozdíl v rychlosti traverzování tedy činil pouze 20%. Na poslední a nejnáročnější scéně, San Miguel, jsem za stejného nastavení naměřil průměrně 110% rozdíl.

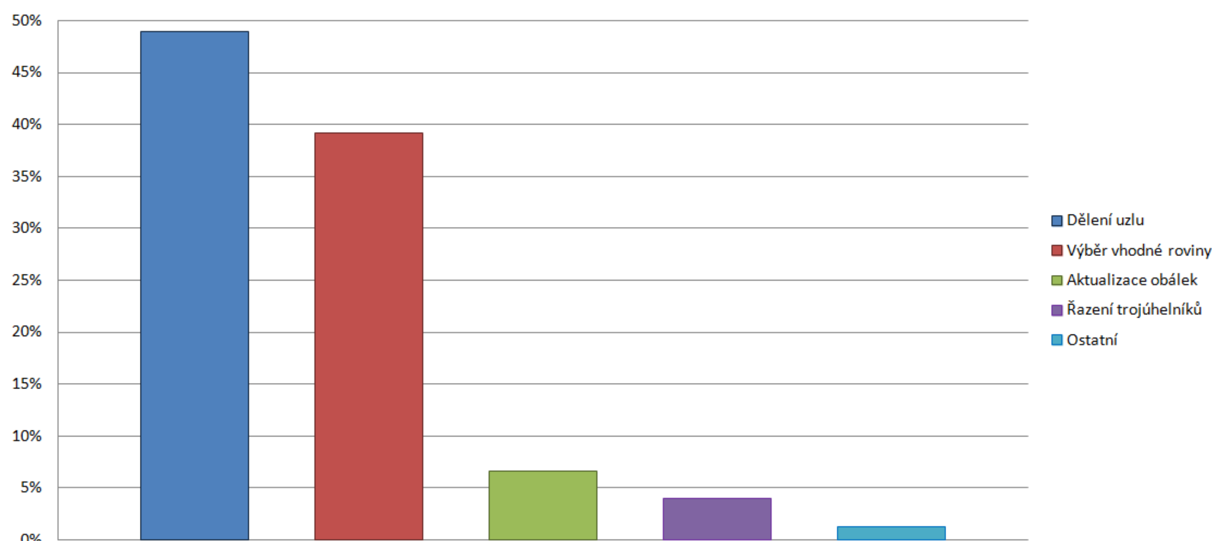
4.1.4 Vliv více vláknového zpracování

Při implementaci vícevláknového zpracování jsem využil direktivy OpenMP. Testování jsem prováděl na sestavě PC64 a pomocí AVX instrukcí, která podávala na všech scénách nejlepší výsledky. Přínos u jednotlivých testovaných scén ukazuje tabulka č.4.5. Sloupce u jednotlivých jader opět ukazují hodnoty v traverzovaných milionech primárních paprsků za sekundu. Poslední sloupec uvádí přínos použití čtyř jader oproti jednomu. Z popisu algoritmu sledování paprsku je jasné, že vrhání paprsků na více jádrech bude na většině scén rychlejší, než na jednom.

4.1.5 Profiling aplikace

Z testování statických scén je patrné, kolik vnitřních uzlů a kolik listů musí každý paprsek průměrně traverzovat, než je jeho sledování ukončeno. Zajímavé výsledky by mohl přinést profiling aplikace, ze kterého bude patrné, kolik procent výpočetního času se tráví v jakých částech běžící aplikace. Profiling aplikace proběhl v Microsoft Visual Studio 2013 a byl spuštěn pro statickou scénu San Miguel na počítačové sestavě PC64. Hierachie obálek byla postavena s maximálně dvaceti trojúhelníky v jednom listu. Profilingem jsem chtěl odhalit místa, která jsou za běhu prováděna nejčastěji a tudíž by jejich optimalizace mohla přinést další zrychlení vykreslování. Zaměřil jsem se hlavně na proces stavby hierarchie obálek a na průběh traverzace paprsků. Z grafu č.4.2 je patrné, že nejdelší čas stavby hierarchie zabere dělení jednotlivých vnitřních uzlů a výběr vhodné roviny pro další dělení. Tyto dvě činnosti dohromady představují 88% celé doby stavby. Třetí nejnáročnější operací je aktualizace obalových těles u nově vzniklých uzlů. Čtyři procenta času stavby byla strávená přesouváním trojúhelníků ve vektoru tak, aby se trojúhelníky nacházející se v jednom listu, byly umístěny vedle sebe.

Další sledovanou činností byla traverzace paprsků. Výsledky v tabulce č.4.4 ukazují, že průměrně každý paprsek traverzuje 125 vnitřních uzlů a 13 listů. Z profilingu traverzace ovšem vyplynulo, že z celkového času traverzace připadá na testy průsečíku paprsku s obalovým tělesem 46% a na testy průsečíku paprsku s trojúhelníky v listech 43,5%. Ostatních



Obrázek 4.2: Výsledky profilingu hierarchie obálek

10,5% je stráveno při dalších operacích jako je ukládání a odebírání listů ze zásobníku a dalších.

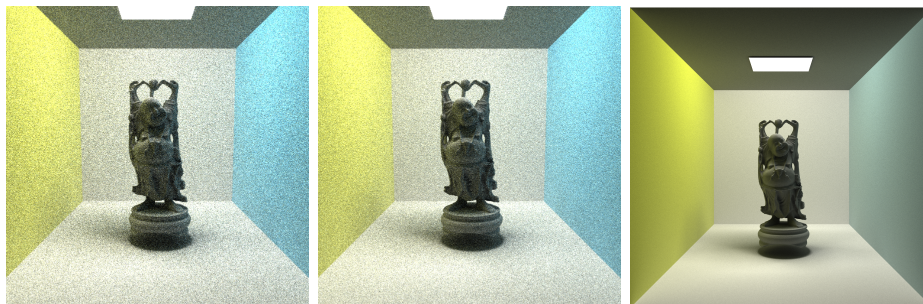
4.2 Porovnání lokální a globální osvětlovací metody

Jak již bylo zmíněno výše, testovací aplikace je schopna scénu vykreslit jak pomocí sledování paprsku, tak pomocí sledování cesty. Přišlo mi zajímavé do této kapitoly zahrnout také porovnání těchto dvou metod, protože rozdíl v kvalitě výsledků je značný. Úplně vlevo na sérii obrázků č. 4.3 je známá scéna Cornell Box s jedním kvádrem ze zrcadlového materiálu vykreslená pomocí sledování paprsku. Plošný zdroj světla je nahrazen bodovým zhruba na stejné pozici. Na obrázku úplně vlevo si můžeme všimnout ostrých stínů, které vznikají při použití bodového světelného zdroje a absence jevů, které vznikají při výpočtu globálního osvětlení. Chtěl bych ještě upozornit na odvrácenou stranu menšího kvádru, která se sice nachází ve stínu, ale dopadá na ni světlo odražené od zrcadlového povrchu, takže by neměla být celá černá. Na druhou stranu, obrázek uprostřed je vykreslen pomocí sledování cesty a již na první pohled vidíme, že působí reálnějším dojmem, než ten úplně vlevo. Na každý pixel bylo použito 4000 primárních paprsků s minimálně dvěmi odrazy ve scéně, sledováním ukončeným pomocí ruské rulety a vzorkováním odražených paprsků podle důležitosti. Plošný zdroj světla byl v každém bodě navzorkován jedním vzorkem. Můžeme si všimnout měkkých stínů, osvětlení odvrácené strany menšího kvádru a difuzního přenosu barev (např. nazelenalá levá strana menšího kvádru). V detailu úplně vpravo je patrné, že výsledný obrázek stále obsahuje šum, který je pro metodu sledování cesty typický a pro jeho odstranění by bylo potřeba vrhnout více primární paprsků každým pixelem.

Dále jsme se rozhodl otestovat různé způsoby vzorkování směru difuzního odrazu. Po dopadu paprsku na čistě difuzní povrch je potřeba rozhodnout, kam se paprsek odrazí (pokud není sledování ukončeno ruskou ruletou). Na obrázku č.4.4 je úplně vlevo vykreslena scéna

Obrázek 4.3: Porovnání sledování paprsku a sledování cesty

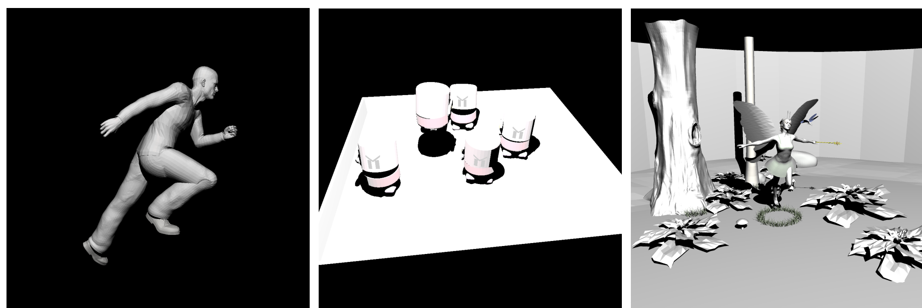
Buddhy v Cornell Boxu za použití uniformního vzorkování ve směru normály se 100 vzorky na jeden pixel výsledného obrázku. Obrázek uprostřed představuje tu samou scénu vykreslenou za použití vzorkování dle důležitosti. Doba vykreslení scény vlevo byla o 2% rychlejší, než doba vykreslení scény uprostřed. Při použitím vzorkování dle důležitosti obsahuje obrázek uprostřed méně šumu, než obrázek vlevo. Pouze pro ilustraci jsem přiložil i obrázek úplně vpravo, na kterém je vidět totožná scéna s 4000 vzorky na jeden pixel. Scéna neobsahuje žádné zrcadlové nebo průhledné materiály, takže i při relativně malém množství primárních paprsků vedených každým pixelem, vzorkování podle důležitosti a ruské rulety, bylo dosaženo poměrně kvalitního výsledku.



Obrázek 4.4: Porovnání vzorkování difuzních odrazů

4.3 Dynamické scény

U dynamických scén jsem se rozhodl testovat hlavně rychlost přestavby hierarchie obálek a vliv přestavby na kvalitu postaveného stromu a tudíž i rychlost traverzace. Zajímavé bylo také porovnat, jaký výkonnostní propad nastane při použití jednoduché přestavby hierarchie proti případu, kdy bychom stavěli celou novou hierarchii pro každý snímek. Z teoretického základu o hierarchii obálek se dá předpokládat, že pouhá přestavba hierarchie bude rychlejší než nové postavení, ale traverzace bude naopak rychlejší u nově postaveného stromu. Dynamické scény pocházejí z The Utah 3D Animation Repository [14], ve kterém je možné najít několik animovaných scén ve formátu OBJ. Každý snímek je tvořen novým OBJ souborem. Při použití knihovny u animovaných scén je důležité, aby trojúhelníky byly definovány v každém snímku ve stejném pořadí, a aby mezi snímky žádné trojúhelníky nevznikaly ani nezanikaly. S tímto případem by si neuměla zvolená metoda přestavby hierarchie poradit.



Obrázek 4.5: Obrázky testovaných dynamických scén

Obrázky z vybraných scén je možné vidět na obrázku č. 4.5. Scéna běžce (Ben) obsahuje 78 089 trojúhelníku a jedná se o typický příklad deformace jednoho modelu. Během animace model zůstává na místě, ale jeho trup, ruce a nohy se hýbou, jako kdyby postava běžela. V typické interaktivní aplikaci ovšem dochází k pohybu více než jednoho modelu, a proto jsem do testů zařadil druhou scénu (Toasters), ve které se pět postaviček pohybuje po rovné ploše. Tyto postavičky do sebe narážejí, odrážejí se od sebe a přeskakují se. Tato scéna obsahuje pouze 11 141 trojúhelníků, ale vykazuje daleko dynamičtější chování, než scéna s běžcem. Poslední scéna (Fairy Forest) by měla naplno otestovat, zda je zvolený způsob přestavby dostatečný pro animované scény. Jedná se o scénu s 174 117 trojúhelníky, ve které tančí lesní víla, létá vážka a navíc se pohybují i jednotlivé rostliny. Jedná se tedy téměř o plně animovanou scénu, ve které dochází ke značným rozdílům v geometrii mezi jednotlivými snímky.

Všechny níže uvedené výsledky obsahují pouze traverzaci primárních paprsků (jejich počet a počet trojúhelníků scén je v tabulce č. 4.6), protože si myslím, že poskytnou dostatečné porovnání kvality postavené akcelerační struktury a rychlosti traverzace. Hierarchie obálek byla postavena s maximálně pěti objekty v každém listu. Pro přehlednost textu jsou všechny grafy zařazeny až na konec této podkapitoly.

Na prvním grafu č. 4.6 vidíme čas strávený stavbou akcelerační struktury, traverzací a výslednou cenu postavené akcelerační struktury za použití stavby nové hierarchie pro každý snímek. Průměrná cena hierarchie obálek přes všechny snímky byla 67,3 s průměrnou dobou

Jméno	**Počet trojúhelníků**	**Primárních paprsků**	**Snímků**
Ben	78 089	1 690 000	29
Toasters	11 141	1 048 576	105
Fairy Forest	174 117	1 440 000	21

Tabulka 4.6: Parametry testovaných dynamických scén

stavby 0,65 sekundy. Průměrná doba traverzace trvala 0,363 vteřiny s tím, že čas traverzace mezi jednotlivými snímky byl v podstatě konstantní, s rozdíly v rámci 10%. Je to logické, protože v modelu nedochází k velkým deformacím, pouze se pohybují některé jeho části. Pokud se podíváme na graf 4.7, kde dochází pouze k přestavbě hierarchie z prvního snímku, trvá přestavba pro každý snímek průměrně 0,007 sekundy a průměrná traverzace trvala 0,399 sekundy. Pozorujeme tedy zhoršení času traverzace o 10%. Průměrná cena při přestavbě stromu pro každý snímek byla 67,3 a při úpravě původního stromu mezi snímky byla 92,9. I přestože není hierarchie tak kvalitní, jako když jí stavíme pro každý snímek nově, v této scéně je aktualizace původní hierarchie dostačující a podává lepší výsledky, co do celkového času vykreslení animace, než stavba nové pro každý snímek. Ještě pro doplnění bych uvedl, že v této scéně jsem dosáhl rychlosti traverzace 4,23 milionu primárních paprsků za sekundu na testovací konfiguraci PC64 a využití AVX.

Druhá scéna zachycuje pohyb hraček po pevně ohraničeném území. Tato scéna je již o něco dynamičtější než model samotného běžce, protože dochází k pohybu více modelů zároveň, které se navíc od sebe vzdalují, přibližují a přeskakují. Dá se tedy očekávat, že rozdíl výkonu mezi přestavbou hierarchie v každém snímku a aktualizací hierarchie, bude tentokrát markantnější. Při pohledu na výsledky měření v grafech 4.8 a 4.9 vidíme, že až do 77. snímku je cena při aktualizování hierarchie průměrně lepší, než u stavby nové. To může být způsobeno tím, že při stavbě hierarchie pomocí SAH nevybíráme z dostatečně velkého počtu pozic dělících rovin (v mé implementaci jsou to čtyři pozice) a nepodaří se nám najít to nejlepší rozdělení vnitřního uzlu na listy. Když se podíváme detailněji na to, co se začíná ve scéně odehrávat od 77. snímku, začne nám být jasné, čím je dán následný propad kvality postavené hierarchie. Od 77. snímku přes sebe hračky začnou poskakovat, což začne deformovat původní obálky. Od tohoto snímku produkuje lepší hierarchie kompletní přestavba. V celkovém pohledu ceny hierarchie se tedy obě metody liší v průměrné ceně o 0,3% ve prospěch nově postavené hierarchie pro každý snímek. Při testování druhé dynamické scény jsme narazili na případ, kdy se začínají projevovat nedostatky metody aktualizace původní hierarchi. Průměrná rychlost traverzace primárních paprsků v této scéně činila 2,52 milionu paprsků za vteřinu na testovací konfiguraci PC64 s využitím AVX.

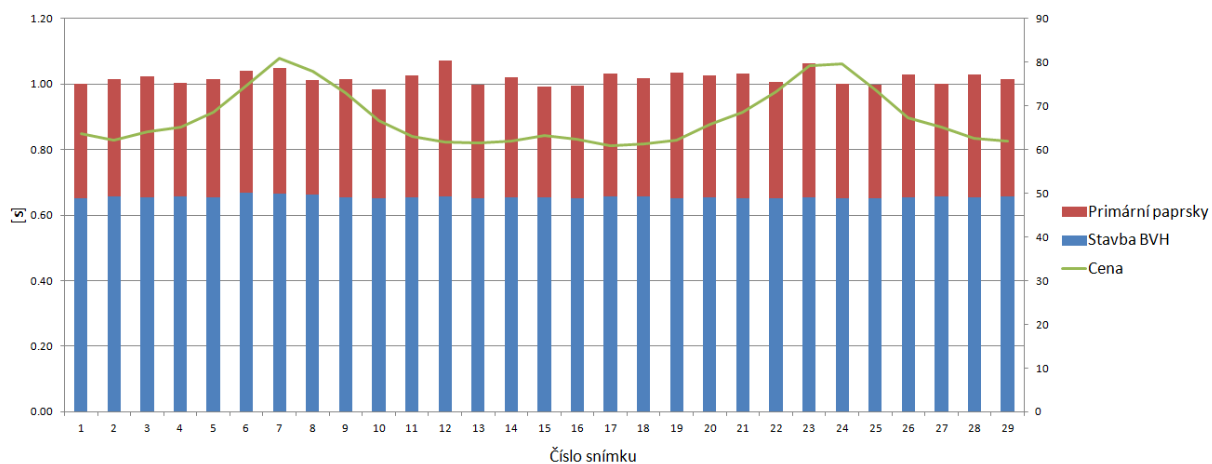
Poslední scéna s lesní vílou obsahuje nejméně snímků, za to je ale nejvíce dynamická a obsahuje nejvíce trojúhelníků. Zde se nejvíce projevil rozdíl v kvalitě hierarchie postavené mezi jednotlivými snímky a metoda stavby nové hierarchie v průměru poskytuje o 20% rychlejší traverzaci. Tyto výsledky pouze potvrzují předpoklady z teoretické části a testování druhé dynamické scény. Jelikož u této testované scény byly rozdíly mezi oběma metodami vykreslování dynamických scén největší, rozhodl jsem se ještě zařadit graf 4.12 znázorňující průměrný počet traverzačních kroků na jeden paprsek v každém snímku. Z grafu je patrné, že u hierarchie upravované aktualizacemi postupně vzrůstá počet traverzačních kroků. U

	Kompletní přestavba			**Aktualizace hierarchie**		
Scéna	**Úprava [s]**	**Cena**	**Traverzace PP**	**Úprava [s]**	**Cena**	**Traverzace PP**
Ben	0.66	67.304	4.64	0.03	92.9	4.24
Toasters	0.05	150.13	2.46	0.001	150.64	2.58
Fairy Forest	0.862	102.7	1.74	0.053	107.9	1.46

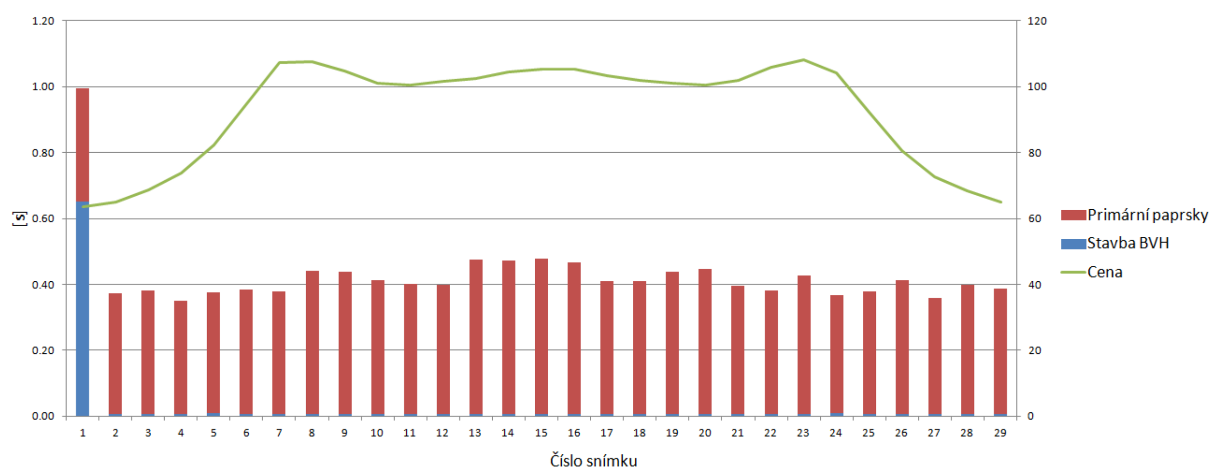
Tabulka 4.7: Shrnutí výsledků dynamických scén - všechny hodnoty jsou průměrem ze všech snímků, traverzace měřena na primárních paprscích

nově postavených hierarchií pro každý snímek tuto charakteristiku nepozorujeme. Z grafu je ještě patrné, že průměrný počet traverzačních kroků je přímo úměrný času vykreslování snímku, kdy nejdéle trvalo vykreslení snímků číslo 2, 16 a 19, které mají také největší počet průměrných traverzačních kroků na paprsek.

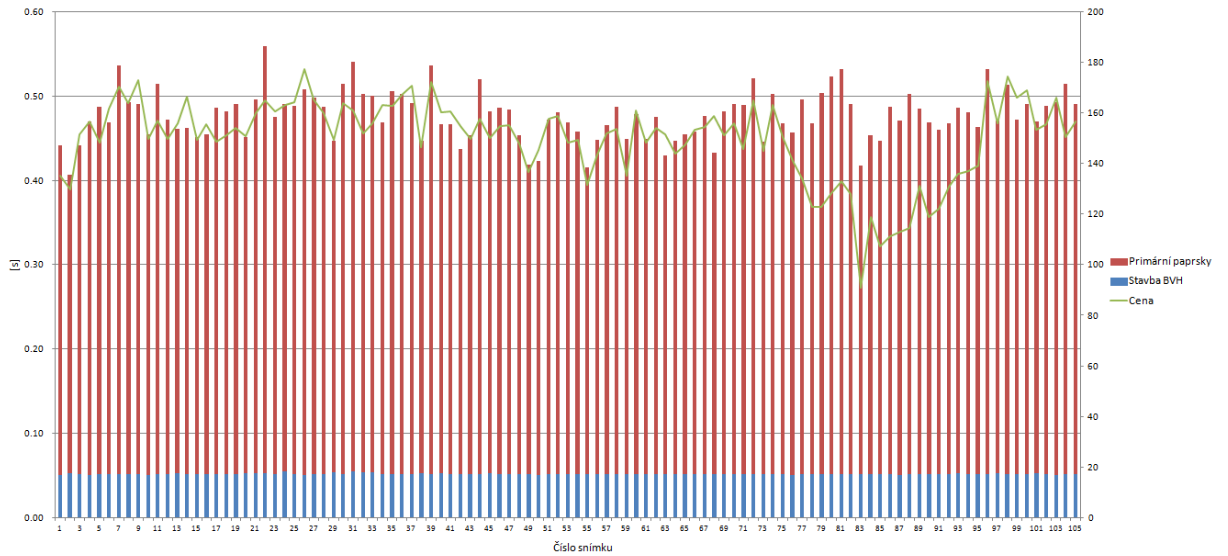
Výsledky testování dynamických scén jsou shrnuty v tabulce č.4.7. Hodnoty ve všech sloupcích tvoří průměr přes všechny snímky naměřené na konfiguraci PC64. Sloupec *prava* udává průměrný čas potřebný k úpravě hierarchie mezi jednotlivými snímky. Hodnoty v sloupci $Traverzace\ PP$ představují traverzované miliony primárních paprsků za vteřinu. Všechny hierarchie byly postaveny s maximálně pěti trojúhelníky v listu. Z výsledků je patrné, že metoda modifikace obalových těles je více než dostačující pro všechny tři vybrané testovací scény a ve všech z nich poskytuje lepší celkový čas potřebný k vykreslení všech snímků animace, než metoda stavby nové hierarchie pro každý snímek. Pokud by docházelo v průběhu aktualizace k velkému rozdílu v aktuální ceně v porovnání s počáteční cenou, můžeme strukturu postavit od znovu a dále pokračovat s aktualizacemi.



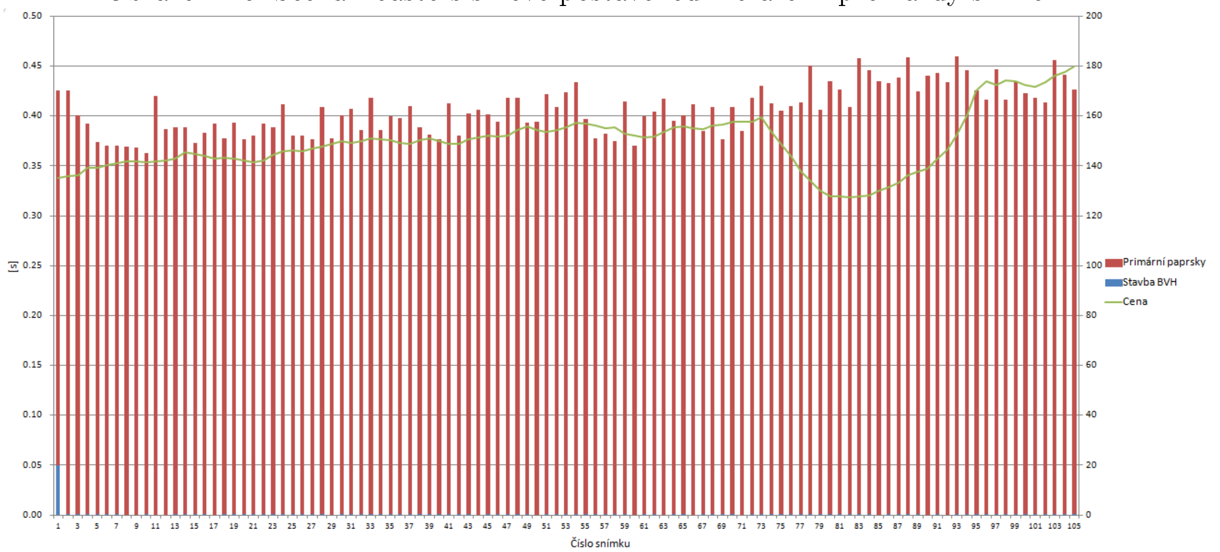
Obrázek 4.6: Scéna běžce s nově postavenou hierarchií pro každý snímek



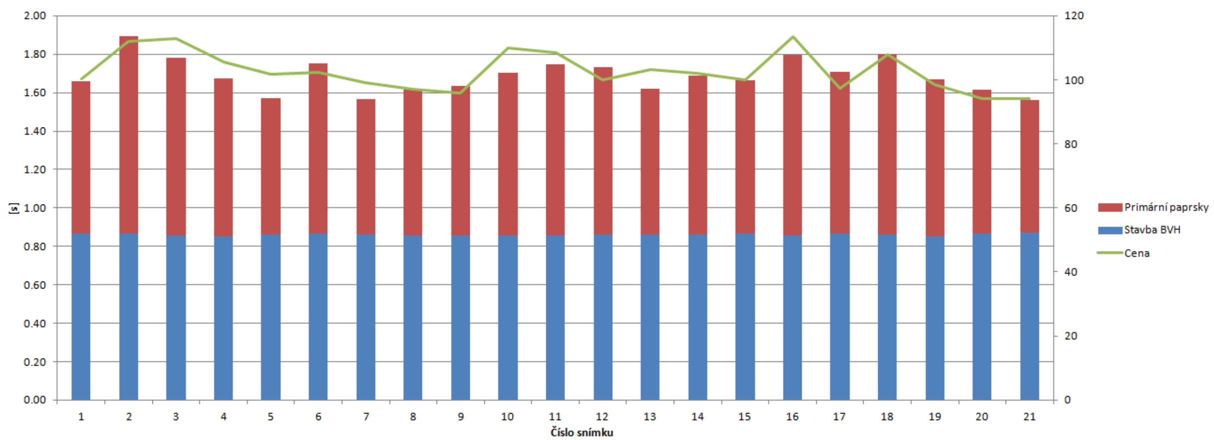
Obrázek 4.7: Scéna běžce s úpravami původní hierarchie



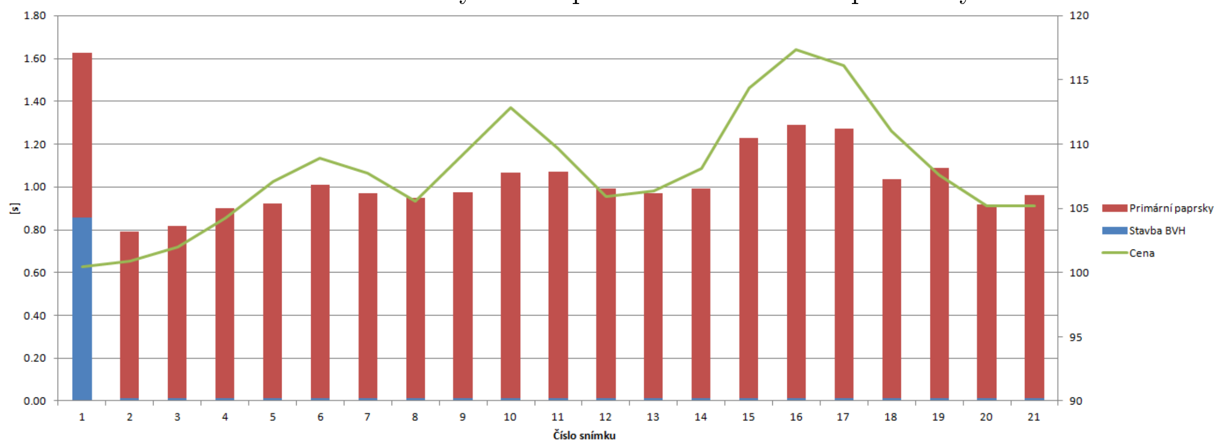
Obrázek 4.8: Scéna Toasters s nově postavenou hierarchií pro každý snímek



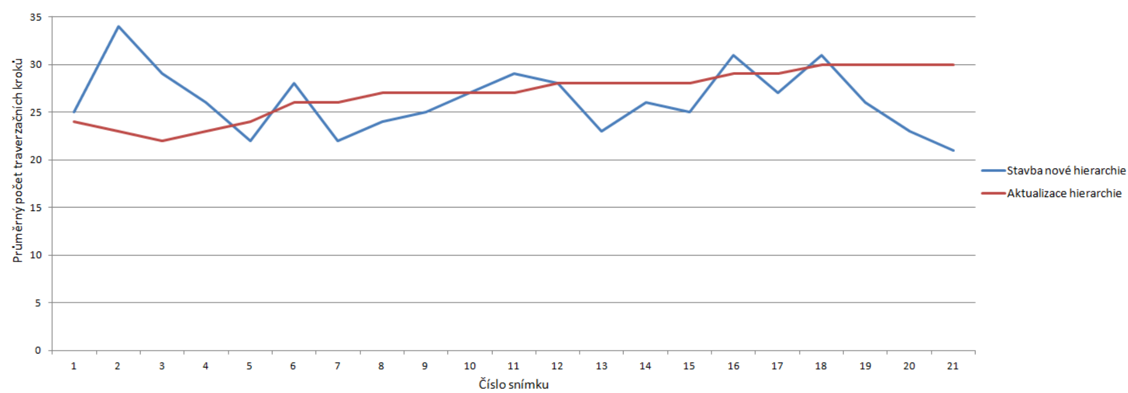
Obrázek 4.9: Scéna Toasters s úpravami původní hierarchie



Obrázek 4.10: Scéna lesní víly s nově postavenou hierarchií pro každý snímek



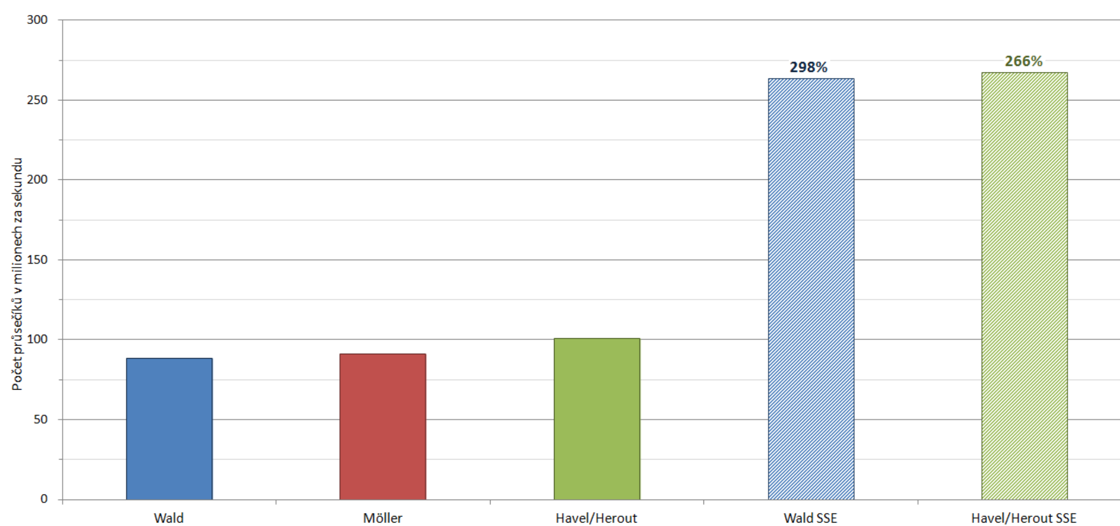
Obrázek 4.11: Scéna lesní víly s úpravami původní hierarchie



Obrázek 4.12: Průměrný počet traverzačních kroků na paprsek u obou metod

4.4 Porovnání naimplementovaných algoritmů

V mé diplomové práci jsem otestoval také výpočetní výkon jednotlivých algoritmů popsaných na začátku kapitoly implementace. Testování probíhalo na primárních paprscích na scéně Cornellova boxu bez akcelerační struktury, v rozlišení 2048 na 2048 pixelů a každý primární paprsek byl otestován na průsečík se všemi objekty ve scéně. Testování rychlosti probíhalo na konfiguraci PC64. Výsledky měření a porovnání algoritmů je k vidění na grafu č. 4.13. Má implementace potvrzuje výsledky naměřené v článku Ing. Jiřího Havla, Ph.D. a Doc. Ing. Adama Herouta, Ph.D. [5]. V jejich článku z měření vyplynulo, že jejich algoritmus je zhruba o 15% rychlejší, než algoritmus navržený Waldem. Mé výsledky standardní implementace potvrzují, že Havel/Heroutův algoritmus je o 14% rychlejší než Waldův a o 10% rychlejší než Möllerův algoritmus. Testování s využitím SSE4 instrukcí probíhalo pro čtveřice paprsků. SIMD varianta Waldova algoritmu je 2,98x rychlejší než standardní implementace, u Havlova/Heroutova algoritmu je zrychlení dvou a půl násobné oproti standardní implementaci. Jak je z grafu patrné, v SSE4 implementaci jsou si Havel/Heroutův a Waldův algoritmus téměř rovni, co do rychlosti výpočtů. Procenta nad sloupci v grafu č. 4.13 znamenají rozdíl mezi standardní a SSE4 implementací.



Obrázek 4.13: Porovnání rychlosti algoritmů pro výpočet průsečíku paprsek/trojúhelník

Kapitola 5

Závěr

5.1 Cíle diplomové práce

Cílem mé diplomové práce bylo naimplementovat knihovnu pro vrhání paprsků, která bude používat hierarchii obálek jako akcelerační strukturu a SSE4 a AVX instrukce pro syntézu statických a dynamických scén. Kvůli tomu jsem provedl analýzu metod sloužících k syntéze obrazu pomocí vrhání paprsků a možnosti využití SIMD instrukcí k urychlení těchto metod. Na základě provedené analýzy jsem stanovil požadavky na knihovnu a případy užití. Dle těchto požadavků jsem naimplementoval jak knihovnu, tak testovací aplikaci, která demonstruje její možnosti. Knihovna byla otestována na třech statických a třech dynamických scénách. Z testování vyplynulo, že za použítí SSE4 instrukcí jsme schopni na testovací sestavě dosáhnout průměrně 2,5x zrychlení traverzace vůči standardní sekvenční implementaci. Použití AVX instrukcí, které poskytují dvakrát větší registry, než SSE4 instrukce, se dostaneme až na 3,4x násobné zrychlení traverzace vůči standardní sekvenční implementaci. Knihovna také demonstruje použití SIMD instrukcí při sledování paprsků a jejich traverzaci v hierarchii obálek. U dynamických scén jsem se při testování hlavně zaměřil na porovnání rychlosti vykreslení celé animace při stavbě nové hierarchie pro každý snímek, a při pouhé aktualizaci hierarchie z prvního snímku. Z tohoto testování vyplynulo, že metoda aktualizace hierarchie podává na všech třech testovaných scénách lepší výsledky, co do rychlosti vykreslení celé animace, než metoda stavby nové hierarchie pro každý snímek.

Hlavní přínos vzniklé knihovny vidím v možnosti syntézy jak statických, tak dynamických scén metodou sledování paprsku, nebo sledování cesty. Knihovna navíc dokáže využívat moderní instrukční sady SSE4 a AVX a vrhat paprsky na více jádrech procesoru.

5.2 Možná rozšíření

Co se týká možných rozšíření knihovny, kterými by bylo možné knihovnu posunout dál, napadají mě následující vylepšení:

- Sofistikovanější způsob přestavby hierarchie obálek - aktuálně se v knihovně používá jeden z nejjednodušších způsobů přestavby hierarchie obálek pro dynamické scény (ačkoliv je plně dostatečný, jak bylo dokázáno v části věnované testování). Existují sofistikovanější způsoby přestavby například pomocí rotací jednotlivých částí stromu [7].

- Optimalizace [2] již postavené hierarchie obálek, které vede k urychlení traverzace.
- Traverzace větších svazků paprsků - knihovna aktuálně podporuje pouze traverzaci svazků o velikosti 2x2 pro SSE4 a 4x2 pro AVX. Je možné používat i daleko větší svazky, než jsou v současné době implementovány. Pro traverzaci větších svazků slouží algoritmy popsané v samostatné kapitole o hierarchii obálek.
- Rozšíření o další akcelerační struktury - knihovna je velice snadno rozšiřitelná o další akcelerační struktury, takže není problém doplnit například strukturu kD-stromu, který je také velice používaný jako akcelerační struktura pro vrhání paprsků.

Literatura

[1] J. Arvo and D. Kirk. Particle transport and image synthesis. *SIGGRAPH Comput. Graph.*, 24(4):63–66, Sept. 1990.

[2] J. Bittner, M. Hapala, and V. Havran. Fast insertion-based optimization of bounding volume hierarchies. *Comput. Graph. Forum*, 32(1):85–100, 2013.

[3] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, Jan. 1984.

[4] A. T. Áfra. Improving BVH Ray Tracing Speed Using the AVX Instruction Set. pages 27–28.

[5] J. Havel and A. Herout. Yet faster ray-triangle intersection (using sse4). *IEEE Transactions on Visualization and Computer Graphics*, 2010(3):434–438, 2010.

[6] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278, Aug. 1986.

[7] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler. Fast, effective bvh updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 197–204, 2012.

[8] T. Möller and B. Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, Oct. 1997.

[9] R. Overbeck, R. Ramamoorthi, and W. Mark. Large ray packets for real-time whitted ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 41–48, 2008.

[10] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[11] J. Žára, B. Beneš, and P. Felkel. *Moderní počítačová grafika.* Computer Press s.r.o, Brno, 1st edition, 1998. In Czech.

[12] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[13] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), Jan. 2007.

[14] The utah 3d animation repository.
http://www.sci.utah.edu/~wald/animrep/, stav z 01. 05. 2014.

[15] Knihovna embree.
http://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels, stav z 01. 05. 2014.

[16] Knihovna kixor pro načítání obj souborů.
http://www.kixor.net/dev/objloader/, stav z 01. 05. 2014.

[17] Mcguire graphic data.
http://graphics.cs.williams.edu/data/meshes.xml, stav z 01. 05. 2014.

[18] Minimax.
http://dcgi.felk.cvut.cz/home/bittner/minimax/, stav z 01. 05. 2014.

[19] Mitsuba rendered.
http://www.mitsuba-renderer.org/, stav z 01. 05. 2014.

[20] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

Příloha A

Porovnání standardní a SSE implementace

```
/*
* PŘEDPOČÍTANÉ HODNOTY PRO KAŽDÝ TROJÚHELNÍK
*/
int matIndex = 0;
rt::Vertex normal = Vertex::CrossProd(v2-v1, v3-v1);
float nx = normal.x;
float d = Vertex::DotProd(v1, normal);
rt::Vertex N1 = Vertex::CrossProd((v3-v1), normal)/(Vertex::Length(normal)*Vertex::Length(normal));
float d1 = Vertex::DotProd(N1*-1, v1);
rt::Vertex N2 = Vertex::CrossProd(normal, (v2-v1))/(Vertex::Length(normal)*Vertex::Length(normal));
float d2 = Vertex::DotProd(N2*-1, v1);

/*
* VLASTNÍ VÝPOČET PRŮSEČÍKU
*/
float det = Vertex::DotProd(ray.direction, normal);
float tCur = d - Vertex::DotProd(ray.origin, normal);
if(tCur<0 && (det*t-tCur)>0)
    return false;
else if(tCur>0 && (det*t-tCur)<0)
    return false;

Vertex P = ray.origin*det + ray.direction*tCur;
float u = Vertex::DotProd(P, N1)+det*d1;
if(u<0 && (det-u)>0)
    return false;
else if(u>0 && (det-u)<0)
    return false;

float v = Vertex::DotProd(P, N2)+det*d2;
if(v<0 && (det-u-v)>0)
    return false;
else if(v>0 && (det-u-v)<0)
    return false;

t = tCur/det;
return true;
```

Obrázek A.1: Standardní implementace Havel/Heroutova algoritmu

```
/*
* PŘEDPOČÍTANÉ HODNOTY PRO KAŽDÝ TROJÚHELNÍK
*/
rt::Vertex normal = Vertex::CrossProd(v2-v1, v3-v1);
rt::Vertex N1 = Vertex::CrossProd((v3-v1), normal)/(Vertex::Length(normal)*Vertex::Length(normal));
rt::Vertex N2 = Vertex::CrossProd(normal, (v2-v1))/(Vertex::Length(normal)*Vertex::Length(normal));
/*
* VLASTNÍ VÝPOČET PRŮSEČÍKU
*/
const float arr[4] = {-1,-1,-1,1};
const __m128 int_coef = _mm_load_ps(arr);
const __m128 o = ray.origin.t[0];
const __m128 d = ray.direction.t[0];
const __m128 n = _mm_set1_ps(this->nx);
const __m128 det = _mm_dp_ps(n, d, 0x7f);
const __m128 dett = _mm_dp_ps(_mm_mul_ps(int_coef, n), o, 0xff);
const __m128 oldt = _mm_set1_ps(*t);
if((_mm_movemask_ps(_mm_xor_ps(dett, _mm_sub_ss(_mm_mul_ss(oldt, det),dett)))&1) == 0)
{
    const __m128 detp = _mm_add_ps(_mm_mul_ps(o, det),_mm_mul_ps(dett, d));
    const __m128 tmpN1 = _mm_set1_ps(N1.x);
    const __m128 detu = _mm_dp_ps(detp, tmpN1, 0xf1);
    if((_mm_movemask_ps(_mm_xor_ps(detu,_mm_sub_ss(det, detu)))&1) == 0)
    {
        const __m128 tmpN2 = _mm_set1_ps(N2.x);
        const __m128 detv = _mm_dp_ps(detp,tmpN2, 0xf1);
        if((_mm_movemask_ps(_mm_xor_ps(detv,_mm_sub_ss(det, _mm_add_ss(detu, detv))))&1) == 0)
        {
            const __m128 ones = _mm_set1_ps(1.0f);
            const __m128 inv_det = _mm_div_ps(ones, det);
            const __m128 curT = _mm_mul_ss(dett,inv_det);
            _mm_store_ss(t, curT);
            return true;
        }
    }
}
return false;
```

Obrázek A.2: SSE implementace Havel/Heroutova algoritmu

Příloha B

Instalační a uživatelská příručka

K běhu aplikace je nutné, aby se soubor `app.exe` a statická knihovna nacházely ve stejném adresáři. Konfigurační soubor musí být uložen ve složce `data`, mít jméno `config.properties` a jeho atributy musejí splňovat definici popsanou v části implementace. Jelikož aplikace využívá některé knihovny z C++ verze 11 a byla zkompilována pomocí Microsoft Visual Studia 2013, je nutné si před spuštěním nainstalovat Visual C++ Redistributable for Visual Studio 2013.

Pro využití SSE4 nebo AVX instrukcí je potřeba se ujistit, že je procesor opravdu podporuje. Pokud ne, aplikace bude ukončena. Pro vykreslení dynamických scén se v aplikaci přiložené na CD používá aktualizace původní hierarchie.

Příloha C

Obsah přiloženého CD

Přiložené CD obsahuje:

- zdrojové kódy knihovny ve složce `src\library`
- zdrojové kódy aplikace ve složce `src\app`
- spustitelnou verzi aplikace pro statické scény ve složce `executable\static`
- spustitelnou verzi aplikace pro dynamické scény ve složce `executable\dynamic`
- text diplomové práce ve složce `thesis`
- instalační manuál ve složce `install`

K demonstraci aplikace je k dispozici jedna statická a jedna dynamická scéna, které se nacházejí ve složce data v patřičných adresářích.