VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY

DEPARTMENT OF MICROELECTRONICS

GENEROVÁNÍ OBRAZU METODOU SLEDOVÁNÍ PAPRSKU

PICTURE GENERATION USING PATH TRACING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **Ondřej Áč**
AUTHOR

VEDOUCÍ PRÁCE **Ing. Michal Pavlík, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Mikroelektronika a technologie**

Ústav mikroelektroniky

Student: Ondřej Áč ***ID:*** 230452

Ročník: 3 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Generování obrazu metodou sledování paprsku

POKYNY PRO VYPRACOVÁNÍ:

Nastudujte problematiku počítačem generovaných obrázků. Popište zobrazovací rovnici a její možné způsoby řešení. Zaměřte se na řešení zobrazovací rovnice za pomocí metody sledování cest založené na integraci metodou Monte Carlo. Uveďte možné optimalizace samotného algoritmu sledování cest spolu s mikro-optimalizacemi specifickými pro architekturu x86-64. Vytvořte počítačový program schopný generovat obrázky za pomocí metody sledování cest, umožňující náhled v reálném čase.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 1.6.2023

Vedoucí práce: Ing. Michal Pavlík, Ph.D.

doc. Ing. Pavel Šteffan, Ph.D.
předseda rady studijního programu

UPOZORNĚNÍ:

Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

Abstrakt

Tato práce se zabývá problematikou počítačem generovaných obrázků prostřednictvím metody sledování cest. Cílem práce je tvorba interaktivního počítačového programu, jenž umožňuje upravovat a vykreslovat fotorealistické snímky různých scén v reálném čase. V teoretické části práce je představen koncept zobrazovací rovnice, včetně jejích známých metod řešení. Podrobněji je popsán algoritmus sledování cest, založený na integraci metodou Monte Carlo, spolu s výhodami, které přináší oproti ostatním řešením. Jsou dále prezentovány základní hardwarové i softwarové optimalizace. Praktická část práce je poté zaměřena na rozbor zdrojového kódu v jazyku C++ a zkompilovaného strojového kódu při využití ručních optimalizací SIMD. Nedílnou součástí práce je také demonstrace hlavních funkcí programu, včetně měření výkonnostních přínosů při použití ručních optimalizací.

Klíčová slova

CGI, zobrazovací rovnice, sledování cest, Monte Carlo, C++, SIMD, x86-64

Abstract

This thesis deals with the problematics of computer-generated imagery using path tracing. The goal of this work is to create interactive computer program, which allows editing and rendering of photorealistic images of various scenes in real time. The work presents the concept of rendering equation, along with its known solutions, in the theoretical part of the work. Thesis describes in detail the solution using path tracing, based on the Monte Carlo integration technique, along with the benefits, it provides compared to the other techniques. Several hardware and software optimizations are then presented. Practical part of the work focuses on analysis of C++ source code and compiled assembly code whilst using hardware specific SIMD optimizations. Mandatory part of work is also the demonstration of program's functionality, along with the measurements of achieved performance gains using manual optimizations.

Keywords

CGI, rendering equation, path tracing, Monte Carlo, C++, SIMD, x86-64

Bibliografická citace

ÁČ, Ondřej. Generování obrazu metodou sledování paprsku. Brno, 2023. Dostupné také z: https://www.vut.cz/studenti/zav-prace/detail/152249. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav mikroelektroniky. Vedoucí práce Michal Pavlík.

Prohlášení autora o původnosti díla

Jméno a příjmení studenta:	*Ondřej Áč*
VUT ID studenta:	*230452*
Typ práce:	*Bakalářská práce*
Akademický rok:	*2022/23*
Téma závěrečné práce:	*Generování obrazu metodou sledování paprsku*

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

.

V Brně dne 1. června 2023 ---------------------------------------

podpis autora

Obsah

SEZNAM OBRÁZKŮ ... **8**

SEZNAM TABULEK ... **9**

ÚVOD ... **10**

1. POČÍTAČOVÁ GRAFIKA ... **11**

1.1 Radiometrie ... 11
1.1.1 Definice veličin ... 11
1.1.2 Principiální zákony ... 12
1.2 Zobrazovací rovnice ... 14
1.2.1 Plošná formulace ... 14
1.2.2 Úhelná formulace ... 14
1.2.3 Konvenční metody řešení ... 15
1.3 Metoda sledování cest ... 17
1.3.1 Integrace metodou Monte Carlo ... 17
1.3.2 Řešení zobrazovací rovnice ... 18
1.3.3 Popis algoritmu sledování cest ... 18
1.4 Optimalizace metody sledování cest ... 20
1.4.1 Akcelerační struktury ... 20
1.4.2 Hardwarové optimalizace SIMD ... 23

2. TVORBA PROGRAMU ... **24**

2.1 Prerekvizity ... 24
2.1.1 Programovací model ... 24
2.1.2 Uživatelské rozhrání ... 25
2.2 Jádro grafického programu ... 25
2.2.1 Hlavní smyčka programu ... 26
2.2.2 Zpracování vstupu ... 27
2.2.3 Kontextové nabídky ... 27
2.2.4 Vykreslení obrazu ... 28
2.3 Reprezentace objektů v prostoru ... 29
2.3.1 Vektory ... 29
2.3.2 Matice ... 30
2.4 Geometrické primitivy ... 31
2.4.1 Paprsek ... 31
2.4.2 Ohraničující objem ... 31
2.4.3 Obecná struktura primitivů ... 32
2.4.4 Koule ... 33
2.4.5 Krychle ... 33
2.4.6 Čtyřúhelník ... 33
2.4.1 Trojúhelník ... 34
2.5 Scéna ... 35
2.5.1 Kamera ... 35
2.5.2 Seznam objektů ... 36
2.5.3 Varianty ... 37

2.5.4 Objekty ... *38*
2.5.5 Materiály ... *39*
2.5.6 Renderování ... *40*
2.6 VSTUPNĚ VÝSTUPNÍ SYSTÉM ... 42
2.6.1 Načítání modelů ... *42*
2.6.2 Načítání a ukládání scén ... *43*
2.6.3 Ukládání obrázků ... *44*
2.7 OPTIMALIZACE ... 45
2.7.1 Optimalizace SIMD ... *45*
2.7.2 Akcelerační struktura BVH ... *46*
3. VÝSTUPY PROGRAMU ... 49
3.1 HLAVNÍ FUNKCE PROGRAMU ... 49
3.1.1 Náhled v reálném čase ... *49*
3.1.2 Načítání a ukládání dat ... *49*
3.1.3 Možnosti úprav ... *50*
3.1.4 Optimalizace kódu ... *50*
3.2 UŽIVATELSKÉ PROSTŘEDÍ PROGRAMU ... 52
3.2.1 Rozložení okna ... *52*
3.2.2 Ovládání ... *53*
3.3 GENEROVANÉ OBRÁZKY ... 53
3.4 MĚŘENÍ VÝKONU ... 54
3.4.1 Měření výkonu s použitím optimalizací SIMD ... *54*
3.4.2 Měření výkonu s použitím struktury BVH ... *55*
4. ZÁVĚR ... 57
LITERATURA ... 58
SEZNAM SYMBOLŮ A ZKRATEK ... 60
SEZNAM PŘÍLOH ... 62

SEZNAM OBRÁZKŮ

1.1 Grafické znázornění Lambertova zákona 13
1.2 Grafické znázornění zákona inverzní druhé mocniny 13
1.3 Vizualizace členů Neumannovy řady, inspirováno [5] a [10] 19
1.4 Vizualizace datové struktury Octree ve 2D, upraveno z [12] 21
1.5 Dělení uzlu BVH pomocí prostorového a objektového mediánu [12] 22
2.1 Základní hierarchie programu 24
3.1 Porovnání kvality obrazu pro implicitní a explicitní vzorkování 51
3.2 Rozložení okna programu 52
3.3 Fyzikální vlastnosti obrázku vygenerovaného programem 53
3.4 Graf snímkových frekvencí s použitím optimalizací SIMD 54
3.5 Graf závislosti času vykreslení snímků na počtu primitivů 55
3.6 Graf závislosti poměru rychlosti vykreslování na počtu primitivů 56

SEZNAM TABULEK

3.1 Seznam ovládání programu 53
3.2 Naměřená data snímkových frekvencí s použitím optimalizací SIMD 54
3.3 Naměřená data času vykreslení snímků s použitím struktury BVH 55

ÚVOD

Technika CGI (*Computer Generated Imagery*) pro generování počítačových obrázků se začala využívat již na počátku 70. let 20. století. Od té doby došlo k významným pokrokům jak v oblasti vykreslovacích algoritmů, tak i ve výkonu výpočetního hardwaru. To umožnilo rozšíření této technologie do mnoha oblastí počítačové grafiky, zejména pak do oblasti real-time renderování.

V této oblasti se již přes 20 let používá technika rasterizace, která nabízí velmi rychlý způsob testování viditelnosti, avšak znemožňuje přesné výpočty osvětlení. Pro eliminaci tohoto omezení se v posledních 5 letech rozšířila aplikace tzv. hybridního renderování, jehož cílem je použití metody sledování paprsků (*ray tracing*) pro nahrazení nepřesných grafických efektů pracujících v oblasti obrazovky (*screen space*).

V současnosti však dochází i k nahrazování této metody na úkor generování obrázků výhradně prostřednictvím metody sledování cest (*path tracing*). Ta rozšiřuje metodu sledování paprsků o probabilistické jevy, čímž umožňuje schopnost nalezení objektivního řešení tzv. zobrazovací rovnice (*rendering equation*), jež slouží k matematickému popisu propagace světelné energie v prostoru.

Cílem této práce je tedy seznámit čtenáře se základními principy, na nichž je metoda sledování cest postavena, včetně praktické implementace tohoto algoritmu v počítačovém programu, jenž umožňuje vykreslování fotorealistických snímků, spolu s dynamickými úpravami různých scén v reálném čase.

V teoretické části práce je nejdříve popsán fyzikální obor radiometrie, na němž je založena podstata zobrazovací rovnice. Jsou představeny některé konvenční metody řešení rovnice, přičemž podrobněji je poté popsán algoritmus sledování cest, založený na integraci metodou Monte Carlo, spolu s výhodami, které přináší oproti ostatním řešením. Jsou dále uvedeny optimalizace algoritmu v podobě prostorových akceleračních struktur i v podobě hardwarových optimalizací SIMD (*Single Instruction Multiple Data*).

V praktické části práce je popsána základní hierarchie programu, včetně její praktické implementace v podobě objektově orientovaného kódu v jazyku C++. Je poté proveden podrobný rozbor vybraných třídních metod i funkcí za účelem vytvoření jasné představy o interním fungování celého programu. Na závěr jsou popsány optimalizace s využitím instrukční sady SSE (*Streaming SIMD Extensions*) k akceleraci vektorových výpočtů i optimalizace s využitím struktury BVH (*Bounding Volume Hierarchy*) k akceleraci algoritmu pro nalezení nejbližšího bodu průniku paprsku a scény.

Na závěr jsou popsány nejdůležitější funkce a výstupy programu, např. ovládání a podoba náhledového okna s uživatelským rozhráním, možnosti úprav tvořených scén, ukládání a načítání modelů, textur a textových souborů scén z pevného disku, anebo ukládání a popis vlastností generovaných obrázků. Nakonec jsou zde uvedeny i výsledky měření dosažených výkonnostních zisků v rychlosti renderování programu při použití optimalizací SIMD i akcelerační struktury BVH.

1. POČÍTAČOVÁ GRAFIKA

Počítačová grafika je jedním z mnoha oborů výpočetní techniky. Jejím hlavní cílem je vizualizace dat pomocí počítače, ať už v abstraktním uměleckém slova smyslu, nebo přímo konkrétních deterministických dat, např. primitivů, objektů a scén.

Tato práce se zabývá oborem trojrozměrné počítačové grafiky, konkrétně pak oblastí renderování, kde je cílem převedení trojrozměrné scény ve dvourozměrný obraz, přičemž jsou obvykle simulovány fyzikální principy přenosu světla v reálném světě. Podoby renderování mohou být jak stylistické (kresby), tak i realistické (syntézy). Při stylistickém zaměření je snahou vytvořit co nejvíce vizuálně přívětivý obraz, avšak ne vždy zcela realistický. A naopak při realistickém zaměření je snahou co nejlépe napodobit realitu řadou vizuálně přínosných fyzikálních fenoménů.

1.1 Radiometrie

Radiometrie je odvětví optiky zabývající se měřením elektromagnetického záření v prostoru a času. Zkoumá energii celého spektra elektromagnetického záření pomocí objektivních veličin, na rozdíl od fotometrie, jež zkoumá pouze viditelnou část záření a je vztažena na spektrální citlivost lidského oka [1]. Zobrazovací rovnice je však založena na radiometrii a nikoliv fotometrii, ačkoliv je uvažováno pouze viditelné spektrum, a to zejména kvůli objektivitě výsledků.

1.1.1 Definice veličin

Tato kapitola slouží k definici základních vztahů a veličin z oboru radiometrie, na jejichž principech zakládá i zobrazovací rovnice.

Zářivá energie

Zářivou energií se rozumí množství energie vyzářené zdrojem záření do svého okolí. Pro zdroj elektromagnetického záření je energie rovna:

$$Q_e = h \cdot f \; [J] \, , \tag{1.1}$$

kde $h = 6{,}626 \cdot 10^{-34} \, J \cdot s$ je Planckova konstanta a $f \; [Hz]$ je frekvence záření.

Zářivý tok

Vyjadřuje množství energie procházející určitou plochou za jednotku času [1]:

$$\phi_e = \frac{dQ_e}{dt} \; [W] \, , \tag{1.2}$$

kde $Q_e \; [J]$ je zářivá energie a $t \; [s]$ je čas.

Zářivost (hustota zářivého toku)

Vyjadřuje množství zářivého toku v jednotce prostorového úhlu [1]:

$$I_e = \frac{d\phi_e}{d\Omega} \; [W \cdot sr^{-1}] \, , \tag{1.3}$$

kde ϕ_e $[W]$ je zářivý tok a $d\Omega$ [sr] je element prostorového úhlu vyjadřující velikost úhlu, jenž je vytnutý průnikem kužele s jednotkovou sférickou plochou. Prostorový úhel lze vypočíst ze vztahu [2]:

$$\Omega = \frac{A}{r^2}\,[sr]\,, \tag{1.4}$$

kde A $[m^2]$ je plocha vytnutá kuželem na sférické ploše a r $[m]$ je poloměr koule.

Plošná zářivost

Je určena podílem zářivosti elementární plošky zdroje ve zvoleném směru a kolmého průmětu plošky v tomto směru [2]:

$$L_e = \frac{dI_e}{dS \cdot \cos\theta} = \frac{1}{\cos\theta}\frac{d^2\phi_e}{dS\, d\Omega}\ [W \cdot m^{-2} \cdot sr^{-1}]\,, \tag{1.5}$$

kde I_e $[W \cdot sr^{-1}]$ je zářivost, dS $[m^2]$ je plocha a $\cos\theta$ je úhel incidence.

Intenzita ozáření a vyzařování

Intenzita ozáření vyjadřuje množství zářivého toku dopadající na elementární plošku v prostoru [1]:

$$E_e = \frac{d\phi_e}{dA} = \frac{I_e}{r^2} \cdot \cos\theta\ [W \cdot m^{-2}]\,, \tag{1.6}$$

kde ϕ_e $[W]$ je zářivý tok, dA $[m^2]$ je plocha, I_e $[W \cdot sr^{-1}]$ je zářivost, $r[m]$ je vzdálenost a θ je úhel dopadu záření. Intenzita tedy klesá se čtvercem vzdálenosti od zdroje záření a se zvyšujícím se úhlem dopadu.

Intenzita vyzařování vyjadřuje množství zářivého toku vyzářeného z elementární plošky dS do poloprostoru [1]:

$$M_e = \frac{d\phi_e}{dS} = \frac{I_e}{r^2} \cdot \cos\theta\ [W \cdot m^{-2}]\,, \tag{1.7}$$

kde ϕ_e $[W]$ je zářivý tok a dS $[m^2]$ je obsah elementární plošky.

1.1.2 Principiální zákony

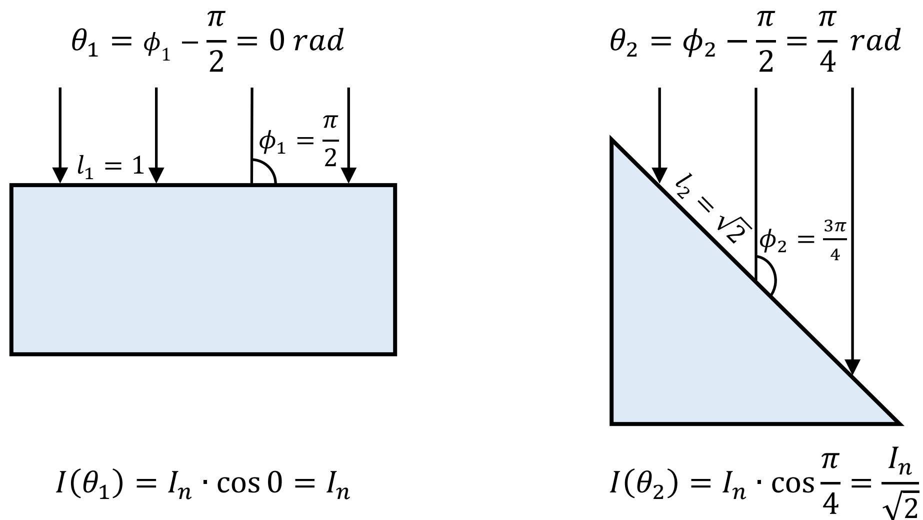
V této kapitole jsou popsány principiální zákony, které se vztahují ke členům $\cos\theta$ a $\frac{1}{r^2}$ z předchozí kapitoly. Na těchto zákonech je poté založena i podstata zobrazovací rovnice.

Lambertův kosinový zákon

Je možno se setkat se dvěma obdobnými formulacemi pocházejících ze dvou různých pohledů. První formulace říká, že jas pozorované plochy difuzního materiálu se s úhlem pohledu nemění. Druhá formulace říká, že zářivost dopadající na plochu S pozorovaného objektu je úměrná vstupní zářivosti I_n $[W \cdot sr^{-1}]$ a kosinu úhlu θ $[rad]$, který svírá dopadající záření s normálou plochy, nebo také sinu úhlu ϕ $[rad]$, jenž svírá dopadající záření s vektorem rovnoběžným s plochou [2]:

$$I(\theta) = I_n \cdot \cos\theta = I_n \cdot \cos\left(\phi - \frac{\pi}{2}\right) = I_n \cdot \sin\phi\ [W \cdot sr^{-1}]\,. \tag{1.8}$$

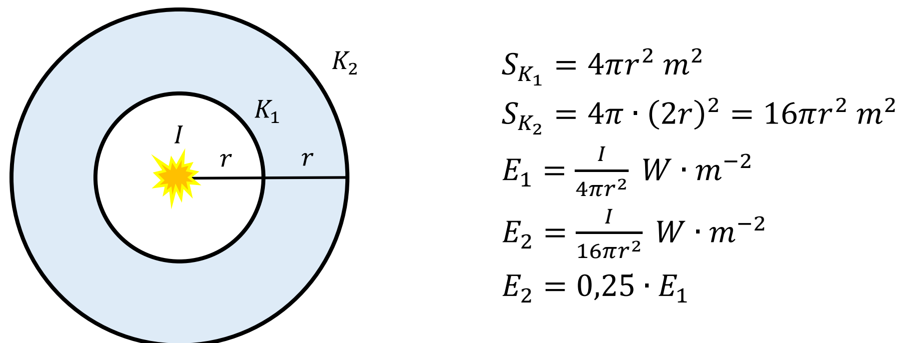
Graficky si lze situaci představit jako měnící se vzdálenost dopadu l ekvidistantně rozmístěných paprsků na plochu pod různými úhly ϕ.



Obrázek 1.1 Grafické znázornění Lambertova zákona

Zákon inverzní druhé mocniny

Tento zákon říká, že intenzita ozáření klesá s druhou mocninou vzdálenosti plošky od zdroje záření, nebo také že intenzita vyzařování klesá s druhou mocninou vzdálenosti od zdroje záření. Tato formulace byla již uvedena ve vztazích (1.6) a (1.7). Graficky si lze problém představit jako rozložení energie na dvě kulovité plochy s různým poloměrem.



Obrázek 1.2 Grafické znázornění zákona inverzní druhé mocniny

Shrnutí

Spolu tyto dva zákony tvoří základ pro veškeré potřebné výpočty v oblasti zobrazovací rovnice, kde je snahou vypočíst intenzitu osvětlení v kterémkoliv bodě scény. Nachází pak konkrétně využití při výpočtech intenzity odraženého záření i při výpočtech vážené pravděpodobnosti při explicitním vzorkování světelných zdrojů.

1.2 Zobrazovací rovnice

Zobrazovací rovnice slouží k matematickému popisu propagace světelné energie v prostředí. Vychází výhradně z fyzikálních oborů geometrické optiky a radiometrie. V této kapitole jsou popsány dvě formulace rovnice lišící se integrační doménou.

1.2.1 Plošná formulace

Zobrazovací rovnice byla poprvé představena v roce 1986 ve stejnojmenné publikaci „*The rendering equation*“ napsané Jamesem Kajiyou. Slouží k porovnání různých vykreslovacích algoritmů pomocí jedné unifikované rovnice, konkrétně pak k vyjádření přesnosti aproximací jejího řešení za pomocí těchto algoritmů. Její formální zápis zní [3]:

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'')\, dx'' \right], \tag{1.9}$$

kde:

$I(x, x')$ je celková intenzita světla procházejícího z bodu x' do bodu x,
$g(x, x')$ je geometrický člen (popisuje viditelnost plochy a vlastnosti prostředí),
$\epsilon(x, x')$ je intenzita světla vyzářeného z x' do x (vlastní emise),
$\rho(x, x', x'')$ je intenzita rozptýleného (odraženého) světla z x'' do x skrz bod x'.

Zobrazovací rovnice tedy říká, že celkové množství světla přeneseného do bodu x je dáno jako součet emitovaného a odraženého světla z bodu x'. Množství odraženého světla závisí na přenosové funkci v bodě x'a součtu jednotlivých příspěvků světla ze všech bodů x'' umístěných na všech uvažovaných plochách S [3]. Jedná se tudíž o zápis s integrací v rámci jednotlivých ploch.

1.2.2 Úhelná formulace

V současné literatuře se lze často setkat s alternativní formou zápisu vycházející z formy zobrazovací rovnice popsané Immelem v r. 1986. Zde dochází k integraci všech ploch promítnutých na jednotkovou hemisféru polohovanou nad bodem x' u nějž pozorujeme radianci ve směru ω_o [4]. Její formulace zní [5]:

$$\begin{aligned} L_o(x, \omega_o) &= L_e(x, \omega_o) + L_r(x, \omega_o) \\ L_o(x, \omega_o) &= L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) \cdot f_r(x, \omega_i \to \omega_o) \cdot \cos\theta_i \; d\omega_i \,, \end{aligned} \tag{1.10}$$

kde:

$L_o(x, \omega_o)$ je celková výstupní zář (radiance) z bodu x ve směru ω_o,
$L_e(x, \omega_o)$ je emitovaná zář z bodu x ve směru ω_o,
$L_r(x, \omega_o)$ je odražená zář z bodu x ve směru ω_o,
$L_i(x, \omega_i)$ je příchozí zář ze směru ω_i do bodu x,
$f_r(x, \omega_i \to \omega_o)$ je BRDF (*Bidirectional Reflectance Distribution Function*) v bodě x,
$\cos\theta_i$ je kosinus úhlu mezi normálou plochy a příchozím směrem ω_i.

Tato formulace tedy říká, že výstupní zář ve směru ω_o z bodu x je rovna součtu emitované záře a odražené záře v bodě x. Odražená zář je úměrná součtu všech příspěvků záře ze všech příchozích směrů ω_i na hemisféře Ω, jež je umístěná nad bodem x. Vstupní zář L_i v jediném směru ω_i lze poté určit řešením zobrazovací rovnice pro bod, ze kterého zář pochází. Množství odražené energie a směr odrazu pak určuje BRDF materiálu.

1.2.3 Konvenční metody řešení

Z uvedených definic v kapitolách 1.2.1 a 1.2.2 plyne, že je prakticky nemožné řešit tuto rovnici analyticky, neboť pro znalost vstupní záře v jediném bodě $L_i(x, \Omega)$ je nutno integrovat nekonečné množství bodů, u nichž je také potřeba zjistit vstupní zář integrací nekonečna bodů. Je tedy zřejmé, že se jedná o nekonečně rekurzivní algoritmus, který nelze řešit analyticky, řešení zobrazovací rovnice je tudíž nutno aproximovat numericky.

Během posledních 40 let došlo k velkému pokroku v oblasti počítačové grafiky, a to především díky pokrokům ve výzkumu vykreslovacích a LT (*Light Transport*) algoritmů, které doprovázel rychlý vzrůst hrubého výpočetního výkonu hardwaru.

Výsledkem této snahy je nemalá řada vykreslovacích algoritmů, které řeší problém globální iluminace, a tudíž i zobrazovací rovnice, různými způsoby s rozlišnou přesností. Pro účely této práce stačí jmenovat nejznámější zástupce, tak aby je bylo možno porovnat s metodou sledování cest, kterou se tato práce primárně zabývá.

Je rovněž nutno podotknout, že níže uvedené algoritmy řeší primárně problém viditelnosti, zdali je konkrétní bod viditelný z určitého úhlu pohledu a zdali není například okludován nějakým bližším objektem [6]. Implementace výpočtů o viditelnosti poté dále omezuje, anebo naopak umožňuje různé implementace LT algoritmů [7].

Rasterizace

Princip rasterizace spočívá v dekompozici objektů scény do trojúhelníků (*polygons*) nebo obecných mnohoúhelníků, jejichž vrcholové body (*vertices*) jsou následně za pomocí projekční matice, která reprezentuje kameru, transformovány na dvourozměrnou plochu reprezentující obrazovku. Pomocí rasterizačního algoritmu jsou poté interpolací vyplněny pixely ohraničené těmito body. Každý pixel pak uchovává hodnotu hloubky (*Z-Buffer*), která určuje nejbližší vzdálenost průniku, podle které dochází k přepsání pixelu pouze v případě, že je nová hodnota vzdálenosti menší než ta předešlá [6].

Jedná se tedy o velmi efektivní algoritmus, neboť nemusí testovat každý pixel oproti všem objektům. Zároveň dokáže eliminovat většinu polygonů již na začátku renderování, neboť se nenachází v oblasti obrazovky, a taktéž dokáže interpolovat mezi vrcholovými body polygonů, aniž by musel počítat náročné výpočty geometrických průniků [6].

Tento algoritmus má však své nevýhody. Jednou z nich je ztráta důležitých informací o rozložení trojrozměrné scény při transformaci primitivů na dvourozměrnou plochu a při interpolaci mezi jednotlivými body. Algoritmu je známo pouze rozložení scény, tak jak je projektováno na obrazovku, což výrazně komplikuje implementaci efektů pro výpočet osvětlení, stínů i odrazů.

Rasterizace aproximuje zobrazovací rovnici prostřednictvím diskrétní sumy, která počítá celkové osvětlení na základě konečného počtu bodových zdrojů světla. Pro zjištění, zdali je konkrétní bod osvětlen daným zdrojem světla, je potřeba vykreslit scénu znovu z pohledu zdroje a zkontrolovat, jestli je daný bod viditelný dle jeho projektované hloubky. Tato technika tedy provádí výpočet přímého osvětlení (*direct illumination*).

Pro veliké množství světelných zdrojů je nerealistické vykreslovat scénu pro každý z nich v plném rozlišení, proto je mnohdy rozlišení omezeno, nebo je tento krok úplně vynechán, a pro viditelnost jsou použity značně zjednodušené výpočty [7]. To způsobuje řadu problémů, zejména pak únik světla (*light leaking*) a nemožnost vytvářet realistické měkké stíny (*soft shadows*) bez výrazných modifikací algoritmu, neboť renderování z pohledu světel uvažuje pouze binární hodnoty viditelnosti.

Mimo přímého osvětlení existuje v rasterizaci i pojem nepřímého osvětlení (*indirect illumination*), jenž má za úkol napodobit realistickou propagaci světla ve scéně, kdy se uvažuje možnost více odrazů světla od objektů.

V tomto ohledu se nejčastěji používají algoritmy pracující pouze v oblasti obrazovky, např. SSAO (*Screen Space Ambient Occlusion*), jenž simuluje lokální okluzi blízkých povrchů, bounce lighting, jenž simuluje více odrazů světla, anebo SSR (*Screen Space Reflections*), jenž napodobuje odrazy od lesklých povrchů.

Metoda sledování paprsků

Metoda sledování paprsků zavádí do renderování koncept sledování cest jednotlivých paprsků ve scéně, jejichž primární účel je určení viditelnosti mezi jednotlivými body všech uvažovaných objektů. Algoritmus využívá k výpočtům osvětlení zpětného postupu, kdy jsou všechny paprsky generovány z kamery, odkud se dále odráží ve scéně, dokud nenarazí na zdroj světla. Tímto je zvýšena efektivita algoritmu, neboť oproti paprskům pocházejících ze světelných zdrojů je garantováno, že všechny narazí na kameru [7].

Výše uvedený odstavec již napovídá, že pro výpočty osvětlení tento algoritmus používá rekurze jednotlivých paprsků, avšak jejich směr odrazu je zcela deterministický, neboť umožňuje pouze perfektní odraz nebo refrakci v případě lesklých a průhledných materiálů. V případě difuzních materiálů umožňuje vypočíst přesné přímé osvětlení pomocí zkonstruování tzv. stínových paprsků ve směru bodového zdroje světla, které mají za účel ověřit, zdali je světelný zdroj viditelný a není okludován ostatními objekty. Dle toho, zda je světelný zdroj okludován, lze poté danému bodu přiřadit určitou hodnotu osvětlení dle vzdálenosti zdroje a úhlu odraženého paprsku [7].

Při tomto procesu tedy dochází k jistému způsobu řešení zobrazovací rovnice (řešení geometrické viditelnosti, rekurzivita), avšak doména původního integrálů je tak značně omezena, neboť dochází k integraci vždy pouze v jediném směru.

V současné době tento algoritmus nachází využití v kombinaci s rasterizací v tzv. hybridním renderování, kde je základní obraz vykreslen pomocí rasterizace a následně doplněn o přesnější grafické efekty s využitím metody sledování paprsků, které primárně nahrazuje nepřesné SS algoritmy výpočtů osvětlení, stínů, odrazů, ambientní okluze aj.

1.3 Metoda sledování cest

Metoda sledování cest je založena na obdobných principech jako metoda sledování paprsků. Opět dochází k rekurzivnímu šíření paprsků ve scéně počínaje od kamery, avšak jejich směr již není deterministický, ale zcela náhodný dle určitého rozložení hustoty pravděpodobnosti na jednotkové hemisféře.

Tento postup umožňuje zcela korektní řešení zobrazovací rovnice za pomocí metody integrace Monte Carlo, ve které dochází k postupné akumulaci a zprůměrování vzorků záření v čase. Princip metody je popsán v následující kapitole.

1.3.1 Integrace metodou Monte Carlo

Monte Carlo je soubor numerických metod, které využívají náhodného vzorkování k aproximaci výsledků funkcí. V tomto kontextu je metoda aplikována k numerické integraci multidimenzionálního integrálu zobrazovací rovnice.

Jednoduchým příkladem metody je poté integrace jednorozměrné funkce $f(x)$ na intervalu (a, b), kde je výsledek integrálu aproximován za pomocí zprůměrování N vzorků funkce, za předpokladu N vstupních náhodných hodnot s uniformní distribucí na tomto intervalu. Tzv. estimátor integrálu má poté následující podobu [8]:

$$F = \int_a^b f(x)\, dx = \frac{1}{N-1} \sum_{i=0}^{N} \frac{f(X_i)}{p(X_i)} = \frac{b-a}{N-1} \sum_{i=0}^{N} f(X_i)\,, \tag{1.11}$$

kde:

F je výsledná hodnota integrálu,
(a, b) je integrační interval,
$f(x)$ je integrovaná funkce,
$p(x)$ je funkce hustoty pravděpodobnosti funkce $f(x)$,
X_i je náhodná proměnná v intervalu (a, b),
$f(X_i)$ je hodnota funkce f v bodě X_i,
$p(X_i)$ je hodnota funkce hustoty pravděpodobnosti p v bodě X_i.

V příkladu je uvažováno, že distribuce hodnot náhodné proměnné $D(X_i)$ je uniformní na intervalu (a, b), a tudíž hodnota hustoty pravděpodobnosti je konstantní $p(X_i) = \frac{1}{b-a}$.

Směrodatná odchylka estimátoru je definována následujícím vztahem [8]:

$$\sigma = [\langle F^N \rangle] \propto \frac{1}{\sqrt{N}}\,, \tag{1.12}$$

chyba je tudíž nepřímo úměrná odmocnině počtu vzorků. Při nekonečném množství vzorků se pak limitně blíží k nule [8]. Chybu lze dodatečně redukovat za pomocí metody důležitostního vzorkování (*importance sampling*). Tato metoda umožňuje analyticky odvodit distribuci vzorků $D(X_i)$, spolu s hustotou pravděpodobnosti $p(X_i)$, na základě funkčních hodnot integrované funkce v různých intervalech.

1.3.2 Řešení zobrazovací rovnice

Dosazením integrálu zobrazovací rovnice (1.10) do vztahu (1.11) lze získat obecné řešení estimátoru pro odraženou zář $L_r(x, \omega_o)$ ve formě:

$$L_r(x, \omega_o) = \frac{1}{N-1} \sum_{i=0}^{N} \frac{f_r(x, \omega_i \to \omega_o) \cdot L_i(x, \omega_i) \cdot cos\, \theta_i}{p(\omega_i)} . \quad (1.13)$$

Při aplikaci metody Monte Carlo na integrál zobrazovací rovnice lze generovat náhodné vzorky s uniformní distribucí v jednotkové hemisféře, hustota pravděpodobnosti má tedy konstantní hodnotu $p(\omega_i) = \frac{1}{2\pi}$. V případě difuzního materiálu je hodnota BRDF rovněž konstantní $f_r(x, \omega_i \to \omega_o) = \frac{c}{\pi}$, kde c je reflektivita (barva) materiálu. Dosazením členů do vztahu (1.13) lze dojít k následnému řešení estimátoru:

$$L_r(x, \omega_o) = \frac{1}{N-1} \sum_{i=0}^{N} \frac{\frac{c}{\pi}}{\frac{1}{2\pi}} L_i(x, \omega_i) \cdot cos\, \theta_i = \frac{2 \cdot c}{N-1} \sum_{i=0}^{N} L_i(x, \omega_i) \cdot cos\, \theta_i . \quad (1.14)$$

Za účelem snížení variance estimátoru se používá metoda důležitostního vzorkování, ve které je rovnoměrná distribuce vzorků nahrazena vhodnější distribucí, jež s vyšší pravděpodobností vzorkuje ty části BRDF, při kterých nabývá nejvyšších hodnot [8].

Pro difuzní materiály lze například využít Kosinovu váženou distribuci, jež s vyšší pravděpodobností vzorkuje horní část hemisféry a jejíž funkce hustoty pravděpodobnosti je $p(\omega_i) = \frac{cos\, \theta_i}{\pi}$. Po dosazení do vztahu (1.13) je řešení estimátoru následovné:

$$L_r(x, \omega_o) = \frac{1}{N-1} \sum_{i=0}^{N} \frac{\frac{c}{\pi}}{\frac{cos\, \theta_i}{\pi}} L_i(x, \omega_i) \cdot cos\, \theta_i = \frac{c}{N-1} \sum_{i=0}^{N} L_i(x, \omega_i) , \quad (1.15)$$

výpočet byl tudíž zjednodušen na aritmetický průměr všech vzorků příchozího záření vynásobený materiálovou konstantou c. Podobný princip pak lze aplikovat i na ostatní modely BRDF, např. vzorkování VNDF (*Visible Normal Distribution Function*) mikro-fazetového modelu GGX [9].

1.3.3 Popis algoritmu sledování cest

Zobrazovací rovnici (1.10) lze zkráceně vyjádřit v operátorové formě. Všechny členy jsou lineární, a lze proto použít ekvivalentních úprav rovnic pro nalezení řešení $\boldsymbol{S}$ [10]:

$$\begin{gathered} L = L_e + \boldsymbol{T}L \\ (\boldsymbol{I} - \boldsymbol{T})L = L_e \\ L = (\boldsymbol{I} - \boldsymbol{T})^{-1} L_e \\ L = \boldsymbol{S} L_e \ , \end{gathered} \quad (1.16)$$

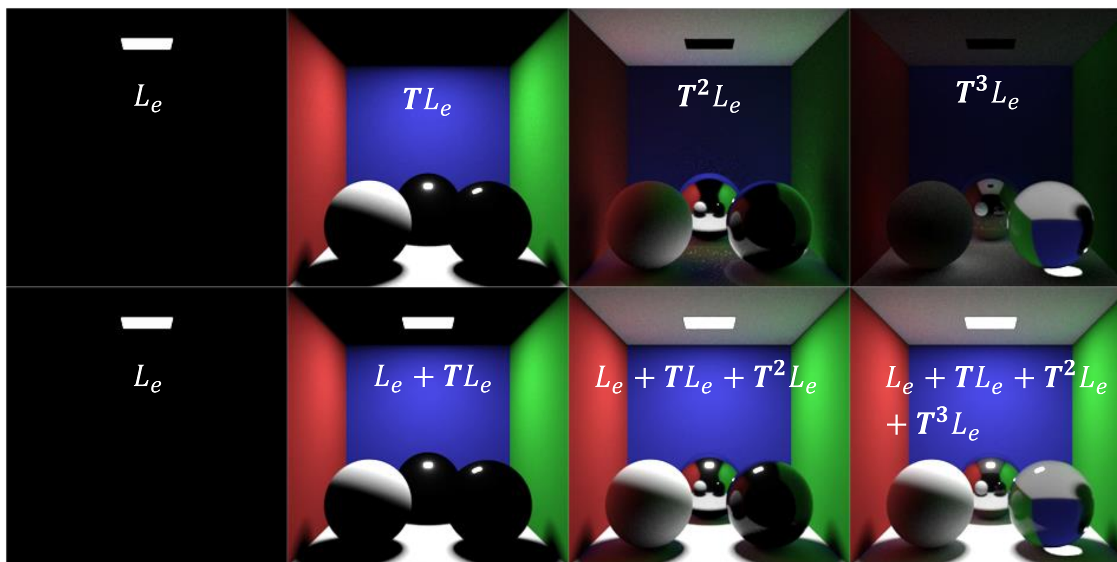
kde L označuje celkovou zář, L_e označuje emitovanou zář, $\boldsymbol{T}$ je transportní operátor, člen $\boldsymbol{I}$ označuje identitu a člen $\boldsymbol{S} = (\boldsymbol{I} - \boldsymbol{T})^{-1}$ lze poté označit za operátor řešení.

Výsledek operátoru řešení $\boldsymbol{S}$ lze získat aplikací Neumannovy řady. Jedná se o iterační metodu určenou k aproximaci řešení inverzních matic. Vztah (1.16) lze tedy vyjádřit [10]:

$$L = \boldsymbol{S}L_e = (\boldsymbol{I} - \boldsymbol{T})^{-1}L_e = \sum_{k=0}^{\infty} \boldsymbol{T}^k L_e = L_e + \boldsymbol{T}L_e + \boldsymbol{T}^2 L_e + \boldsymbol{T}^3 L_e \dots, \tag{1.17}$$

kde platí $|T^k| \leq 1$ pro $k \geq 1$ a pro modely BRDF dodržující zákon o zachování energie.

Z praktického hlediska každý člen $\boldsymbol{T}^k L_e$ vyjadřuje jeden krok odrazu nebo propagace světla ve scéně [5]. Na následujícím obrázku lze pozorovat vliv jednotlivých dílčích členů $\boldsymbol{T}^k L_e$ na celkové řešení zobrazovací rovnice $\sum_{k=0}^{\infty} \boldsymbol{T}^k L_e$. Lze vidět, že každý následující člen přispívá k celkovému řešení s menší vahou než člen předešlý, jinými slovy množství odražené energie je vždy menší nebo rovno množství příchozí energie.



Obrázek 1.3 Vizualizace členů Neumannovy řady, inspirováno [5] a [10]

Iterační formulaci zobrazovací rovnice (1.17) lze prakticky implementovat za pomocí rekurzivního algoritmu, jenž vyhodnocuje integrál zobrazovací rovnice při každém kroku odrazu prostřednictvím metody Monte Carlo. Hlavní podstata algoritmu je zachycena v následujícím pseudokódu.

```
Path_Trace(ray, depth) {
 if (depth > max_depth) return 0;          //Limit počtu odrazů
 intersection = scene.intersection(ray);   //Geometrický průnik
 if (!intersection.any_hit) return ambient; //Okolní osvětlení
 material = intersection material;         //Materiál objektu
 new_ray = material.BRDF.generate(ray, intersection);     //Odražený paprsek
 fr = material.BRDF.evaluate(ray, new_ray, intersection); //BRDF
 p_wi = material.BRDF.p_wi(new_ray, intersection);        //Pravděpodobnost
 cos_theta = dot(intersection.normal, new_ray.direction); //Úhel dopadu
 Le = material.emission(ray, intersection); //Emise
 Li = Path_Trace(new_ray, depth + 1);      //Příchozí zář
 return Le + Li * fr * cos_theta / p_wi;   //Emise + Odraz
}
```

První podmínka v programu slouží k omezení maximálního počtu odrazů, tak aby nemohlo dojít k zacyklení. Dále je nalezen nejbližší průnik paprsku s geometrií scény, kdy v případě nenalezení žádného průniku je vráceno okolní osvětlení. V druhém případě je vygenerován nový paprsek počínající z průsečíku x se směrem určeným dle rozložení distribuce BRDF. Rovněž je vyhodnocena odrazivost $f_r(x, \omega_i \rightarrow \omega_o)$ na základě směrů příchozího a odchozího paprsku, normály povrchu a odrazivosti materiálu, včetně hustoty pravděpodobnosti $p(\omega_i)$ pro směr odraženého paprsku. Dále jsou vyhodnoceny členy $\cos\theta_i$ a $L_e(x, \omega_o)$.

Hodnota příchozí záře $L_i(x, \omega_i)$ je následně získána rekurzivním voláním metody `Path_Trace()` s argumenty v podobě odraženého paprsku a inkrementované hloubky. Funkce tak vrací numerický výsledek ve formě (1.13), přičemž substitucí rekurzivních volání funkce `Path_Trace()` za L_i lze zpětně vyvodit vztah (1.17), pro jehož členy nyní platí: $\boldsymbol{T} = f_r(x, \omega_i \rightarrow \omega_o) \cdot cos\,\theta_i$ a $L_e = L_e(x, \omega_o)$.

1.4 Optimalizace metody sledování cest

Tato kapitola je rozdělena do dvou částí. V první části jsou obsaženy algoritmické optimalizace pro testování průniků paprsků s geometrií v podobě akceleračních struktur. V druhé části jsou poté obsaženy hardwarové optimalizace výpočtů v plovoucí desetinné čárce (*floating point)* za pomocí technik SIMD s využitím instrukční sady SSE specifické pro architekturu x86-64.

1.4.1 Akcelerační struktury

Jedním z problémů metody sledování paprsků je lineární časová komplexita algoritmu pro nalezení nejbližšího bodu průniku paprsku s geometrií - $O(N)$, kde N značí celkový počet primitivů obsažených ve scéně. Pro nalezení tohoto bodu je tedy nutno otestovat průsečíky všech objektů, z nichž je poté vybrán ten nejbližší.

K vyřešení tohoto problému lze aplikovat různé akcelerační struktury. Jedná se o prostorové datové struktury umožňující hierarchické rozdělení objektů scény (nejčastěji v podobě stromu). Procházením těchto struktur lze nalézt informaci o nejbližším bodu průniku paprsku s primitivy obecně v čase $O(\log N)$ [11]. Hlavními zástupci jsou poté: pravidelná mřížka, Octree, Kd-tree, BSP (*Binary Space Partitioning*) anebo BVH [12].

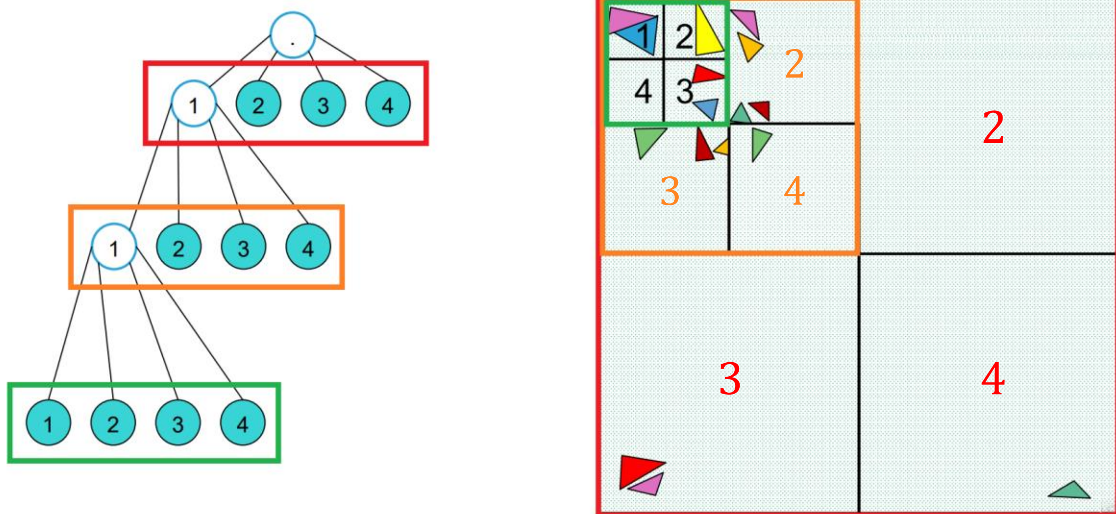
Struktura Octree

Struktura Octree je založena na principu rekurzivního dělení ohraničujících objemů scény do oktantů a následného přiřazení jednotlivých primitivů do vzniklých buněk. Dělení je pozastaveno při dosažení maximálního počtu subdivizí, anebo při dosažení minimálního počtu primitivů v buňkách [12].

Metoda zakládá na principu dělení prostoru do nepřekrývajících se oblastí, kdy případě, že dochází k překryvu primitivu s více buňkami, je primitiv přiřazen do všech buněk, ve kterých se vyskytuje. Může tak docházet k nadbytečnému využití paměti i ke

snížení rychlosti procházení struktury. Existují však varianty algoritmu, jenž tento problém minimalizují použitím arbitrárních dělících rovin, např. BSP nebo Kd-tree [12].

Vzniklou datovou strukturu si u Octree, jak již název napovídá, lze představit jako strom, kde každý z rodičů má 8 potomků, které ukazují buď na další buňky, nebo přímo na obsažené primitivy. Na následujícím obrázku je zachycena 2D reprezentace datové struktury Octree (Quadtree) za účely zjednodušení vizualizace dělení prostoru do buněk.



Obrázek 1.4 Vizualizace datové struktury Octree ve 2D, upraveno z [12]

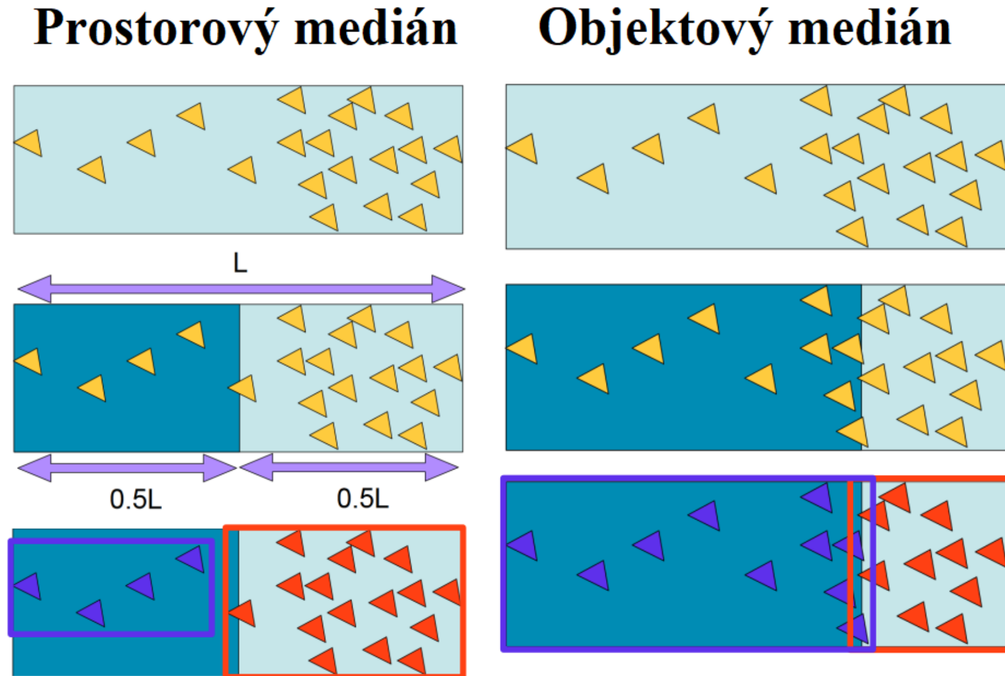
Na obrázku lze vidět kořenový uzel stromu (vlevo) a odpovídající ohraničující objem primitivů (vpravo) zvýrazněný červeně. Ten je dále rozdělen do 4 kvadrantů, přičemž kvadranty 2, 3 a 4 jsou listové uzly stromu (nemají žádné další potomky) a kvadrant č. 1 je rekurzivně rozdělen do dalších podléhajících kvadrantů do dosažení minimálního počtu primitivů v každém uzlu stromu. Postup při budování Octree ve 3D je identický, avšak jak již bylo zmíněno, tak ohraničující objemy jsou děleny do osmi namísto čtyř částí.

Struktura BVH

Struktura BVH je na rozdíl od Octree založena na principu rekurzivního dělení objektů, nikoliv prostoru. Jednou z hlavních charakteristik této struktury je pak možnost překrytí jednotlivých ohraničujících objemů potomků nadřazeného uzlu stromu. Tato vlastnost umožňuje vyšší flexibilitu při tvorbě podléhající akcelerační struktury, neboť lze zajistit, že každý z primitivů je obsažen v koncových uzlech nejvýše jednou [12]. Nevýhodou poté může být složitější způsob procházení této struktury, neboť v místě překryvu dvou ohraničujících objemů je nutno zkontrolovat průsečíky všech příslušných primitivů obou uzlů, z nichž je následně vybrán ten nejbližší.

Při tvorbě struktury BVH lze rozhodnout o rozdělení nadřazeného uzlu za pomocí tří heuristik: prostorový a objektový medián nebo SAH (*Surface Area Heuristics*) [12]. První zmíněná technika je založena na principu dělení nadřazeného ohraničujícího objemu na základě jeho nejdelší strany. Dochází tak roztřídění jednotlivých primitivů na základě jejich pozice vůči zvolené dělící rovině, jež je umístěna uprostřed nadřazeného objemu.

Druhá technika jednoduše volí nejdelší osu nadřazeného objemu, vůči níž jsou všechny primitivy roztříděny do seznamu, jenž je následně rozdělen do dvou stejných polovin na základě počtu primitivů. Princip obou technik je znázorněn na následujícím obrázku.



Obrázek 1.5 Dělení uzlu BVH pomocí prostorového a objektového mediánu [12]

Na obrázku vlevo lze vidět rozdělení primitivů do dvou skupin na základě jejich pozic vůči ploše, jež dělí prostor nadřazeného uzlu do dvou polovin. Lze pozorovat, že počet primitivů v levém a pravém uzlu není stejný, důsledkem čehož může být zkonstruován nevyvážený binární strom. Vpravo lze vidět rozdělení dílčích primitivů dle objektového mediánu, přičemž lze pozorovat, že počet primitivů v pravém i levém uzlu je stejný. Na rozdíl od první struktury je tak zkonstruován vyvážený binární strom. Při procházení struktury je tedy počet kroků ke koncovým uzlům konzistentní.

Třetí technika poté využívá matematické funkce SAH, jež umožňuje vybrat optimální dělící rovinu na základě minimalizace ceny rozdělení. Tuto cenu je možno vyjádřit jako podmíněnou pravděpodobnost průniku paprsku s ohraničujícím objemem podřazeného uzlu v případě, že byl již nalezen průnik s nadřazeným uzlem. Tuto pravděpodobnost lze vypočíst na základě povrchu ohraničujících objemů dílčích uzlů [13]:

$$P(N_c|N)^{SAH} = \frac{SA(N_c)}{SA(N)}, \tag{1.18}$$

kde N_c je podřazený uzel, N je nadřazený uzel a SA je funkce pro výpočet povrchu uzlu.

Hodnota funkce SAH může být ovlivněna jak samotným povrchem ohraničujícího objemu, tak i počtem primitivů obsažených v daném uzlu. Rekurzivní dělení lze poté ukončit buď při dosažení minimálního počtu primitivů v uzlu, nebo pokud je součet cen obou rozdělených uzlů vyšší než cena nadřazeného uzlu [14].

1.4.2 Hardwarové optimalizace SIMD

Jedná se o druh paralelní výpočetní architektury, která umožňuje aplikovat operaci, zakódovanou v jediné instrukci, na větší počet datových operandů zároveň. Jedná se tak o metodu paralelismu pracující na úrovni dat (*data level parallelism*), která umožňuje výrazně zvýšit prostupnost vykonávání aritmetických operací v procesoru, zejména pak u velkých datových souborů [15].

V komerčních produktech lze tuto architekturu nalézt buď v procesorech (*CPU*) v podobě vektorových rozšíření instrukční sady, nebo v grafických procesorech (*GPU*) v podobě specializovaných hardwarových jednotek – streamovacích multiprocesorů [15].

Pro tuto práci jsou relevantní vektorová rozšíření instrukční sady x86-64, mezi něž patří např. SSE, AVX (*Advanced Vector Extensions*), AVX2 anebo AVX512. Následující kapitola je konkrétně zaměřena na rozbor instrukční sady SSE.

Instrukční sada SSE

Tato instrukční sada poskytuje šestnáct 128 bitových registrů, které umožňují uchovávat buď čtyři 32 bitové čísla, nebo dvě 64 bitové čísla, jak v celočíselném formátu, tak i ve formátu s plovoucí desetinnou čárkou [16]. Sada rovněž poskytuje řadu vektorových instrukcí, které paralelně manipulují všechny datové členy v registrech. Instrukce plní celou řadu funkcí, např. přenosové, aritmetické, bitové, porovnávací anebo převodní.

Vektorových optimalizací kódu lze dosáhnout ve vysokoúrovňových jazycích buď automaticky prostřednictvím kompilátoru, anebo ručně prostřednictvím inline assembly kódu, případně použitím standardizovaných intrinsických funkcí [17].

V jazyku C a C++ podporuje naprostá většina kompilátorů intrinsické vektorové funkce a datové typy. Příkladem použití může být jednoduchý program, jenž načte čtyři čísla ze dvou polí typu `float[4]` do SSE registrů a uloží jejich součet do dalšího registru.

```
__m128 add(float x[4], float y[4]) {
    __m128 m1 = _mm_loadu_ps(x); //Načtení dat do prvního registru
    __m128 m2 = _mm_loadu_ps(y); //Načtení dat do druhého registru
    return _mm_add_ps(m1,m2); //Uložení výsledku operace součtu do registru
}
```

Ve výše uvedené funkci se vyskytuje datový typ `__m128` reprezentující SSE registr, který obsahuje 4 proměnné typu `float`. Dále lze vidět použití intrinsických funkcí ve formátu `_mm_xxx_ps()`, které představují použití konkrétních instrukcí. Po kompilaci lze získat následující strojový kód [18].

```
add(float*, float*):
vmovups xmm0, xmmword ptr [rsi]        # Načtení dat z 1. pole
vaddps  xmm0, xmm0, xmmword ptr [rdi]  # Součet dat v registru a 2. pole
ret
```

Lze pozorovat, že i přes použití intrinsických funkcí byl kód dále optimalizován, neboť druhá instrukce `vmovups` odpovídající `_mm_loadu_ps(y)` byla nahrazena použitím ukazatele na paměť druhého pole ve třetím operandu instrukce `vaddps`. Tím bylo rovněž dosaženo toho, že celá funkce využívá pouze jediný registr. Nízká okupace registrů pak umožňuje řetězení instrukcí za účelem skrytí latencí spojených se čtením dat z paměti.

2. TVORBA PROGRAMU

V této kapitole jsou popsány základní principy fungování tvořeného programu. Uvedené ústřižky kódu slouží převážně k vytvoření představy o tom, jak vypadá vnitřní struktura programu. Byly značně zjednodušeny a místy zkráceny do stylu spíše připomínajícího pseudokód, tak aby na nich bylo možno stručně vysvětlit jejich hlavní podstatu fungování.

2.1 Prerekvizity

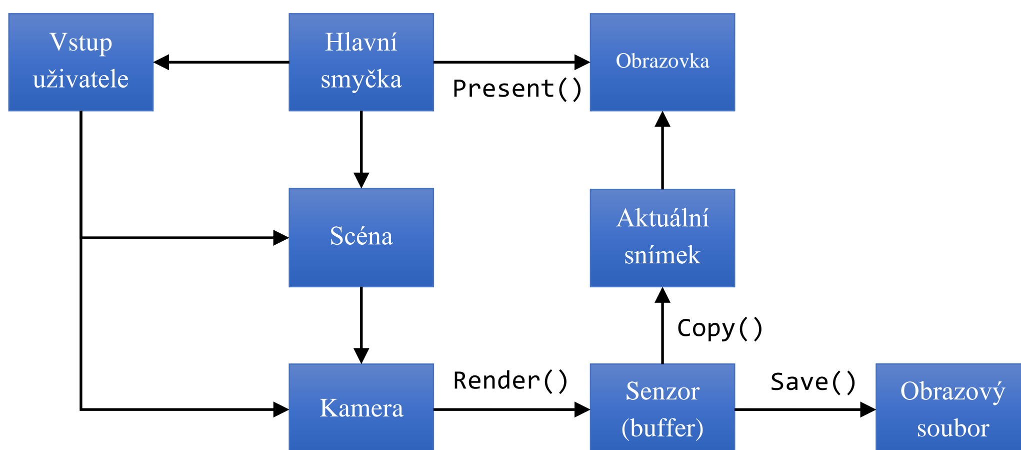
V této kapitole jsou popsány základní rozhodnutí, které bylo potřeba provést ještě před naprogramováním jádra programu, založeného na vykreslování obrázků prostřednictvím metody sledování cest. Jsou popsány základní požadavky na vstupně-výstupní funkce, např. ovládání, vizualizace obrázků anebo načítání a ukládání dat na disk. Je rovněž odůvodněno použití konkrétních knihoven za účelem implementace těchto funkcí.

2.1.1 Programovací model

Pro tvorbu tohoto programu byl zvolen jazyk C++, neboť nabízí širokou škálu užitečných nástrojů umožňujících abstrakci dat, která je nezbytná k vytvoření grafického programu se schopností vykreslování různých typů grafických primitivů s různorodými materiály.

Hlavní komponenty programu jsou rozděleny do tříd obsahujících třídní metody. V modelu je tak umožněna tvorba jednotlivých modulárních komponent, které je možno upravovat a vylepšovat nezávisle na sobě, čímž se velmi zjednodušuje přidávání nových funkcionalit, aniž by docházelo ke kolizím s ostatními komponenty.

Struktura programu se skládá z hlavní smyčky obsahující zpracování uživatelského vstupu, aktualizaci dat scény, akumulaci vykreslených vzorků do bufferu, převod dat z bufferu do snímku a jeho zobrazení na obrazovce. Jeden cyklus smyčky pak představuje jeden vykreslený snímek. Dodatečně lze uložit obsah bufferu do obrazového souboru.



Obrázek 2.1 Základní hierarchie programu

2.1.2 Uživatelské rozhrání

Nejjednodušší způsob komunikace s programem by tvořilo rozhrání v konzoli, kde by se daly zadávat příkazy před počatím výpočtů, např. volba scény, poloha kamery, rozlišení a ostatní parametry. Následně by proběhlo offline renderování do paměťového bufferu, jehož obsah by se poté dal uložit do souboru na pevném disku v grafickém formátu.

Hlavní funkcí tvořeného programu je však schopnost online renderování, má být tedy schopen zobrazovat v reálném čase renderované snímky, které se mění v závislosti na vstupu uživatele. Rovněž pak umožňuje zachytit aktuální snímek do souboru na disku. V následujících kapitolách je popsáno, jak tohoto cíle bylo dosaženo.

Správa oken a vstupu uživatele

Interaktivní program lze realizovat prostřednictvím nativního grafického okna pro daný operační systém. Okno poté umožňuje vykreslení obsahu prostřednictvím nativních API (*Application Programming Interface*), včetně zpracování uživatelského vstupu z myši, klávesnice, joysticku aj. V systému Windows by tento krok zahrnoval použití nativní API DirectX, avšak program by nebylo možno spustit na ostatních systémech bez složitého přepisování kódu pro API ostatních systémů (Metal, OpenGL, Vulkan).

Přesně za tímto účelem byla použita knihovna SDL (*Simple DirectMedia Layer*) [19], jež nabízí řadu abstraktních příkazů, které je schopna implicitně překládat do nativních příkazů zvoleného systému bez nutnosti zásahu programátora.

Uživatelské prostředí

Knihovna SDL postrádá jakékoliv funkce pro tvorbu uživatelského prostředí, které je nezbytné k vytvoření uživatelsky přívětivého ovládání. K tomuto účelu slouží knihovna Dear ImGui [20], která umožňuje nativní integraci univerzálních widgetů do velkého množství grafických API a backendů, včetně SDL.

Vstupně výstupní systém

Vstupně výstupní (IO) systém slouží k načítání a ukládání souborů do úložiště. V tomto programu je potřeba načítat textury různých formátů do paměti pro použití v grafickém enginu. Zároveň je potřeba ukládat data do obrazového souboru v případě zachycení snímku obrazovky. K tomuto účelu je použita knihovna STB Image Write [21], která umožňuje načítání a ukládání souborů v grafickém formátu.

2.2 Jádro grafického programu

Teoretická hierarchie programu, jež byla popsána v předchozí kapitole, byla prakticky implementována prostřednictvím třídy `Engine`, jejíž definici lze nalézt v hlavičkovém souboru „Engine.h“. Ve třídě jsou zapouzdřeny všechny nezbytné objekty knihovny SDL, které umožňují tvorbu okna, textury a rendereru, anebo zpracování vstupu uživatele. Ve třídě je dále umístěna instance třídy `Scene`, jejíž funkcionality jsou popsány v dalších kapitolách. Třída rovněž obsahuje členské metody, které rozdělují hlavní vykreslovací smyčku programu do dílčích úloh, například zpracování vstupu, sestavení kontextových

nabídek, rozčlenění jednotlivých prvků obrazovky, omezení snímkové frekvence anebo vykreslení obsahu okna.

```
#include <SDL.h>
class Engine {
public:
	Engine(...);			//Konstruktor třídy
	void Render_loop();		//Hlavní vykreslovací smyčka
private:
	void Delay(double sec);	//Prodleva vykreslení
	void Resize();			//Změna velikosti okna
	void Process_input();		//Zpracování vstupu
	void Process_overlay();	//Zpracování kontextových nabídek
	void Draw_scene();		//Vykreslení scény
	void Object_menu();		//Nabídka upravení objektů
	void Camera_menu();		//Nabídka nastavení scény/kamery
	void Add_object();		//Nabídka přidání objektu
	void Add_material();		//Nabídka přidání materiálu
	Scene scene;			//Instance scény
	SDL_Window* window;		//Grafické systémové okno
	SDL_Renderer* renderer;	//Renderer (zobrazovací plocha okna)
	SDL_Texture* frame;		//Aktuální snímek
	SDL_Rect screen, viewport, menu;//Proporce okna
	SDL_Event event;			//Proměnná uživatelského vstupu
	Event_handler handle;		//Zpracování uživatelského vstupu
	... //Další stavové proměnné a metody (running, overlay, dt, timer()…)
};
```

V konstruktoru třídy jsou inicializovány všechny nezbytné komponenty enginu v podobě grafických objektů knihoven SDL a ImGui a načtení přednastavené scény č. 1.

```
Engine::Engine(...) {
	SDL_Init(SDL_INIT_EVERYTHING);	//Inicializace SDL
	window = SDL_CreateWindow(...);	//Vytvoření okna
	renderer = SDL_CreateRenderer(window,...);	//Vytvoření rendereru
	frame = SDL_CreateTexture(renderer,...);	//Vytvoření snímku
	ImGui::CreateContext();	//Inicializace ImGui
	ImGui::StyleColorsDark(); //Barevný motiv ImGui
	ImGui_ImplSDL2_InitForSDLRenderer(window, renderer); //Přiřazení okna
	ImGui_ImplSDLRenderer_Init(renderer); //Inicializace ImGui pro SDL
	scn_load(scene, 1);//Načtení výchozí scény
}
```

2.2.1 Hlavní smyčka programu

Celá funkcionalita grafického programu je abstrahována ve třídě `Engine`. Vytvořením instance této třídy v metodě `main()` a zavoláním metody `Render_loop()` je tak dosaženo úhledné implementace celého jádra programu ve 2 řádcích kódu. Zdrojový kód programu tak lze lehce importovat jako knihovnu do jiných, rozsáhlejších projektů. Po ukončení této metody dojde k destrukci objektu `engine`, a program tak může pokračovat v exekuci.

```
#include "Engine.h"
int main() {
	PTCR::Engine engine(90);	//Instance třídy Engine, 90 FPS
	engine.Render_loop();		//Hlavní smyčka
}
```

V hlavní smyčce programu jsou postupně volány třídní metody pro zpracování vstupu a kontextových nabídek, vykreslení obrazu a případného omezení snímkové frekvence. Tyto metody jsou podrobněji popsány v následujících kapitolách.

```
void Engine::Render_loop() {
	while (running) {
		Process_input();   //Zpracování vstupu uživatele
		Process_overlay(); //Zpracování překrytí a kontextových nabídek
		Render();          //Vykreslení scény, nabídek, okna
		Delay(...);        //Omezení snímkové frekvence
	}
}
```

2.2.2 Zpracování vstupu

V následujícím kódu lze vidět příklad pro zpracování vstupu s využitím knihovny SDL, včetně komunikace s podléhající třídou `Scene`. Pomocí metody `SDL_PollEvent()` jsou v cyklu načteny všechny vstupní stavy z fronty, které jsou dále předány knihovně ImGui a uloženy v instanci třídy `Event_handler`. Speciální vstupy, jako je třeba uzavření nebo změna velikosti okna, jsou zpracovány přímo v cyklu. Načtením ostatních stavů do pole lze zachytit všechny stisknuté klávesy během jednoho snímku. Je tak možno interpretovat kombinaci stisknutých kláves jako plynulý pohyb kamery ve více směrech naráz.

```
void Engine::Process_input() {
	while (SDL_PollEvent(&event)) { //Čtení vstupů z fronty
		ImGui_ImplSDL2_ProcessEvent(&event); //Předání vstupu ImGui
		handle.scan(event); //Uložení vstupu do pole
		if (event.type == ...) {...} //Zpracování speciálních vstupů
	}
	//Interpretace načtených vstupů pro pohyb a rotaci kamery
	vec4 dir(handle[SDL_SCANCODE_W] - handle[SDL_SCANCODE_S], ...);
	vec4 pan(handle[SDL_SCANCODE_LEFT] - handle[SDL_SCANCODE_RIGHT], ...);
	scene.cam.move(dir);
	scene.cam.rotate(pan);
}
```

2.2.3 Kontextové nabídky

V této kapitole jsou obsaženy základní příkazy knihovny ImGui určené k vytvoření uživatelského rozhrání s ovládacími prvky. V následujícím úryvku kódu je pak zobrazena třídní funkce `Process_overlay()`, která seskupuje všechny funkce definující jednotlivé okna s ovládacími prvky, spolu se samotnými příkazy pro řízení logiky knihovny ImGui.

```
void Engine::Process_overlay() {
	if (overlay) {
		ImGui_ImplSDLRenderer_NewFrame(); //Započetí snímků pro SDL
		ImGui_ImplSDL2_NewFrame();
		ImGui::NewFrame(); //Započetí interního snímku ImGui
		Camera_menu(); //Okno s nastavením parametrů kamery
		Object_menu(); //Okno s nastavením parametrů světa a objektů
		Add_object();      //Okno pro přidání nového objektu
		Add_material();    //Okno pro přidání nového materiálu
		ImGui::Render();   //Vykreslení interního snímku ImGui
	}
}
```

Jako příklad kontextové nabídky lze použít úryvek kódu z funkce `Add_material()`, jež slouží k vytvoření okna, které umožňuje přidávání materiálů s nastavením parametrů prostřednictvím různých widgetů. Ostatní okna jsou vytvořena obdobným způsobem.

```
void Engine::Add_material() {
	if (add_mat) { //Viditelnost okna na základě proměnné add_mat
		ImGui::Begin("Add material", &add_mat); //Začátek okna
		ImGui::SliderInt(...)//Posuvník pro celá čísla
		ImGui::Checkbox(...); //Zaškrtávací políčko
		ImGui::ColorEdit4(...); //Posuvník pro nastavení barvy RGBA
		... //Další widgety
		if (ImGui::Button("Add")) { //Akce při stisku tlačítka
		scene.world.add_mat(...); //Přidání materiálu na základě param.
		}
		ImGui::End(); //Konec okna
	}
}
```

2.2.4 Vykreslení obrazu

Funkce `Render()` slouží k vykreslení všech prvků obrazovky v podobě nabídek ImGui i samotného plátna obsahujícího vykreslený snímek ze třídy `Scene`.

V prvé řadě je prostřednictvím metody `SDL_LockTexture()` vyhrazena část paměti RAM, do níž lze zapisovat nová data textury. Ukazatel na tuto paměť je předán funkci `Scene::Render()`, která do ní zapíše nová data vygenerovaného snímku. Pomocí funkce `SDL_LockTexture()` jsou tyto data následně nahrána do paměti grafické karty. Obsah textury je poté zkopírován do rendereru okna funkcí `SDL_RenderCopy()`.

V druhém kroku jsou do rendereru zkopírovány data snímku s nabídkami knihovny ImGui, které byly vytvořeny na začátku hlavní smyčky ve funkci `Process_overlay()`. Jelikož je tento příkaz zavolán až po vykreslení snímku scény, tak existující data v okně jsou přepsána a kontextové nabídky jsou tak zobrazeny navrchu.

Následně je obsah rendereru okna prezentován na obrazovce prostřednictvím metody `SDL_RenderPresent()`. Tento snímek pak zůstává zobrazen po celou dobu zpracování nadcházejícího snímku.

```
void Engine::Render() {
	uint* disp = nullptr; //Ukazatel na texturu pro vykreslení obrazu
	int pitch = viewport.w * 4; //Šířka textury
	SDL_LockTexture(frame, 0, (void**)&disp, &pitch); //Přiřazení textury
	scene.Render(disp, pitch / 4); //Vykreslení scény do textury v RAM
	SDL_UnlockTexture(frame); //Nahrání textury do GPU
	//Kopie obsahu textury do rendereru (vykreslení scény)
	SDL_RenderCopy(renderer, frame, 0, &viewport);
	//Kopie bufferu ImGui do rendereru (vykreslení nabídky)
	ImGui_ImplSDLRenderer_RenderDrawData(ImGui::GetDrawData());
	//Zobrazení obsahu rendereru v okně
	SDL_RenderPresent(renderer);
}
```

2.3 Reprezentace objektů v prostoru

V této kapitole jsou uvedeny datové typy, které jsou v programu využity k popisu poloh, směrů anebo transformací v trojrozměrném prostoru.

2.3.1 Vektory

V kontextu této práce se jedná o geometrické vektory, které slouží k popisu nějakého směru nebo polohy v prostoru. Mají obecně 3 složky, které reprezentují vzdálenost od počátku souřadného systému ve třech základních osách. V grafice mohou mít i čtvrtou složku, která reprezentuje, zdali se jedná o bod, nebo směrový vektor.

V programu je vektor implementován se všemi čtyřmi složkami, čtvrtá složka ničemu nepřekáží, neboť všechny geometrické operace selektivně počítají pouze s prvními třemi složkami. Čtvrtá složka naopak zlepšuje fyzické rozložení paměti díky ukládání dat na 16B rozhrání a zároveň tak umožňuje načítání celistvých vektorů do 128b SSE registrů bez nutnosti maskování. Tento datový typ lze pak rovněž použít k reprezentaci barev se složkami RGBA (*Red Green Blue Alpha*).

Datovou část tvoří pole `xyz[4]`, ve kterém jsou uloženy jednotlivé složky XYZ a W. Struktura obsahuje konstruktor pro implicitní převod z jedné skalární hodnoty na vektor, konstruktor ze 2 až 4 skalárních hodnot a řadu dalších členských funkcí.

```
struct vec4 {
	vec4() : xyz{} {} //Základní konstruktor
	vec4(float t) : xyz{ t,t,t,t } {} //Skalární konstruktor
	vec4(vec4 v, float w) :xyz{ v.x(),v.y(),v.z(),w }{} //Vec3 + 4. složka
	vec4(float x, float y, float z = 0, float w = 0) : xyz{ x,y,z,w } {}
	float x() const { return xyz[0]; } //Dílčí složky XYZW
	float y() const { return xyz[1]; }
	float z() const { return xyz[2]; }
	float w() const { return xyz[3]; } //Délka vektoru
	float len() const { return sqrtf(x() * x() + y() * y() + z() * z()); }
	... //Další členské metody (dir, print, operator…)
	float xyz[4]; //Pole 4x float (16B)
};
```

V souborech „vec4.h“, „vec4_scalar.h“ a „vec4_sse.h“ je implementována rozsáhlá řada funkcí pracující s vektory. Soubor s příznakem „sse“ obsahuje implementace funkcí optimalizované pro instrukční sadu SSE a soubor „scalar“ poté obsahuje jejich základní verze bez optimalizací.

Mezi nejdůležitější funkce v oblasti geometrie patří skalární a vektorový součin, které jsou v kódu implementovány dle jejich příslušných matematických definic.

```
float dot(vec4 u, vec4 v) { //Skalární součin vektorů
	return u.x() * v.x() + u.y() * v.y() + u.z() * v.z();
}
vec4 cross(vec4 u, vec4 v) { //Vektorový součin vektorů
	float x = u.y() * v.z() - u.z() * v.y();
	float y = u.z() * v.x() - u.x() * v.z();
	float z = u.x() * v.y() - u.y() * v.x();
	return vec4(x, y, z);
}
```

2.3.2 Matice

Pro reprezentaci transformací (rotace, škálování a translace) se používá transformačních matic. Jedná se o matice s rozměry 4x4, jejichž první tři sloupce reprezentují rotaci prostřednictvím Eulerových úhlů a poslední sloupec reprezentuje translaci. Vynásobením matice s daným vektorem dojde k jeho transformaci.

Matice je v programu implementována pomocí 4 vektorů, kde první 3 uchovávají data o transformaci a 4. explicitně ukládá úhly rotace v osách XYZ. Konstruktor má tři vstupní parametry – `P` (pozice), `A` (úhel) a `S` (měřítko). Za pomocí funkce `Compose()` jsou do matice zakódovány všechny tři transformační složky.

Třída dále nabízí řadu algebraických funkcí, např. transpozici, inverzi anebo násobení matice vektorem, příp. maticí. Funkce `vec()` a `pnt()` poté slouží k výpočtu transformace směrového vektoru, resp. bodu. Celou implementaci lze nalézt v souboru „mat4.h“.

```
struct mat4 {
	mat4() { //Jednotková matice
		x = vec4(1, 0, 0, 0);
		y = vec4(0, 1, 0, 0);
		z = vec4(0, 0, 1, 0);
		w = vec4(0, 0, 0, 1);
	}
	mat4(vec4 P, vec4 A = 0, vec4 S = 1) { //Translace, Rotace, Škála
		Compose(P, A, S);
	}
	//Přímy konstruktor ze 4 vektorů
	mat4(vec4 r0, vec4 r1, vec4 r2, vec4 r3) : R{ r0,r1,r2,r3 } {}
	//Dekompozice matice do dílčích složek
	vec4 P() const {return vec4(x.w(), y.w(), z.w(), w.w());}
	vec4 S() const {return vec4(x.len(), y.len(), z.len());}
	vec4 A() const {return w;}
	mat4 transpose() const; //Transponovaná matice
	mat4 inverse() const; //Inverzní matice
	mat4 operator*(mat4 T) const; //Násobení matice maticí
	vec4 operator*(vec4 u) const { //Násobení matice vektorem
		return vec4(dot4(u, x), dot4(u, y), dot4(u, z), u.w());
	}
	vec4 vec(vec4 u) const; //Transformace směru
	vec4 pnt(vec4 u) const; //Transformace bodu
	void Compose(vec4 P, vec4 A, vec4 S) { //Konstrukce ze 3 parametrů
		Rotation(A); //Alternativně lze vytvořit 3 transformační matice
		Scale(S);    //a vynásobit je v tomto pořadí
		Position(P);
	}
	void Scale(vec4 S); //Škálovací složka
	void Position(vec4 P); //Translační složka
	void Rotation(vec4 A); //Rotační složka
	... //Další metody
	union {
		vec4 R[4]; //4 řádky matice o 4 sloupcích
		struct { vec4 x, y, z, w; }; //Alias
	};
};
```

2.4 Geometrické primitivy

V této kapitole jsou uvedeny všechny geometrické primitivy, které jsou obsaženy v programu. Paprsek a ohraničující objem jsou speciální typy primitivů, neboť je nelze vykreslit přímo. Slouží pouze k vytvoření teoretického modelu, jenž umožňuje definování funkcí pro výpočty průniků s paprsky a hierarchické rozřazení těchto primitivů ve scéně.

2.4.1 Paprsek

Paprsek si lze představit jako orientovanou úsečku v prostoru, má tedy nějaký počáteční bod a směr. Tuto skutečnost lze vyjádřit pomocí 2 vektorů – `O` a `D`. První vektor označuje počáteční bod a druhý směr polopřímky. Dodatečně obsahuje 3. vektor – `iD`, v němž je uložena invertovaná hodnota směrového vektoru, jehož účel bud popsán později. Jedinou členskou funkcí třídy je `at()`, která vrací polohu bodu vypočteného dosazením proměnné `t` do parametrické rovnice paprsku. Viz hlavičkový soubor „ray.h“.

```
struct ray {
	ray(vec4 _O, vec4 _D) : O(_O), D(norm(_D)), iD(1.f / D) {}
	vec4 at(float t) const {
		return O + t * D; //Poloha bodu ve vzdálenosti t od počátku
	}
	vec4 O, D, iD;
};
```

Algoritmy pro výpočty průniků paprsků s primitivy jsou poté přímo implementovány ve třídních metodách jednotlivých primitivů.

2.4.2 Ohraničující objem

Ohraničujícím objemem AABB (*Axis Aligned Bounding Box*) se rozumí krychle nebo kvádr, jehož stěny jsou rovnoběžné s osami souřadného systému. V programu je využíván k ohraničení ostatních primitivů, celistvých objektů anebo uzlů akceleračních struktur. Je tak možno urychlit testování průniků paprsků s objekty.

Urychlení spočívá ve faktu, že blízko umístěné primitivy lze rozřadit do skupin, kterým lze přiřadit jeden AABB. Poté lze pro každou skupinu nejdříve otestovat průnik s jejím AABB a pouze v případě průniku s ním jsou poté otestovány i ostatní primitivy.

AABB je implementován v hlavičkovém souboru „aabb.h“, jeho strukturu tvoří dva body – `pmin` a `pmax`, ve kterých jsou uloženy nejnižší, resp. nejvyšší hodnoty souřadnic, které ohraničuje. Konstruktor pak přijímá 2 body, z nichž vypočítá jejich minimální a maximální souřadnice, které uloží do bodů `pmin` a `pmax`. Ve třídě jsou implementovány metody pro výpočet průniku s paprskem, určení středu a spojení dvou AABB.

```
struct aabb {
	aabb(vec4 p1, vec4 p2) : pmin(min(p1, p2)), pmax(max(p1, p2)) {}
	bool hit(const ray& r) const; //Výpočet průniku
	vec4 pmid()const { return 0.5f * (pmin + pmax);} //Střed objemu
	void join(const aabb& box); //Spojení dvou AABB
	... //Další metody (area(), move(), print()…)
	vec4 pmin = infp, pmax = infn; //Min a Max souřadnice AABB
};
```

Průnik paprsku s AABB je realizován jednoduchým ověřením, zdali alespoň jeden průsečík s jednou z nekonečných ploch, reprezentující jeho prodloužené stěny, spadá do hranic vytyčenými body AABB. Základní 2D verze algoritmu byla převzata z [22], byla pak formálně upravena a rozšířena o třetí rozměr. Zde nachází využití již dříve zmíněný inverzní směrový vektor paprsku – `iD`.

```
bool aabb::hit(const ray& r) const {
      vec4 t1 = (pmin - r.O) * r.iD; //Průnik s první plochou
      vec4 t2 = (pmax - r.O) * r.iD; //Průnik s druhou plochou
      vec4 tmin = min(t1, t2); //Minimální průsečík
      vec4 tmax = max(t1, t2); //Maximální průsečík
      float mint = max(tmin); //Min vzdálenost
      float maxt = min(tmax); //Max vzdálenost
      return mint < maxt && maxt > 0; //Bod leží na polopřímce paprsku
}
```

2.4.3 Obecná struktura primitivů

Všechny ostatní primitivy mají stejnou podobu vnitřní struktury. Mají tedy stejnojmenné funkce se stejnými vstupními i výstupními proměnnými, liší se pouze implementací a obsahem dat. Je nutno zmínit, že následující třída není základní třídou těchto primitivů, slouží pouze jako teoretická šablona implementace, v reálném kódu se tudíž nenachází.

Třídy primitivů tedy obsahují funkce pro výpočet průniku s paprskem, transformaci, konstrukci AABB, generování náhodných směrů, výpočet pravděpodobnosti průniku pro daný paprsek, včetně funkcí pro výpočet povrchu.

```
class primitive {
      primitive(); //Konstruktor
      bool hit(const ray& r, hitrec& rec) const; //Průnik paprsku
      aabb get_box()const; //Ohraničující objem objektu
      primitive trans(const matrix& T) const; //Transformace (trvalá)
      primitive move(const vec4& P) const; //Translace (dočasná)
      float pdf(const ray& r)const; //Pravděpodobnost průniku paprskem
      float area()const; //Povrch objektu (pro výpočet pdf)
      vec4 rand_to(vec4 O) const; //Náhodný směr k povrchu primitivu
      vec4 rand_from() const; //Náhodný směr z povrchu primitivu
      ... //Třídní proměnné (body, rozměry)
};
```

V kódu se vyskytuje nový datový typ: `hitrec`. Jedná se o strukturu uchovávající údaje o nejbližším průniku, obsahuje tedy normálu, průsečík, vzdálenost, UV souřadnice, index objektu a orientaci stěny. Při konstrukci struktury je nastavena počáteční hodnota vzdálenosti na 10^4, což má za následek to, že vzdálenější objekty již nebudou vykresleny. Hodnota `idx=-1` poté značí, že nebyl nalezen žádný průnik. Viz soubor „obj.h“.

```
struct hitrec {
      vec4 N, P;          //Normála, Průsečík
      float t = infp;     //Vzdálenost průsečíku, infp = 1e4f
      float u, v;         //UV souřadnice (pro texturu)
      uint idx = -1;      //Index objektu
      bool face;          //Orientace stěny (vnitřní/vnější)
};
```

2.4.4 Koule

Kouli lze definovat prostřednictvím souřadnic jejího středu a poloměru. V programu je využito skutečnosti, že vektor je schopen uchovávat 4 složky, což umožňuje použít první tři složky pro uchování souřadnic středu a čtvrtou složku pro poloměr.

```
class sphere {
	sphere(vec4 Q, float r) :Qr(Q, r) {} //Střed Q + poloměr r
	bool hit(const ray& r, hitrec& rec) const; //Průnik s paprskem
	... //Další třídní metody dle šablony
	vec4 Qr;
};
```

Metoda `hit()` řeší kvadratickou rovnici, která vznikne dosazením parametrické rovnice paprsku do implicitní rovnice koule. Výsledkem řešení této rovnice jsou jeden nebo dva průsečíky, z nichž je vybrán ten nejbližší. Viz hlavičkový soubor „sphere.h".

2.4.5 Krychle

Krychli lze stejně jako kouli definovat pomocí jediného vektoru, v němž první tři složky reprezentují souřadnice středu a čtvrtá pak délku jedné hrany.

```
class voxel {
	voxel(vec4 Q, float a) :Qa(Q, a) {} //Střed Q + délka strany a
	bool hit(const ray& r, hitrec& rec) const; //Průnik s paprskem
	... //Další třídní metody dle šablony
	vec4 Qa;
};
```

Metoda `hit()` je založena na metodě průniku paprsku s AABB, přičemž je doplněna o výpočet průsečíku, UV souřadnic a normály. Transformace tohoto primitivu je omezena pouze na translaci, neboť jeho stěny musí být rovnoběžné s osami souřadného systému. Viz hlavičkový soubor „voxel.h".

2.4.6 Čtyřúhelník

Čtyřúhelník lze definovat za pomocí tří vektorů QUV. První vektor reprezentuje zvolený vrcholový bod a další dva představují úsečky z tohoto bodu do dvou sousedních bodů. Poslední bod je osově souměrný vůči počátečnímu bodu, není ho proto nutno explicitně ukládat. Implementace rovněž obsahuje čtvrtý vektor N, v němž je uložena normála (první 3 složky) a povrch tvaru (4. složka).

Konstruktor třídy přijímá 3 body, z nichž vypočítá úsečky hran, normálu a obsah. Metoda `hit()` využívá Möller-Trumbore algoritmus [23] pro výpočet barycentrických souřadnic čtyřúhelníku. Průsečík spadá do plochy čtyřúhelníku v případě, že souřadnice nabývají hodnot v intervalu $0 < u < 1$ a $0 < v < 1$. Viz hlavičkový soubor „quad.h".

```
class quad {
	//3 body ABC, vrchol Q, úsečky U = AB, V = AC a normála N = cross(U,V)
	quad(vec4 A, vec4 B, vec4 C):Q(A), U(B-A), V(C-A), N(quad_ns(U, V)) {}
	bool hit(const ray& r, hitrec& rec) const; //Möller-Trumbore alg.
	... //Další třídní metody dle šablony
	vec4 Q, U, V, N;
};
```

Normála čtyřúhelníku je vypočtena vektorovým součinem dvou úseček, přičemž jeho obsah je úměrný délce tohoto vektoru.

```
vec4 quad_ns(vec4 u, vec4 v) {
	vec4 uv = cross(u, v);  //Vektorový součin UV
	vec4 N = norm(uv); //Normalizace vektoru
	N.xyz[3] = uv.len(); //Délka vektoru UV ve 4. složce N (plocha)
	return N;
}
```

2.4.1 Trojúhelník

Trojúhelník je stejně jako čtyřúhelník definován pomocí 3 vektorů QUV, avšak není již implicitně předpokládán čtvrtý vrcholový bod. Pří výpočtu bodu průniku jsou intervaly pro barycentrické souřadnice dodatečně omezeny na $0 < u < 1$ a $0 < v < 1 - u$.

Třída poly navíc umožňuje použití tří normálových vektorů příslušících jednotlivým vrcholovým bodům. Mezi těmito normálami lze interpolovat prostřednictvím souřadnic UV při výpočtu průniku paprsku, čímž je umožněno realizovat hladké stínování, které maskuje ostré hrany sousedících primitivů v modelu. Tyto normály bývají obsaženy přímo v datech 3D modelu, nebo je lze vypočíst na základě orientace sousedních trojúhelníků daného bodu při načítání modelu. Viz soubory „poly.h“ a „mesh.cpp“.

V následujícím úryvku kódu lze nalézt metodu hit(), v níž jsou vypočteny UV souřadnice se vzdáleností. V případě, že souřadnice spadají do stanovených intervalů a vypočtená vzdálenost je menší než v záznamu, dojde k přepsání záznamu novými údaji.

```
class poly {
	poly(vec4 Q, vec4 U, vec4 V, vec4 n0, vec4 n1, vec4 n2)
	: Q(Q), U(U), V(V), n0(n0), n1(n1), n2(n2) {}
	bool hit(const ray& r, hitrec& rec) const{
		vec4 pV = cross(r.D, V); //Vektor kolmý k paprsku a úsečce V
		float D = dot(U, pV); //Nenulový determinant
		float iD = 1.f / D;
		vec4 tV = r.O - Q; //Vektor z vrcholu do počátku paprsku
		vec4 qV = cross(tV, U); //Vektor kolmý k tV a úsečce U
		float u = dot(tV, pV) * iD; //UV souřadnice, vzdálenost průniku
		float v = dot(r.D, qV) * iD;
		float t = dot(V, qV) * iD; //Hraniční intervaly, min. vzdálenost
		if (within(u,0,1) && within(v,0,1-u) && inside(t,0,rec.t)) {
			rec.face = D > 0; //Orientace plochy, normála
			rec.N = rec.face ? normal(u, v) : -normal(u, v);
			rec.P = Q + u * U + v * V; //Určení průsečíku
			rec.u = u; rec.v = v; rec.t = t; //UV souřadnice, vzdál.
			return true; //Průnik byl nalezen
		}
		return false; //Průnik nebyl nalezen
	} //Interpolace normály z UV souřadnic
	vec4 normal(float u, float v)const {
		return norm((1.f - u - v) * n0 + u * n1 + v * n2);
	}
	... //Další třídní metody dle šablony
	vec4 Q, U, V;
	vec4 n0, n1, n2;
};
```

2.5 Scéna

Scéna obsahuje seznam všech objektů a materiálů, nastavení, kameru a členské funkce určené k vykreslení celistvých snímků i k výpočtu barev jednotlivých vzorků za pomocí metody sledování cest. Její implementace se nachází v hlavičkovém souboru „Scene.h“. V následujících kapitolách jsou podrobněji popsány její dílčí části.

```
class Scene {
	Scene(camera cam, obj_list world = {}) :cam(cam), world(world) {}
	camera cam;  //Kamera
	obj_list world; //Objekty scény
	mat4 sun_pos; //Poloha slunce na obloze
	scene_opt opt; //Nastavení scény
	texture skybox; //Textura pozadí
	void Render(uint* disp, uint pitch); //Vykreslení snímku
	void Screenshot(bool reproject = 0)const; //Uložení snímku
	vec4 raycol(const ray& r)const; //Výpočet barvy pixelu
	//Algoritmus sledování cest
	vec4 iterative_pt(const ray& sr, const hitrec& srec, int depth) const;
	... //Další třídní metody (volumetrika, vzorkování, debug…)
};
```

2.5.1 Kamera

Kamera modeluje reálné fyzikální vlastnosti fotoaparátu s objektivem. Má tedy nějakou polohu a rotaci, senzor s nastavením expozice, fokální délku a světelnost objektivu a řadu dalších interních parametrů. Viz hlavičkový soubor „camera.h“.

```
class camera {
	camera(uint w, uint h, float fov, mat4 _T = mat4()); //Konstruktor
	vec4 SS(vec4 xy) const; //Převod na screen space souřadnice
	ray optical_ray(vec4 xy) const; //Optický paprsek
	ray pinhole_ray(vec4 xy) const; //Geometrický paprsek
	mat4 T = mat4(); //Poloha a rotace kamery
	sensor CCD = sensor(1280, 720); //Obrazový senzor
	float foc_t = 0, foc_l = 0.0216f; //Vzdál. ostření, ohnisková vzdál.
	float fstop = 16.f , exposure = 1.f; //Světelnost, expozice
	const float frame = 0.035, diagonal = 0.0432; //Fyz. proporce senzoru
	bool autofocus = 1, bokeh = 1; //Automatické ostření, hloubka ostrosti
};
```

Nejdůležitější třídní metodou je `optical_ray()`. Vstupním parametrem metody jsou XY souřadnice pixelu umístěného na senzoru, které jsou následně převedeny do lokálního souřadného systému v rozsahu $\langle -1{,}1 \rangle$. Z těchto souřadnic je dále zkonstruován vektor orientovaný ve směru záporné osy Z. Finálně je sestrojen paprsek, jehož směrový vektor je transformován z lokálních souřadnic do souřadnic světa. Paprsek je následně odsazen v náhodném směru vzhledem k nastavené vzdálenosti ostření, fokální délce a světelnosti objektivu. Je tak docíleno realistického efektu hloubky ostrosti.

```
ray camera::optical_ray(vec4 xy) const {
	if (!bokeh)return pinhole_ray(xy); //Paprsek bez efektů optiky
	vec4 D = T.vec(foc_t * SS(xy)); //Směrový vektor (world space)
	vec4 r = T.vec(foc_l * sa_disk() / fstop); //Rozostření na disku
	return ray(P - r, D + r); //Odsazený “optický” paprsek
}
```

Instance třídy senzoru `CCD` uchovává akumulační pole `data` a ukazatel na výstupní texturu `disp`. V následujícím úryvku je zobrazena funkce `add()`, jež slouží k akumulaci vzorků do pole za pomocí klouzavého průměru. Čtvrtá složka vektoru slouží k uchování hloubky pixelu. Informace o hloubce lze využít při zpracování výsledného obrazu anebo při reprojekci. Další funkcí je `out()`, která slouží k uložení obsahu akumulačního bufferu do výstupní textury. Viz soubor „sensor.h“.

```
class sensor {
	sensor(uint w, uint h): data(w*h), buff(w*h), w(w), h(h), n(w*h) {}
	vector<vec4> data; //Akumulační buffer
	vector<vec4> buff; //Sekundární buffer (postproccesing, reprojekce)
	uint* disp = nullptr; //Ukazatel na výstupní texturu
	uint w = 0, h = 0, n = 0, pitch = 0; //Rozměry senzoru, šířka textury
	float time = 0.f; //Časová proměnná (počet vzorků)
	void add(uint i, uint j, vec4 rgb) {
		uint off = i * w + j; //Jednorozměrný offset z 2D indexů
		rgb = fixnan(rgb); //Potlačení NaN a akumulace do bufferu
		data[off] = (1 – 1 / time) * data[off] + rgb / time;
	}
	void out(uint i, uint j) {
		uint off = i * w + j;
		uint off2 = i * pitch + j; //Šířka textury je závislá na SDL
		disp[off2] = vec2bgr(data[off]); //Zakódování uchar[4] barvy
	}
};
```

Funkce `out()` využívá funkci `vec2bgr()`, která slouží k převodu barvy ve formátu `vec4` RGBA do celočíselného formátu ARGB8888 typu `uint`, který umí grafická karta dále zpracovat a zobrazit na obrazovce. Viz hlavičkový soubor „vec4.h“.

2.5.2 Seznam objektů

V seznamu objektů jsou uloženy všechny objekty a materiály, které jsou použity v dané konkrétní scéně. Obsahuje metody pro přidávání objektů i materiálů a metodu `hit()`, jež vrací informace o nejbližším průniku se všemi objekty. Celý seznam je navíc ohraničen AABB, lze tudíž eliminovat testování paprsků, které směřují mimo hranice scény. Ostatní paprsky jsou otestovány vůči všem objektům nebo struktuře BVH, jejíž implementace je popsána v dalších kapitolách, případně ji lze nalézt v souboru „obj_list.cpp“.

```
class obj_list {
	aabb bbox; //Ohraničující objem
	vector<mesh_var> objects; //Seznam objektů
	vector<mesh_raw> obj_bvh; //Seznam primitivů v objektech pro BVH
	vector<bvh_node> bvh; //Struktura BVH
	vector<mat_var> materials; //Seznam materiálů
	vector<uint> lights; //Seznam indexů světelných zdrojů
	vector<uint> nonbvh; //Seznam indexů objektů, jenž nepatří do BVH
	bool hit(const ray& sr, hitrec& rec) const; //Nejbližší průnik paprsku
	float pdf(const ray& r)const; //PDF pro zdroje osvětlění
	vec4 rand_to(vec4 O) const; //Náhodný směr ke zdrojům osvěltení
	void add_mat(const mat_var& mat); //Přidání materiálu
	void add_obj(const mesh_var& object); //Přidání objektu
	... //Další třídní metody (konstrukce BVH, aktualizace objektů…)
};
```

V následujícím úryvku kódu je zobrazena implementace metody `hit()`.

```
bool obj_list::hit(const ray& r, hitrec& rec) const {
if (bbox.hit(r)) { //Průnik ohraničujícího objemu scény
	if (en_bvh) { //Struktura BVH je zapnuta
		for (const auto& obj : nonbvh) //Průnik objektů mimo BVH
			rec.idx = objects[obj].hit(r, rec) ? obj : rec.idx;
		if (bvh[0].bbox.hit(r)) //Průnik prvního uzlu BVH
			traverse_bvh(r, rec, 0); //Procházení BVH
	}
	else { //Průnik všech objektů
		for (uint obj = 0; obj < objects.size(); obj++)
			rec.idx = objects[obj].hit(r, rec) ? obj : rec.idx;
	}
}
return rec.idx != -1; //Byl nalezen průnik
}
```

V kódu třídy lze dále vidět, že materiály i objekty jsou uchovávány v dynamickém poli typu `vector`, jejichž datový typ nese příznak `_var`, čímž je indikováno, že se jedná o speciální datový typ – varianty, jejichž princip je popsán v následující kapitole.

2.5.3 Varianty

Varianta je datová struktura uchovávající union všech požadovaných datových typů, spolu s identifikátorem, který určuje současný uložený datový typ. Dle identifikátoru lze poté správně interpretovat uložená data, anebo volat jejich příslušné třídní funkce. Je tak docíleno dynamického polymorfismu, který nabízí řadu výhod oproti použití virtuálních funkcí a pole ukazatelů na objekty. Jednou z výhod je pak datová lokalita, neboť data uložených objektů v poli se nachází sekvenčně v paměti, což má za následek zvýšení efektivity čtení dat z paměti RAM i zvýšení efektivity mezipaměti procesoru.

Zde je příklad implementace varianty třídy `mesh`, obsahující polymorfickou funkci `hit()`. Celou implementaci této struktury lze nalézt v hlavičkovém souboru „mesh.h".

```
struct mesh_var { //První konstruktor slouží k načtení modelu ze souboru
	mesh_var(const char* name,...) : p(load_mesh(name)),flag(o_pol, ...)
	mesh_var(const mesh<poly>& m, ...)  : p(m), flag(o_pol, ...) {}
	mesh_var(const mesh<quad>& m, ...)  : q(m), flag(o_qua, ...) {}
	mesh_var(const mesh<sphere>& m,...) : s(m), flag(o_sph, ...) {}
	mesh_var(const mesh<voxel>& m, ...) : v(m), flag(o_vox, ...) {}
	bool hit(const ray& r, hitrec& rec) const; //Průnik s paprskem
	... //Další třídní metody (pdf, rand_to, set_trans…)
	union { //Členy okupují stejnou adresu v paměti
		mesh<poly> p;
		mesh<quad> q;
		mesh<sphere> s;
		mesh<voxel> v;
	};
	c_str name;  //Název objektu (pro načtené modely ze souboru)
	obj_flags flag; //Identifikátor typu + nastavení (BVH, světlo, mlha…)
};
```

Lze vidět, že třída obsahuje stejný počet konstruktorů, jako počet uchovávaných datových typů, spolu s konstruktorem pro načtení modelu ze souboru. Každý konstruktor pak automaticky nastavuje správnou hodnotu enumerativního identifikátoru `obj_enum`.

Pomocí struktury `switch()` v metodě `hit()` je poté volána správná metoda pro výpočet průniku paprsku. Tento postup je aplikován i v ostatních metodách s využitím makra.

```
bool mesh_var::hit(const ray& r, hitrec& rec) const {
	if (!s.get_box().hit(r))return false; //Průnik AABB objektu paprskem
	switch (flag.type()) { //Switch pro vyhodnocení enumerativního typu
	case o_pol: return p.hit(r, rec); //Trojúhelník
	case o_qua: return q.hit(r, rec); //Čtyřúhelník
	case o_sph: return s.hit(r, rec); //Koule
	case o_vox: return v.hit(r, rec); //Krychle
	default: return false;
	}
}
```

2.5.4 Objekty

Objekt představuje kolekci základních primitivů tvořící jeden celek, který má nějakou společnou transformaci, materiál a ohraničující objem. V kódu je implementován jako třída s názvem `mesh` ve stejnojmenné hlavičce „mesh.h“.

Třída je postavena na C++ šabloně, která slouží k automatické generaci kódu, kdy je abstraktní datový typ, třída nebo funkce nahrazena konkrétní implementací dle vstupních argumentů šablony během kompilace. V tomto případě je argumentem obecná třída `primitive`, která je při kompilaci nahrazena jednou z konkrétních existujících tříd: `poly`, `quad`, `sphere` a `voxel`.

Třída má dva konstruktory, u prvního je vstupní argument jediný primitiv a u druhého je to větší množství primitivů uložených v dynamickém poli `vector`. Druhým vstupním argumentem je u obou konstruktorů ID materiálu. Při konstrukci je dynamicky alokována potřebná paměť pro všechny primitivy, přičemž je vytvořena interní kopie vstupních dat. Destruktor poté uvolňuje dynamicky alokovanou paměť. Metoda `hit()` iteruje skrze pole primitivů za účelem nalezení nejbližšího průsečíku. Nachází se zde také metody `fit()` a `set_trans()`, první slouží k vytvoření AABB z primitivů a druhá k jejich transformaci.

```
template <class primitive> //Šablona
class mesh {
	mesh(const primitive& _prim, uint _mat) : //Konstruktor z 1 primitivu
	prim(new primitive[1]{ _prim }), mat(_mat), size(1) { fit();}
	//Konstruktor z pole primitivů
	mesh(const vector<primitive>& _prim, uint _mat) :
	prim(new primitive[_prim.size()]), mat(_mat), size(_prim.size()) {
		memcpy(prim, _prim.data(), size * sizeof(primitive));
		fit();
	}
	bool hit(const ray& r, hitrec& rec) const; //Nejbližší průnik
	mesh& set_trans(const mat4& T); //Transformace všech primitivů
	void fit(); //Vytvoření hranic AABB
	... //Další třídní metody (pdf, rand_to, copy…)
	aabb bbox; //Ohraničující objem
	vec4 P, A; //Poloha, rotace
	primitive* prim; //Ukazatel na pole primitivů
	uint mat, size; //Index materiálu, počet primitivů
};
```

2.5.5 Materiály

Materiály jsou rovněž uloženy jako varianty s tím rozdílem, že každý typ materiálu není vlastní třídou, ale pouze funkcí. Kolekce funkcí představující materiály se pak nachází ve jmenném prostoru `material` obsaženém v souboru „material.h“.

Ve variantě je uložena textura pod názvem `albedo`, ve které je uchovávána uniformní barva nebo obrázek. Typ materiálu je poté určen pomocí enumerativního typu `mat_enum`.

```
struct mat_var {
      mat_var(const albedo& tex, mat_enum type) : tex(tex), type(type) {}
      void sample(const ray& r, const hitrec& rec, matrec& mat)const {
            switch (type) { //Switch pro volání funkcí dle typu materiálu
            case mat_mix: return material::mixed(r, rec, tex, mat);
            case mat_ggx: return material::ggx(r, rec, tex, mat);
            case mat_vnd: return material::vndf(r, rec, tex, mat);
            case mat_lig: return material::light(r, rec, tex, mat);
            case mat_las: return material::laser(r, rec, tex, mat);
            default: return;
            };
      }
      albedo tex; //Trísložková textura materiálu
      mat_enum type; //Enumerativní typ materiálu
};
```

V kódu se vyskytuje další typ záznamu – `matrec`, jenž uchovává výstupní data při vzorkování materiálu: modifikovanou normálu, bod průniku, směr odraženého paprsku, zesílení odraženého světla, emisi a enumerativní typ odraženého paprsku `refl_type`.

```
struct matrec {
      vec4 N, P, L; //Normála, počáteční bod a směr paprsku
      vec4 aten, emis; //Atenuace a emise materiálu
      float a = 0, ir = 1.f; //Hrubost a index refrakce
      uint refl = refl_none; //Typ vygenerovaného paprsku
};
enum refl_type { //Typ odrazu: žádný, difuzní, spekulární, transmisní
      refl_none, refl_diff, refl_spec, refl_tran
};
```

Nejjednodušším příkladem materiálu je poté světlo, které negeneruje žádné odražené paprsky, pouze vrací hodnotu vlastní emise. Vstupní parametry vzorkovací funkce jsou: paprsek, záznam průniku, textura a reference záznamu materiálu, jehož hodnoty upravuje. Nejdříve je vzorkována základní RGB barva textury a poté i její složka MER (*Metallic Emissive Rougness*). Do materiálového záznamu tudíž funkce zapisuje hodnotu emisního koeficientu vynásobeného základní barvou.

```
void light(const ray& r, const hitrec& rec, const albedo& tex, matrec& mat) {
      vec4 rgb = tex.rgb(rec.u, rec.v); //Základní barva textury RGB
      vec4 mer = tex.mer(rec.u, rec.v); //Kov, emise, drsnost povrchu
      float em = mer.y(); //Emisní koeficient
      mat.emis = em * rgb; //Emitované světlo
}
```

Ostatní materiály navíc vzorkují i normálovou složku textury, která umožňuje lokálně měnit směr povrchové normály, a tudíž i směr odraženého paprsku. Typ odrazu paprsku je pak závislý na metalické složce textury MER. Viz hlavičkový soubor „material.h“.

2.5.6 Renderování

V této kapitole jsou popsány důležité členské metody scény, které slouží k vykreslení obrazu pomocí algoritmu sledování cest. Hlavní metodou je pak `Render()`, která přijímá ukazatel na výstupní texturu, do níž zapisuje data vykresleného obrázku.

Před začátkem renderovací smyčky je inkrementován interní čítač senzoru za pomocí funkce `dt()`. Dále je uložen ukazatel na výstupní texturu do senzoru a je provedeno automatické ostření kamery. Následuje direktiva `#pragma omp parallel`, která slouží k automatické paralelizaci výpočtů dílčích iterací smyčky na všech jádrech procesoru.

Každá iterace smyčky počítá jeden pixel na senzoru. Ten je v případě pohybu kamery vynulován, tak aby nedocházelo k rozmazání obrazu. Dále jsou převedeny indexy `i`, `j` na xy souřadnice pixelu, ke kterým je přičtena malá náhodná hodnota v rozsahu -0,5 až 0,5, čímž je zajištěno vzorkování celé oblasti pixelu, a nikoliv pouze jeho středu, a tak je docíleno vyhlazování hran. Z daných souřadnic je poté zkonstruován paprsek metodou `optical_ray()`, která byla popsána v kapitole 2.5.1. Z paprsku je poté vypočtena barva vzorku prostřednictvím funkce `raycol()`. Vypočtený vzorek je akumulován do senzoru a zapsán do výstupní textury s použitím funkce `cam.display()`.

```
void Scene::Render(uint* disp, uint pitch) {
	if (cam.moving)cam.CCD.reset(); //Vynulování čítače v senzoru
	cam.CCD.dt(opt.samples); //Inkrementace čítače
	cam.CCD.set_disp(disp, pitch); //Nastavení výstupní textury
	cam_autofocus(); //Zaostření kamery
#pragma omp parallel for collapse(2) schedule(dynamic, 100) //Paralelizace
	for (int i = 0; i < cam.h; i++) { //Výška obrazu
		for (int j = 0; j < cam.w; j ++) { //Šířka obrazu
			if (cam.moving) cam.CCD.clear(i, j); //Vynulování pixelu
			vec4 xy = vec4(j, i) + 0.5f * ravec(); //XY souřadnice
			ray r = cam.optical_ray(xy); //Paprsek z kamery
			cam.add(i, j, raycol(r)); //Akumulace barvy v bufferu
			cam.display(i, j); //Zápis pixelu do výstupní textury
		}
	}
	cam.moving = 0; //Resetování pohybu kamery
}
```

Funkce `raycol()` akumuluje barvu pro určitý počet vzorků během jednoho snímku. Ta je následně vydělena počtem vzorků, čímž dochází k jejímu zprůměrování. Je využito optimalizace, kdy paprsky z kamery mají během jednoho snímku stejný směr, a tudíž lze první průnik scény otestovat pouze jednou a případně vrátit barvu pozadí funkcí `sky()`. Barva vzorku je určena funkcí `iterative_pt()`, která obsahuje iterativní implementaci rekurzivního algoritmu sledování cest.

```
vec4 Scene::raycol(const ray& r)const {
	vec4 col;  hitrec rec; //Akumulovaná barva, záznam průniku
	... //Volumetrické renderování
	if (!world.hit<1>(r, rec)) col = sky(r.D); //Barva pozadí
	else for (int i = 0; i < opt.samples; i++) //Akumalace vzorků
		col += opt.inv_sa * iterative_pt(r, rec, opt.bounces);
	return vec4(col, rec.t); //Vrácení barvy vzorku se vzdáleností
}
```

Iterativní algoritmus v metodě `iterative_pt()` má čtyři stavové proměnné. První proměnná představuje akumulované světlo, druhá současné zesílení světelné cesty, třetí současný sledovaný paprsek a poslední pak materiálový záznam. Při první iteraci je bod průniku již vypočítán v nadřazené funkci `raycol()`, je tudíž pouze zkopírován záznam o průniku. V ostatních iteracích je průnik vypočten v těle smyčky a v případě nenalezení průniku je vrácena barva pozadí. Rovněž je aplikována metoda ruské rulety, jež s určitou pravděpodobností terminuje sledovaný paprsek.

V případě nalezení průniku je tedy vzorkován materiál daného objektu ze seznamu materiálů. Dochází poté k akumulaci emitovaného světla vynásobeného zesílením cesty. Směr sekundárního paprsku je následně rozhodnut dle typu odrazu v záznamu `mat.refl`.

V případě žádného odrazu `refl_none` se jedná o zdroj světla, tudíž lze opustit smyčku a vrátit akumulované světlo. V dalších případech je zkonstruován nový paprsek, kdy navíc v případě difuzního a spekulárního paprsku jsou volány metody `sa_diff()` a `sa_spec()`, jež umožňují explicitní vzorkování světelných zdrojů, spolu s výpočtem vážené pravděpodobnosti `p1/p2`. Metody poté vrací vygenerovaný paprsek, jež s 50% pravděpodobností vzorkuje buď BRDF materiálu, nebo světelný zdroj.

Při odrazu je současný paprsek nahrazen tím novým, a zesílení cesty je vynásobeno zesílením (barvou) materiálu. Smyčka se tedy opakuje až do dovršení maximálního počtu odrazů, kde je pak na konci funkce vráceno celkové akumulované světlo daného vzorku.

```
vec4 iterative_pt(const ray& sr, const hitrec& srec, int depth) const {
	vec4 col(0), aten(1.f); //Akumulované světlo, zesílení cesty
	ray r = sr; matrec mat; //Současný paprsek, materiál
	for (int i = 0; i < depth + 1; i++) { //Smyčka s max. počtem odrazů
		hitrec rec; mat.refl = refl_none; //Záznam o průniku
		if (i == 0) rec = srec; //První průnik, barva pozadí
		else if (!world.hit<1>(r, rec)) return col += aten * sky(r.D);
		else if (rafl() >= opt.p_life) break; //Ruská ruleta
		else aten *= opt.i_life; //Vážení cesty
		sample_material(r, rec, mat); //Vzorek materiálu
		col += mat.emis * aten; //Akumulace světla z emise a zesílení
		if (mat.refl) { //Odraz -> mat.refl != refl_none
			if (mat.refl == refl_diff || mat.refl == refl_spec) {
				float p1, p2; //Vážená pravděpodobnost
				r = mat.refl == refl_diff ? sa_diff(mat, p1, p2)
				: sa_spec(mat, -r.D, p1, p2); //Difuzní/Spekulární
				if (p1 > 0)aten *= (p1 / p2); //Váha cesty
				else break; //Konec smyčky
			}
			else { //Transmisní paprsek
				r = ray(mat.P, mat.L, true);
			}
			aten *= mat.aten; //Násobení zesílení cesty barvou mat.
		}
		else break; //Zdroj osvětlení, konec smyčky
	}
	return col; //Vrácení akumulovaného světla
}
```

2.6 Vstupně výstupní systém

V této kapitole jsou popsány všechny třídy a funkce sloužící ke čtení a ukládání různých datových souborů z pevného disku. Program je schopen tedy zpracovávat nebo generovat soubory různých formátů, např. textové, binární a grafické.

2.6.1 Načítání modelů

Program umožňuje importování základních 3D modelů v textovém formátu OBJ, s nímž umí pracovat řada komerčních modelovacích programů, např. Maya, Rhino a Blender. Program umožňuje tyto soubory konvertovat do vlastního binárního formátu za účelem úspory a zvýšení rychlosti čtení dat z úložiště.

Ke konverzi modelů z textového do binárního formátu slouží funkce `OBJ_to_MSH()`, umístěná v souboru „mesh.cpp“, jež načítá seznam všech vrcholů `vert` a seznam všech stěn `face` z textového souboru OBJ. Vytváří poté binární soubor, do nějž jsou uloženy velikosti obou seznamů, spolu se samotnými daty v binárním formátu.

```
bool OBJ_to_MSH(path name) { //Cesta k souboru
	std::ifstream file(name); //Deklarace a načtení souboru
	if (!file.is_open()) //Soubor nebyl nalezen
		return false;
	vector<float3> vert; vert.reserve(0xffff); //Seznam vrcholů
	vector<uint3> face; face.reserve(0xffff); //Seznam stěn
	string line; line.reserve(256); //Řetezec s řádkem + rezervace paměti
	while (std::getline(file, line)) { //Načtení řádku do řetězce
	  float3 f3; uint3 u3; uint u4 = 0; //Dočasné proměnné
	  const char* cstr = line.c_str(); //Převod na řetězec charů
	  if (sscanf_s(cstr, "v %f %f %f", &f3.x, &f3.y, &f3.z) > 1) {
		vert.emplace_back(f3); //Přidání vrcholu
	  }
	  else if(sscanf_s(cstr,"f %u %u %u %u",&u3.x,&u3.y,&u3.z,&u4) > 2) {
		u3.x -= 1; u3.y -= 1; u3.z -= 1; u4 -= 1; //Nulové indexování
		face.emplace_back(u3); //Přidání stěny
		if (u4 != -1) //Triangulace v případě, že stěna má 4 vrcholy
			face.emplace_back(uint3(u3.x, u3.z, u4));
	  }
	}
	name.replace_extension(".msh"); //Otevření binárního souboru
	std::ofstream out(name, std::ios_base::binary | std::ios_base::out);
	uint vf[2] = { (uint)vert.size(),  (uint)face.size() };
	out.write((char*)vf, 2 * sizeof(uint)); //Zápis velikostí seznamů
	out.write((char*)&vert[0], sizeof(float3) * vf[0]); //Zápis bin. dat
	out.write((char*)&face[0], sizeof(uint3) * vf[1]);
	printf("Generated .msh file!\n");
	return true;
}
```

Soubory formátu MSH lze následně načíst prostřednictvím funkce `load_MSH()`, která otevře daný binární soubor a přečte první dvě hodnoty typu `uint` reprezentující velikost uložených dat. Dle velikostí alokuje paměť obou seznamů, do nichž následně překopíruje data ze souboru, a vytvoří z nich tak seznam trojúhelníků, který dále předává konstruktoru třídy `mesh::mesh(const vector<poly>& prim)`.

2.6.2 Načítání a ukládání scén

V programu je rovněž implementováno rozhrání pro ukládání a načítání souborů scén v deskriptivním textovém formátu. V souboru „Scenes.cpp“ se nachází příslušné metody `scn_save()` a `scn_load()`, které slouží k uložení a načtení souborů v textovém formátu SCN. Vstupními parametry obou metod jsou poté reference na instanci scény a textový řetězec obsahující název souboru.

Metoda `scn_save()` slouží k vytvoření anebo přepisu textového souboru scény. Pro převod různých proměnných a parametrů do textové podoby využívá funkci standardní knihovny `sprintf()`. Zápis je nejdříve proveden do dynamického pole typu `vector<ln>` obsahující jednotlivé řádky textu. Všechny řádky uložené v poli jsou poté zapsány do textového souboru prostřednictvím jediného cyklu `for()`.

```
bool scn_save(Scene& scn, const char* filename) {
	path name(u8path(filename)); //Cesta k souboru (kódování UTF-8)
	if (name.empty()) return false; //Cesta je prázdná
	std::ofstream file(name); //Otevření souboru
	if (!file.is_open()) return false; //Soubor nebyl nalezen
	ln line; vector<ln> lines; //Současný řádek a seznam řádků
	lines.push_back("*Objects"); //Přidání řádku s nadpisem objekty
	for (const auto& obj : scn.world.objects) { //Pro každý objekt scény
	  if (obj.name.empty()) { //Objekt nemá název, jsou uložena jeho data
		if (obj.type() == o_sph) {...} //Koule
		else if (...) {...} //Ostatní objekty
	  }
	  else { //Objekt obsahuje název modelu načteného ze souboru .msh
		sprintf(line,"mesh P=%g,%g,%g,%g A=%g,%g,%g mat=%d bvh=%d lig=%d
		fog=%d {%s}",...); //Naplnění obsahu řádku formát. daty objektu
		lines.push_back(line); //Přidání řádku do pole
	  }
	}
	lines.push_back("*Materials"); //Začátek materiálů
	for (const auto& mat : scn.world.materials) { //Pro všechny materiály
		sprintf(line, "albedo type=%d rgb=%s mer=%s nor=%s scl=%g ir=%g
		tint=%g,%g,%g,%g",...); //Naplnění řádku daty materiálu
		lines.push_back(line); //Přidání řádku do pole
	}
	lines.push_back("*Params"); //Začátek parametrů
	sprintf(line, "bounces=%d", scn.opt.bounces); //Načtení počtu odrazů
	lines.push_back(line); //Přidání řádku s počtem odrazů
	... //Ostatní parametry scény
	for (const auto& line : lines) //Zápise všech řádků z pole do souboru
		file << line.x << std::endl; //Zalomení řádku
	file.close(); //Uzavření souboru
	cout << "Saved scene: " << filename; //Zápis do konzole
	return true; //Operace byla úspěšná
}
```

Struktura `ln` slouží k zapouzdření pole `char[256]` pro použití v úložišti `vector`.

```
struct ln {
	ln() {}
	ln(const char* text) { strcpy_s(x, 255, text); } //Zkopírování textu
	operator char* () { return x; } //Ukazatel na data
	char x[256] = {}; //Úložiště
};
```

K načítání scén uložených v tomto formátu slouží metoda `scn_load()`. Tato metoda využívá k převodu formátovaného textu do proměnných funkci `sscanf_s()`, ve které je možno specifikovat stejný formát, jaký byl použit při zápisu dat ve funkci `sprintf()`. Načítání dat probíhá po řádek po řádku z paměťového bufferu, do něhož byl načten celý textový soubor. Každý řádek je tedy otestován vůči všem implementovaným příkazům, kdy v případě shody dochází k provedení příslušné akce. Celou implementaci lze nalézt v hlavičkovém souboru „Scenes.cpp“.

2.6.3 Ukládání obrázků

Program poskytuje možnost ukládání vykreslených obrázků do souboru ve třídní funkci `Scene::Screenshot()`, která se nachází v souboru „Scene.cpp“. Je tedy převeden obsah akumulačního bufferu `data` do pole s celočíselným formátem RGBA, které je uloženo do souboru ve formátu PNG pomocí funkce `stbi_write_png()` z knihovny STB. Rovněž lze uložit snímek obrazovky sestávající z reprojekce bodů vykresleného obrázku.

```
void Scene::Screenshot(bool reproject) const {
	int spp = cam.CCD.spp;  //Celkový počet vzorků
	uint wh = cam.CCD.n; //Celkový počet pixelů v bufferu
	uint w = cam.CCD.w; //Šířka bufferu
	uint h = cam.CCD.h; //Výška bufferu
	char name[40] = {}; //Textové pole pro uložení časové známky
	std::tm tm; time_t now = time(0);
	localtime_s(&tm, &now); //Získání systémového času
	strftime(name, sizeof(name), "%Y%m%d_%H%M%S", &tm); //Převod na text
	string file; //Název souboru
	string name_spp = string(name) + "_" + std::to_string(spp) + "SPP_";
	vector<uint> buff(wh); //Výstupní buffer – 4 složky x 8bitů - RGBA
	file = "screenshots\\" + name_spp + ".png"; //Celý název souboru
	if (reproject) { //Zápis obsahu bufferu s reprojekcí
		for (uint i = 0; i < wh; i++)
			buff[i] = vec2rgb(cam.CCD.buff[i]) | (255 << 24);
	}
	else { //Zápis obsahu bufferu s vykresleným snímkem
		for (uint i = 0; i < wh; i++)
			buff[i] = vec2rgb(cam.CCD.data[i]) | (255 << 24);

	} //Vytvoření a zápis do souboru
	stbi_write_png(file.c_str(), w, h, 4, buff.data(), 4 * w);
	cout << "Saved file in: " << file << "\n"; //Výpis stavu v konzoli
}
```

Případná data pro reprojekci jsou vygenerována ve funkci `Gen_projection()`, která je součástí třídy `Scene.` Funkce využívá data hloubky jednotlivých pixelů v akumulačním bufferu pro zkonstruování bodů v prostoru. Tyto body jsou následně promítnuty zpět na zobrazovací plochu kamery s odlišnou transformací. Je tak vytvořen trojrozměrný graf scény, který nachází využití především v moderních technikách temporálního odstranění šumu, anebo generování umělých snímků. Viz hlavičkový soubor „Scene.cpp“.

2.7 Optimalizace

Tato kapitola je rozdělena do dvou částí. V první časti jsou popsány hardwarové optimalizace SIMD vektorové třídy `vec4`. V druhé části je poté popsána implementace akcelerační struktury BVH ve třídě `obj_list`.

2.7.1 Optimalizace SIMD

Tyto optimalizace byly aplikovány v podobě intrinsických funkcí instrukční sady SSE, jež byly popsány v kapitole 1.4.2. Byla tedy modifikována základní třída vektoru `vec4` použitím unionu obsahující pole `float _xyz[4]` a SSE vektorový registr `__m128 xyz`. Použitím této syntaxe je kompilátor schopen automaticky vygenerovat potřebné instrukce pro přesun dat z paměti do registrů. Oproti definici vektoru v kapitole 2.3.1 byl do třídy přidán konstruktor z vektorového typu `__m128`.

```
struct vec4 {
	vec4(const __m128 &t) :xyz(t) {}
	... //Původní členské metody a konstruktory
	union {
		__m128 xyz; //Vektorový registr
		float _xyz[4]; //Pole v paměti
	};
};
```

Jako ukázku lze použít přetížený operátor `vec4 operator+=()`, jenž přičítá hodnoty v druhém vektoru do hodnot v prvním vektoru. V případě skalární implementace jsou tedy postupně sečteny všechny složky vektoru.

```
vec4& operator+=(const vec4 &u) {
	//Postupný součet a uložení všech složek vektoru
	xyz[0] += u.x();
	xyz[1] += u.y();
	xyz[2] += u.z();
	xyz[3] += u.w();
	return *this;
}
```

V následujícím bloku kódu lze vidět zkompilovaný strojový kód pomocí kompilátoru Clang 16.0.0 s využitím parametrů kompilace `-Ofast` a `-march=haswell` [18].

```
vmovsd  xmm0, qword ptr [rdi]         # xmm0 = mem[0],zero
vmovsd  xmm1, qword ptr [rdi + 8]     # xmm1 = mem[0],zero
vmovsd  xmm2, qword ptr [rsi]         # xmm2 = mem[0],zero
vaddps  xmm0, xmm2, xmm0              # Součet prvních 2 prvků
vmovsd  xmm2, qword ptr [rsi + 8]     # xmm2 = mem[0],zero
vaddps  xmm1, xmm2, xmm1              # Součet posledních 2 prvků
ret
```

Lze vidět, že všechny prvky jsou přesunuty z paměti do vektorových registrů `xmm0` až `xmm2` s využitím instrukcí `vmosd`. Po přesunutí je poté využito instrukcí pro vektorový součet `vaddps`. Kompilátor tedy do jisté míry provedl vektorizaci výpočtů, avšak z nevysvětlitelných důvodů generuje dvě instrukce pro součet po dvou prvcích, ačkoliv by stačila jediná instrukce pro součet čtyř prvků naráz.

Ruční optimalizace tento problém eliminují, neboť s použitím intrinsických funkcí je kompilátoru jasně předurčeno, že se jedná o vektorové výpočty. Zároveň tak dochází ke zjednodušení syntaxe většiny matematických funkcí.

```
vec4& operator+=(const vec4& u) {
	xyz = _mm_add_ps(xyz, u.xyz); //Součet všech složek vektoru
	return *this;
}
```

S použitím totožných parametrů kompilace jako v přechozím případě lze získat následující strojový kód [18].

```
vmovaps xmm0, xmmword ptr [rdi]        # Načtení dat z paměti 1. vektoru
vaddps  xmm0, xmm0, xmmword ptr [rsi]  # Součet registru a paměti 2. vektoru
ret
```

Lze vidět, že v tomto případě dochází k načtení celého vektoru do registru `xmm0` prostřednictvím instrukce `vmovaps`, a k následnému součtu obsahu registru a obsahu paměti druhého vektoru jedinou instrukcí `vaddps`.

V podobném stylu jsou optimalizovány i ostatní elementární funkce, např. `sqrt()`, `min()`, `abs()` a mnoho dalších. Implementace vlastních intrinsických funkcí lze poté nalézt v hlavičkovém souboru „SSE.h“. Mezi skalární a SSE verzí lze přepínat při kompilaci programu prostřednictvím makra `#define USE_SSE` v souboru „defines.h“.

2.7.2 Akcelerační struktura BVH

Akcelerační struktura BVH je v programu implementována ve třídě `obj_list` za pomocí dvou dynamických polí `vector<bvh_node>` a `vector<mesh_raw>`, viz kapitola 2.5.2.

Datové typy

Struktura `bvh_node` reprezentuje jeden uzel stromové struktury BVH. Obsahuje tedy ohraničující objem, ukazatele na další uzly a indikátor koncového uzlu. Viz „obj.h“.

```
struct bvh_node {
	bvh_node(aabb bbox, uint n1, uint n2, bool parent) :
	bbox(bbox), n1(n1), n2(n2), parent(parent) {}
	aabb bbox; //Ohraničující objem uzlu
	uint n1, n2; //Levý a pravý potomek, nebo rozsah primitivů
	bool parent; //Koncový uzel -> parent == 0
};
```

Další strukturou je `mesh_raw`, která slouží jako ukazatel na konkrétní primitivy v jednotlivých objektech. Obsahuje tedy index objektu a index primitivu. Indexy jsou použity namísto ukazatelů, neboť ukazatele nelze spolehlivě použít na objekty umístěné v dynamickém úložišti `vector`. Třídní metody poté využívají ukazatele na pole objektů, do něhož indexují prostřednictvím uložených indexů. Viz „mesh.h“.

```
struct mesh_raw {
	mesh_raw(aabb bbox, uint obje_id, uint prim_id) :
	obje_id(obje_id), prim_id(prim_id) {}
	bool hit(const mesh_var* obj, const ray& r, hitrec& rec) const;
	float pdf(const mesh_var* obj, const ray& r) const;
	aabb get_box(const mesh_var* obj)const; //Ukazatel na pole objektů
	uint obje_id; //Index objektu
	uint prim_id; //Index primitivu v objektu
};
```

Konstrukce struktury

Při konstrukci struktury je nejdříve naplněn seznam všech indexů `vector<mesh_raw> obj_bvh` ve funkci `obj_create()`. Funkce iteruje skrze všechny objekty a v případě, že objekt patří do BVH, jsou do pole přidány indexy ukazující na všechny jeho primitivy.

```
void obj_list::obj_create() {
	obj_bvh.clear(); //Vyčištění seznamu indexů
	if (objects.empty()) return; //Scéna neobsahuje žádné objekty
	for (uint i = 0; i < objects.size(); i++) { //Všechny objekty
		if (objects[i].bvh()) { //Pokud má objekt patřit do BVH
		//Je přidán záznam indexů pro všechny primitivy v objektu
		for (uint j = 0; j < objects[i].get_size(); j++)
		//Přidání záznamu s AABB, indexem objektu a indexem primitivu
		obj_bvh.emplace_back(objects[i].get_box(j), i, j);
		}
	}
}
```

Dále je volána funkce pro zkonstruování samotné struktury BVH, jejíž uzly jsou uloženy v dynamickém poli `vector<bvh_node> bvh`. Metoda vytváří první uzel, jenž obsahuje celý ohraničující objem všech objektů a indexy ukazující na začátek a konec seznamu primitivů. Je poté volána metoda `split_bvh()`, která rekurzivně dělí tento uzel.

```
void obj_list::bvh_builder(bool print, uint node_size) {
	bvh.clear(); //Vyčištění seznamu uzlů
	if (obj_bvh.empty()) return; //Seznam indexů je prázdný
	bvh.reserve(obj_bvh.size()); //Rezervace paměti
	//Přidání prvního uzlu obsahující všechny objekty
	bvh.emplace_back(box_from(0, obj_bvh.size()), 0, obj_bvh.size(), 0);
	split_bvh(0, node_size); //Rekurzivní dělení prvního uzlu
}
```

Metoda `split_bvh()` rekurzivně dělí nadřazený uzel do dvou potomků na základě nejnižší ceny vypočtené pomocí funkce `split_cost()`. V případě, že vypočtená cena obou potomků je nižší než cena rodiče a počet primitivů v uzlu je větší než minimální počet, dochází k přidání obou potomků do seznamu, přičemž je rekurzivně volána metoda `split_bvh()`. V druhém případě se jedná o koncový uzel, který nemá žádné potomky.

```
void obj_list::split_bvh(uint node, uint node_size) {
	uint be = bvh[node].n1, en = bvh[node].n2; //Krajní indexy primitivů
	uint size = en - be, mi = (be + en) / 2;  //Velikost, střed
	if (size > node_size) { //Počet primitivů je větší než min. počet
	  float pcost = bvh[node].bbox.area() * size; //Cena SAH rodiče
	  float cost = split_cost(be, en, mi, bvh[node].bbox); //SAH potomků
	  if (mi > be && mi < en && cost < pcost) { //Minimalizace SAH
	    bvh[node].parent = true; //Rodičovský uzel
	    bvh[node].n1 = bvh.size(); //Index levého uzlu v poli
	    bvh[node].n2 = bvh[node].n1 + 1; //Index pravého uzlu v poli
	    //Přidání levého a pravého uzlu do pole
	    bvh.emplace_back(box_from(be, mi), be, mi, 0);
	    bvh.emplace_back(box_from(mi, en), mi, en, 0);
	    split_bvh(bvh[node].n1, node_size); //Dělení levého uzlu
	    split_bvh(bvh[node].n2, node_size); //Dělení pravého uzlu
	  }
	} //Uzel implicitně obsahuje rozsah primitivů
}
```

Procházení struktury

K procházení struktury a nalezení nejbližšího bodu průniku paprsku s primitivy, slouží metoda `traverse_bvh()`, která je součástí třídy `obj_list`. Tato metoda rekurzivně prochází stromovou strukturu BVH, kdy pro každý nadřazený uzel kontroluje průniky s ohraničujícími objemy AABB obou jeho potomků. V případě průniku s AABB jsou poté rekurzivně ověřeny i další podléhající uzly. Je zde využito metody postupného procházení, která prioritně navštěvuje nejbližší uzly. Algoritmus se opakuje po dosažení všech koncových uzlů s primitivy, z nichž je poté vybrán nejbližší bod průniku.

```
uchar traverse_bvh(const ray& r, hitrec& rec, uint n0 = 0) const {
	const bvh_node& node = bvh[n0]; //Alias současného uzlu
	if (node.parent) //Průběžný uzel, odkazuje se na další uzly
	{
		float t1 = rec.t;
		float t2 = rec.t;
		//Ověření průniku obou potomků, spolu se vzdálenostmi t1 a t2
		bool h1 = bvh[node.n1].bbox.hit(r, t1);
		bool h2 = bvh[node.n2].bbox.hit(r, t2);
		if (h1 && h2) //Paprsek protíná oba uzly
		{
			//Seřazení uzlů dle vzdálenosti
			bool swap = t2 < t1;
			uint n1 = swap ? node.n2 : node.n1;
			uint n2 = swap ? node.n1 : node.n2;
			float t = swap ? t1 : t2;
			//Procházení nejbližšího uzlu
			uchar first = traverse_bvh(r, rec, n1);
			//Nalezený průnik je blíže než hranice 2. AABB
			if (first && rec.t < t) return 1;
			//Procházení vzdálenějšího uzlu
			else return traverse_bvh(r, rec, n2) | first;
		}
		//Paprsek protíná pouze jeden uzel
		else if (h1)return traverse_bvh(r, rec, node.n1);
		else if (h2)return traverse_bvh(r, rec, node.n2);
		else return 0; //Paprsek neprotíná žádný uzel
	}
	else //Koncový uzel, obsahuje rozsah primitivů
	{
		uchar any_hit = 0;
		//Metoda mesh_raw::hit() interně přepisuje rec.idx a rec.t
		for (uint i = node.n1; i < node.n2; i++)
			any_hit |= obj_bvh[i].hit(objects.data(), r, rec);
		return any_hit; //Průnik byl nalezen
	}
}
```

Funkce je použita v kombinaci s testováním průniků objektů vyřazených ze struktury BVH ve třídní metodě `obj_list::hit()`, jež byla uvedena v kapitole 2.5.2.

3. VÝSTUPY PROGRAMU

V této kapitole jsou popsány hlavní funkce zhotoveného programu, včetně jeho ovládání i hlavních charakteristik vygenerovaných obrázků, spolu s měřením výkonu programu při použití různých optimalizací.

3.1 Hlavní funkce programu

V této kapitole jsou jmenovány hlavní funkce programu, např. renderování a úpravy v reálném čase, načítání a ukládání souborů anebo obecné optimalizace programu.

3.1.1 Náhled v reálném čase

Hlavní funkcí programu je generování obrázků pomocí metody sledování cest v reálném čase. V tomto ohledu nabízí program náhledové okno s vysokým rozlišením, které je schopno reflektovat okamžité změny ve virtuálním světě, např. transformace objektů, úpravy materiálů, nastavení pozadí a volumetrické mlhy, změny parametrů vykreslování i kamery a spoustu dalších.

Program rovněž nabízí řadu diagnostických nástrojů, např. monitorování a měření průměrné snímkové frekvence a rychlosti renderování, omezení maximální snímkové frekvence a maximálního počtu vykreslených vzorků, anebo pozastavení vykreslování. Je také umožněna vizualizace interních dat algoritmu sledování cest v podobě světových normál, UV souřadnic textur, hran primitivů, vzdáleností průniků, struktury BVH, anebo zesílení jednotlivých materiálů. Na zobrazený snímek je rovněž možno aplikovat základní mediánový filtr pro redukci šumu, jenž má minimální dopad na rychlost vykreslování.

V neposlední řadě program nabízí schopnost trojrozměrné reprojekce, která využívá data barvy a hloubky již vykreslených pixelů k sestrojení interaktivního trojrozměrného grafu obrazovky, ve kterém se lze volně pohybovat s kamerou.

3.1.2 Načítání a ukládání dat

Program v prvé řadě umožňuje ukládání a načítání scén v textovém formátu SCN. Je využito intuitivních příkazů, s velmi jednoduchou syntaxí, které umožňují definování parametrů renderování, popis jednotlivých objektů a materiálů přímo v souboru, anebo odkazování na externí soubory obsahující modely i textury. Příklad takového souboru lze nalézt v příloze C, v níž je zachycen textový popis scény „Bent_Fiber.scn“, která slouží k vykreslení šesti zahnutých optických vláken s rozdílnými indexy lomu.

V programu lze dále načítat trojrozměrné modely anebo textury. V případě modelů je podporován textový formát OBJ, jenž je programem převeden na kompaktní binární soubor vlastního formátu MSH. Textury lze číst v téměř jakémkoliv grafickém formátu díky použití knihovny STB [21].

Ukládání obrázků je v programu realizováno zachycením a následným převedením obsahu akumulačního bufferu obrazového senzoru do souboru grafického formátu PNG. Při převodu je rovněž aplikován filtr pro redukci šumu, čímž je docíleno toho, že je uložen přesně ten stejný obrázek, jaký byl zobrazen v náhledu. Program tímto způsobem tedy napodobuje funkci reálného fotoaparátu.

Pro načítání souborů formátu OBJ, MSH a SCN lze rovněž využít funkci přetažení souborů z operačního systému (*drag and drop*). Program poté automaticky rozhodne, o jaký typ souboru se jedná a provede příslušnou akci.

3.1.3 Možnosti úprav

Program obsahuje osm přednastavených scén, sestávajících z jednoduchých primitivů i celistvých polygonových modelů, načtených z přiložených souborů formátu MSH [24].

Všechny objekty a na ně vázané materiály lze po označení upravovat v kontextové nabídce. U objektů lze měnit jejich polohu, rotaci a přiřazený materiál. Je s nimi možno také pohybovat v náhledu prostřednictvím myši.

U materiálů lze volit jejich typ, použité textury, fyzikální vlastnosti i index refrakce. Jednotlivé textury materiálů mohou poté obsahovat solidní barvy i obrázkové textury s nastavitelným měřítkem, mezi nimiž lze přepínat pomocí zaškrtávacích políček.

Nové objekty i materiály lze přidávat do scény pomocí dvou kontextových nabídek, které umožňují vytváření jednoduchých objektů a textur, anebo načítání trojrozměrných modelů i obrázků textur ze souborů. Alternativně lze v hlavním menu duplikovat anebo mazat již existující položky objektů i materiálů.

Program rovněž obsahuje matematický model pozadí sestávající z oblohy a slunce, spolu se simulací rozptylu světla v atmosféře v podobě volumetrické mlhy. U pozadí lze měnit celou řadu parametrů, čímž je možno ovlivnit způsob osvětlení scény. V kombinaci s nastavením hustoty a barvy volumetrické mlhy lze pak dosáhnout velmi autenticky vypadajících obrázků. U pozadí scény lze rovněž nastavit enviromentální texturu.

V programu jsou dále implementovány nastavení kamery, které slouží k úpravám finální podoby obrázku. Lze tedy nastavit např. rozlišení, expozici, zorné pole, hloubku ostrosti anebo vzdálenost ostření.

Další nastavení poté slouží k nastavení kvality a výkonu renderování, lze tak nastavit počet odrazů paprsků, počet vzorků na pixel, rozlišení snímku, taktiku vzorkování anebo míru redukce šumu zobrazeného obrázku.

3.1.4 Optimalizace kódu

V programu je aplikována rozsáhlá řada optimalizací, díky kterým je schopen v reálném čase vykreslovat obrázky vysokého rozlišení bez větších kompromisů.

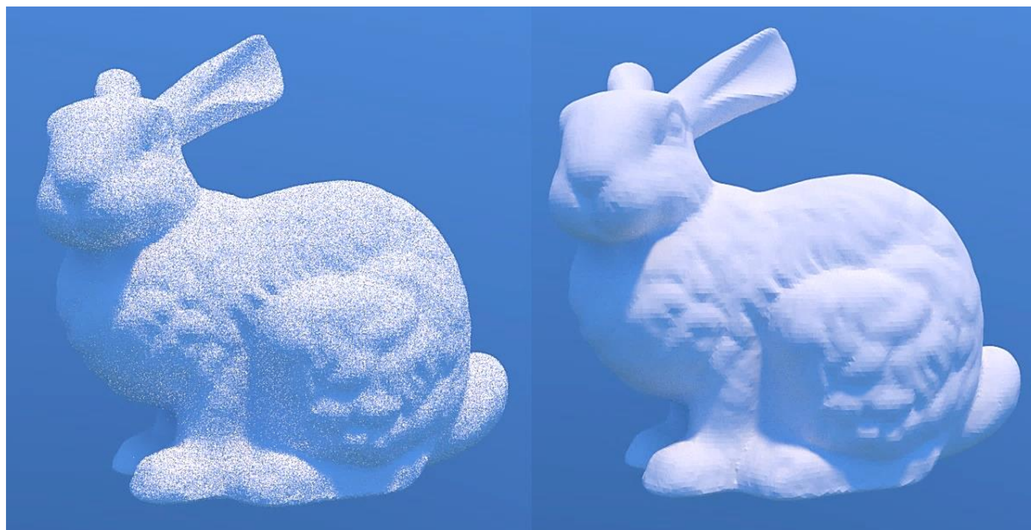
V prvé řadě se jedná o multithreading. Hlavní smyčka pro generaci obrázku je tedy rozdělena do částí, jejichž výpočty jsou distribuovány na všech jádrech procesoru. Dochází tak k téměř lineárnímu škálování výkonu vzhledem k počtu dostupných jader.

Program rovněž umožňuje dvě hybridní metody vykreslování. První metoda slouží k akceleraci testování průniků paprsků pocházejících z kamery, kdy pro větší počet vzorků je použit stejný záznam o průniku. Druhá metoda urychluje renderování tím, že vykresluje pouze polovinu pixelů v šachovnicovém vzoru a pro doplnění zbývajících pixelů využívá algoritmu založeného na výpočtu mediánu okolních pixelů.

Dále jsou implementovány optimalizace výpočtů v podobě instrukční sady SSE, která redukuje počet instrukcí potřebných k realizaci matematických operací s vektory, čímž je dosaženo výrazného urychlení výpočtů. Je také využito aproximačních matematických funkcí uzpůsobených pro použití s instrukční sadou SSE. Lze tak například vypočíst goniometrické funkce čtyř různých proměnných naráz.

Samotný algoritmus sledování cest je urychlen pomocí akcelerační struktury BVH, jež umožňuje logaritmicky snížit počet testovaných primitivů při hledání nejbližšího bodu průniku scény s paprskem. S použitím této struktury lze tak v reálném čase vykreslovat scény obsahující i miliony trojúhelníku. Toto tvrzení si lze potvrdit v programu v přednastavené scéně č. 7, v níž se nachází trojrozměrný model obsahující 250 tisíc trojúhelníků. Při zapnutém algoritmu BVH je jeden snímek vykreslen za 11 ms. V případě jeho vypnutí trvá vykreslení jediného snímku přes 100 s. Podrobná data měření rychlosti vykreslování při různém počtu primitivů lze nalézt v kapitole 3.4.

V programu je dále implementováno explicitní vzorkování světelných zdrojů, čímž je zvýšena efektivita integrace metodou Monte Carlo. Pro dosažení stejné kvality obrazu je tedy potřeba vypočíst výrazně nižší celkový počet vzorků. Rozdíl kvality pro explicitní vzorkování světel a běžné implicitní vzorkování lze vidět na následujícím obrázku.



Obrázek 3.1 Porovnání kvality obrazu pro implicitní a explicitní vzorkování

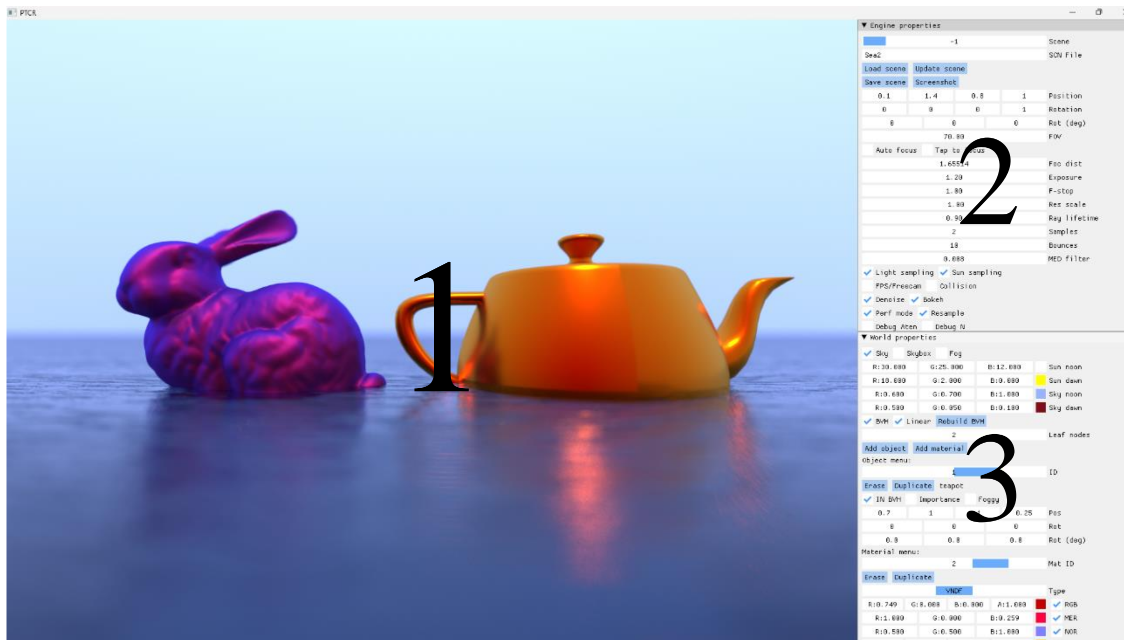
Výše uvedený snímek byl pořízen pro tisíc vzorků pro oba typy vzorkování. Lze vidět, že s použitím explicitního vzorkování světelného zdroje slunce (obrázek vpravo) došlo k výraznému poklesu šumu na okem téměř nepostřehnutelnou úroveň.

3.2 Uživatelské prostředí programu

V této kapitole je představeno uživatelské prostředí a základní ovládání programu.

3.2.1 Rozložení okna

Okno programu má fixní rozlišení 1280x720 pixelů, přičemž plocha 960x720 pixelů je vyhrazena pro náhled v reálném čase a zbylá část obrazovky pak obsahuje ovládací prvky.



Obrázek 3.2 Rozložení okna programu

Legenda k obrázku 3.2:

1) Náhledové okno (*viewport*)
2) Menu pro načtení scény, nastavení kamery a kvality obrazu a monitorování výkonu
3) Menu pro nastavení parametrů scény, BVH a zvoleného objektu

V druhém okně jsou obsaženy všechny nastavení grafického enginu. Lze tady najít například posuvník anebo textové pole určené k načítání scén. Dále jsou zde obsaženy posuvníky pro různá nastavení kamery, např. polohu a rotaci, velikost zorného pole, ostřící vzdálenost, expozici anebo světelnost objektivu. Ostatní ovládací prvky umožňují nastavení kvality renderování, např. počet vzorků a odrazů, intenzitu filtru, anebo nastavení různých vzorkovacích strategií. Nakonec jsou zde obsaženy přepínače pro zobrazení diagnostických dat, spolu s měřením průměrného a okamžitého výkonu.

Ve třetím okně lze najít různá nastavení parametrů světa, např. parametry pozadí, volumetrické mlhy, struktury BVH, anebo tlačítka pro přidání nových objektů nebo materiálů. Rovněž se zde nachází posuvník pro zvolení konkrétního objektu scény. U zvoleného objektu lze upravovat jeho interní vlastnosti, např. transformaci, explicitní vzorkování anebo přiřazený materiál. U materiálů lze poté upravovat jejich typ, použité textury, měřítko, index refrakce a zabarvení.

3.2.2 Ovládání

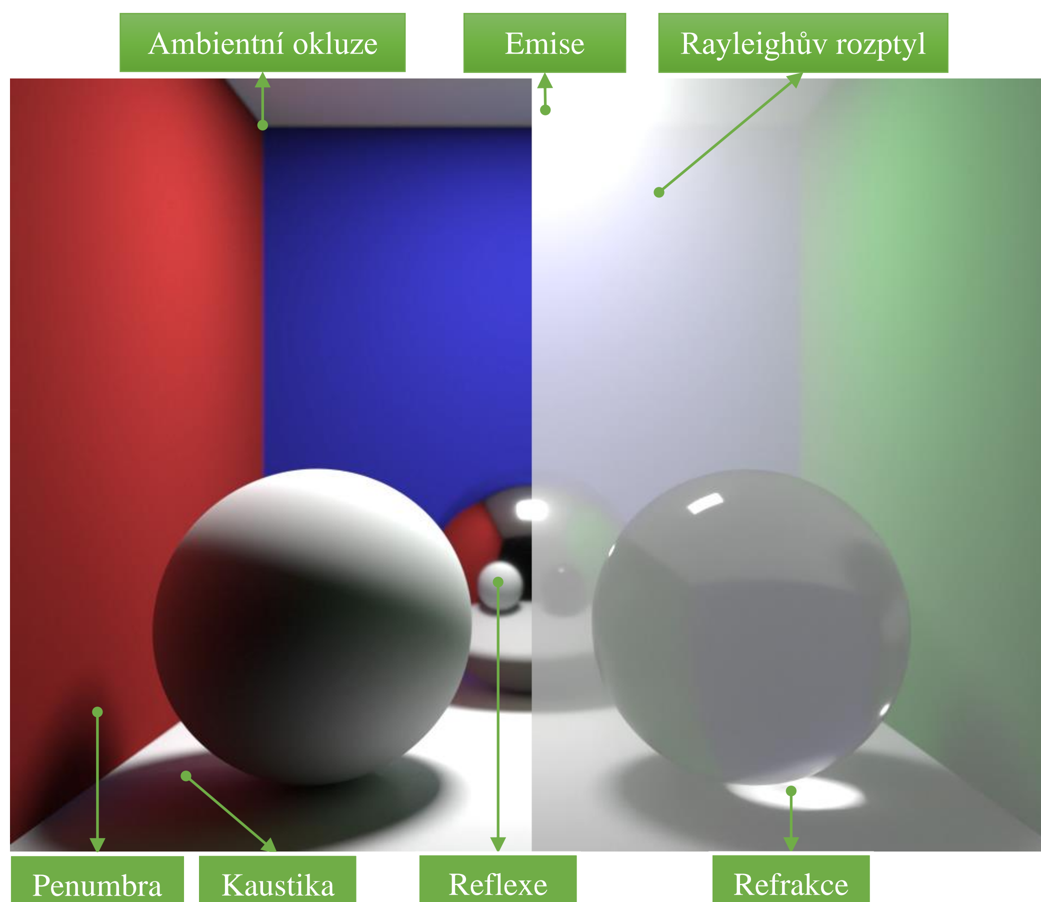
K ovládání programu je využito klávesnice a myši. Klávesnice slouží převážně k pohybu kamery a případnému zadávání číselných hodnot do ovládacích prvků. Myš poté slouží k ovládání rotace kamery i pozice zvolených objektů, včetně ovládacích prvků v menu. V následující tabulce jsou uvedena základní ovládání programu.

Tabulka 3.1 Seznam ovládání programu

Klávesa	Akce	Klávesa	Akce
WASD	Horizontální pohyb	SPACE / LCTRL	Vertikální pohyb
Šipky / LMB	Rozhlížení	LSHIFT	Urychlení pohybu
MMB / RMB+ALT	Označení objektu	RMB	Pohyb objektem
F1	Zapnutí překrytí	ESC	Konec

3.3 Generované obrázky

K popisu charakteristik obrázků generovaných programem byl použit snímek obrazovky přednastavené scény č. 6. Na následujícím obrázku lze tedy vidět popis všech fyzikálních jevů, které je program schopen simulovat s využitím algoritmu sledování cest.



Obrázek 3.3 Fyzikální vlastnosti obrázku vygenerovaného programem

3.4 Měření výkonu

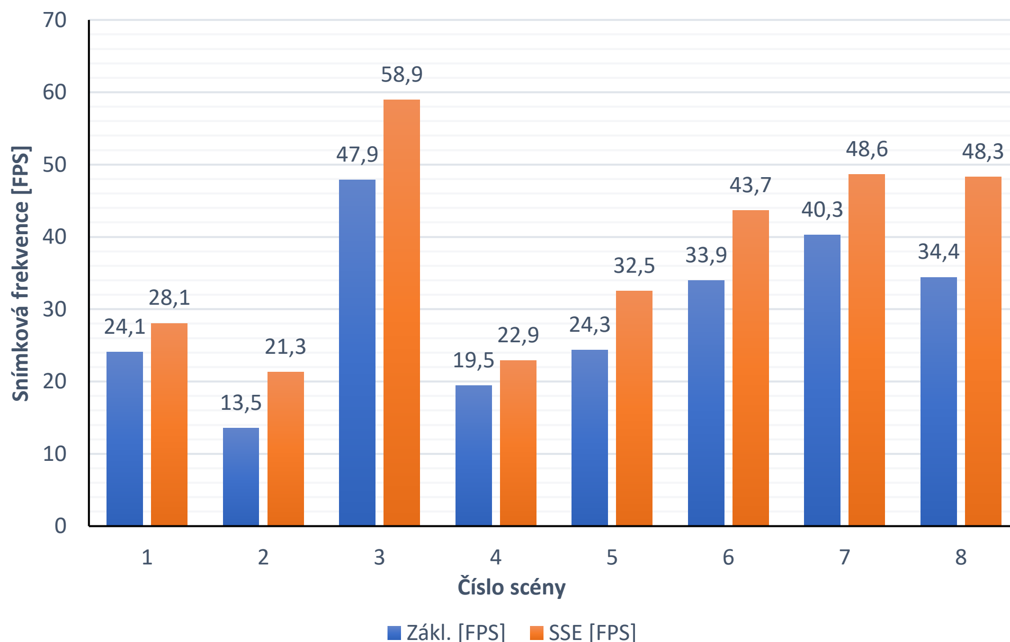
V této kapitole jsou uvedeny výsledky měření rychlosti renderování programu při použití optimalizací SIMD i struktury BVH. Oba typy optimalizací jsou následně porovnány se základní verzí. K měření byl použit PC s následujícími parametry: AMD Ryzen 7 5800H 8C/16T CPU, 32 GB 3200 MHz DDR4 RAM. Program byl zkompilován prostřednictvím kompilátoru Clang 15.0.1 s konfigurací `-Ofast -march=haswell`.

3.4.1 Měření výkonu s použitím optimalizací SIMD

Měření výkonu bylo provedeno s použitím přednastavených scén v programu a definicí makra `#define TEST 1` v souboru „defines.h“, které vypíná hloubku ostrosti a hybridní renderování. Byla změřena snímková frekvence (FPS) základní a optimalizované verze programu, včetně výpočtu procentuálního poměru výkonu obou verzí. Data byla získána zprůměrováním času vykreslení 500 snímků pomocí třídní metody `Meas_fps()` ve třídě `Engine` a odečtena z kontextové nabídky programu.

Tabulka 3.2 Naměřená data snímkových frekvencí s použitím optimalizací SIMD

Scéna	1	2	3	4	5	6	7	8	Průměr
Poč. primitivů	75772	8	121	75773	9	5	249882	4	50197
Zákl. [FPS]	24,1	13,5	47,9	19,5	24,3	33,9	40,3	34,4	29,7
SSE [FPS]	28,1	21,3	58,9	22,9	32,5	43,7	48,6	48,3	38,0
Poměr [%]	116,6	157,6	123,1	117,8	133,6	128,6	120,8	140,3	129,8



Obrázek 3.4 Graf snímkových frekvencí s použitím optimalizací SIMD

Z naměřených dat lze vidět, že optimalizovaná verze je rychlejší než základní verze ve všech měřených scénách. Průměrný nárust snímkové frekvence tedy činí 29,8 %. Nejvyšší nárůst 57,6 % je možno pozorovat u scény č. 2, která obsahuje malý počet

primitivů, avšak uzavřený prostor s vysokým počtem odrazů. Naopak nejnižší přínos lze pozorovat u scén, které obsahují vysoký počet primitivů v otevřeném prostoru (1, 4, 7).

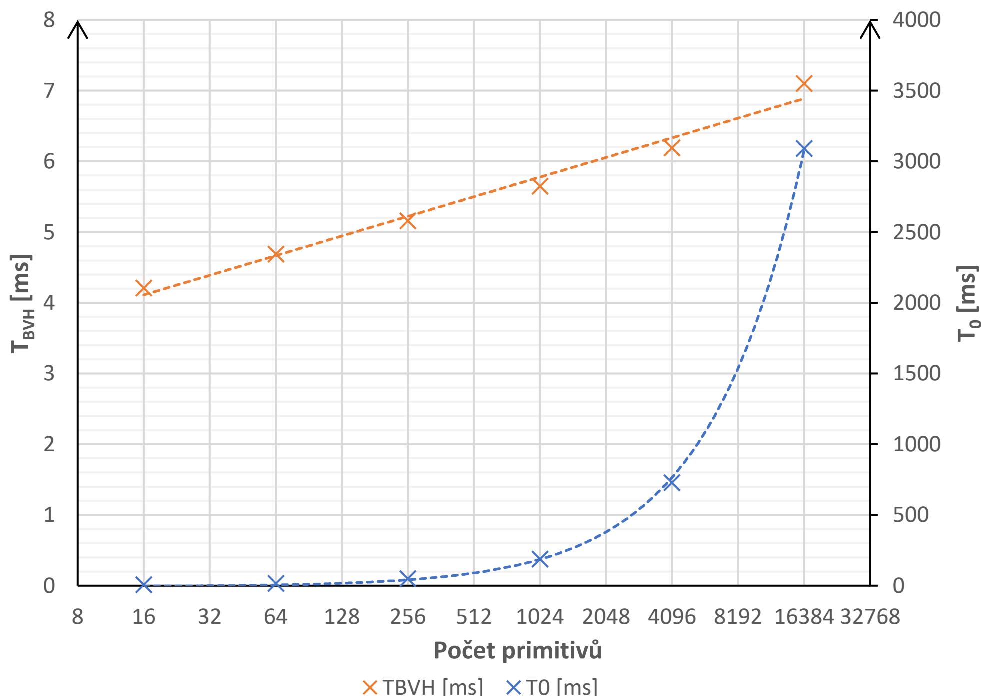
Existuje tedy jistá míra korelace mezi nárůstem výkonu a strukturou scény. Nejvyšší nárůst výkonu lze pozorovat u scén s nízkým počtem primitivů, kde většinu výpočetního času stráví proces u hrubých vektorových výpočtů průniků, odrazů anebo osvětlení. Naopak u scén s vysokým počtem primitivů je většina výpočetního času strávena při procházení struktury BVH, kde je rychlost procesu limitována především prostupností paměti, optimalizace vektorových výpočtů tudíž nemají skoro žádný vliv na výkon.

3.4.2 Měření výkonu s použitím struktury BVH

V tomto měření bylo využito scén obsahující čtvercový útvar, s rostoucím počtem dílčích primitivů, zabírající konstantní plochu na obrazovce. Použité scény lze nalézt ve složce programu pod názvy dle počtu obsažených primitivů „bvh16.scn, bvh64.scn, …“. Jako v předchozím měření bylo použito makra `TEST` a interní metody pro měření průměrného času vykreslení 500 snímků. Bylo rovněž použito diagnostické vykreslení vzdálenosti namísto klasické metody renderování, tak aby byl eliminován vliv ostatních výpočtů, např. osvětlení anebo odrazů, na výsledky měření.

Tabulka 3.3 Naměřená data času vykreslení snímků s použitím struktury BVH

Poč. primitivů	16	64	256	1024	4096	16384
T_0 [ms]	5,76	14,7	49,4	187,0	727,8	3090,1
T_{BVH} [ms]	4,21	4,7	5,2	5,7	6,2	7,1
T_0/T_{BVH} [-]	1,4	3,1	9,6	33,1	117,6	435,2

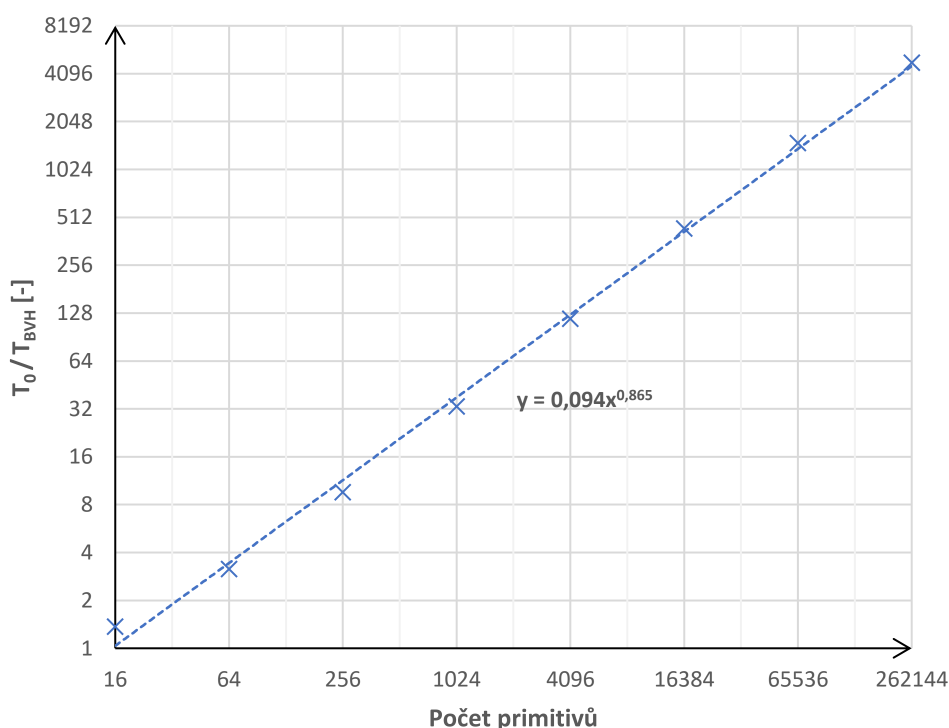


Obrázek 3.5 Graf závislosti času vykreslení snímků na počtu primitivů

Z naměřených dat lze pozorovat, že čas potřebný k vykreslení jednoho snímku roste lineárně s počtem primitivů u neoptimalizované verze programu. S použitím struktury BVH tento čas roste logaritmicky. Vzhledem k logaritmickému měřítku horizontální osy se tyto závislosti jeví v grafu jako exponenciála a přímka. Měřením tedy bylo ověřeno původní tvrzení o rozdílných asymptotických časových komplexitách obou algoritmů, jež bylo uvedeno v teoretickém úvodu práce v kapitole 1.4.1.

Rozdíl v rychlostech vykreslování obou verzí programu tedy drasticky roste s počtem primitivů. Ve scéně s nejnižším počtem primitivů je optimalizovaná verze pouze 1,4 krát rychlejší, neboť vykreslení snímků obou verzí trvá v řádu několika milisekund. U scény s nejvyšším počtem primitivů je však optimalizovaná verze již 435 krát rychlejší, neboť u ní trvá vykreslení snímku 7 ms, kdežto u základní verze tento čas činí přes 3 sekundy.

Na následujícím obrázku lze vidět graf poměru rychlosti vykreslování obou algoritmů v závislosti na počtu primitivů, spolu se spojnicí trendu v mocninném tvaru.



Obrázek 3.6 Graf závislosti poměru rychlosti vykreslování na počtu primitivů

Z grafu lze pozorovat, že závislost roste přibližně s odmocninou počtu primitivů, její průběh lze tedy popsat funkcí trendu $y = 0{,}094 \cdot x^{0{,}865}$. Tento vztah je možno použít k extrapolaci předpokládaného nárůstu výkonu i pro vyšší počet vykreslených primitivů.

4. ZÁVĚR

V práci byl popsán úvod do problematiky počítačové grafiky, včetně základů fyzikálního oboru radiometrie. Byly vysvětleny principy Lambertova zákona a zákona inverzní druhé mocniny, na nichž je založena podstata zobrazovací rovnice. Byly představeny známé metody řešení zobrazovací rovnice používané v průmyslu, přičemž hlavní zaměření bylo na popis algoritmu sledování cest založeného na integraci metodou Monte Carlo, u níž byl popsán princip vzorkování různých BRDF. Dále byly popsány základní hardwarové i softwarové optimalizace v podobě technik SIMD i různých typů akceleračních struktur. Podrobněji bylo poté popsáno využití instrukční sady SSE prostřednictvím intrinsických funkcí a akcelerační struktury BVH včetně známých algoritmů dělení prostoru.

Teoretické poznatky byly aplikovány při tvorbě počítačového programu, důsledkem čehož byl implementován algoritmus sledování cest, umožňující trasování světelných paprsků ve scéně, za současného výpočtu řešení integrálu zobrazovací rovnice. Výsledky algoritmu byly následně zpracovány do grafické podoby a zobrazeny na obrazovce.

Pro reprezentaci vstupních dat algoritmu byla zkonstruována třídní hierarchie scény obsahující pozadí, přenosové médium, kameru a seznam objektů skládajících se z dílčích primitivů. Byl taktéž implementován systém materiálů umožňující simulaci různých typů interakcí světla s povrchy objektů.

V programu bylo implementováno 8 přednastavených scén určených k demonstraci jeho hlavních funkcí. Bylo vytvořeno uživatelsky přívětivé prostředí s ovládacími prvky umožňující úpravy parametrů renderování i zvolených objektů, včetně jejich materiálů, v reálném čase. Rovněž byl vytvořen vstupně výstupní systém umožňující načítání textur, trojrozměrných modelů i souborů scén ve vlastním textovém formátu.

V programu byly dále implementovány specifické optimalizace architektury x86-64 s použitím instrukční sady SSE pro akceleraci vektorových počtů. V případě optimalizací algoritmu sledování cest byla implementována akcelerační struktura BVH sloužící ke snížení efektivního počtu vypočtených průniků paprsků s geometrií.

S použitím optimalizací SIMD bylo možno dosáhnout až 60% nárůstu snímkové frekvence oproti základní verzi dle typu zobrazené scény. U optimalizací struktury BVH poté byla ověřena časová komplexita obou algoritmů, jejíž závislost byla u základní verze lineární a u verze využívající struktury BVH logaritmická. Výkonnostní poměr obou verzí byl následně vynesen do grafu, z něhož byla odečtena mocninná funkce trendu.

Tato práce úspěšně splnila všechny požadavky uvedené v zadání, včetně všech cílů, které byly dodatečně stanoveny při tvorbě semestrální práce. V budoucnu by bylo možno program dále upravit a rozšířit o nové funkce, např. o podporu instancování objektů, spolu s rozdělením akcelerační struktury do úrovní. Tyto funkce by následně umožnily zvýšení komplexity scén za současného snížení využití paměti. V neposlední řadě by bylo možno přepsat zdrojový kód programu v jazyku CUDA, jenž slouží k programování grafických procesorů, které poskytují řádově vyšší výpočetní výkon než běžné procesory.

LITERATURA

[1] BAUMRUK, Vladimír. *Fotometrie a radiometrie* [online]. Praha: Univerzita Karlova, 2016 [cit. 2022-10-29]. Dostupné z: http://fu.mff.cuni.cz/biomolecules/media/files/courses/Fotometrie_a_radiometrie.pdf

[2] OHNO, Yoshi. *OSA Handbook of Optics, Volume III Visual Optics and Vision Chapter for Photometry and Radiometry* [online]. Maryland USA: National Institute of Standards and Technology, 1999 [cit. 2022-10-30]. Dostupné z: https://www.physics.muni.cz/~jancely/PPL/Texty/IntegracniKoule/Photometry%20and%20Radiometry.pdf

[3] KAJIYA, James T. The rendering equation. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86* [online]. New York, USA: ACM Press, 1986, s. 143-150 [cit. 2022-10-30]. ISBN 0897911962. DOI: 10.1145/15922.15902. Dostupné z: https://dl.acm.org/doi/pdf/10.1145/15922.15902

[4] IMMEL, David S., Michael F. COHEN a Donald P. GREENBERG. A radiosity method for non-diffuse environments. *ACM SIGGRAPH Computer Graphics* [online]. 1986, 20(4), 133-142 [cit. 2022-10-30]. ISSN 0097-8930. DOI: 10.1145/15886.15901. Dostupné z: https://dl.acm.org/doi/pdf/10.1145/15886.15901

[5] KŘIVÁNEK, Jaroslav. *Computer graphics III – Rendering equation and its solution* [online]. Praha: Univerzita Karlova, 2015 [cit. 2022-10-30]. Dostupné z: https://cgg.mff.cuni.cz/~jaroslav/teaching/2015-npgr010/slides/07%20-%20npgr010-2015%20-%20rendering%20equation.pdf

[6] SCRATCHAPIXEL. *Rasterization: a Practical Implementation* [online]. @2009-2022 [cit. 2023-05-20]. Dostupné z: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html

[7] WHITTED, Turner. An improved illumination model for shaded display. *Communications of the ACM* [online]. 1980, 23(6), 343-349 [cit. 2022-11-13]. ISSN 0001-0782. DOI: 10.1145/358876.358882. Dostupné z: https://dl.acm.org/doi/pdf/10.1145/358876.358882

[8] JAROSZ, Wojciech. *Efficient Monte Carlo methods for light transport in scattering media* [online]. San Diego, USA: University of California, 2008 [cit. 2022-11-18]. Dostupné z: https://cs.dartmouth.edu/~wjarosz/publications/dissertation/dissertation-web.pdf

[9] HEITZ, Eric. Sampling the GGX Distribution of Visible Normals. *Journal of Computer Graphics Techniques (JCGT)* [online]. vol. 7, no. 4, 1-13, 2018 [cit. 2023-04-13]. Dostupné z: http://jcgt.org/published/0007/04/01/

[10] CELAREK, Adam. *Rendering: The Rendering Equation* [online]. Austria: TU Wien, 2020 [cit. 2023-04-14]. Dostupné z:

https://www.cg.tuwien.ac.at/courses/Rendering/2020/slides/04_The_Rendering_Equation_v20200515.pdf
[11] SUBRAMANIAN, K.R. *Spatial Data Structures and Acceleration Algorithms* [online]. North Carolina, USA: UNC Charlotte, 2006 [cit. 2023-04-16]. Dostupné z: https://webpages.charlotte.edu/krs/courses/5010/ged/lectures/spatial1.pdf
[12] KERBL, Bernhard. *Rendering: Spatial Acceleration Structures* [online]. Austria: TU Wien, 2020 [cit. 2023-04-16]. Dostupné z: https://www.cg.tuwien.ac.at/courses/Rendering/2020/slides/01_spatial_acceleration.pdf
[13] GALVAN, Alain. *Ray Tracing Acceleration Structures* [online]. 2022 [cit. 2023-04-22]. Dostupné z: https://alain.xyz/blog/ray-tracing-acceleration-structures
[14] BIKKER, Jacco. *How to build a BVH – part 2: Faster Rays* [online]. 2022 [cit. 2023-04-22]. Dostupné z: https://jacco.ompf2.com/2022/04/18/how-to-build-a-bvh-part-2-faster-rays/
[15] BATEMAN, Rob a Jon MACEY. *Introduction to SIMD* [online]. 2019 [cit. 2023-04-30]. Dostupné z: https://nccastaff.bournemouth.ac.uk/jmacey/Lectures/SIMD/
[16] JOHNSON, Jeremy a Timothy A. CHAGNON. *SSE and SSE2* [online]. Pennsylvania, USA: Drexel University, 2009 [cit. 2023-05-12]. Dostupné z: https://www.cs.drexel.edu/~jjohnson/2009-10/fall/cs540/lectures/sse.pdf
[17] INTEL. *Intel Intrinsics Guide* [online]. 2022 [cit. 2023-04-30]. Dostupné z: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
[18] GODBOLT, Matt. *Compiler Explorer* [software]. 2023 [cit. 2023-04-30]. Dostupné z: https://godbolt.org/
[19] LIBSDL-ORG. *Simple DirectMedia Layer* [software]. 2.26.5 [cit. 2023-04-30]. Dostupné z: https://github.com/libsdl-org/SDL
[20] CORNUT, Omar. *Dear ImGui* [software]. v1.89.5 [cit. 2023-04-30]. Dostupné z: https://github.com/ocornut/imgui
[21] BARRETT, Sean. *STB Image (Write)* [software]. 2.27 (1.16) [cit. 2022-12-1]. Dostupné z: https://github.com/nothings/stb
[22] BARNES, Tavian. Fast, Branchless Ray/Bounding Box Intersections [online]. 2011 [cit. 2022-12-1]. Dostupné z: https://tavianator.com/2011/ray_box.html
[23] SCRATCHAPIXEL. *Ray Tracing: Rendering a Triangle* [online]. @2009-2022 [cit. 2022-12-1]. Dostupné z: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection
[24] JACOBSON, Alec. Common 3D test models [online]. 2022 [cit. 2022-12-9]. Dostupné z: https://github.com/alecjacobson/common-3d-test-models

SEZNAM SYMBOLŮ A ZKRATEK

Zkratky:

2D	Dvourozměrný
3D	Trojrozměrný
AABB	Axis Aligned Bounding Box
API	Application Programming Interface
BRDF	Bidirectional Reflectance Distribution Function
BSP	Binary Space Partitioning
BVH	Bounding Volume Hierarchy
CGI	Computer-Generated Imagery
CPU	Central Processing Unit
FPS	Frames Per Second
GPU	Graphics Processing Unit
IO	Input/Output
LT	Light Transport
MER	Metallic Emissive Roughness
PNG	Portable Network Graphics
RAM	Random Access Memory
RGBA	Red Green Blue Alpha
SAH	Surface Area Heuristics
SDL	Simple DirectMedia Layer
SIMD	Single Instruction Multiple Data
SS	Screen Space
SSAO	Screen Space Ambient Occlusion
SSE	Streaming SIMD Extensions
SSR	Screen Space Reflections
VNDF	Visible Normal Distribution Function

Symboly:

Q_e	Zářivá energie	$[J]$
h	Planckova konstanta	$[J \cdot s]$
f	Frekvence	$[Hz]$
ϕ_e	Zářivý tok	$[W]$
t	Čas	$[s]$
I_e	Zářivost	$[W \cdot sr^{-1}]$
Ω	Prostorový úhel	$[sr]$
A	Plocha	$[m^2]$
r	Poloměr	$[m]$
L_e	Plošná zářivost	$[W \cdot m^{-2} \cdot sr^{-1}]$

θ	Úhel incidence	$[rad]$
E_e	Intenzita ozáření	$[W \cdot m^{-2}]$
M_e	Intenzita vyzařování	$[W \cdot m^{-2}]$
I	Světelná intenzita	$[-]$
g	Geometrický člen	$[-]$
ϵ	Emisní člen	$[-]$
ρ	Reflektivní člen	$[-]$
L_o	Výstupní zář	$[-]$
L_e	Emitovaná zář	$[-]$
L_i	Příchozí zář	$[-]$
f_r	BRDF materiálu	$[-]$
$cos\,\theta_i$	Kosinus úhlu incidence	$[-]$
$F(x)$	Primitivní funkce	$[-]$
$f(x)$	Integrovaná funkce	$[-]$
$p(x)$	Hustota pravděpodobnosti	$[-]$
X_i	Náhodná proměnná	$[-]$
σ	Směrodatný odchylka	$[-]$
L	Celková zář	$[-]$
$\boldsymbol{T}$	Transportní operátor	$[-]$
$\boldsymbol{I}$	Operátor identity	$[-]$
$\boldsymbol{S}$	Operátor řešení	$[-]$
$O(N)$	Algoritmická komplexita	$[-]$
$P(N_c\|N)^{SAH}$	Funkce SAH	$[-]$
SA	Povrch uzlu	$[-]$
N_c	Podřazený uzel	$[-]$
N	Nadřazený uzel	$[-]$

SEZNAM PŘÍLOH

PŘÍLOHA A - ZDROJOVÝ KÓD PROGRAMU 63

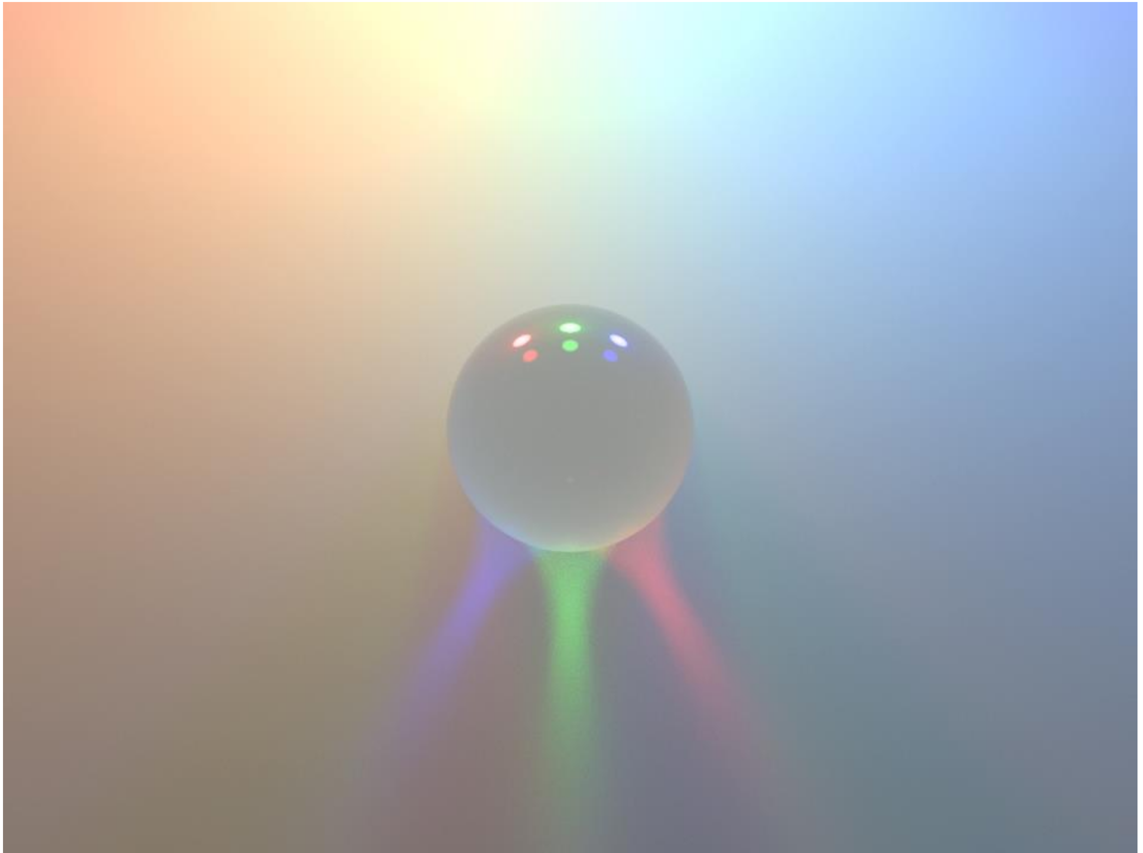
PŘÍLOHA B - OBRÁZKY VYGENEROVANÉ PROGRAMEM 64

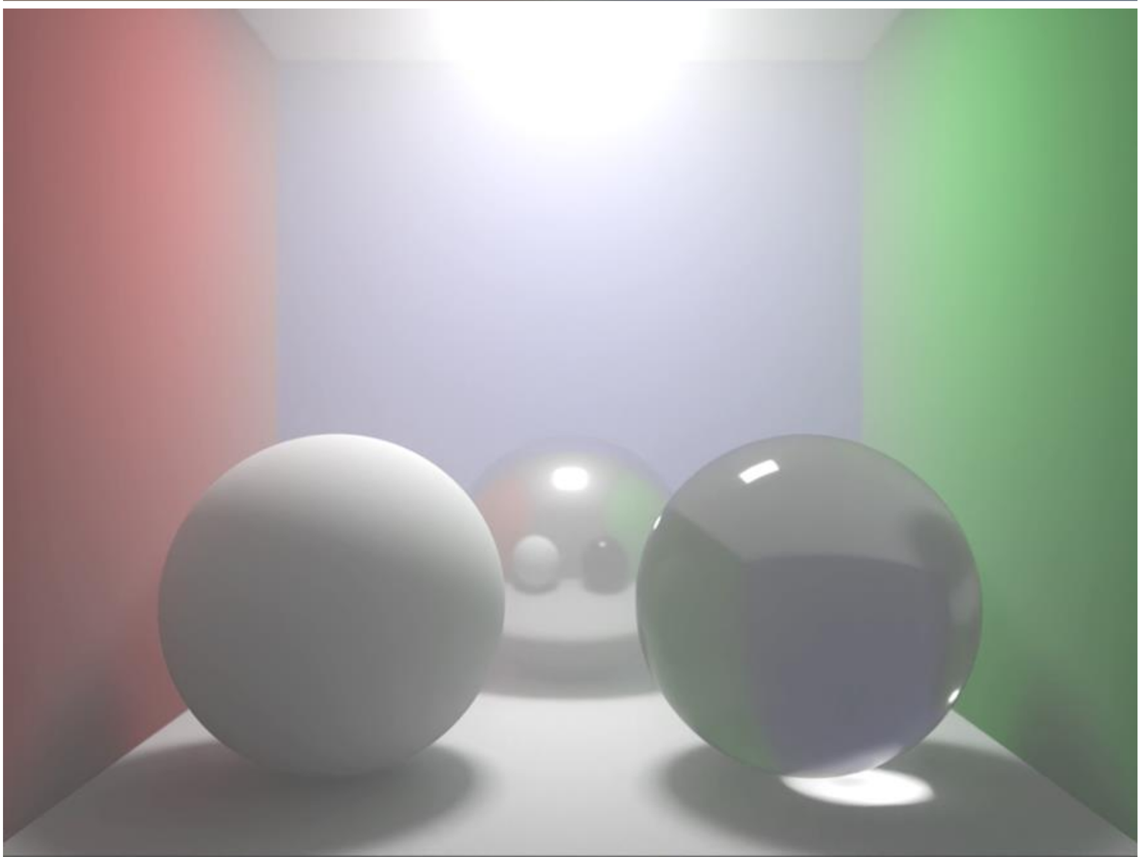
PŘÍLOHA C - PŘÍKLAD TEXTOVÉHO SOUBORU SCÉNY 67

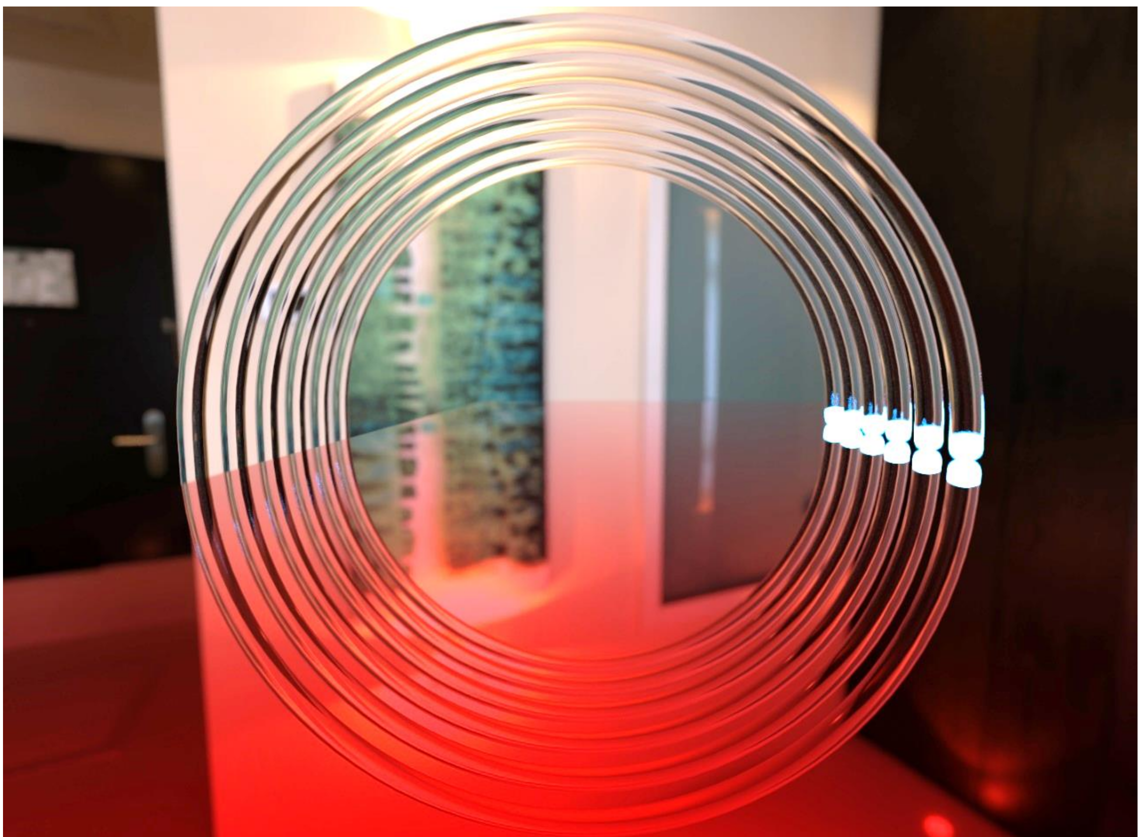
Příloha A - Zdrojový kód programu

Viz elektronická příloha nebo online repositář: https://github.com/Panjaksli/PTCR2.0

Příloha B - Obrázky vygenerované programem







Příloha C - Příklad textového souboru scény

```
*Objects
sphere P=-0.01,0.05,0.06,1 A=0,0,0 mat=0 bvh=0 lig=0 fog=0 {0,0,0,0.001}
sphere P=-0.005,0.05,0.06,1 A=0,0,0 mat=0 bvh=0 lig=0 fog=0 {0,0,0,0.001}
sphere P=0,0.05,0.06,1 A=0,0,0 mat=0 bvh=0 lig=0 fog=0 {0,0,0,0.001}
sphere P=0.005,0.05,0.06,1 A=0,0,0 mat=0 bvh=0 lig=0 fog=0 {0,0,0,0.001}
sphere P=0.01,0.05,0.06,1 A=0,0,0 mat=0 bvh=0 lig=0 fog=0 {0,0,0,0.001}
sphere P=0.015,0.05,0.06,1 A=0,0,0 mat=0 bvh=0 lig=0 fog=0 {0,0,0,0.001}
quad P=0,-0.052,0,0.1 A=0,0,0 mat=1 bvh=0 lig=0 fog=0 {-1,0,-1,1,0,-1,-1,0,1}
quad P=0,0.049,0.12,0.1 A=0,0,0 mat=2 bvh=0 lig=0 fog=0 {-1,0,-1,1,0,-1,-1,0,1}
mesh P=-0.01,0,0,1 A=0,0,0 mat=3 bvh=1 lig=0 fog=0 {fiber_bent2}
mesh P=-0.005,0,0,1 A=0,0,0 mat=4 bvh=1 lig=0 fog=0 {fiber_bent2}
mesh P=0,0,0,1 A=0,0,0 mat=5 bvh=1 lig=0 fog=0 {fiber_bent2}
mesh P=0.005,0,0,1 A=0,0,0 mat=6 bvh=1 lig=0 fog=0 {fiber_bent2}
mesh P=0.01,0,0,1 A=0,0,0 mat=7 bvh=1 lig=0 fog=0 {fiber_bent2}
mesh P=0.015,0,0,1 A=0,0,0 mat=8 bvh=1 lig=0 fog=0 {fiber_bent2}
*Materials
albedo type=3 rgb=0,0.1,0.559,1,0 mer=0,0,10,0 nor=0,0.5,0.5,1 scl=1 ir=1
albedo type=0 rgb=0,1,1,1,1 mer=0,0,0,1 nor=0,0.5,0.5,1 scl=1 ir=1
albedo type=3 rgb=0,0,0,0,0 mer=0,0,0,0 nor=0,0.5,0.5,1 scl=1 ir=1
albedo type=0 rgb=0,0.999,0.999,0.999,0 mer=0,0,0,0 nor=0,0.5,0.5,1 scl=1 ir=1
albedo type=0 rgb=0,0.999,0.999,0.999,0 mer=0,0,0,0 nor=0,0.5,0.5,1 scl=1 ir=1.02
albedo type=0 rgb=0,0.999,0.999,0.999,0 mer=0,0,0,0 nor=0,0.5,0.5,1 scl=1 ir=1.04
albedo type=0 rgb=0,0.999,0.999,0.999,0 mer=0,0,0,0 nor=0,0.5,0.5,1 scl=1 ir=1.06
albedo type=0 rgb=0,0.999,0.999,0.999,0 mer=0,0,0,0 nor=0,0.5,0.5,1 scl=1 ir=1.08
albedo type=0 rgb=0,0.999,0.999,0.999,0 mer=0,0,0,0 nor=0,0.5,0.5,1 scl=1 ir=1.1
*Params
skybox=0
ambient=0,0,0
sky_noon=0.6,0.7,1
sky_dawn=0.5,0.05,0.1
sun_noon=30,25,12
sun_dawn=18,2,0
fog_col=1,1,1
sun_pos=1,0,0.32
bounces=100
samples=2
en_fog=0
en_bvh=1
en_sky=0
en_box=0
fog_dens=-5
cam_collision=0
cam_blur=0
cam_auto=0
cam_fov=90
cam_fstop=1
cam_exp=1
cam_foc_t=1.556
cam_speed=0.02
cam_pos=0.0025,-0.05,0.012
cam_rot=0,0,0
```