VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PATH TRACING NA GPU

PATH TRACING ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DAVID NOVÁK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MICHAL TÓTH**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Novák David**
Obor: Informační technologie
Téma: **Path tracing na GPU**
Path Tracing on GPU
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte knihovnu OpenGL a její nadstavby.
2. Nastudujte metodu Path tracing a její varianty
3. Nastudujte možnosti paralelizace zvolené metody na GPU
4. Navrhněte aplikaci demonstrující zvolenou metodu
5. Implementujte navrženou aplikaci
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu
7. Vytvořte video pro prezentování projektu.

Literatura:

- Podle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 4
- Funkční prototyp aplikace

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Tóth Michal, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L. S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Cílem této bakalářské práce je implementace a následná akcelerace algoritmu sledování cest. Algoritmus bude implementován na GPU pomocí OpenGL. Nad vykreslovanou scénou bude sestavena akcelerační datová struktura oktalový strom. Dále bude naměřeno, jakého zrychlení bylo dosaženo pomocí dané akcelerační datové struktury.

Abstract

The aim of this bachelor's thesis is an implemetation and following acceleration of Path Tracing algorithm. This algorithm will be implemented on the GPU using OpenGL. Above rendered scene will be built Octree data structure. Then the acceleration, which was achieved using this data structure, will be measured.

Klíčová slova

Sledování cest, Sledování paprsku, Globální osvětlení, OpenGL, Oktalový strom.

Keywords

Path Tracing, Ray Tracing, Global Illumination, OpenGL, Octree.

Citace

NOVÁK, David. *Path Tracing na GPU*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Tóth Michal.

Path Tracing na GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Tótha. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
David Novák
9. května 2017

Poděkování

Rád bych zde poděkoval svému vedoucímu za jeho rady a připomínky při tvorbě této práce.

Obsah

Kapitola 1

Úvod

Počátky fenoménu tvorby fotorealistických snímků, dnes známých jako globální osvětlení, sahají již do 60. let minulého století. První a pravděpodobně nejznámnější metodou je Ray Tracing. Dnes ale není zdaleka jediný, existuje celá řada technik, mezi nejznámnější patří například radiozita, Photon Mapping nebo právě Path Tracing. Tyto techniky dokáží simulovat optické jevy jakými jsou odrazy světla, kaustiky, nepřímé osvětlení, zlom paprsku, jejichž simulování je jinak jen velmi obtížné, ne-li nemožné.

Metoda Path Tracing vychází právě z metody Ray Tracing. Hlavním rozdílem těchto metod je, že Path Tracing sleduje nejen jednu cestu, ale různé cesty paprsku skrz scénu, čímž se daří lépe simulovat fyzikální chování světla, a tím pádem získat lepší výsledky.

Obecným problémem nejen Path Tracingu, ale i ostatních metod globálního osvětlení je však jejich značná výpočetní náročnost, což se ukazovalo jako značný problém především v dobách zrození globalního osvětlení. Ačkoliv dnešní možnosti výpočetní techniky jsou již na jiné úrovni (výkonné CPU i GPU, umožňující využití masivního paralelismu), stále tak zůstává výzvou kreslení komplexních scén těmito technikami v reálném čase. Zde je zapotřebí využití různých akceleračních struktur jako například oktalový strom nebo k-d strom.

Hlavním cílem práce je implemetace metody Path Tracing, která poběží na GPU prostřednictvím OpenGL. Tato metoda bude akcelerována datovou strukturou oktalový strom, která bude sestavena nad zobrazovanou scénou. Aplikace bude interaktivní a uživatel bude mít možnost měnit různé faktory scény a pozorovat výsledky.

Práce v kapitole 2 popisuje teoretické základy pro pochopení principu metody Path Tracing. Kapitola 3 řeší téma datové struktury oktalový strom, používaného k akceleraci výpočtu algoritmu. Kapitola 4 představí existující implementace této metody. Kapitola 5 se bude zabývat návrhem a implementací dané aplikace, konkrétně jak CPU části, tak samotného jádra, které poběží na GPU. V následující kapitole bude probrán způsob testování a jeho výsledky. V závěru budou zhodnoceny dosažené cíle a představeny možnosti dalšího rozšíření vytvořené aplikace.

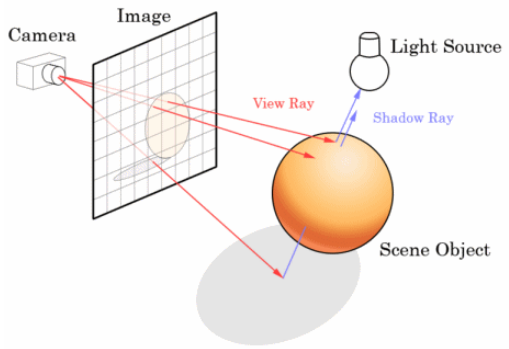
Kapitola 2

Metoda Path Tracing

V této kapitole bude vysvětlen princip metody Path Tracing a metody Ray Tracing z níž je právě Path Tracing odvozen, dále bude představen a vysvětlen matematický aparát potřebný při implementaci dané metody.

2.1 Ray Tracing

První algoritmus pro výpočet globálního osvětlení byl právě Ray Tracing [10] (neboli sledování paprsku). Původní myšlenka tohoto algoritmu je taková, že paprsek je vyslán ze zdroje světla a pohybuje se skrze scénu až k pozorovateli, tedy simulace chování reálného paprsku. U tohoto přístupu se však vyskytuje problém, že se velký počet paprsků ze světelného zdroje nedostane k pozorovateli, tudíž se nepodílejí na výsledku, a to znamená, že byly počítány zbytečně. Z toho důvodu je mnohem efektivnější sledovat paprsek opačným směrem tzn. od pozorovatele ke zdroji světla (princip algoritmu je ilustrován na obrázku 2.1), kde paprsek vychází od pozorovatele skrze jednotlivé pixely průmětny. Při průchodu scénou se testuje výskyt kolize paprsku se všemi objekty ve scéně. Při výskytu kolize je paprsek odražen, a tak pokračuje až ke zdroji světla. Výpočet kolize se všemi objekty však může být u rozsáhlých scén značně výpočetně náročný, navíc s ohledem, že cesta paprsku ke zdroji světla může být teoreticky nekonečná, se mnohdy délka cesty ohraničí nějakou hodnotou. Výhody tohoto algoritmu jsou však nesporné a celkem bez větších problémů je možné simulovat např. odlesky, lom světla nebo kaustiky.



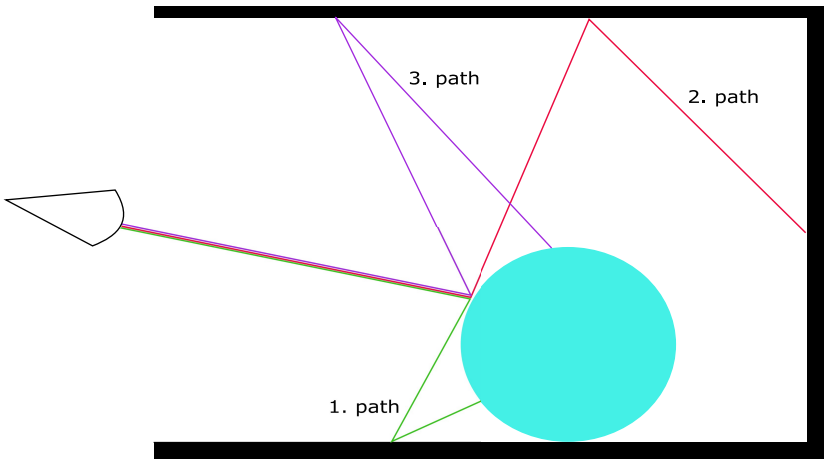
Obrázek 2.1: Princip algoritmu sledování paprsku. Paprsek prochází skrze průmětnu do scény, kde se protíná s objekty a je případně odrážen. Převzato z webu[2].

2.2 Path Tracing

Za jednoho z následníků Ray Tracingu se dá označit Path Tracing [4] (neboli sledování cest). Jeho publikováním se zasloužil ve druhé polovině 80. let minulého století James Kajiya. Jedná se o stochastickou metodu, která si bere základ z Ray Tracingu. Hlavním rozdílem těchto metod je ten, že Path Tracing nesleduje pouze jednu stejnou cestu paprsku scénou, ale různé cesty (viz obrázek 2.2).

Path Tracing zohledňuje myšlenku, že na konkrétní bod dopadá světlo nejen ze zdroje světla, ale i ze všech směrů z okolí[3]. Množství takových paprsků, které dopadají na určitý bod, je nekonečné, a to představuje značný problém při implementaci. Path Tracing využívá k řešení tohoto problému metodu Monte Carlo. Výsledek je tedy složen z mnoha vzorků respektive cest paprsku, kde první vzorky ovlivní výsledek více než ty později spočítané vzorky. Nevýhodnou vlastností Monte Carlo integrace je však velká chyba při prvních vzorcích, která se na výsledném obrazu projevuje značným šumem. S přibývajícím počtem vzorků se však chyba snižuje a šum mizí. Výsledek tedy není naprosto přesný, ale jedná se pouze o jeho aproximaci.

Cesta paprsku je především určená materiály objektů se kterými se paprsek protnul. Na základě tohoto materiálu se určí směr, kterým se paprsek odrazí, případně se neodrazí a cesta je ukončena. Paprsek tím pádem začíná stejně jako u Ray Tracingu, od prvního odrazu už bývá cesta paprsku jiná.



Obrázek 2.2: Ilustrace různých cest sledovaných paprsků.

[2] https://upload.wikimedia.org/wikipedia/commons/thumb/8/83/Ray_trace_diagram.svg/300px-Ray_trace_diagram.svg.png

[3] Toto popisuje zobrazovací rovnice.

2.3 Parpsek a jeho reprezentace

Jelikož je metoda Path Tracing založena na sledování cest paprsku, je také nutné vysvětlit, jak je takový paprsek vytvořen a jakým způsobem je reprezentován. Paprsek si lze představit jako polopřímku s počátkem v určitém bodě ve scéně, tím pádem je k jeho definování zapotřebí dvou hodnot, a to bod počátku a jednotkový vektor určující směr.

Na začátku výpočtu je vytvořen primární paprsek, který má počátek v pozici pozorovatele a směřuje skrze určitý bod průmětny. Paprsky, které jsou odražené, lomové případně stínové jsou označovány jako sekundární paprsky. Směr těchto paprsků je buď určen stochasticky, nebo deterministicky jako například u zrcadlového povrchu, kde je paprsek odražen ideálním odrazem.

Takřka životně důležitou součástí algoritmu Path Tracing (ale také i Ray Tracing) je testovaní na průsečík paprsku s objektem ve scéně. Při této operaci je kromě určení přítomnosti kolize také důležité určení jejího umístění. Pro zjištění pozice paprsku, a tudíž i místa kolize slouží následující rovnice (2.1).

$$P(t) = O + t\vec{D} \mid t \geq 0 \tag{2.1}$$

Kde O značí počátek paprsku, $\vec{D}$ je jeho směr a t je délka paprsku. Při výpočtu průsečíku získáme hodnotu t, jejímž dosazením do rovnice výše dostaneme právě bod kolize. Pokud tato proměnná nabývá záporných hodnot, znamená to, že paprsek směřuje opačným směrem, než je současný směr paprsku, tudíž není potřeba s ním dále počítat. Paprsek však může protnout více objektů ve scéně, zde se bere v úvahu nejbližší bod kolize[4].

Zvláštní úlohu mají již zmíněné stínové paprsky, ty jsou využívány při výpočtu osvětlovacího modelu v daném bodě, zároveň také slouží k určení, zda daný bod leží ve stínu (mezi tímto bodem a zdrojem světla leží jiný objekt).

2.3.1 Výpočet průsečíku paprsku s trojúhelníkem

Velice důležitou operací prováděnou s paprskem, je hledání jeho průsečíku s určitým objektem, neboli spíše primitivem scény. Geometrie scény bývá obvykle popsána pomocí trojúhelníků (především díky jejich vlastnostem). Důležitým kritériem pro tuto operaci je výběr vhodného algoritmu pro nalezení kolize. Hojně využívaným pro tento účel je algoritmus Möller-Trumbore [7]. Tento algoritmus je založený na řešení následující rovnice (2.2).

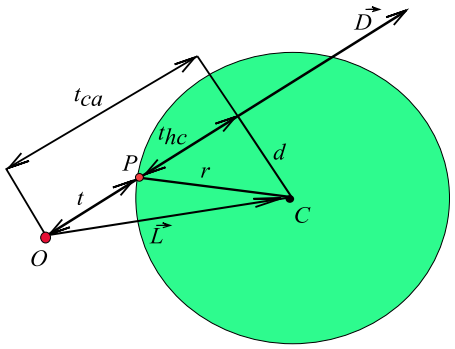
$$O + t\vec{D} = (1 - u - v)V_0 + uV_1 + vV_2 \tag{2.2}$$

Kde levá strana rovnice vychází z rovnice 2.1, která popisuje bod nacházející se na paprsku. V_0, V_1 a V_2 jsou vrcholy trojúhelníku a u, v jsou barycentrické souřadnice. Celkově tedy pravá strana vyjadřuje bod ležící v rámci trojúhelníku. Aby však bod (tudíž i bod kolize) ležel uvnitř trojúhelníku musí platit, že $u \geq 0 \wedge v \geq 0 \wedge (u + v) \leq 1$. V případě neplatnosti jedné z těchto podmínek kolize nenastala.

2.3.2 Výpočet průsečíku paprsku s koulí

Dalším primitivem které se ve scéně může nacházet je koule (ta však může být složená z trojúhelníků). Nalezení průsečíku je naznačeno a popsáno na obrázku 2.3.

[4] Nejmenší hodnota proměnné t.



Obrázek 2.3: Princip nalezení kolize paprsku a koule. Nejprve je určen vektor $\vec{L}$ (odečtení bodu počátku paprsku od bodu středu koule). V dalším kroku se pomocí skalárního součinu vektorů $\vec{D}$ a $\vec{L}$ získá hodnota t_{ca}. Poté pomocí pythagorovy věty spočítáme hodnotu d, jíž využijeme spolu s velikostí poloměru koule k výpočtu hodnoty t_{hc} (opět pomocí pythagorovy věty). Odečtením hodnoty t_{hc} od t_{ca} získáme hodnotou t, jejímž dosazením do rovnice 2.1 získáme pozici bodu kolize.

2.4 Zobrazovací rovnice

Zobrazovací rovnice [4] slouží k výpočtu množství světla v určitém bodě, kde došlo k průniku paprsku a objektu. Takové množství světla je složeno z emitovaného světla v daném bodě a také množství světla příchozího z okolí, s čímž souvisí charakterisktika materiálu daného objektu. Metody globálního osvětlení řeší tuto rovnici různými způsoby, například metoda radiozity rozděluje povrch objektů na malé plošky a počítá příspěvky těchto jednotlivých plošek k výsledku. Oproti tomu metoda Path Tracing používá k řešení metodu Monte Carlo. Zobrazovací rovnice má následující podobu (2.3).

$$L_o(x,\omega) = L_e(x,\omega) + \int_\Omega f_r(x,\omega',w) L_i(w,\omega')(\omega' \cdot n) d\omega' \tag{2.3}$$

Kde $L_o(x,\omega)$ udává odchozí světlo v bodě x a směru ω, $L_e(x,\omega)$ reprezentuje vyzařované světlo v bodě x a směru ω, $f_r(x,\omega',w)$ značí funkci BRDF, která určuje poměr mezi odraženým a přijatým světlem, $L_i(w,\omega')$ určuje příchozí světlo ve směru ω' a $(\omega' \cdot n)$ je tzv. zeslabovací faktor intenzity záření. Integrál udává, že se akumulují příspěvky všech paprsků dopadající na polokouli okolo bodu x.

Celkově řečeno, množství světla v daném bodě je světlo, které povrch sám vyzařuje (bývá většinou minimální) a množství světla, jenž přichází z okolí, ať už se jedná o přímé osvětlení od zdroje světla nebo nepřímé od povrchu okolních objektů.

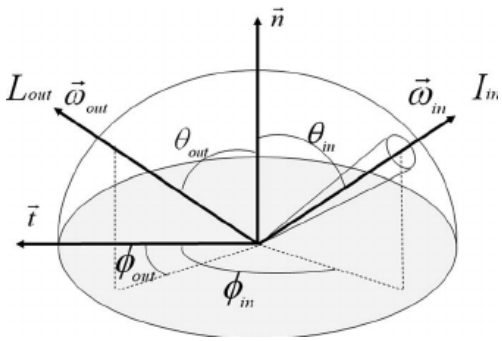
2.5 BRDF

BRDF [8] neboli bidirectional reflectance distribution function se používá pro charakteristiku optických vlastností materiálu určitého povrchu. Různé materiály reagují na světlo různým způsobem, například hladký lesklý povrch odráží případně absorbuje světlo jinak než třeba hrubý povrch. Takovéto chování světla v závislosti na materiálu popisuje právě BRDF. Byla vyjádřena již v 60. letech minulého století Fredem Nicodemusem. Vyjadřuje

vztah mezi odraženým a dopadajícím světlem (graficky znázorněno na obrázku 2.4). BRDF je definována následně (2.4).

$$f_r(x, \omega_i, \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)cos(\theta_i \mathrm{d}\omega_i)} \tag{2.4}$$

Kde $L_r(\omega_r)$ značí intenzitu odraženého světla ve směru ω_r, $L_i(\omega_i)$ značí intenzitu dopadajícího světla ve směru ω_i, θ_i představuje úhel mezi normálou povrchu a směrem dopadajícího světla. V podstatě udává jaká je pravděpodobnost toho, že se paprsek odrazí právě daným směrem.



Obrázek 2.4: Ilustrace dopadajícího a odraženého světla. Symbol $\vec{\omega_{out}}$ označuje směr dopadajícího paprsku, $\vec{\omega_{in}}$ značí směr odraženého paprsku, kůžel okolo odraženého paprsku značí prostorový úhel, v jehož rámci mohou být potenciálně odraženy paprsky od plochy daného materiálu. Převzato z webu[6].

2.6 Monte Carlo integrace

Název Monte Carlo [2] nenese pouze integrace (což je jedna z aplikací), nýbrž existuje celá řada metod. Základní princip je však u všech stejný, k získání výsledku se dojde tak, že jsou nad daným problémem (např. určitý integrál nějaké funkce) prováděny experimenty s náhodnými vzorky. Monte Carlo má mnoho oblastí uplatnění, kromě počítačové grafiky se využívá také v oblasti počítačových simulací nebo například v umělé inteligenci.

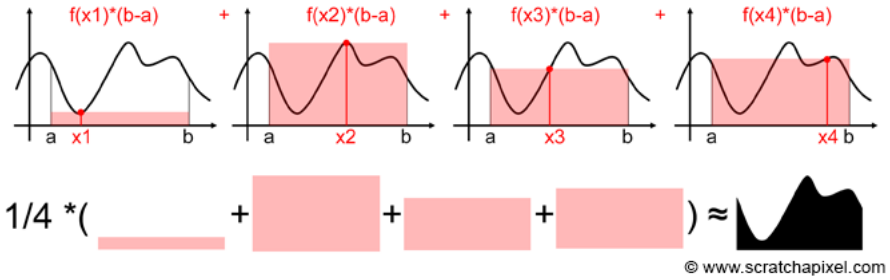
Monte Carlo integrace slouží k numerickému řešení určitého integrálu. K výsledku dojde tím, že vyhodnocuje integrovanou funkci v náhodně zvolených bodech. Jednotlivé výsledky jsou sečteny a poděleny počtem provedených experimentů. Získaná hodnota je vynásobena délkou oblasti na které se počítá určitý integrál. Demonstrace použití techniky Monte Carlo při výpočtu určitého integrálu je naznačena na obrázku 2.5.

S pomocí Monte Carlo se však nezíská naprosto přesný výsledek, ten je zatížen chybou, a to poměrně značnou. Tato chyba je závislá na počtu provedených experimentů. Taková chyba je určená vztahem níže (2.5).

$$e = \frac{1}{\sqrt{N}} \tag{2.5}$$

[6] https://www.researchgate.net/profile/Duck_Bong_Kim2/publication/252647257/figure/fig1/AS:298074032361479@1448077795754/Fig-1-BRDF-Geometry-of-surface-reflection.png

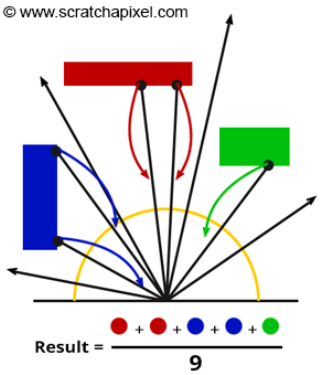
[8] https://www.scratchapixel.com/images/upload/monte-carlo-methods-practice/MCIntegration03.png



Obrázek 2.5: Demonstrace principu Monte Carlo integrování. Provedení několika náhodných experimentů nad zadanou funkcí a následné odhadnutí výsledku. Převzato z webu[8].

Kde N udává počet provedených experimentů. K výsledku se tím pádem konverguje, navíc poměrně pomalu, jelikož chyba menší než například 1% se projevuje až při 10 000 a více experimentech.

Nespornou výhodou Monte Carlo integrace je však schopnost spočítat integrály, které by se jinak řešili jen velmi obtížně. U metody Path Tracing se pomocí tohoto přístupu řeší zobrazovací rovnice, což je právě případ, kde by bylo analytické řešení velmi náročné. Takové řešení probíhá tím způsobem, že v bodě kolize paprsku s objektem jsou vysílány paprsky s náhodně zvoleným směrem a je získán jejich barevný příspěvek, který je započítán do výsledku (viz obrázek 2.6).



Obrázek 2.6: Použití Monte Carlo integrace k výpočtu nepřímého osvětlení v daném bodě. Z konkrétního bodu jsou vysílány (resp. odraženy) paprsky s náhodně určeným směrem, jenž získají barevný příspěvek z okolí bodu. Při samotném výpočtu je ještě využita BRDF charakterizující materiál povrchu. Převzato z webu[10].

2.7 Generování náhodných čísel

Jelikož je metoda Path Tracing stochastická, dá se označit za jednu z kritických součástí generování náhodných čísel [5]. Hlavním úkolem takového generátoru je generování čísel z rozsahu <0, 1> podle určitého rozložení (rovnoměrné, exponenciální atd.). Generování

[10] https://www.scratchapixel.com/images/upload/shading-intro2/shad2-globalillum2.png

skutečně náhodných čísel, tedy, že jsou generovány podle nějakého stochastického procesu, je nevýhodné v několika ohledech. Tím prvním je poněkud náročnější implementace takového generátoru, další nevýhodou je malý počet výstupních dat. Z těchto důvodů se používají pseudonáhodné generátory čísel. Ty sice nezískávají náhodná čísla na základě nějakého stochastického procesu (nýbrž deterministickým způsobem), na druhou stranu však celkem věrně dokáží simulovat generátor náhodných čísel, navíc odstraňují nevýhody s náročností implementace a malým počtem získaných dat.

Typickým představitelem pseudonáhodného generátoru čísel je lineární kongruentní generátor. Jeho princip je poměrně prostý, pamatuje si předchozí vygenerovanou hodnotu z níž vytvoří novou hodnotu pomocí vztahu 2.6.

$$x_{i+1} = (ax_i + c) \ \ mod \ \ m \tag{2.6}$$

kde x_{i+1} je nově vygenerovaná hodnota, x_i je předchozí vygenerovaná hodnota, a, c jsou vhodně zvolené konstanty a m označuje rozsah generátoru, tedy velikost množiny čísel, které je možno vygenerovat. Jelikož je tato množina konečná, tak se po určitém počtu vygenerovaných čísel začne opakovat stejná posloupnost. Takový počet čísel se označuje jako perioda generátoru.

Perioda generátoru se vyskytuje u všech pseudonáhodných generátorů, což plyne z jejich podstaty. V určitých případech (např. velmi složité simulace) se však může tato vlastnost projevovat negativně. Zde se dají využít generátory s větší periodou, typickým, hojně používaným generátorem v takovýchto případech je Mersenne Twister, jenž s periodou $2^{19937} - 1$ představuje robustní řešení.

Důležitou součástí generování náhodných čísel je také jejich testování, které čítá několik technik. První možností může být vygenerování souboru čísel a jejich následné zanesení do grafu. Lepší variantou je spíše generování dvojic případně trojic čísel, jelikož dokáže ukázat případné závislosti mezi jednotlivými hodnotami, a tudíž nekvalitní způsob generování. Dalšími technikami testování jsou například test dobré shody χ^2, poker test nebo Hammingův test.

Kapitola 3

Datová struktura oktalový strom

Grafické aplikace obecně patří mezi výpočetně náročné úlohy. Při požadavcích běhu těchto úloh v reálném čase jsou na místě různé optimalizace. Mnohdy používanou optimalizací je paralelizace dané úlohy, což provádí grafická karta při zpracování úloh. V některých případech ani paralelizace nestačí k dostatečnému urychlení běhu aplikace. V takovýchto případech se používá optimalizace pomocí různých datových struktur, které umožňují různým způsobem rozložit scénu na jednotlivé podčásti a tím snížit počet zbytečně provedených operací.

Právě metoda Path Tracing se dá rozhodně označit za značně vypočetně náročnou úlohu, kde je kritickou pasáží testování vzniku průsečíku paprsku a primitiva. U jednoduchých scén se problém s výkonem vyskytovat nemusí, avšak u scén se složitou geometrií může výpočet jednoho snímku trvat až jednotky sekund.

Tato kapitola popisuje základní myšlenku principu oktalového stromu a způsob práce s touto datovou strukturou.

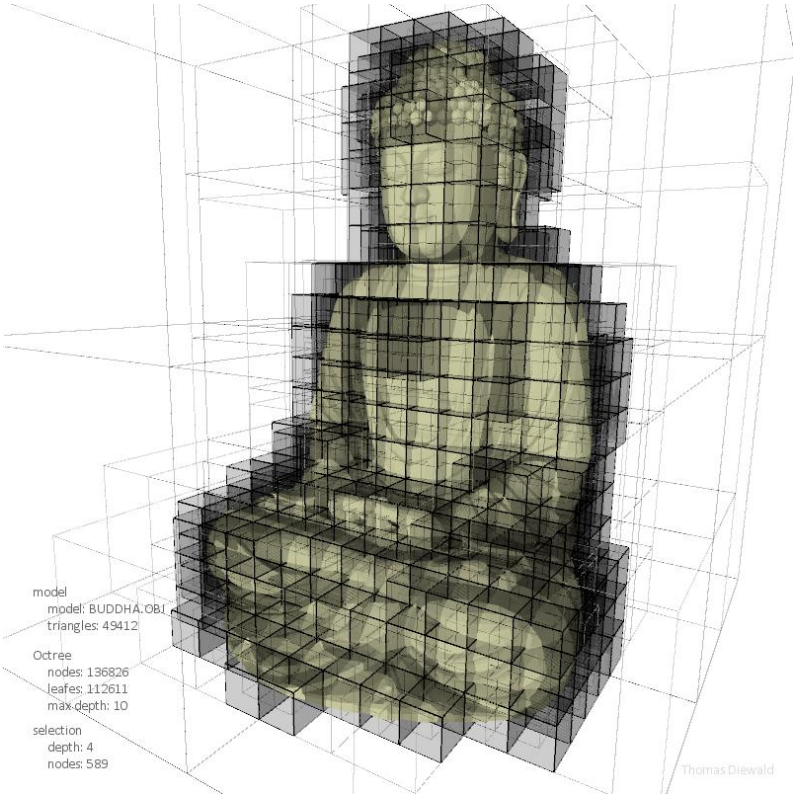
3.1 Princip oktalového stromu

Základní myšlenka oktalového stromu [6] vychází z dělení scény na osm stejných částí. Takové části pak mají poloviční rozměry oproti svému předchůdci. Samotná scéna tedy představuje kořenový uzel a jednotlivé podčásti jsou jejími potomky. Takto můžeme scénu dělit teoreticky nekonečně. Oktalový strom se skládá z uzlů a listů (uzly, které nemají potomka nebo jinými slovy, jsou terminální), kde právě v listech jsou uložena samotná primitiva.

Díky rozdělení primitiv mezi jednotlivé listové uzly není potřeba testovat průnik se všemi primitivy ve scéně, ale pouze s těmi, které leží v zasažených listových uzlech. Tímto způsobem se dá získat až několikanásobné urychlení vykreslování. V této souvislosti je ale zapotřebí rozdělit jednotlivé uzly správným způsobem (optimální počet uzlů, případně výška stromu). Pokud je scéna rozdělena do malého počtu uzlů potom je zrychlení zanedbatelné, naproti tomu rozdělení scény do příliš velkého množství uzlů se může projevit většími nároky na paměť.

Oktalový strom nachází uplatnění i u jiných aplikací než jsou Raytracery resp. Pathtracery. Mnohdy se využívá v oblasti počítačových her, kde by bylo nevýhodné kompletně vykreslovat velké scény s množstvím modelů, proto se scéna rozdělí do oktalového stromu a vykreslují se pouze modely v nejbližším okolí. Další oblastí, která využívá této datové struktury, jsou volumetrické techniky, kde jsou do určitých listových uzlů vloženy jednot-

livé voxely[1]. Ukázku rozdělení trojrozměrného modelu do oktalového stromu je možné vidět na obrázku 3.1.



Obrázek 3.1: Rozdělení modelu do datové struktury oktalový strom. V oblastech výskytu geometrie je strom členěn na menší uzly. Převzato z webu[3].

3.2 Sestavení a procházení oktalového stromu

Princip sestavení oktalového stromu není nijak složitý, testuje se přítomnost průsečíku primitiva a uzlu stromu, v kladném případě je do uzlu vloženo dané primitivum. Pokud uzel obsahuje velký počet primitiv, je uzel rozdělen a označen za neterminální. Tato činnost se dá optimalizovat například tím, že se netestuje průsečík se všemi primitivy, ale jen s těmi, které se nachází v rodičovském uzlu.

Rychlost samotné konstrukce nebývá kritická u statických scén, kde je strom sestaven pouze jednou před vykreslováním. Problém nastává u dynamických scén, kde je strom opakovaně rekonstruován při změně pozic objektů ve scéně. V takovýchto případech je možné využití paralelizace tohoto procesu.

Kritičtější operací je procházení oktalového stromu. Rychlost průchodu v této struktuře je rozhodující pro akceleraci vykreslování. Princip této operace je takový, že se testuje průsečík se všemi uzly na dané úrovni. Vzdálenosti jednotlivých průsečíku jsou seřazeny vzestupně podle jejich velikosti, a v tomto pořadí jsou procházeny, případně probíhá za-

[1] Elementární částice u volumetrických technik, nebo-li prostorový pixel.
[3] http://thomasdiewald.com/blog/wp-content/uploads/2012/09/diewald_BUDDHA_Octree_800x800_selection.jpg

noření do nižší úrovně. Jak jde vidět, nejedná se o triviální operaci, naopak v případech, kdy je nutné procházet značné množství uzlů, může být výsledné zrychlení velmi malé. V dnešní době však existují algoritmy umožňující rychlejší procházení oktalovým stromem. Příkladem mohou být algoritmus HERO [1] nebo často používaný parametrický algoritmus od Jorge Revellese [9], využívající znalost o směru paprsku a vlastnosti oktalového stromu.

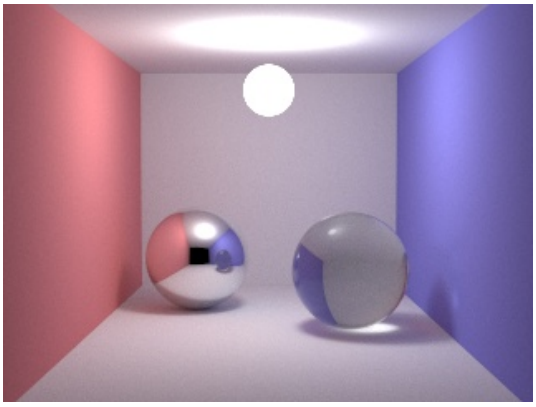
Kapitola 4

Existující implementace

Na webu je možné v současné době nalézt řadu implementací metody Path Tracing, které jsou řešeny různými přístupy za pomoci různých technologií. V této části bude představeno několik již existujících řešení s jejich krátkým popisem.

4.1 SmallptGPU2

Prvním zástupcem je aplikace SmallptGPU2, jejímž autorem je David Bucciarelli. Aplikace je založena na implementaci Path Tracingu od Kevina Beasona[1], která však žádným způsobem nevyužívá pro výpočet grafický procesor. Dá se tedy označit za rozšíření této implementace. Pro výpočet snímku na GPU je využito technologie OpenCL. Aplikace může rovněž běžet na CPU a GPU současně, což logicky umožňuje ještě rychlejší chod aplikace. Oproti implementaci, ze které vychází, je tedy výrazně rychlejší, rychleji konverguje k výsledku (šum se projevuje méně již při prvních snímcích) a navíc nabízí interakci se scénou. Jistou nevýhodou je fakt, že zobrazuje pouze jedinou scénu.

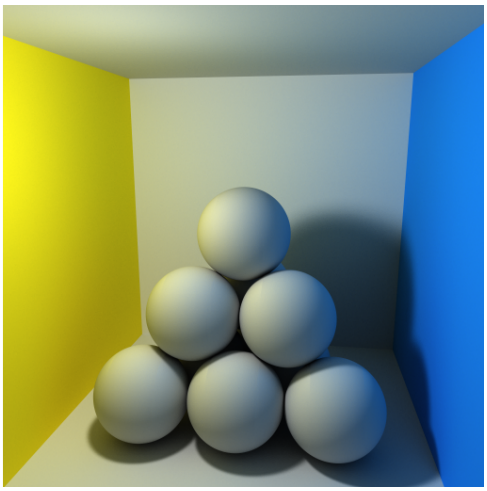


Obrázek 4.1: Snímek z aplikace SmallptGPU2.

[1] viz webová stránka: http://www.kevinbeason.com/smallpt/

4.2 WebGL Path Tracing

Další implementací metody Path Tracing je výtvor z dílny Evana Wallace. Jak již název napovídá, pro implementaci bylo využito WebGL API. Pro svůj běh využívá prostředí webového prohlížeče. Samotná aplikace zobrazuje scénu Cornell box s různým obsahem. Umožňuje také jistou interakci se scénou v podobě pohybu kamery, světla a jednotlivých objektů scény. Navíc umožňuje měnit barvu stěn a materiály objektů. Jistou nevýhodou může být, že zobrazuje pouze základní analytické objekty (kvádr, koule, rovina), nikoliv komplexnější modely. Snímek z této aplikace je možné vidět na obrázku 4.2.



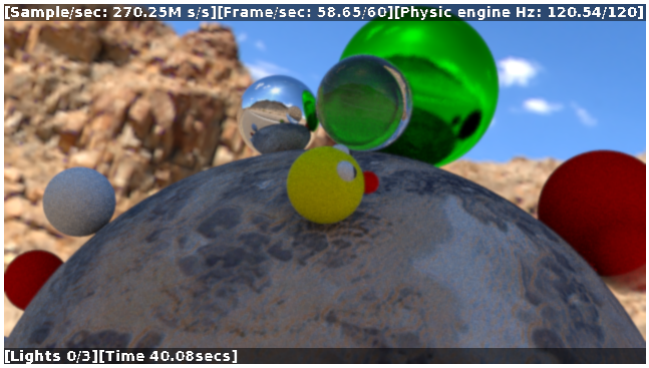
Obrázek 4.2: Snímek z aplikace WebGL Path Tracing.

4.3 Octane Render

V tomto případě se jedná o komerční produkt. Umožňuje vykreslovat scénu metodou Path Tracing pomocí GPU, za použití technologie CUDA. Nástroj obsahuje velké množství funkcionality v podobě nastavení mnoha vlastností scény, správy textur případně nastavení způsobu kreslení. Dostupný je rovněž jako plugin pro modelovací nástroje (3DS Max, Blender, CINEMA 4D a další).

4.4 Sfera

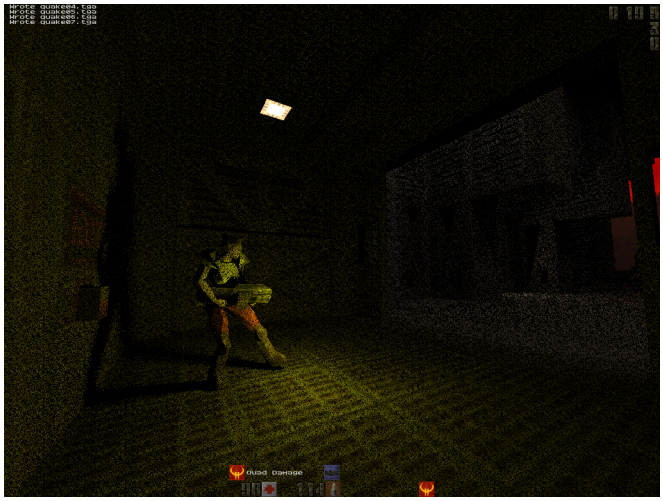
S myšlenkou využití Path Tracingu v jednoduché hře přichází již zmíněný David Buccia-relli. Pro akceleraci na GPU je využito OpenCL. Jak již samotný název naznačuje, jedinými objekty ve hře jsou koule (viz obrázek 4.3), scéna tedy není reprezentována komplexní troj-úhelníkovou sítí, což umožňuje běh v reálném čase. Při překreslování se pochopitleně projevuje šum, který může působit rušivě, přesto hra běží díky GPU akceleraci poměrně svižně. Vyskytuje se zde také možnost vypnutí akcelerace, což značně zpomalí výpočet snímků, a tím pádem není možný běh v reálném čase. Obecně využití Path Tracingu v počítačových hrách může být zajímavé, přesto je potřeba se vypořádat s množstvím výkonnostních překážek řešených v podobě různých optimalizací. Důkazem, že i toto je možné, budiž následující aplikace.



Obrázek 4.3: Snímek z aplikace Sfera. Jedinými primitivy jsou koule různých barev a materiálů. Prostředí je řešeno pomocí Skyboxu.

4.5 Quake 2 Realtime GPU Pathtracing

Velmi zajímavá je implementace počítačové hry Quake 2, kde je zobrazování řešeno metodou Path Tracing, jejímž autorem je Edd Biddulph. Pro akceleraci na GPU je využito OpenGL, to však samo o sobě pro běh v reálném čase nestačí, s ohledem na složitou geometrii scény a množství světel je potřeba další optimalizace. Takovou optimalizací je ukládání scény do datové struktury BSP[2] strom, kterou si autor ještě speciálně upravil pro větší zrychlení aplikace. Ta je dynamicky sestavována při každém novém snímku. Ve výsledku tedy vznikla v reálném čase běžící hra na bázi Path Tracingu, což bylo ku příkladu v dobách implementace samotné hry Quake 2 (rok 1997) vzhledem k tehdejším technologiím naprosto nemyslitelné. Obrázek 4.4 zobrazuje snímek z této aplikace.



Obrázek 4.4: Snímek z aplikace Quake 2 Realtime GPU Pathtracing.

[2]Binary Space Partitioning.

Kapitola 5

Návrh a implementace aplikace

Nabyté znalosti budou využity při tvorbě aplikace demonstrující zobrazování scény metodou Path Tracing, akcelerovanou oktalovým stromem. Nejprve budou představeny technologie použité při řešení a návrh aplikace. V další části kapitoly bude popsána samotná implementace realizovaná v jazycích C++ a GLSL. Zde budou popsány nejprve hlavní úkoly a funkčnost části aplikace, která poběží na procesoru (CPU) a poté popis úloh části aplikace bežící na grafickém procesoru (GPU).

5.1 Použité technologie

Tato sekce se bude věnovat rozboru použitých knihoven při praktickém řešení metody Path Tracing. Popíše účel knihoven v implementované aplikaci, jejich vlastnosti, případně jejich vliv při návrhu.

Vzhledem k implementačnímu jazyku C++ je zapotřebí nejprve vybrat knihovnu umožňující vytvoření, správu okna a zpracování událostí, jenž bude pro daný jazyk dostupná. Dalším bodem je knihovna pro zajištění běhu aplikace na GPU, jejíž princip zásadně ovlivní další návrh aplikace. Tento princip bude rovněž rozebrán a vysvětlen.

5.1.1 GLFW

První použitou knihovnou je open-source knihovna GLFW. Její použití je podmíněno tím, že OpenGL, narozdíl od DirectX samo o sobě neobsahuje funkcionalitu pro vytvoření a správu grafického kontextu, případně obsluhy událostí od vstupních zařízení (myš, klávesnice atd.).

V tomto případě by mohlo být nasnadě použití WinAPI, což by však bylo příliš nízkoúrovňové řešení, navíc by se aplikace stala platformě závislou (fungovala by pouze na OS firmy Microsoft). Zde se projevuje další výhoda knihovny GLFW, která je multiplatformní.

Ke knihovně GLFW existuje také několik alternativ, pravděpodobně nejznámější je knihovna SDL, dále mohou být jmenovány ku příkladu GLUT, GLAUX nebo SFML.

5.1.2 OpenGL

OpenGL (Open Graphics Library) je API především pro 3D, ale také 2D grafiku, která je akcelerována pomocí grafické karty. Je podporována ve velkém množství programovacích jazyků (C++, Java, Python ad.). OpenGL je spravováno konsorciem Khronos Group, vydáno bylo ovšem skupinou Silicon Graphics roku 1992.

Pro akceleraci jej využívají CAD systémy, grafické editory (Adobe Photoshop), GUI frameworky (např. Qt) nebo dnes i poměrně rozšířená virtuální realita. Nasnadě je použití OpenGL pro tvorbu počítačových her. Ve skutečnosti je to možné (důkazem mohou být hry Tomb Raider nebo Left 4 Dead případně i jiné), avšak v současnosti většina her využívá knihovnu společnosti Microsoft, DirectX. Jedním důvodem může být kupříkladu nutnost propojení OpenGL s dalšími knihovnami podporující obsluhu událostí, načtení textur ad., což u DirectX není zapotřebí.

Kromě OpenGL existuje dále OpenGL ES, jenž je používáno na mobilních zařízeních a WebGL podporující grafiku ve webových prohlížečích. Snahou konsorcia Khronos Group je sjednocení zmíněných API do jediného. Výsledkem této snahy je relativně nedávno vydané (v únoru 2016) API Vulkan.

Princip vykreslování pomocí OpenGL je založen na vykreslovací pipeline (viz obrázek 5.1). Ta je složená z řady sekcí, kde na počátku jsou údaje o geometrii vykreslované scény a na konci samotný vykreslený snímek. Některé sekce jsou programovatelné, jiné nikoliv. Programovatelným částem se říka shadery, a jedná se o programy běžící na grafickém procesoru. Existuje několik druhů shaderů Vertex, Tessellation Control a Evaluation, Geometry, Fragment a speciální Compute shader.

Vertex shader pracuje s jednotlivými vrcholy (pro každý vrchol běží jedna instance Vertex shaderu). Zde je možné různým způsobem transformovat vrcholy, častou operací je zde násobení vrcholů MVP maticí[1]. Transformace vrcholů lze využít případně i v animacích, kde je v závislosti na čase transformován každý vrchol. Výstupem této sekce je tedy nová pozice vrcholu (přesněji řečenou pozice vrcholu v clip space).

Další částí pipeline je dvojice Tesselation shaderů. Tím prvním je Tessellation Control shader, běžící ve stejném počtu invokací jako Vertex shader. Tento shader umožňuje určit, jak bude Tessellator generovat nové vrcholy (jsou generovány interpolací mezi již existujícími vrcholy) na základě stávajících. Dalším v řadě je Tessellation Evaluation shader, jenž má především za úkol transformovat vrcholy, které jsou výstupem Tessellatoru.

Po Tessellation shaderech následuje Geometry shader. V této části je možné z již existujícího vrcholu tvořit nové vrcholy a utvořit z nich nová primitiva. Toto například umožňuje z malého množství bodů vytvořit procedurálním způsobem nějakou složitější geometrii. Výstupem Geometry shaderu je primitivum (příp. primitiva) složené z vytvořených vrcholů.

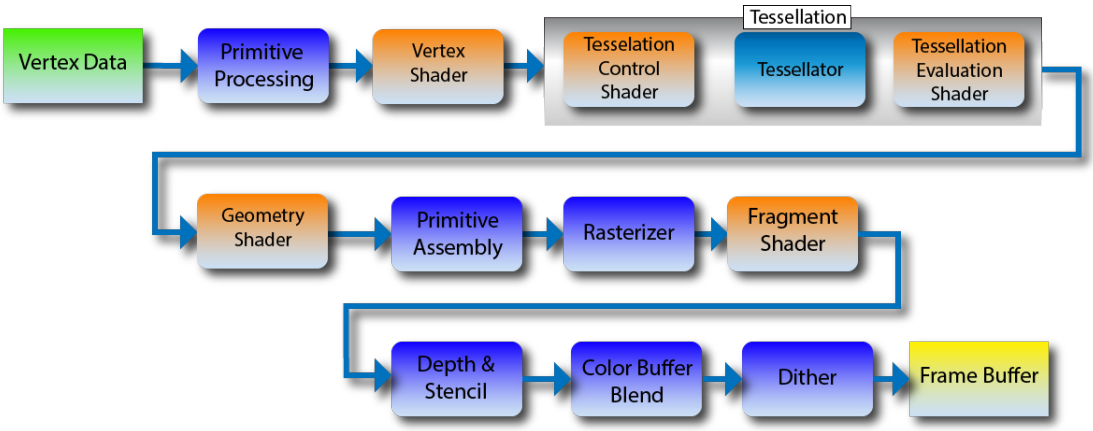
Po rasterizaci, kde jsou převedena data popisující geometrii scény na jednotlivé fragmenty, následuje Fragment shader. Pro každý fragment je spuštěna jedna jeho instance (obvykle odpovídá počtu pixelů). Jednotlivé atributy vrcholu (např. texturovací souřadnice) jsou získány pomocí barycentrické interpolace. Hlavním účelem této sekce je výpočet výsledné barvy fragmentu. Ta může být předána jako konstantní barva nebo jako barva z textury. Navíc v případě 3D grafiky je obvykle počítán osvětlovací model (např. Phongův). Po dokončení Fragment shaderu jsou ještě provedeny per-fragment operace (Blending, Stencil test) a následně je výsledná barva uložena do Framebufferu.

Posledním a speciálním typem shaderu je Compute shader. Ten se v klasické vykreslovací pipeline nenachází, což je způsobeno jeho určením. Je určen pro GPGPU (General purpose computing on GPU), tedy výpočetně náročné úlohy, které je možné akcelerovat jejich paralelním během na grafické jednotce. Takovouto funkcionalitu nabízejí knihovny OpenCL nebo CUDA (schopná běhu pouze na GPU značky nVidia). Jedná se o nejnovější shader, jelikož byl přidán ve verzi 4.3 (aktuální je 4.5), proto je možné že jej některé starší grafické karty nemusí podporovat.

[1] Model View Projection matice - transformuje pozice vrcholů do tzv. clip space.

Programovacím jazykem pro psaní shader programů je jazyk GLSL (GL shading language). Ten svojí syntaxí připomíná jazyk C. Podporuje i podobné konstrukce jako jsou makra, tvorba struktur, unií nebo funkcí. Není však možné využít ukazatelů nebo rekurzivního volání funkcí, což je způsobeno konstrukcí GPU jednotek.

Při tvorbě aplikace bude OpenGL použito pro vykreslování scény pomocí metody Path Tracing, přičemž budou použity Vertex a Fragment shadery.



Obrázek 5.1: Vykreslovací pipeline v OpenGL 4.5 (oranžově označené sekce jsou programovatelné). Převzato z webu[3].

5.2 Rozvržení tříd

Při návrhu aplikace je dobré využití objektově orientovaného programování. Jednotlivé komponenty celku jsou tak logicky odděleny do jednotlivých tříd a jejich data zapouzdřena.

Třída `Window` zajišťuje práci s hlavním oknem aplikace, počínaje jeho vytvořením přes obsluhu událostí od klávesnice případně myši, až po jeho ukončení a uvolnění jím alokovaných zdrojů. Tato třída bude kooperovat s knihovnou GLFW a poskytovat tak jakési rozhraní pro práci s potřebnými funkcemi této knihovny (např. dotázání na výskyt událostí, prohození bufferů ad.).

Třída `Shader` má na starosti rutiny, jako jsou načtení souboru se zdrojovým kódem shader programu v jazyce GLSL, zkompilování a sestavení tohoto programu a také jeho nastavení jako aktivní shader program, který bude vykonáván.

Informace o scéně, konkrétně jednotlivé objekty, jejich počet, informace o světle apod., sdružuje třída `Scene`. Ta taktéž tyto informace poskytuje k jejich odeslání do grafické paměti, kde jsou využity při kreslení snímku. Kameru a její atributy zapouzdřuje třída `Camera`. Primitiva, jakými jsou koule, zapouzdřuje třída `Sphere`.

Zahájení kreslení, nastavování uniformních proměnných, předání dat do grafické paměti, toto jsou hlavní úkoly třídy `Render`. V podstatě by se dalo říct, že je tato třída prostředníkem mezi částí programu běžící na CPU a druhou částí na GPU. Třída tedy logicky pracuje s knihovnou OpenGL.

[3] http://www.opengl2go.net/wp-content/uploads/2015/11/OpenGL-4.0-Programmable-Shader-Pipeline1-1024x400.png

Načítání modelů ze souboru zajišťuje třída `ModelLoader`. Jedná se o jednoduchý parser, jenž je schopen načítat modely ve formátu OBJ. Načtená data poté mohou být předány ve formě vektoru jednotlivých trojúhelníků, z nichž je model složen.

Třída `Octree`, jak název napovídá, zapouzdřuje oktalový strom, konkrétně zajišťuje operace pro jeho sestavení nad zadanými daty. Výsledek, ve formě jednotlivých uzlů oktalového stromu a indexů na jednotlivé trojúhelníky (jsou zde z důvodu nepodporovaných ukazatelů v jazyce GLSL), bývá předáván třídě `Render`, která zajistí nahrání těchto dat na grafickou kartu.

Pro diagnostické účely v návrhu figuruje třída `FPSMeter`, což je jednoduchá třída pro měření času vykreslení snímku a celkového času kreslení. Pro měření času je využito OpenGL queries (asynchronní dotazy na určité údaje od OpenGL), které zajišťují vyšší přesnost než v případě měření pomocí hodnoty systémového času.

5.3 Implementace CPU části

Průběh celé aplikace bude řízen ze strany CPU. Ta bude mít na starosti načtení potřebných dat, sestavení datových struktur a jejich předání do grafické paměti, zpracování vnějších událostí a především v neposlední řadě hlavní část, a to řízení vykreslování a následného zobrazování jednotlivých snímků.

Následující sekce se věnuje popisu a praktické realizaci výše zmíněných částí aplikace.

5.3.1 Zobrazení aplikace

Prvním zásadním úkolem je zobrazení aplikace, konkrétně vytvoření okna, kam bude výsledek zobrazován a také OpenGL kontextu (obsahuje buffery, textury a další objekty využívané při kreslení). Tyto úkony jsou zajištěny pomocí knihovny GLFW, která při vytváření okna zajistí i vytvoření OpenGL kontextu. Na počátku jsou však knihovně předány informace, jako jsou rozměry okna, viditelnost nebo např. titulek okna.

Dále je zapotřebí zajistit odchytávání událostí. To je zajištěno tím, že se nastaví k příslušné vzniknuvší události ukazatel na funkci (tzv. callback), která zajistí patřičnou akci při jejím vzniku. V této aplikaci je využito odchytávání událostí při stisknutí klávesy, myši, změny polohy myši a změně velikosti okna.

Před vykreslováním je nutné také nastavit vytvořený kontext jako aktivní, aby se vykreslovalo do daného okna. Dále je potřeba aktivovat blending (průhlednost), jelikož jednotlivé snímky budou postupně tvořit výsledný snímek (přesněji řečeno bude se k němu konvergovat). Váhu daného snímku bude udávat hodnota jeho alpha kanálu. Právě ten je OpenGL ignorován v případě nepovoleného blendingu.

Jednou z možností aplikace je zachycení výsledného vykresleného snímku, jenž je následně uložen do formátu PNG. Samotné získání dat z framebufferu není obtížné, složitější částí je komprese těchto dat do formátu PNG. Z tohoto důvodu je použita knihovna lodepng[4], jenž umožňuje snadnou práci se soubory v tomto formátu (kromě ukládání do souboru dokáže např. načítat textury).

Knihovna GLFW bude kromě vytvoření kontextu zajišťovat i jeho udržování a při ukončení aplikace také jeho odstranění a odalokování použitých zdrojů. Veškerá práce s oknem je v aplikaci zapouzdřena do třídy `Window`.

[4] Webové stránky projektu: http://lodev.org/lodepng/

5.3.2 Reprezentace scény

Další důležitá část implementace se týká reprezentace, uložení geometrie scény a jejího předání do grafické paměti. Správa nad celou scénou je starostí třídy `Scene`, jenž uchovává objekty ve formě vektoru jednotlivých primitiv. Do scény je možné vložit komplexnější objekt jakým je trojrozměrný model. Pro jeho načtení je využívána třída `ModelLoader`, jenž dokáže načítat modely ve formátu OBJ. Načtení probíhá v několika fázích, nejprve jsou přečteny a dočasně uloženy údaje o pozicích vrcholů, normály příp. i texturovací souřadnice. Následně jsou tyto údaje využity při sestavování jednotlivých primitiv (trojúhelníků).

Předání údajů o geometrii je možné za pomoci uniformních proměnných (jejich hodnota je nastavována ze strany CPU a je pro všechny invokace shader programů stejná). Ty jsou v aplikaci využity především pro předání informací, jako jsou pozice světla, pozorovatele příp. počet objektů ve scéně. Využití uniformních proměnných pro předání geometrie scény na grafickou paměť však není vhodné z několika hledisek. Tím hlavním je relativně omezené množství dat, které lze pomocí uniformních proměnných nastavit (záleží na konkrétní grafické kartě), dalším hlediskem je nutnost znát předem množství nastavovaných dat, což není vždy možné.

Údaje o geometrii scény jsou předávány pomocí SSBO (Shader Storage Buffer Object), jenž odstraňuje nevýhody spojené s uniformními proměnnými. Velikost přenášených dat je omezena pouze kapacitou grafické paměti a množství těchto dat stačí znát až před samotným vykreslováním. Před samotným předáním dat je nutné uvážit fakt, že na rozdíl od operační paměti jsou data na grafické paměti zarovnávána (konkrétně u SSBO podle normy std430). Z tohoto důvodu je třeba data v operační paměti správně upravit (co se týče jejich uložení v paměti), jinak hrozí chybná interpretace předaných dat na straně GPU.

5.3.3 Sestavení oktalového stromu

Při sestavování oktalového stromu nad danou scénou je zapotřebí brát v potaz nemožnost použití ukazatelů na GPU jednotce. Z tohoto důvodu jsou jednotlivé uzly stromu vkládány do vektoru, přičemž namísto ukazatelů jsou přistupovány pomocí jejich indexu v daném vektoru. Dále je nutné si u každého uzlu pamatovat, která primitiva se v něm nachází, což by se opět ideálně řešilo pomocí ukazatelů, ale je nutné použít stejné řešení jako při ukládání jednotlivých uzlů stromu, tedy uložit patřičná primitiva do vektoru a uzlu předat index prvního z těchto primitiv ve vektoru a také jejich počet.

Samotné sestavení stromu není příliš složité, a jelikož se v rámci této aplikace nejedná o kritickou část, nejsou nezbytně nutné různé optimalizace. Princip je založen na testování přítomnosti primitiva v daném uzlu, v případě příliš velkého počtu primitiv v daném uzlu dojde k jeho rozdělení. Takto se pokračuje, dokud strom není optimálně rozdělen. Pseudokód algoritmu pro sestavení oktalového stromu je zobrazen níže (viz algoritmus 1).

Po sestavení ještě probíhá odstranění redundantních informací vzniklých při tomto procesu. Konkrétně se jedná o seznam primitiv náležící neterminálním uzlům.

Pro následné předání nově vzniknutých struktur oktalového stromu do grafické paměti je použito stejně jako u geometrie scény SSBO. I zde platí, že je nutné respektovat stejné ukládání do operační paměti, jako je ukládáno do grafické paměti.

Algorithm 1 Sestavení oktalového stromu

```
function buildOctree
    foreach uzel in uzly v octree
        if listový uzel
            foreach primitivum in geometrie scény
                if primitivum protíná uzel
                    vlož primitivum do uzlu
                end if
            end foreach
            if počet primitiv v uzlu > MAX
                rozděl uzel
            end if
        end if
    end foreach
end function
```

5.3.4 Hlavní smyčka aplikace

Jádrem celé aplikace je smyčka, která ve svém těle zajišťuje vykreslení nového snímku. Podmínkou ukončení smyčky je zavření okna aplikace.

Obsahem této smyčky je aktualizace hodnot některých uniformních proměnných (např. pozice pozorovatele, světla), zjištění času vykreslování (pomocí OpenGL queries) a jeho zobrazení, rutinní funkce jako dotázání na výskyt událostí a prohození framebufferu (standardně se používají dva framebuffery - jeden pro ukládání výsledku kreslení a druhý pro zobrazování) a především zajištění vykreslování.

K vykreslování jsou předávány souřadnice průmětny prostřednictvím VBO (Vertex Buffer Object), tedy způsobem, jímž bývá standardně předávána geometrie na vykreslení. Tato průmětna je však ještě rozdělena na menší části. Důvod jejího rozdělení pramení z výpočetní náročnosti Path Tracingu, a zároveň nutnosti provedení shader programu během krátké časové periody (v případě nedodržení může nastat pád ovladače grafické karty). Snímek je tím pádem konstruován postupně po částech.

5.4 Implementace GPU části

Výpočet snímků scény bude probíhat na grafické jednotce, jenž umožní rychlejší získání výsledků než při běhu na procesoru. Hlavním úkolem zde bude tedy výpočet barvy pixelů, přičemž za touto samotnou funkcí se skrývá několik dalších součástí. Tou možná nejzásadnější je hledání průsečíku paprsku s objekty, tím pádem důležitou roli hraje i strategie průchodu oktalovým stromem, jímž je vykreslování akcelerováno.

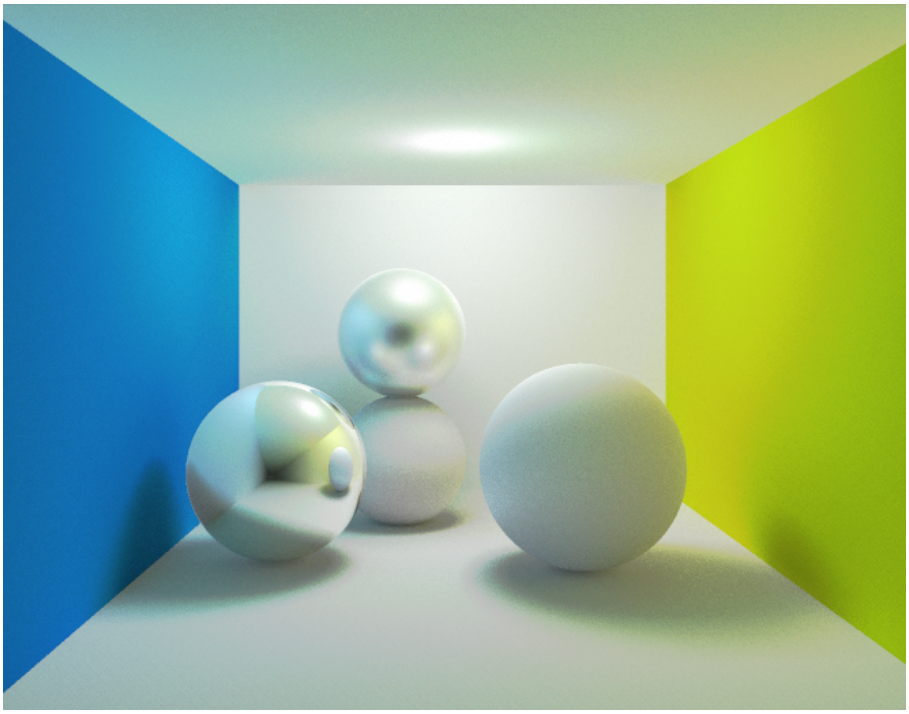
Následující část popisuje hlavní funkcionalitu a realizaci programové části, běžící na GPU.

5.4.1 Výpočet barvy

Výpočet výsledné barvy pixelu je poměrně komplexní činnost skládající se různých funkcí zajišťujících např. detekci kolize paprsku s objektem, výpočet osvětlení v daném bodě, generování náhodných čísel ad. Při implementaci této části je taktéž nutné brát v potaz jistá omezení jazyka GLSL.

Nejprve je nutné simulovat rekurzi sloužící ke sledování jednotlivých úseků cesty paprsku. Ta je simulována cyklem, jehož ukončovací podmínka je dosažení maximální délky cesty. V těle cyklu se nejprve testuje přítomnost kolize paprsku a objektu ve scéně (jejichž matematický princip je popsán v kapitole o Path Tracingu). K akceleraci této činnosti se využívá oktalového stromu a procházení jeho struktury (podrobněji rozebráno v následující části). Následuje výpočet osvětlení v daném bodě, který je připočítán k výsledné barvě. Na závěr může být cesta předčasně ukončena pomocí Russian Roulette (technika sloužící zde pro ukončení cesty na základě určité pravděpodobnosti, ta bývá často založena na relfektivitě materiálu v daném bodě, tím pádem v tmavých místech je větší pravěpodobnost ukončení) nebo se v ní pokračuje dál, v tom případě se náhodně zvolí směr kudy bude cesta paprsku pokračovat. Po vytvoření nového směru je však ještě nutné ho převést z globálního souřadného systému do lokálního souřadného systému (osy jsou odvozeny od normály v daném bodě, samotná normála je pak jednou z os, další dvě se označují jako tangenta a binormála). Pseudokód algoritmu sledování cest je ukázán v algoritmu 2.

Jak jde vidět, důležitou roli zde hraje náhodnost. To znamená, že jde o kritickou část algoritmu, kdy je potřeba zajistit kvalitní zdroj náhodných čísel. Jistým problémem může být, že výpočet jednotlivých pixelů běží nezávisle na sobě, což by např. v případě implementace lineárního kongruentního generátoru v shaderu mohlo mít za následek stejná čísla v každém pixelu. Důležité je tedy zajištění generování náhodných čísel nezávislého na pixelu a zároveň nezávisle na vykreslovaném snímku. Toto je zajištěno tím, že je na straně CPU pomocí Mersenne Twister vygenerováno při každém snímku určité množství náhodných čísel a předáno do GPU za použití uniformních proměnných. Přistupováno je k nim nakonec pomocí souřadnic daného pixelu. Výsledný vyrenderovaný snímek lze vidět na obrázku 5.2.



Obrázek 5.2: Výsledný snímek z implementované aplikace.

Z důvodu výkonnosti aplikace je taktéž důležité mít na paměti, jak správně psát kód pro GPU. Především je nutné minimalizovat počet větvení toku, jelikož se v některých případech mohou provést všechny větve. Dále je třeba mít na paměti, že data shaderu jsou ukládána do registrů a v případě velkého množství dat, dojde k obsazení velkého množství registrů, vedoucího k redukci počtu současně běžících invokací shaderu.

Algorithm 2 **Nerekurzivní verze algoritmu sledování cest**

```
function pathTrace
    while delka < MAX_DELKA
        if nenalezen průsečík paprsku s objektem
            return ČERNÁ
        end if
        získání informací o bodu průsečíku
        výsledná barva = výsledná barva + Osvětlení(bod průsečíku)
        if náhodné ukončení cesty pomocí Russian Roulette
            return výsledná barva
        end if
        nový směr paprsku = náhodně zvolený směr podle materiálu
        delka = delka + 1
    end while
    return výsledná barva
end function
```

5.4.2 Průchod oktalovým stromem

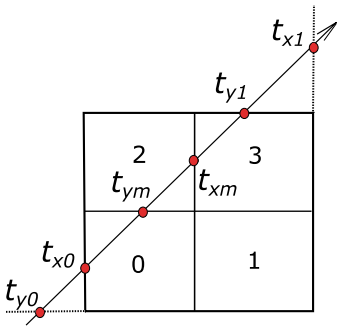
Místem, na kterém stojí rychlost výpočtu, je právě průchod oktalovým stromem. Algoritmy určené pro tuto činnost jsou stejně jako výpočet barvy založeny na rekurzi. Ta je zde taktéž nahrazena cyklem, avšak zde je nutné si pamatovat informace o uzlech, kterými se procházelo (z důvodu možného backtrackingu při průchodu). Pro ukládání těchto informací slouží struktura simulující funkci zásobníku. Při samotné implementaci je důležité si promyslet, jaké množství a jakým způsobem budou tyto informace ukládány. Toto rozhodnutí může mít v důsledku přímý dopad na rychlost, jelikož, jak už bylo zmíněno, může být obsazeno velké množství registrů, což způsobí menší počet současně běžících invokací shaderů.

Samotný princip procházení této datové struktury byl naznačen v kapitole o oktalovém stromu. Jelikož se jedná o poměrně kritickou část pro běh aplikace, byly zde také navrženy rychlejší varianty algoritmů procházení. V této aplikaci je konkrétně využito zmíněného algoritmu od Jorge Revellese.

Myšlenka tohoto algoritmu spočívá v tom, že není potřeba pro každý uzel nalézat průsečíky. Výpočet průsečíku probíhá pouze jednou, a to na jeho výskyt paprsku s kořenovým uzlem stromu. Přesněji řečeno nepočítá průsečík jako takový, ale průsečík paprsku s jednotlivými rovinami. Na základě jejich hodnot pak lze pomocí tabulky určit, kterým uzlem bude paprsek vstupovat, a na základě další tabulky spolu s hodnotami průsečíků je možné určit, jakým uzlem bude paprsek dále postupovat (ukázka základního principu algoritmu je ilustrována na obrázku 5.3). Algoritmus je dále navržen tak, že směr paprsku v rámci jednotlivých os musí být vždy kladný, v opačném případě je paprsek zrcadlen podle příslušné osy a uložen o této skutečnosti příznak, jenž bude zohledněn při procházení.

Díky tomuto přístupu lze ušetřit značné množství výpočtů a ukládat menší množství informací o daném uzlu, navíc není potřeba průsečíky nějakým způsobem řadit podle vzdá-

lenosti, což by taktéž znamenalo nepříjemné zpoždění a mělo tak logicky negativný vliv na rychlost kreslení.



Obrázek 5.3: Ilustrace průchodu paprsku skrz uzel oktalového stromu. Použitý algoritmus využívá znalosti hodnot průsečíků s jednotlivými rovinami k určení procházených uzlů. V tomto případě, kdy průsečík t_{x0} má menší hodnotu než t_{ym} lze určit, že paprsek vstupuje do uzlu 0. Při výběru, jakým uzlem se bude pokračovat jsou využity pouze průsečíky t_{xm} a t_{ym} (jenž leží uprostřed mezi t_{x0} a t_{x1} respektive t_{y0} a t_{y1}). Zde má menší hodnotu t_{ym}, což určuje pokračování uzlem 2 a následně 3 (kde by měl paprsek končit vždy).

Kapitola 6

Testování aplikace

Důležitým aspektem při zobrazování scény jakoukoliv technikou je rychlost výpočtu snímku. U metody Path Tracing toto platí dvojnásob, jelikož je tato technika zatížena šumem a pro získání kvalitního výsledku je zapotřebí výpočet určitého množství snímků zajišťující kvalitní aproximaci výsledného snímku.

V této kapitole bude naměřena rychlost kreslení různě složitých scén při akceleraci oktalovým stromem a porovnána s neakcelerovanou variantou.

6.1 Popis testování

Testování aplikace, jak již bylo zmíněno, proběhne nad scénami s různě složitou geometrií, a tím pádem i různými výpočetními a paměťovými nároky. Testováno bude na sestavě, jenž je specifikována v tabulce 6.1.

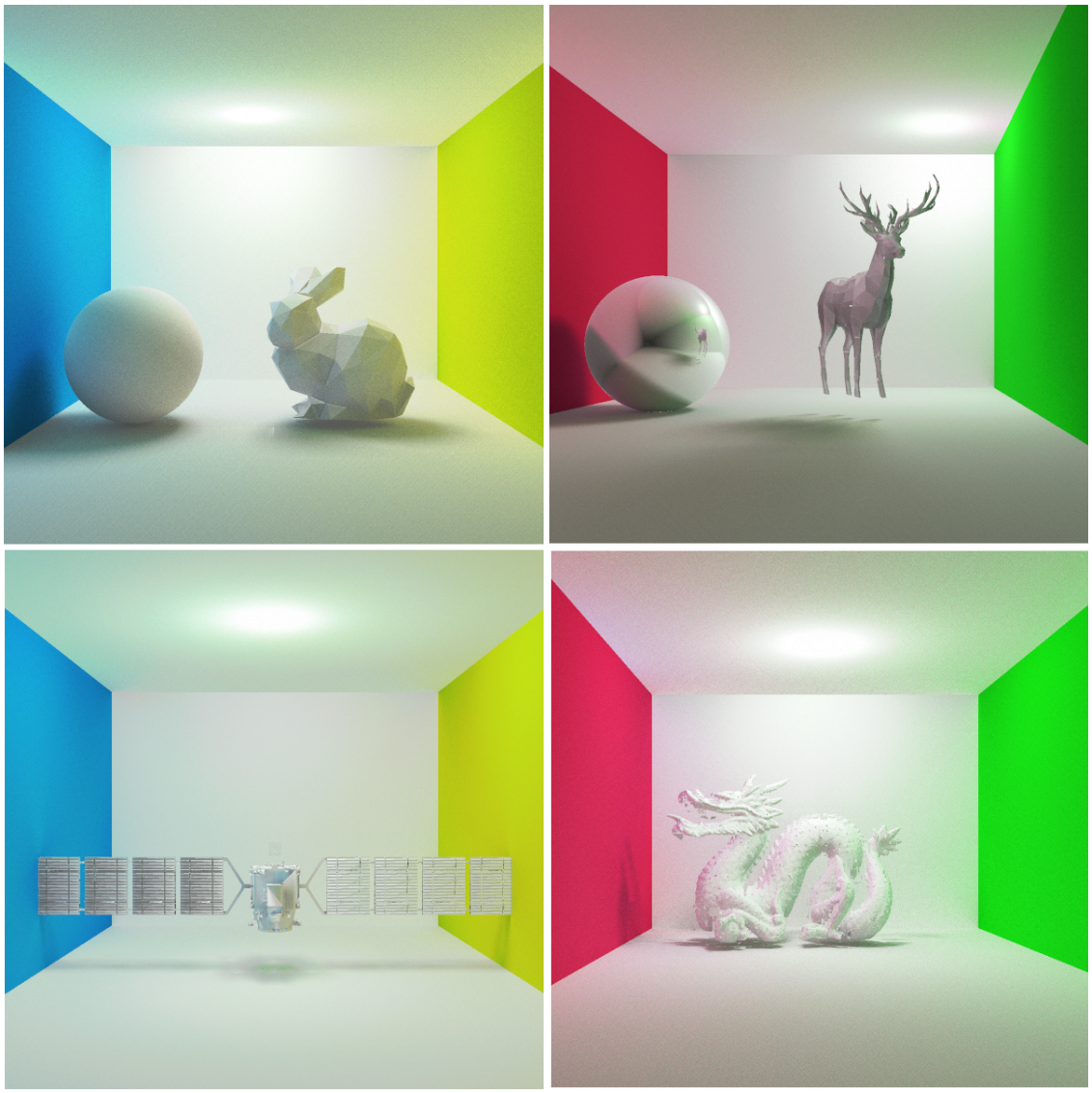
OS	Windows 10 Pro 64 bit
CPU	Intel Core i5 6402P
GPU	nVidia GeForce GTX 950
RAM	8GB DDR4

Tabulka 6.1: Specifikace sestavy na níž bude probíhat testování.

Zobrazování bude probíhat v okně s rozlišením 600 × 600 pixelů, což znamená výpočet 360 000 primárních paprsků (vyslaných od pozorovatele do scény) v jednom snímku. Výsledný snímek pro testovací účely se bude skládat z 1 000 snímků (nebo-li vzorků na pixel), což je hodnota, kdy šum ve výsledném snímku běžně nebývá prakticky viditelný.

Sledovány budou údaje o celkovém čase kreslení, paměťové nároky aplikace, údaje o oktalovém stromu (výška, počet uzlů) a počet paprsků za sekundu.

Pro účely testování jsou připraveny 4 scény. Jedná se o scény Cornell box, kdy v každém případě obsahuje jiné objekty. První obsahuje zjednodušený model Stanford Bunny, skládající se z 336 trojúhelníků, druhá scéna již obsahuje složitější geometrii, konkrétně model jelena čítající 1 503 trojúhelníků. Další scéna obsahuje model vesmírného teleskopu skládající se z 13 812 trojúhelníků. Nejsložitější geometrií disponuje poslední scéna obsahující model Stanford Dragon složený ze 100 000 trojúhelníků. Podoby jednotlivých scén jsou vyobrazeny na obrázku 6.1.



Obrázek 6.1: Obrázky scén na nichž bude probíhat měření. Vlevo nahoře se nachází první scéna s modelem zajíce čítající 336 trojúhelníků, napravo je zobrazena scéna s modelem jelena sestávající z 1 503 trojúhelníků a dole jsou zobrazeny scény s nejsložitější geometrií obsahující model vesmírného teleskopu skládajícího se ze 13 812 trojúhelníků a model draka tvořeného 100 000 trojúhelníky.

6.2 Výsledky testování

Získané výsledky jsou zobrazeny v tabulce 6.2, kde celkový čas udává čas kreslení 1 000 snímků, využitá paměť udává množství využité grafické paměti při běhu aplikace, následují informace o oktalovém stromu jako jsou jeho výška a počet uzlů z kterých je složen, posledním měřeným údajem je počet paprsků za sekundu. Tento údaj říká množství zpracovaných paprsků (nalezly svojí kolizi s některým objektem ve scéně), přičemž je započítáván každý nově odražený paprsek a také stínové paprsky.

Scéna	Varianta	Celkový čas [s]	Využitá paměť [MB]	Počet uzlů Octree	Výška Octree	Paprsky za sekundu
1	**bez Octree**	535.2	57	-	-	4 439 461.8
	s Octree	181.4	59	641	7	13 098 125.7
2	**bez Octree**	3 586.9	57	-	-	662 410.4
	s Octree	177.4	60	3 761	9	13 393 461.1
3	**bez Octree**	-	-	-	-	-
	s Octree	198.1	67	34 537	10	11 993 942.5
4	**bez Octree**	-	-	-	-	-
	s Octree	238.1	86	127 865	12	9 979 000.4

Tabulka 6.2: Získané hodnoty při vykreslování jednotlivých scén.

První scénou, nad níž probíhalo měření, byla scéna s modelem zajíce. Výsledky jsou v prvních dvou řádcích tabulky. Jak jde z dat vidět, průměrná doba kreslení jednoho snímku je přibližně 500 ms, což je poměrně dlouhá doba. Toto zároveň demonstruje náročnost této zobrazovací metody, kdy již při ne příliš složité geometrii scény (336 trojúhelníků) běží aplikace se zobrazovací frekvencí přibližně 2 snímky za sekundu.

V případě použití oktalového stromu bylo dosaženo mírného zrychlení, konkrétně 2.3 krát rychlejší vykreslování. Takto mírné zrychlení lze odůvodnit tím, že samotné procházení strukturou oktalového stromu vyžaduje určitou režii. Geometrie scény navíc také není příliš složitá, tudíž sekvenční testování průsečíku s jednotlivými primitivy je zatím výpočetně únosné. Jednoduše řečeno, scéna ještě není dostatečně složitá, aby oktalový strom dokázal značněji urychlit výpočet snímku.

Co se týče paměťových nároků, tak ty jsou v obou případech téměř podobné. Rozdíl tvoří struktury využívané oktalovým stromem (tedy uzly a indexy vrcholů).

Výrazně pomalejší vykreslování se již projevuje u druhé scény, kdy vykreslení 1 000 snímků trvá necelou hodinu, přičemž jeden snímek se vykresluje více než 3 sekundy. Zde se již akcelerace jeví téměř jako nutnost.

Právě akcelerovaný výpočet dosahuje v tomto případě na rozdíl od předchozího velmi výrazného zrychlení. Důvodem této akcelerace (přibližně 20-ti násobné zrychlení) oproti minulému je ten, že paprsek v nejhorším případě provede pouze pár desítek testů na průsečík, navíc mnoho paprsků pouze projde skrze prázdné uzly a protne se se stěnou, zatímco v neakcelerované variantě se pro každý paprsek provádí více než 1 000 testů na přítomnost průsečíku.

V tomto případě se oktalový strom skládá z 3 761 uzlů, přičemž jeho výška je 9. Paměťová režie spojená s oktalovým stromem narostla jen mírným způsobem. Dá se říct, že zde značně urychlí výpočet za přijatelné paměťové nároky.

Další zvláštností může být rychlejší výpočet této scény i přes fakt, že minulá scéna obsahovala mnohem méně trojúhelníků. Podobný čas kreslení je možné vysvětlit tím, že se procházení stromu provádí do podobné hloubky a v listových uzlech leží taktéž podobný počet primitiv (tzn. podobné množství výpočtů průsečíku). Mírné zrychlení lze odůvodnit především tvarem modelu umožňujícím adaptivnější sestavení oktalového stromu. V tomto případě strom obsahuje více prázdných listových uzlů, kterými paprsek jen projde aniž by se musely provádět nějaké výpočty.

Výsledky z měření u třetí scény se nacházejí v 5. a 6. řádku tabulky. Jde vidět, že varianta bez oktalového stromu již nezvládla výpočet této scény. S ohledem na rychlost výpočtu u minulých scén lze odhadovat, že by výpočet mohl trvat i desítky sekund.

Varianta využívající oktalový strom si však v této části připisuje velmi dobré výsledky, prakticky totožné jako u minulých dvou scén i přes fakt, že geometrie scény čítá daleko více primitiv. Důvodem dosaženého času vykreslování je podobná činnost jako u předchozích dvou scén (tedy procházení stromem do podobné hloubky a testování průniku paprsku s až desítkami trojúhelníků).

U této scény sestává strom z 34 537 uzlů, kdy paměťové nároky aplikace činí 67 MB. Nárůst oproti minulé scéně je tedy 7 MB, odpovídající paměťové režii oktalového stromu.

Jelikož třetí scénu již neakcelerovaná varianta nevykreslila, logicky by si neporadila ani s poslední scénou s výrazně vyšším počtem trojúhelníků. Čas kreslení lze tedy jen odhadovat. Konkrétní odhad by byl již v řádech minut na výpočet 1 snímku, přičemž výpočet 1 000 snímků by mohl trvat až mnoho hodin.

Akcelerovaná varianta je oproti předchozím případům sice pomalejší, přesto se dá říci, že vzhledem ke složitosti geometrie je vykreslovací čas celkem slušný. Výška stromu oproti předchozí scéně narostla jen mírně. Výrazněji se však zvýšil počet uzlů oktalového stromu. Paměťové nároky vzrostly oproti akcelerované variantě u minulé scény o necelých 20 MB, což se dá označit za přijatelné vzhledem k dosaženému výkonu.

Ve výsledku se dá o oktalovém stromu říci, že za mírně vyšší paměťové nároky v řádech MB, což je v současnosti, kdy mají grafické paměti kapacitu v jednotkách GB, relativně malé množství, získáme možnost kreslit scény se složitější geometrií ve slušném čase.

Kapitola 7

Závěr

Cílem této bakalářské práce byla implementace aplikace demonstrující zobrazování scény pomocí techniky globálního osvětlení, konkrétně pomocí metody sledování cest. Na počátku bylo však nastudování této metody a pochopení matematických postupů potřebných při implementaci tohoto algoritmu.

Vzhledem k výpočetní náročnosti tohoto algoritmu byla akcelerována na GPU za pomoci API OpenGL. Tato skutečnost měla vliv na návrh implementované aplikace, kdy musel být brán v úvahu princip funkčnosti OpenGL. Další fáze akcelerace běhu implementované aplikace spočívala ve využití datové struktury oktalový strom. Tato část rovněž požadovala nastudovaní principů sestavování této struktury nad určitou geometrií a především nastudování technik procházení touto datovou strukturou.

Výsledkem práce je aplikace schopná zobrazovat snímky technikou sledování cest. V jejíchž možnostech je vykreslování jednodušších scén v reálném čase (doba kreslení jednoho snímku se pohybovala okolo 20ms), složitější scény je možné akcelerovat pomocí oktalového stromu, kdy je možné významně urychlit běh vykreslování a v relativně krátkém čase kreslit i složitější scény. Důkazem toho může být i přibližně dvacetinásobné zrychlení běhu při relativně přijatelných paměťových nárocích, nebo samotná možnost vykreslení scén s velice složitou geometrií, kde neakcelerovaná varianta výkonnostně selhává. Vykreslování jako takové dokáže poměrně rychle konvergovat, což může být důsledkem vzorkování v závislosti na daném materiálu (nejen čistě náhodně).

Co se týče rozšíření stávající aplikace, tak zde existuje několik ohledů, ve kterých by bylo možné aplikaci obohatit. První rozšíření by mohlo spočívat v zajištění větší interakce uživatele se scénou, konkrétně by si uživatel mohl libovolně rozmisťovat objekty v rámci scény, v tomto případě by bylo zapotřebí implementace techniky Pickingu. Další rozšíření by se mohlo týkat akcelerace aplikace pomocí jiné datové struktury než oktalovým stromem a poté vzájemné porovnání jejich zrychlení vykreslování a paměťových nároků. Takovou strukturou by mohlo být BVH (Bounding Volume Hierarchy) [3], kdy scéna není dělena uniformně, ale do hierarchie obalových těles, jenž mohou mít různé podoby (koule, hranol, konvexní těleso). BVH tak umožňuje adaptivnější dělení scény a obecně se udává, že BVH bývá rychlejší než oktalový strom. Toto by však mohlo být předmětem zkoumání při pokračování v této práci. Pro důkladné porovnání těchto struktur by mohly být použity scény s modely čítající statisíce popř. miliony trojúhelníků.

Literatura

[1] Agate, M.: The HERO algorithm for ray-tracing octrees. In *Advances in Computer Graphics Hardware IV*, New York: Springer-Verlag, 1991, ISBN 978-3-642-76300-7, s. 61–73.

[2] Binder, K.: *Mathematical Tools for Physicists*. Weinheim: Wiley, 2005, ISBN 3-527-40548-8, s. 249–280.

[3] Ericson, C.: *Real-Time Collision Detection*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, ISBN 1-55860-732-3, s. 236–237.

[4] Kajiya, J. T.: The Rendering Equation. *SIGGRAPH*, ročník 20, č. 4, 1986: s. 143–150, ISSN 0097-8930, doi:10.1145/15922.15902.
URL http://doi.acm.org/10.1145/15922.15902

[5] Marsaglia, G.: Random Number Generation. In *Encyclopedia of Computer Science*, Chichester, UK: John Wiley and Sons Ltd., ISBN 0-470-86412-5, s. 1499–1503.
URL http://dl.acm.org/citation.cfm?id=1074100.1074752

[6] Meagher, D.: Geometric modeling using octree encoding. *Computer graphics and image processing*, ročník 19, č. 2, 1982: s. 129–147.
URL https://www.researchgate.net/profile/Donald_Meagher/publication/222384835_Geometric_Modeling_Using_Octree-Encoding/links/58adb8eeaca2725b540dd5df/Geometric-Modeling-Using-Octree-Encoding.pdf

[7] Möller, T.; Trumbore, B.: Fast, Minimum Storage Ray/Triangle Intersection. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA: ACM, 2005, doi:10.1145/1198555.1198746.
URL http://doi.acm.org/10.1145/1198555.1198746

[8] Nicodemus, F. E.: Directional Reflectance and Emissivity of an Opaque Surface. *Appl. Opt.*, ročník 4, č. 7, Jul 1965: s. 767–775, doi:10.1364/AO.4.000767.
URL http://ao.osa.org/abstract.cfm?URI=ao-4-7-767

[9] Revelles, J.; Urena, C.; Lastra, M.: An efficient parametric algorithm for octree traversal. *Journal of WSCG*, ročník 8, č. 1-3, 2000, ISSN 1213-6972.
URL https://otik.uk.zcu.cz/bitstream/11025/15821/1/X31.pdf

[10] Whitted, T.: An Improved Illumination Model for Shaded Display. *Commun. ACM*, ročník 23, č. 6, Červen 1980: s. 343–349, ISSN 0001-0782, doi:10.1145/358876.358882.
URL http://doi.acm.org/10.1145/358876.358882