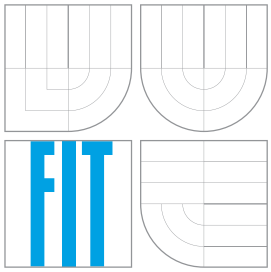
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PATHTRACING NA GPU

PATHTRACING ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ONDŘEJ HÁJEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUKÁŠ POLOK

BRNO 2016

Abstrakt

Tato práce se zaměřuje na akceleraci metody globálního osvětlení Pathtracing. Cílem této práce je demonstrace a srovnání výkonnosti různých implementací této metody a to na GPU i CPU. Implementace bude obsahovat dvě scény, jednodušší pro ověření korektnosti implementace algoritmu a složitější pro demonstraci různých efektů. Efektivita a účinnost jednotlivých implementací bude rozebrána a porovnána mezi sebou. Dále bude diskutována možnost rozšíření aktuálního programu.

Abstract

This bachelor thesis focuses on acceleration of global illumination method called Pathtracing. The goal of this thesis is a demonstration and performance comparison of different implementations of this method on GPU as well as CPU. Implementation will compose of two scenes. One simpler to verify correctnes of used algorithm and one more complicated to demonstrate various effects. The effectiveness and efficiency of said implementations will be analyzed and compared. Additionally, future program expansion and improvements will be discussed.

Klíčová slova

OpenGL, OpenCL, GLFW, GPGPU, Sledování cest, Sledování paprsku

Keywords

OpenGL, OpenCL, GLFW, GPGPU, Pathtracing, Raytracing

Citace

HÁJEK, Ondřej. *Pathtracing na GPU*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Polok Lukáš.

Pathtracing na GPU

Prohlášení

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

.....
Ondřej Hájek
18. května 2016

Poděkování

Rád bych poděkoval svému úžasnému vedoucímu, panu Ing. Lukáši Polokovi za jeho čas a trpělivost při zodpovídání mých otázek. Také bych rád poděkoval mým nejbližším přátelům, kteří mi vždy dodali energii při řešení této práce.

Obsah

Kapitola 1

Úvod

Nalezení ideálního řešení vykreslování, které je věrné realitě, už dnes nepředstavuje takový problém jako v 80. letech. Moderní doba nabízí nepřeberné množství algoritmů, které tuto problematiku řeší, ať už v podobě technického dema nebo jako renderovací engine v grafickém modelovacím programu. Výzvu v dnešní době představuje vykreslování, které je nejen věrné realitě, ale především dokáže vykreslovat scénu v reálném čase (tj. není potřeba čekat na výsledný *render*). Díky neustále zrychlujícímu se hardware a příchodu knihoven umožňujících paralelní zpracování operací(viz. kapitoly 4.3 a 4.4) se ukazuje, že jednou bude možné využít metod realistického zobrazování v tak dynamickém prostředí jako jsou hry (příklad použití Pathtracingu ve hře je zmíněn v části 3.3).

Metoda **Pathtracing** (v překladu též sledování cest), je *Monte Carlo* technika (viz. sekce 2.4) pro renderování realistických 3D scén, jejichž vzhled a osvětlení simulují fyzikální zákony. Tato metoda dokonce dokáže zobrazit pokročilé efekty jako jsou odrazy, lom světla, měkké stíny nebo kaustiku[1], a to vše bez dalších modifikací. Tyto efekty je potřeba implicitně přidat do tradiční metody Raytracing, tudíž lze Pathtracing považovat za jeho vylepšenou verzi. Výhody však doprovázejí zásádní problémy, které stojí v cestě nasazení tohoto algoritmu při zobrazování v reálném čase. První očividný nedostatek je časová náročnost kvůli obrovskému množství vrhaných paprsků, jejichž počet roste s rozlišením renderu. Druhým nedostatkem je počáteční zašuměnost vyvolaná náhodným vrháním paprsků. Tento problém se odstraní delším počítáním algoritmu a průměrováním výsledků na základě dostatečného množství vzorků. Zvyšuje se ovšem už tak velký potřebný čas pro dosažení přijatelného výsledku ve formě realisticky vykreslené barvy pixelu.

Práce je rozdělena na teoretickou a praktickou část. Teoretická část nejprve v kapitole 2 popisuje samotný algoritmus Pathtracing spolu s dalšími algoritmy, ze kterých vychází. Následuje kapitola 3 která analyzuje existující řešení, zpracovávající algoritmus Pathtracing. Teoretickou část uzavírá kapitola 4 popisující použité technologie práce.

První kapitolou praktické části je návrhová a implementační kapitola 5. Kapitola nejdříve popisuje v části 5.1 vytvoření okna programu a inicializaci potřebných datových struktur. Další zajímavou částí je část 5.2 popisující samotný vzhled vykreslovaných scén. Nejdůležitější podkapitolou je však sekce 5.3, která popisuje klíčové části algoritmu Pathtracing jako je generování paprsků v sekci 5.3.2 a výpočet barvy pixelů v podkapitole 5.3.3. V neposlední řade stojí za zmínku podkapitola 5.3.4, která podrobně rozebírá využití knihovny OpenCL v programu. Poslední nedílnou částí práce je testovací kapitola 6, ve které jsou prováděny experimenty s výsledným programem.

[1] Obálka světelných paprsků, které jsou lomeny nebo odráženy od zakřivených ploch.

Tato práce si tedy klade za cíl vytvořit demonstrační program, který implementuje metodu Pathracing na GPU pomocí knihovny OpenCL a vykresluje jeho výsledky v reálném čase. Použity budou jednoduché scény jako variace na scénu Cornell Box, která je popsána podrobněji v praktické části 5.2. Program bude možné spustit jak na CPU tak GPU. CPU verze navíc bude používat technologii OpenMP pro paralelní zpracování a využití na všech dostupných jádrech procesoru.

Kapitola 2

Teoretický úvod do algoritmu Pathtracing

Tato kapitola se věnuje teoriteckému základu metod *globálního osvětlení* a současně rozebírá použité techniky a algoritmy spojené s touto možností řešení realistického osvětlování. Protože metoda Pathtracing se snaží simulovat chování světla tak, aby se osvětlená scéna co nejvíce podobala svému skutečnému vzoru. Pro popis chování světla je použita vědní disciplína **radiometrie**.

2.1 Radiometrie

Model světla v počítačové grafice je z větší části založen na popisu chování paprsků z fyzikálního oboru zveného optika. Součástí optiky je i vědní disciplína, konkrétně se zabývající fyzikálním měřením světla, se nazývá radiometrie. Pro jednoduchost a efektivitu řešení se vychází z předpokladu, že se světlo pohybuje v přímé trajektorii (tedy v dokonalém vakuu) a nekonečnou rychlostí.

2.1.1 Energie

Protože světlo je viditelná část energetického vlnění, je potřeba vysvětlit pojem energie. Tok světelné energie [1] (v anglické literatuře pod označením *flux*) je reprezentován symbolem Φ a měřen v jednotkách Watt. Je definován jako množství energie procházející plochou za čas. Tok energie je definován vzorcem:

$$\Phi = \frac{\mathrm{d}E}{\mathrm{d}t} \tag{2.1}$$

kde:

- $\mathbf{d}\boldsymbol{E}$ značí jednotku energie,
- $\mathbf{d}\boldsymbol{t}$ je jednotka času.

2.1.2 Intenzita záření a zářivá exitance

Dalšími důležitými pojmy pro algoritmus Pathtracing jsou **intenzita záření** a **zářivá exitance** [1]. Tok světelné energie je měřen jako proud energie přes celou plochu. Naproti tomu intenzita záření I_e je definována jako přírustek energie $\mathrm{d}\Phi$ na určitou jednotku plochy $\mathrm{d}S$ v

m^2. Pro účely této práce lze použít i specifičtější označení *intenzita ozáření*, se symbolem E_e. Tato veličina je definována vzorcem:

$$I_e = \frac{\mathrm{d}\Phi}{\mathrm{d}S} \tag{2.2}$$

Pokud energie povrch opouští, označuje se jako zářivá exitance M_e. Exitance je definována podobným vzorcem, pouze s lišícím se označením levé části rovnice (M_e):

$$M_e = \frac{\mathrm{d}\Phi}{\mathrm{d}S} \tag{2.3}$$

2.1.3 Prostorový úhel

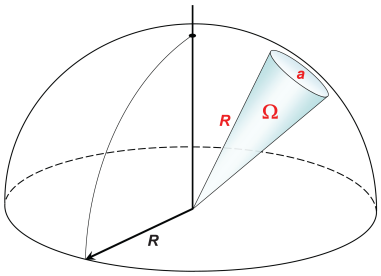
Protože se světlo odráží od povrchů těles zpátky do prostoru, je důležité matematicky popsat směr, kterým se paprsek ubírá ve 3D prostoru. Povrch tedy odráží světlo pouze v prostorovém úhlu, vymezeném jednotkovou polokoulí nad bodem povrchu. Tato polokoule tak popisuje směrové vlastnosti záření. Protostorový úhel jako veličina je reprezentován symbolem Ω a jednotkou je *steradián* [sr]. Úhel je definován jako plocha, kterou vytne kuželosečka na povrchu jednotkové koule, jejíž střed je totožný s vrcholem kuželosečky.

Prostorový úhel je blíže popsán v rovnici 2.4. Tato rovnice popisuje element prostorového úhlu $\mathrm{d}\Omega$, pod nímž je vidět element plochy $\mathrm{d}S$ z bodu P obecné plochy S ve vzdálenosti l.

$$\mathrm{d}\Omega = \frac{\mathrm{d}S \cdot \cos\theta}{l^2} \tag{2.4}$$

kde:

- $\boldsymbol{\theta}$ značí úhel svírající normálu elementu $\mathbf{d}\boldsymbol{S}$ s osou elementárního prostorového úhlu,
- $\boldsymbol{l}$ je vzdálenost od bodu $\boldsymbol{P}$ ke středu plochy $\mathbf{d}\boldsymbol{S}$.



Obrázek 2.1: Definice prostorového úhlu. Převzato z webu[1].

2.1.4 Zářivost

Další důležitým pojmem, který popisuje chování světla je **zářivost** [1]. Množství světelného toku energie je také možné měřit mimo plochu a to přes prostorový úhel $\mathrm{d}\Omega$. Tímto vzniká nový pojem, který se označuje jako zářivost:

$$I_e = \frac{\mathrm{d}\Phi}{\mathrm{d}\Omega} \tag{2.5}$$

[1] http://www.seos-project.eu/modules/laser-rs/images/solid-angle.png

2.1.5 Zář

Posledním pojmem z oblasti radiometrie je **zář** [1]. Záře jako hodnota pro popis světla v metodách globálního osvětlení se nejvíce blíží reprezentaci barvy. Zář (někdy též označovaná jako *radiance* z anglického Radiance) je tedy důležitá vlastnost světla, neboť udává kolik energie vyslané, odražené, přenesené nebo získané se dostane do optického zařízení, které tento povrch sleduje. Udává tedy, jak jasný se povrch jeví a jakou má barvu. Navíc je pro simulaci světla výhodné, že zář nemění se vzdáleností svou hodnotu. Zář je dána vztahem:

$$L_e(x, \Omega) = \frac{\mathrm{d}\Phi}{\mathrm{d}\Omega \cdot \mathrm{d}S \cdot \cos\theta} \tag{2.6}$$

kde:

- $\boldsymbol{\theta}$ je úhel mezi kolmicí na uvažovanou rovinu a měřeným směrem.

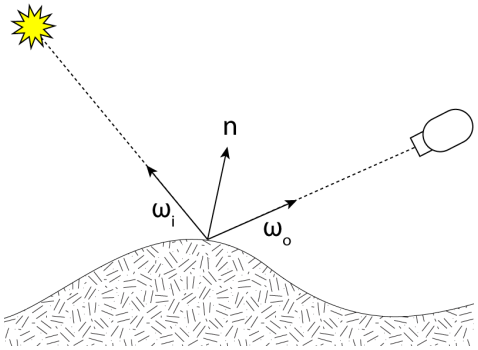
2.2 BRDF

Povrch odráží světlo, proto je potřeba vědět a formálně popsat, jakým způsobem se tento jev děje. Již v roce 1965 Fred Nicodemus formálně popsal odraz paprsku od povrchu pomocí funkce zvané **Bidirectional Reflectance Distribution Function** [5] (dále jen BRDF).

Tato funkce $f_r(x, \omega_i, \omega_r)$ popisuje, kolik světla opouští povrch ve směru ω_r jako následek odrazu světla ve směru ω_i. Jejím výsledkem je tedy poměr odražené záře L_r vůči ozáření E_i. BRDF je definována jako:

$$f_r(x, \omega_i, \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)\cos\theta_i \mathrm{d}\omega_i} \tag{2.7}$$

Důležitou vlastností BRDF je možnost záměny příchozího a odchozího světla a to při zachování stejného výsledku. Tento fakt je klíčový pro metody sledování paprsku, které sledují paprsky směrem od kamery, a ne od zdroje světla, jak je tomu ve skutečnosti.



Obrázek 2.2: Reprezentace BRDF funkce. Oba vektory ω_i a ω_r jsou jednotkové. Převzato z webu[3].

[3] https://upload.wikimedia.org/wikipedia/commons/thumb/e/ed/BRDF_Diagram.svg/300px-BRDF_Diagram.svg.png

BRDF se dá také chápat jako matematický model, který simuluje vlastnosti materiálu. Těchto modelů již existuje nepřeberné množství a některé z nich se zaměřují na simulaci specifického materiálu např. velmi specifický *Oren-Nayar model* je ideální pro simulování povrchu Měsíce, který neodráží světlo stejně jako běžný difúzní povrch nebo např. **Phongův osvětlovací model** pro simulaci lesklých povrchů, (plasty, kovy atp.). U leských povrchů lze stanovit, že mají difúzní a odrazovou (z anglického *specular*) část, ze kterých se materiál skládá. Pro úplnost lze doplnit i třetí část, která navíc bere v potaz okolní osvětlení (z angického *ambient*) dávající barvu objektu. Objekt takto dostává barvu, i když není přímo osvětlen a to umožňuje jeho viditelost i v tmavém prostředí. Každá tato komponenta Phongova modelu se jednotlivě spočítá a pak je složena s ostatními komponenty dohromady tak, aby byl seskládán konečný výsledek v podobě osvětleného objektu. Každá část má i speciální proměnné, kterými lze určovat např. drsnost povrchu a dále tak specifikovat vzhled materiálu.

I přes svou jednoduchost a výpočetní nenáročnost se dnes již tento model používá většinou jen pro výuku stínování a nahradily jej jiné, fyzikálně přesnější modely. Některé z BRDF modelů jsou vytvořeny pro potřeby optiky nebo pro různá měření a jiné mohou být (podobně jako Phongův osvětlovací model) založeny čistě na empirických znalostech daného materiálu.

BRDF funguje pouze za předpokladu, že světlo které dopadá na povrch objektu a světlo které jej opouští do okolí, je jedno a to samé. Příkladem může být střet světla s průsvitným povrchem, kdy se světlo odrazí a část vnikne do objektu, v němž se různě odráží až objekt nakonec opustí. Protože BRDF není schopná tuhle situaci simulovat, je potřeba použít jiný model zvaný *Bidirectional Surface Scattering Distribution Function* (BSSRDF). Dalšími existujícími funkce pro simulaci složitějších chování materiálu jsou např. *BSDF* pro rozptyl světla a *BTDF*, která se chová jako BRDF ale na opačné straně povrchu.

2.3 Zobrazovací rovnice

Tuto rovnici [6] popsal v roce 1986 James Kajiya a popisuje, jak světlo ozařuje objekty a odráží se od nich. Tato rovnice 2.8 je základem pro realistické zobrazování. Jedním ze způsobů řešení rovnice je metodou konečných prvků, která rozděluje spojitý problém do konečné množiny diskrétních částí. Takto hledá řešení rovnice metoda *Radiosity*. Druhým způsobem jsou metody *Monte Carlo*, které jsou využívány velkým množstvím algoritmů (Pathtracing, Photon Mapping a další). Princip těchto metod je popsán v kapitole 2.4.

$$L_o(x,\omega) = L_e(x,\omega) + \int_\Omega f_r(x,\omega',\omega) L_i(w,\omega')(\omega' \cdot n)\mathrm{d}\omega' \tag{2.8}$$

kde:

- $\boldsymbol{L_o(x,\omega)}$ značí odchozí světlo na povrchu v bodě $\boldsymbol{x}$ a daném směru $\boldsymbol{\omega}$,
- $\boldsymbol{L_e(x,\omega)}$ značí jakékoliv světo vyzařované v bodě $\boldsymbol{x}$ a směru $\boldsymbol{\omega}$. Většina materiálů světlo nevyzařuje, takže tato část do rovnice příliš nepřispívá,
- $\boldsymbol{\int_\Omega \mathrm{d}\omega'}$ představuje integrování po polokouli nad bodem $\boldsymbol{x}$ danou normálou $\boldsymbol{n}$. Zjednodušeně řečeno: je potřeba „posbírat“ veškeré příchozí světlo v bodě $\boldsymbol{x}$ na této polokouli,
- $\boldsymbol{f_r(x,\omega',\omega)}$ je funkce BRDF a udává poměr odraženého světla ve směru $\boldsymbol{\omega}$ ku přijatému světlu ve směru $\boldsymbol{\omega'}$,

- $\boldsymbol{L_i(w, \omega')}$ označuje příchozí světlo do bodu $\boldsymbol{x}$ ve směru $\boldsymbol{\omega'}$. Tato funkce nemusí nutně představovat světlo ze světelného zdroje. Může jím být i světlo odražené nebo lomené (nepřímé osvětlení),
- $\boldsymbol{(\omega' \cdot n)}$ je tzv. *snižovací faktor* intenzity záření, neboť je světlo částečně „rozmazáno“ po povrchu podle velikosti úhlu dopadu, který svírá příchozí světlo a normála povrchu. Často se také zapisuje jako $\boldsymbol{\cos \theta_i}$.

2.4 Monte Carlo metody a integrace

Tento název označuje celou třídu algoritmů, které jsou založeny na experimentování s využitím generátoru nahodných čísel. Algoritmy Monte Carlo opakovaně provádí simulaci s náhodnými vzorky, aby získali přibližné numerické řešení. Jejich využití je v celé řadě oblastí, a nejvíce pro výpočet určitých integrálů, především vícerozměrných.

Monte Carlo integrace je aplikace principu Monte Carlo metod [3] právě pro výpočet určitých integrálů. Na rozdíl od jiných algoritmů, při Monte Carlo integrování dochází k náhodnému výběru bodů, ve kterých je funkce vyhodnocena. Jednoduchá ukázka je vidět na rovnici 2.9.

$$\int_a^b f(x)\mathrm{d}x \tag{2.9}$$

Pokud je získáno N náhodných čísel, x_i, $i = 1 \ldots N$ z rozsahu $[a, b]$, je aproximace integrálu funkce $f(x)$ v rozmezí $[a, b]$ definována jako:

$$\frac{(b-a)}{N} \sum_{i=1}^{N} f(x_i)$$

Celková přesnost metody závisí na počtu vzorků (tzv. experimentů), které se provedou. Také konvergence těchto metod není příliš vysoká, což lze odvodit ze vzorce 2.10 pro výpočet relativní chyby.

$$chyba = \frac{1}{\sqrt{N}} \tag{2.10}$$

kde:

- $\boldsymbol{N}$ je počet experimentů.

2.5 Generování náhodných čísel

Jedním ze základních prvků, na kterých je Pathtracing postaven, je generování náhodných čísel. Bez tohot generování by nemohlo fungovat Monte Carlo vzorkování, jehož princip a vliv na algoritmus je vysvětlen v části 2.4. Je proto potřeba tuto část programu kvalitně implementovat, aby generování čísel nebylo náročné, rychlé a hlavně s vysokou periodou.

Základem pro generování jakéhokoli rozložení je generátor, který generuje čísla v intervalu $\langle 0, 1)$. Z tohoto intervalu se poté dále vychází.

Nejjednodušší příkladem generátoru *pseudonáhodných čísel* je *lineární kongruentní generátor* (Linerar Congruent Generator, LCG), jehož princip je vidět v rovnici 2.11.

$$x_{i+1} = (ax_i + b) \bmod m \tag{2.11}$$

kde:

- $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{m}$ jsou vhodně zvolené konstanty pro generátor.

LCG generátor generuje čísla v rozmezí $0 \leq x_i < m$. A protože je počet hodnot v tomto rozsahu omezen, začne se nejpozději po m generovaných číslech opakovat stejná posloupnost hodnot. Doba než se tato posloupnost začne opakovat, se nazývá *perioda* generátoru a je možné docílit plné periody, pokud jsou konstanty generátoru nastaveny na tyto hodnoty:

- $\boldsymbol{a - 1}$ je dělitelné všemi provočíselnými faktory $\boldsymbol{m}$ a zároveň je dělitelné 4, pokud i $\boldsymbol{m}$ je dělitelné 4,
- $\boldsymbol{c}$ a $\boldsymbol{m}$ jsou nesoudělná čísla.

Problémem tohoto generátoru jsou především nevhodně zvolené konstanty, které mohou zapříčinit opakující se posloupnost generovaných čísel. Pro ověření kvality generátoru a pro účely testování jsou generovány dvojice nebo trojice náhodných čísel v rozsahu $[0, 1]$, které jsou následně zaneseny do grafu, ve kterém lze přehledněji odhalit možné chyby generátoru. Některé chyby mohou být odhaleny pouze v určitých dimenzích, a proto se někdy může stát, že se generátor jeví kvalitní ve druhé dimenzi, avšak až ve třetí je možné teprve vidět chybnou posloupnost generovaných hodnot. Tím pádem vygenerovaná čísla kompletně nevyplňují jednotkovou krychli, do které jsou vsazena, a v krychli tak zůstávají volná místa.

Jak již bylo řečeno, generování náhodných čísel má velký podíl na celkové funkčnosti Monte Carlo integrace pro hledání řešení zobrazovací rovnice v algoritmu Pathtracing. Z výše zmíněných důvodů je lineární kongruentní generátor špatnou volbou a je potřeba využít jiný, kvalitnější generátor.

Jako možná náhrada se nabízí generátor *Xorshift*[7], navržený Georgem Marsagliem. Tento generátor generuje nové náhodné číslo opakovanou aplikací bitové operace XOR na bitově posunutou hodnotu sebe sama. To z něj činí na moderních architekturách exterémně rychlý generátor, kde díky plánování instrukcí lze dosáhnout vysokého výkonu výpočtu. Opět je i u tohoto generátoru potřeba vhodně zvolit parametry pro zajištění dostatečně velké periody. Tento typ generátoru je schopný projít většinou statistických testů a jeho případné nedostatky se dají opravit a je tak možné jej využít pro velké množství odvětví a operací. Na obrázku 2.3 je vidět rozložení 1 500 čísel generovaných generátorem Xorshift.

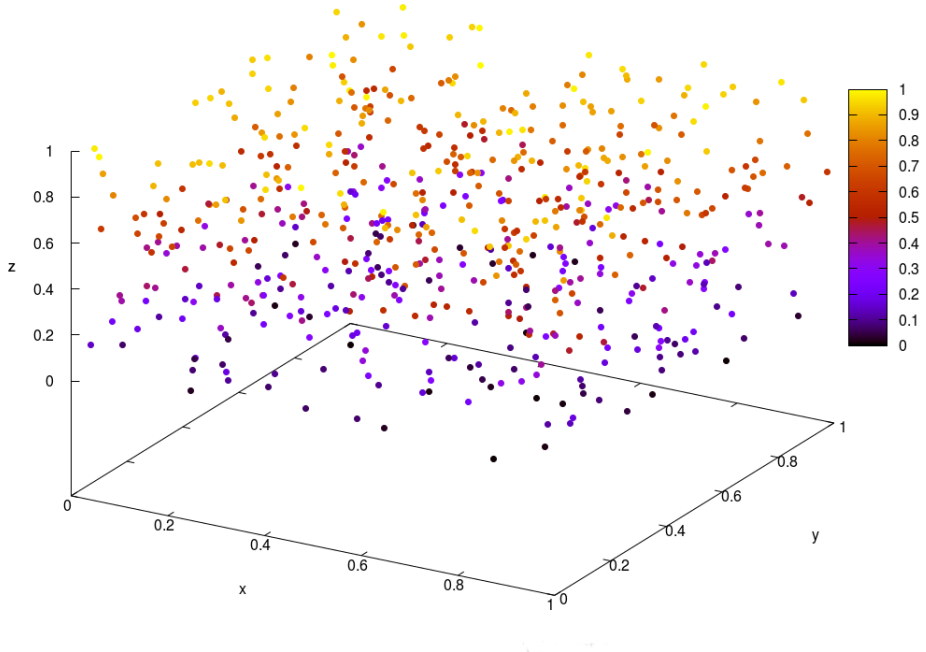
2.6 Russian Roulette

Hloubka zanoření (počet cest paprsku světla při odražení) paprsku může být potenciálně nekonečná a je proto nutné zvolit maximální možnout hloubku zanoření. Dalším důvodem omezení paprsku je, že každé odražení absorbuje určitou část energie. To znamená, že čím více se paprsek odráží při sledování cest, tím méně energie je doručeno do optického zařízení[4].

Russian Roulette je technika, která ukončuje cestu paprsků při každém setkání s povrchem s nenulovou pravděpodobností p a průměruje ji hodnotou $\frac{1}{p}$. Preferované jsou tak kratší cesty a ostatní cesty si zachovají svou intenzitu. Tato technika navíc nepřináší do výsledného zobrazení žádný **bias**[5]

[4] Oko nebo kamera

[5] Výsledek konverguje k očekávanému výsledku.



Obrázek 2.3: Test generování náhodných bodů (trojice náhodných hodnot) generátorem Xorshift.

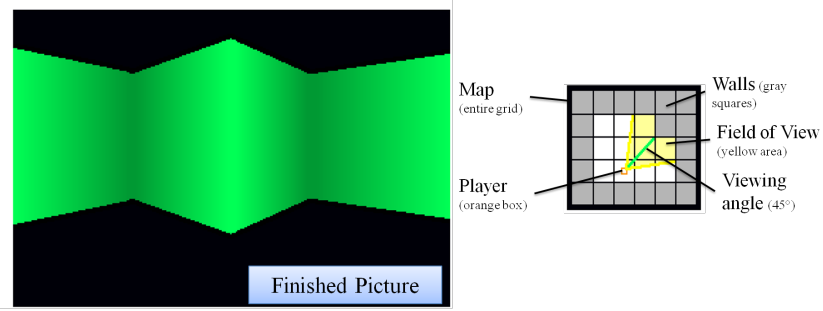
Pravděpodobnost ukončení cesty paprsku je možné zvolit jako globální konstantu, popř. se jako lepší varianta nabízí stanovit ji lokálně. Běžně se proto stanovuje na základě hodnoty reflektivity povrchu. To má za následek, že temné povrchy mají větší pravděpodobnost ukončení.

2.7 Raycasting

První metoda založená na principu vrhání paprsků do scény se nazývá Raycasting[4]. Poprvé byla popsána v roce 1968 Arthurem Appelem a umožňuje na základě 2D popisu vytvořit 3D reprezentaci scény. Metoda funguje na principu vrhání paprsků do scény skrze každý sloupec pixelů na monitoru. Takovýto typ paprsků se označuje pojmem **primární paprsky**. Označují se tak prvotní paprsky, které jsou vyslány od pozorovatele/kamery do scény. Primární paprsky jsou následně testovány na průnik objektů ve scéně. Oproti starším *scanline* renderovacím algoritmům[6] nabízí Raycasting možnost jednoduše vykreslovat i nerovinná tělesa (např. koule, kužel apod.). Tyto tělesa lze matematicky popsat, tudíž je stačí otestovat na protnutí s paprskem. Je-li tedy objekt protnut, vrací paprsek zpátky barvu pixelu objektu a zároveň vzdálenost objektu od kamery. To se děje např. v případě jednoduchých, stejně vysokých stěn, kdy se podle vzdálenosti objektu od pozorovatele vykreslí patřičný sloupec na displeji. Pro tento příklad platí jednoduchá zákonitost: čím blíže je objekt, tím

[6] Algoritmy, které řeší vykreslování po řádcích místo po polygonech.

vetší sloupec se vykreslí a naopak. Dosahuje se tak efektivně 3D vjemu. Tato metoda spolu s uvedeným příkladem je známá pro použití ve hře *Wolfenstein 3D*. Stala se tak hnacím motorem pro posunutí hranic grafického zobrazování. Princip metody je ukázán na obrázku 2.4.



Obrázek 2.4: Princip metody Raycasting a její reprezentace 3D ze 2D popisu. Převzato a upraveno z webu[8].

2.8 Raytracing

Základní metoda založená na principu vrhání paprsků do scény se nazývá **Raytracing** [10]. Tento algoritmus oproti předchozí metodě Raycasting řeší již realistické zobrazení scény. Jedná se o vylepšení Raycastingu, kterou v roce 1979 popsal Turner Whitted. Toto původní vylepšení se označuje pojmem **rekurzivní Raytracing** a vychází z principu, že paprsek, který protne povrch objektu, může generovat tři další typy paprsků rekurzivně, a to *stínový*, *odrazový* a *lomový paprsek*[9]. Tímto přístupem lze dosáhnout realističtějších výsledků a také různých grafických efektů např. *motion blur* (rozmazání pro efekt rychlého pohybu), stínů a hloubky ostrosti. Výpočetní nezávislost jednotlivých paprsků také dovoluje paralelizaci a lze tak urychlit výpočetní čas. Nevýhodou je, že paprsky jsou na protnutí testovány s každým objektem, což zvyšuje náročnost algoritmu na výpočetní výkon. Navíc je zvyšován potřebný renderovací čas k vykreslení scény. Raytracing proto nalézá využití především ve filmovém průmyslu pro vytvoření vizuálních efektů a ve 3D grafických programech jako např. *Blender* pro zhotovení výsledného renderu modelu.

Na Raytracing lze také pohlížet jako na algoritmus řešící čistě viditelnost mezi dvěma body ve scéně, nebo jako na algoritmus pro nalezení nejbližšího průniku po dráze paprsku. Jako příklad lze použít scéna, ve které existuje primitivum, což je jednoduše popsatelný vícerozměrný tvar (např. koule, krychle apod.), a kamera, která na něj směřuje. Paprsky se tvoří s počátkem v pozici kamery a směřují skrze pixely dopadové plochy kamery. Hledáním nebližšího protnutí po směru každého z paprsků je získána informace o viditelnosti objektu (tzn. jestli má být daný pixel obarven nebo ne).

Raytracing lze proto rozdělit do dvou částí. Generování paprsků a testování na protnutí s primitivy.

[8] https://wiki.scratch.mit.edu/w/images/Raycaster_diagram.png
[9] V originále shadow, reflection a refraction ray.

2.8.1 Generování paprsků

Před samotným testováním na průnik paprsků s objekty ve scéně je nejprve potřeba tyto paprsky korektně generovat. Generované paprsky lze rozdělit do dvou skupin: primární a sekundární.

Primární paprsky

Jak již bylo zmíněno výše, primární paprsky se tvoří se svým počátkem v pozici kamery scény a směřují skrze jednotlivé pixely plochy o rozlišení [`šírka`,`výška`]. Primárních paprsků tedy vždy existuje pevné množství `šírka` · `výška`. TADY BUDE ALGORITMUS GENEROVANI

Sekundární paprsky

Všechny ostatní paprsky, jež nejsou primární, se označují pojmem *sekundární paprsky*. Sekundárních paprsků existuje několik druhů, mezi ty nejčastější patří:

- **Stínový paprsek**, jehož počátek leží v bodě protnutí předchozího paprsku a jeho směr směřuje ke zdroji světla.
- **Odrazový paprsek**, jehož počátek leží v bodě protnutí předchozího paprsku a jeho směr je dán materiálem objektu. Pokud se jedná o perfektní odraz (zrcadlo), aplikuje se fyzikální zákon odrazu. Tzn. úhel odrazu se rovná úhlu dopadu. Tento úhel svírá kolmice na povrch objektu v bodě protnutí s dopadajícím paprskem. Tento vztah je vidět v rovnici 2.12.

  $$R = D - 2(N \cdot D)N \tag{2.12}$$

  kde:

  - $\boldsymbol{D}$ je vektor představující dopadající paprsek,
  - $\boldsymbol{R}$ je vektor představující odražený paprsek,
  - $\boldsymbol{N}$ je normála, okolo které se vektor $\boldsymbol{D}$ odráží jako vektor $\boldsymbol{R}$.
- **Lomový paprsek**, jehož počátek leží v bodě protnutí předchozího paprsku a jeho směr je dán *Snelloým zákonem* z rovnice 2.13.

  $$\frac{\sin\theta_i}{\sin\theta_o} = \frac{n_i}{n_o} \tag{2.13}$$
- **Paprsek pro difůzní odraz**, jehož počátek leží v bodě protnutí předchozího paprsku a jeho směr směřuje do bodu na jednotkové polokouli, sestrojené nad bodem průniku.

2.8.2 Protnutí paprsku s objekty

Test na protnutí s objekty je kritickou částí Raytracing algoritmů. Na těchto testech stojí efektivita celého programu, protože se vykonávají pro kažý paprsek Raytracingu. Je tedy potřeba vykonávat je co nejrychleji. Ve scéně mohou existovat různá primitiva, která je potřeba otestovat na průnik s každým paprskem. V této práci jsou pro jednoduchost použity jako primitiva koule, proto bude tato část věnována popisu testu na jejich protnutí s paprsky.

Protnutí paprsku s koulí

Koule jakožto grafické primitivum je velmi snadno matematicky popsatelné a jeho test na protnutí se dá zjednodušit na jednoduchou kvadratickou rovnici. Nejdříve je potřeba začít s vektorovou rovnicí koule. P představuje vektor paprsku a C pozici (střed) koule:

$$(P - C) \cdot (P - C) - r^2 = 0$$

Následně se nahradí P za rovnici přímky $P(t) = O + tD$:

$$(O + tD - C) \cdot (O + tD - C) - r^2 = 0$$

$$(D \cdot D)t^2 + 2D \cdot (O - C)t + (O - C) \cdot (O - C) - r^2 = 0$$

Nakonec se najdou kořeny kvadratické rovnice:

$$a = (D \cdot D)$$

$$b = 2D \cdot (O - C)$$

$$c = (O - C) \cdot (O - C) - r^2$$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Pomocí diskriminantu rovnice lze zjistit vlastnosti protnutí koule. Dá se tak předem zjistit, jestli paprsek kouli trefil, či nikoliv, a případně tak předejít zdlouhavému počítání. Matematika rozpoznává následující typy diskriminantů:

- **Kladný diskriminant**, kdy paprsek kouli protnul. Existují dva kladné kořeny. Pro nejbližší protnutí se vybere ten menší.
- **Záporný diskriminant**, kdy paprsek kouli minul.
- **Diskriminant roven nule**, kdy paprsek je tečnou na kouli.

2.9 Pathtracing

Pathtracing [5] je **Monte Carlo** metoda řešení vykreslování realistických scén a **globálního osvětlení**, ve kterém platí, že není rozdíl mezi světlem vrhaným zdrojem světla a světlem odraženým od povrchu objektu. Každý povrch dostává určitou energii, která jej ozařuje, a to i přesto, že není přímo osvětlen zdrojem světla. Jedná se zde o *stochastickou* metodu, která se snaží aproximovat množství světelné energie, jež doputovalo na daný bod vysláním paprsků do scény. Nutnost aproximace je dána nemožností navzorkovat veškerou příchozí světelnou energii na jednotkové polokouli. Proto je při sběru využita náhodná veličina, která určuje náhodný výběr směru paprsku.

Sledováním těchto paprsků, které se dále ve scéně náhodně odráží, se vypočítá množsví světelné energie přispívající do daného bodu. Náhodnost této metody způsobuje, že prvotní výsledky jsou zašumněné a nejasné. S přibývajícími vzorky se ovšem kvalita zlepšuje, protože každý další vzorek přispívá ke konečné barvě. S každou iterací se však váha vzorku snižuje, neboť přispívá do celkového průměru barvy. Rozdíl tak např. mezi 20000 a 21000 vzorky bude menší, než mezi 1000 a 2000 vzorky.

Na tomto zjednodušeném principu Pathtracingu staví *Naivní Pathtracing* algoritmus. Tento celkem krátký kus kódu hledá řešení zobrazovací rovnice (viz. 2.3), kde jediná známá

proměnná je směr z počátku kamery srkz příslušný pixel do scény. Nejprve je potřeba nalézt bod x, ve kterém se bude počítat osvětlení. Pro tuto potřebu je využit Raytracing vyslaného paprsku a je zjištěno, jestli je některý z objektů trefen a pokud ano, tak ve kterém bodě. Dále je využito metody Monte Carlo integrace pro zjištění příchozí energie pro daný bod. Tato část vyžaduje kvalitní generování náhodných čísel, protože, jak je uvedeno v sekci 2.5, přesnost metod Monte Carlo závisí na jejich kvalitním vygenerování (viz. 2.4). Všechny tyto úkony jsou jednotlivé části zobrazovací rovnice (viz. 2.3), jejímž řešením je získána výsledná barva pixelu a celkový obraz představující realistické globální osvětlení scény.

Kapitola 3

Existující řešení

V této kapitole jsou představeny různé existujicí implementace založené na metodě sledování paprsku. Některá nalezená řešení nemohla být vzhledem k jejich finančnímu modelu vyzkoušena. Z tohoto důvodu se tato kapitola věnuje pouze volně dostupným a oteřeným řešením algoritmu Pathracing a rozebírá použité algoritmy těchto řešení. Dále je u každé implementace uvedena tabulka s naměřenými hodnotami pro ukázku a porovnání rychlosti dané implementace. Měření probíhalo na následujících sestavách:

Tabulka 3.1: Použité sestavy

Sestava	CPU	GPU	RAM
1	Intel i3-4030U	Intel HD 4400	4GB
2	AMD FX-6300	NVIDIA GTX 970	8GB

3.1 Smallpt

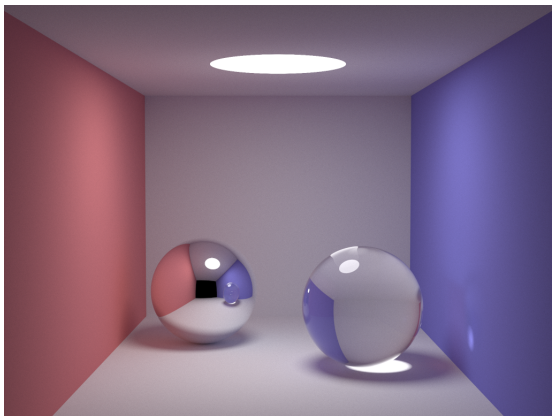
Pravděpodobně nejzajímavější open-source implementací algoritmu Pathtracing je projekt **smallpt** od Kevina Beasona, který se vešel na pouhých 99 řádků. Jako metodu řešení globálního osvětlení je použit *unbiased*[1] Monte Carlo Pathtracing využívající algoritmus *Russian roulette*. Mezi další vlastnosti patří Antialiasing, měkké stíny a více vláknové zpracování pomoci knihovny *OpenMP*. Kompaktnost programu ovšem nedává prostor optimalizaci a je zde také delší doba renderování. I přesto se stal smallpt inspirací pro další Pathtracingové programy a základem, na kterém se dají vytvářet různá vylepšení. Za zmínku stojí například vylepšení, kdy se explicitně vzorkují světla ve scéně. Toto vylepšení výrazně zrychlí celý algoritmus a to za cenu jen pár řádků kódu navíc. Naměřené hodnoty pro vykreslení 32 vzorků na pixel v originálním programu Smallpt jsou v tabulce 3.2.

Program, stejně jako seznam všech jeho vylepšení, je volně dostupný na autorově stránce[2].

[1] Výsledná hodnota konverguje k očekávanému výsledku spolu s rostoucím počtem vzorků.
[2] http://www.kevinbeason.com/smallpt/

Tabulka 3.2: Naměřené hodnoty pro program Smallpt

Sestava	Čas 1 vzorku v ms	Vzorky/sekunda
1	3758.438	0.26
2	2651.563	0.38



Obrázek 3.1: Cornell box scéna programu Smallpt. Převzato z webu[3].

3.2 SmallptGPU

Jednou z modifikací výše zmíněného programu je SmallptGPU autora Davida Bucciarelliho. Je ho hlavní myšlenkou je převedení kódu Smallpt do OpenCL. Výpočty tak mohou probíhat na CPU, GPU nebo na obou současně. Program přitom dovoluje dynamicky nastavit poměr zátěže výpočetních zařízení. Verze pro CPU je čistě jednovláknová bez využití OpenCL nebo OpenMP. Verzi pro GPU, která již OpenCL využívá, je možné spustit jak na CPU, tak na GPU. Zajimavým faktem je, že tato verze spuštěná na CPU má při využití 4 jader stejný výkon jako CPU verze na jednom jádru.

Jako nástupce vznikl prorgram SmallptGPU2, který používá jen OpenCL a to jak pro CPU, tak pro GPU renderování. Naměřené hodnoty v tabulce 3.3 jsou z programu SmallptGPU2, neboť SmallptGPU se nepodařilo zprovoznit. Navíc SmallptGPU2 na sestavě číslo 2 rozpoznal jako OpenCL zařízení pouze procesor a proto chybí data pro grafickou kartu. Hodnoty ukazují zajímavý fakt, kdy 4 jádrový procesor ze sestavy číslo 1 vynikl nad 6-ti jádrovým procesorem v sestavě 2.

Program je volně dostupný na autorově stránce[4].

Tabulka 3.3: Naměřené hodnoty pro program SmallptGPU2

Sestava	Zařízení	Čas 1 vzorku v ms	Vzorky/sekunda
1	CPU	63.00	15.87
	GPU	62.000	16.13
2	CPU	547.00	1.83

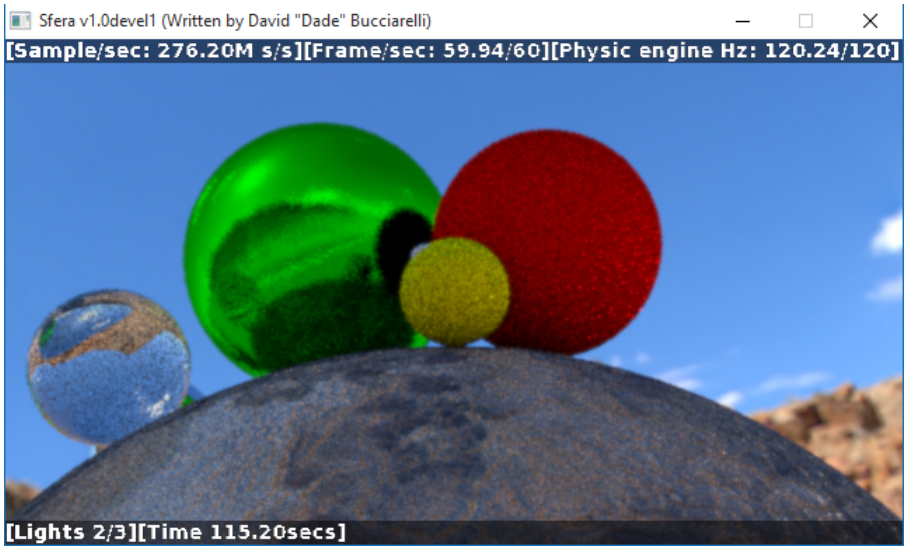
[3] http://www.kevinbeason.com/smallpt/result_25k.png
[4] http://davibu.interfree.it/opencl/smallptgpu/smallptGPU.html

3.3 Sfera

Sfera je zajímavým pokusem autora SmallptGPU2 o využití Pathtracingu v herním odvětví. Sfera k tomu využívá OpenGL pro zobrazování a pro výpočty OpenCL. Na obrázku 3.2 je zobrazen screenshot ze hry. Pro herní logiku je navíc přidána fyzikální knihovna **Bullet Physics**. Hra nabízí dvě možnosti spuštění: mód s využitím OpenCL, nebo bez něj. Zde se projevuje účinek této knihovny, neboť bez použití OpenCL se hra stává téměř nehratelnou. Hra je zdarma ke stažení pod licencí GPL na stránkách Google Code[5]. Naměřené hodnoty lze vidět v tabulce 3.4.

Tabulka 3.4: Naměřené hodnoty pro hru Sfera

Sestava	Zařízení	Čas 1 vzorku v ms	Vzorky/sekunda
1	CPU	156.740	6.38
	GPU	30.912	32.35
2	CPU	115.607	8.65
	GPU	16.681	59.95



Obrázek 3.2: Ukázka ze hry Sfera.

3.4 WebGL Path Tracing

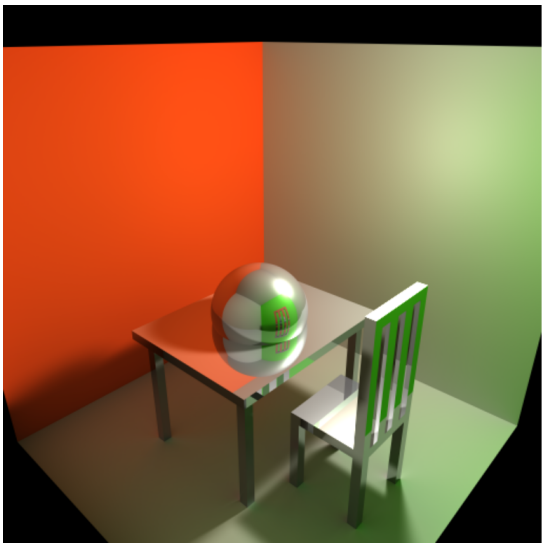
Další zajímavou implementací algoritmu Pathtracing je **WebGL Pathtracing**. Webová aplikace je vytvořena v API *WebGL* pro spuštění přímo v prohlížeči. Aplikaci vytvořil programátor Evan Wallace. Algorimtus je kompletně naimplementován pomocí GLSL shaderů pro WebGL a podporuje vykreslování v reálném čase. Dále lze objektům nastavit různé druhy materiálů (difúzní, reflexivní, matné). Pro demonstraci efektu jsou připraveny dva druhy *Cornell boxu*, do kterých je možné přidávat dva druhy objektů, kouli a krychli, nebo

[5] https://code.google.com/archive/p/sfera/

si vybrat z několika už vytvořených scén. Aplikace také dovoluje pohyb s předměty, světlem a kamerou, přičemž každý z těchto pohybů vyvolá překreslení scény, na kterém je vidět zašumění, které Pathtracing vyvolává.

Pro výpočet barev jednotlivých pixelů je paprsek vystřelen do scény a a jemu dovoleno se 5 krát odrazit. Při každém odrazu se přímé příchozí světlo v daném bodě vynásobí se všemi barvami předchozích materiálů a akumulovány. Jemných stínu autor dosáhl náhodnými změna pozice světla pro každý pixel.

Aplikace je volně dostupná na autorově GitHub stránce[6] a na obrázku 3.3 je vidět její rozhraní.



Obrázek 3.3: Okno aplikace v prohlížeči.

[6] https://github.com/evanw/webgl-path-tracing

Kapitola 4

Použité technologie

Tato kapitola se zabývá popisem jednotlivých nástrojů, které jsou použity v implementaci výsledného demonstračního programu. Jako základ je použit jazyk C++, jednak z důvodu výkonu ale také přirozeného propojení ostatních použitých knihoven, které jsou rovněž implementovány v jazyce C/C++.

Pro vytvoření okna aplikace a získávání uživatelského vstupu je použita multiplatformní knihovna *GLFW*. O samotné zobrazení se stará grafická *API OpenGL 3.3*. Kvůli zachování přenositelnosti a hlavně možnosti spustit program na větším množstvím hardware je pro implementaci samotného Pathtracing algoritmu použita knihovna *OpenCL 1.2*. Alternativou k OpenCL je technologie CUDA, která je bohužel použitelná pouze na GPU firmy Nvidia. Následující sekce se věnují těmto knihovnám.

4.1 GLFW

GLFW je multiplatformní open-ource knihovna pro vývoj programů využívajících OpenGL, OpenGL ES nebo nově také *Vulkan* kontext. Dále obsahuje podporu pro obsluhu vstupních zařízení jako klávesnice, myš a joystick. Pojem OpenGL kontext si lze zjednodušeně představit jako plátno, se kterým dokáže OpenGL pracovat a vykreslovat do něj.

V této práci je GLFW použito pro vytvoření okna programu, získání vstupu z klávesnice a myši a následné řízení programu. Pro tento účel existují další API, jmenovitě např. **SDL**, **SFML** nebo **freeGLUT**. Po vyzkoušení zmíněných knihoven byla vybrána knihovna GLFW. Hlavními důvody jejího výběru se staly:

- přenostitelnost,
- poměrně snadné vytvoření okna i kontextu (především oproti SDL),
- dostupná a kvalitní dokumentace,
- dobře pochopitelný OOP návrh knihovny.

4.2 OpenGL

Open Graphics Library (dále jen OpenGL) [9], je multiplatformní nízkoúrovňové API pro 2D a 3D akcelerovanou grafiku. O vykreslování grafiky se stará GPU. Knihovna se využívá především v grafických programech, v menší míře také pro tvorbu her. V tomto odvětví však stále převládá *DirectX* od společnosti Microsoft. DirectX je přes svou nepřenostitelnost

použit na převládající většině operačních systémů, neboť systémy Microsoft Windows jej nativně podporůjí a výrazně tak ovlivňují majoritu trhu. V poslední době je též zmiňován *API Vulkan*, který by měl jako nástupce OpenGL sjednotit oba své předchůdce, OpenGL a OpenGL ES.

OpenGL také podporuje široké spektrum programovacích jazyků a různých platforem. Tradiční platformy podporující OpenGL jsou především Windows, Mac OS a Linux. Navíc OpenGL ES přináší podporu i pro mobilní a vestavěná zařízení a WebGL pro akcelerovanou grafiku v prohlížečích.

Výše zmíněný Vulkan je snahou konsorcia *Khronos Group* o sjednocení a využití jednoho API pro desktopovou a mobilní grafiku. Zároveň se snaží snížit počet CPU volání, což uvolní CPU a dovolí mu pracovat na ostatních úkolech, ať už renderovacích, nebo ne. Vulkan je založen na *Mantle API* od firmy AMD, která jej věnovala jako open-source základ, na kterém se dá vytvořit nové, nízkoúrovňové API. Určité vlastnosti má společné s dalším novým API *DirectX 12* jako např. snížení vytížení procesoru. API DirectX 12 ovšem nadále zůstává omezeno pouze pro jednu platformu. Vulkan se nabízí jako alternativa k OpenGL, k implementaci této práce však zvolen nebyl a to především z důvodu, že jeho výhody by nebyly v této práci naplno využity. OpenGL je tedy v práci využito pouze pro jednoduché vykreslení textury, pro které bohatě postačí. Také je v tomto ohledu potřeba vzít v úvahu, že inicializace Vulkan API pro renderování je oproti OpenGL zdlouhavá a neexistuje ještě tolik zdrojů informací a dokumentace k této nové knihovně.

Jak bylo řečeno, v této práci je OpenGL použito pro vykreslení výsledného renderu Pathtraceru do okna programu. Další užitečné informace jako renderovací čas jednoho vzorku a celkový čas renderování se zobrazují v terminálovém okně.

4.2.1 OpenGL shadery

Pro účely práce jsou též využity jednoduché **shadery**. Tyto programovatelné části grafické *renderovací pipeline* 4.1 vznikly jako nástupci *ARB assembly language*, který byl příliš komplexní a neintuitivní. Dalším důvodem jejich využití je větší flexibilita při programování pipeline, která do té doby byla fixní. Navíc je jejich syntaxe založená na syntaxi jazyka C, jehož míra abstrakce dopomáhá vytvářet kvalitnější a srozumitelnější shader programy. Jejich využití je především ve videoherním průmyslu pro tvorbu různých grafických efektů.

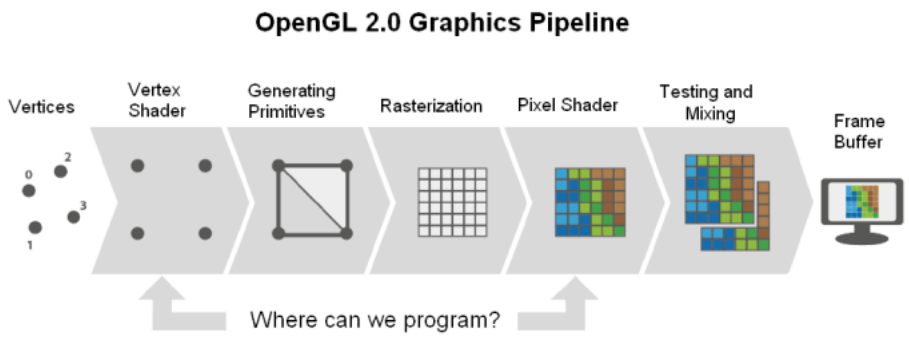
- Vertex shader Je první programovatelnou složkou grafické pipeline, která pracuje na úrovni jednotlivých *vertexů* (bodů), ze kterých je zpracovávaný model složen. Zde jsou řeseny různé transformace bodů modelu, aniž by se jakýmkoliv způsobem zasahovalo do orignálních dat. Tímto způsobem je možné docílit vytvoření jednoduchých animací a transformování tvaru objektu.

- Geometry shader Tento shader se v pipeline objevil ve verzi OpenGL 3.2 a jeho účelem je transformace na úrovni jednotlivých *grafických primitiv* (body, přímky, trojúhelníky, atd.). Na základě vstupních primitiv lze navíc vytvářet nové body a primitiva a tímto způsobem změnit kompletně tvar vstupního objektu. Vstupem může být například jeden jediný bod, který se bude považovat za střed, a okolo něj se offsetem vytvoří nové body, které dohromady dají výsledný objekt.

- Fragment shader Po skončení rasterizace všech primitiv je další částí pipeline Fragment shader, který na vstup dostává jednotlivé pixely po rasterizaci. Má tak možnost

pracovat s barevnou složkou pixelů, což znamená i nastavení barvy podle zadané textury. Zde se vytváří různé barevné filtry a efekty jako například černobílý filtr, ostřící filtr, nebo filtr pro detekci hran.

- Compute shader Od verze OpenGL 4.5 je součástí grafické pipeline i tzv. Compute shader, který je použit pro různé výpočty, a je tak funkčností podobný knihovně OpenCL, která bude probrána v sekci 4.4. Oproti OpenCL tento shader nabízí výhodu integrace do samotného OpenGL, a tudíž není nutné instalovat a nastavovat další knihovny. Další výhodou může být, že shader používá shaderový jazyk *GLSL* a je tak jednodušší pro programátora na vytvoření. Nelze však říci, že by mohl kompletně nahradit OpenCL\CUDA, jelikož nedosahuje veškerých možností těchto knihoven, avšak pro ne příliš složité výpočty postačuje.



Obrázek 4.1: Ilustrace funkčnosti OpenGL pipeline. Převzato z webu[1].

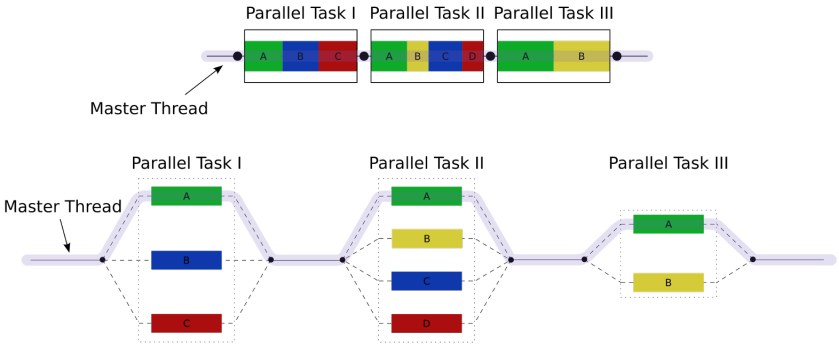
4.3 OpenMP

OpenMP je otevřený standard pro paralelní programování [2] se sdílenou pamětí v jazyce C, C++ a Fortran a je spravován konsorciem OpenMP Architecture Review Board. Obsahuje direktivy pro překladač a procedury, které ovlivňují chování programu za jeho běhu.

OpenMP podporuje paralelismus ve formě *multithreadingu*, kdy existuje jedno hlavní vlákno (Master Thread), které na místech specifikovaných programátorem pomocí daných direktiv vytvoří určité množství podřízených vláken (Slave Thread) a rozdělí mezi ně právě zpracovávaný úkol. Po skončení paralelní sekce, která je opět programátorem označena pomocí konce direktivy, se podřízená vlákna spojí opět do jednoho hlavního vlákna, které pokračuje ve vykonávání programu. Výhodou takového využití direktiv k dosažení paralelního zpracování je, že pokud použitý překladač nepodporuje dané direktivy, je ohraničený úsek ignorován a zpracován klasicky sekvenčně. Díky tomu není potřeba opakovat, či přepisovat části kódu.

V této práci je OpenMP využito pro vytvoření verze programu, která běží na CPU.

[1] https://hackpad-attachments.s3.amazonaws.com/hackpad.com_AJRyvYToxQ6_p.239052_1410571595940_h-GraphicsPipeline-480.png



Obrázek 4.2: Ilustrace funkčnosti OpenMP. Převzato z webu[2].

4.4 OpenCL

OpenCL je otevřený standard pro psaní programů [8], které jsou spustitelné na velké množině heterogenních výpočetních zařízení, což zahrnuje jak tradiční výpočetní zařízení (CPU a GPU), tak i např. *FPGA* a *DSP*[3]. Standard byl původně vytvořen firmou **Apple** a později nabídnut firmě **Khronos group**, která jej dodnes spravuje.

OpenCL API chápe výpočetní systém jako soubor několika výpočetních jednotek, které se dají využít pro datově *paralelní výpočty*, kdy je potřeba provést operace nad velkým množstvím na sobě nezávislých dat. Z důvodu výše zmíněné přenositelnosti bylo nutné zavést některá omezení různých operací. Jedná se hlavně o nemožnost využití rekurze na GPU, dále rozšíření potřebné pro použití čísel s plovoucí destinnou čárkou s dvojitou přesností (*float*) a nemožnost dynamické alokace paměti.

Toto API je primárně vyvíjeno v jazyce C, avšak lze také využít obalovací vrstvy C++. Mnozí OpenCL vývojáři vytvořili i různé porty pro podporu dalších programovacích jazyků, jako například Python, Java apod. Samotný standard ovšem vyžaduje, aby OpenCL frameworky minimálně poskytovaly standardizované knihovny v jazyce C/C++.

Pro samotné psaní výpočetních *kernelů* OpenCL definuje jazyk založený na syntaxi verze C99 (jazyka C), který je podobně jako v případě OpenGL rozšířen o další speciální datové struktury například pro práci důležitý *vektor*. Hlavní předností API pro účely této práce je možnost sdílet paměť s grafickým API OpenGL. Snižuje se tak, zpoždění během přenosu dat mezi CPU a GPU.

Z důvodu výše zmíněné nemožnosti využít rekurzi je dobrým zvykem psát OpenCL programy s využitím iterace. Také by se měla co nejvíce snížit datová závislost jednotlivých vláken tak, aby nedocházelo ke zbytečnému zpomalování probíhajícího výpočtu. Výpočet výsledného renderu zde proto běží paralelně pro každý pixel.

Mezi přednosti standardu OpenCL především patří:

- Přenositelnost

  OpenCL má podobnou filozofii jako jazyk *Java*: „naprogramovat jednou, spustit všude“. Každý výrobce, který dodává hardware schopný práce s OpenCL vytváří i

[2] https://upload.wikimedia.org/wikipedia/commons/thumb/f/f1/Fork_join.svg/2000px-Fork_join.svg.png

[3] Pole programovatelných hradel (Field-programmable gate arrays) a digitální signálový procesor (Digital signal processor).

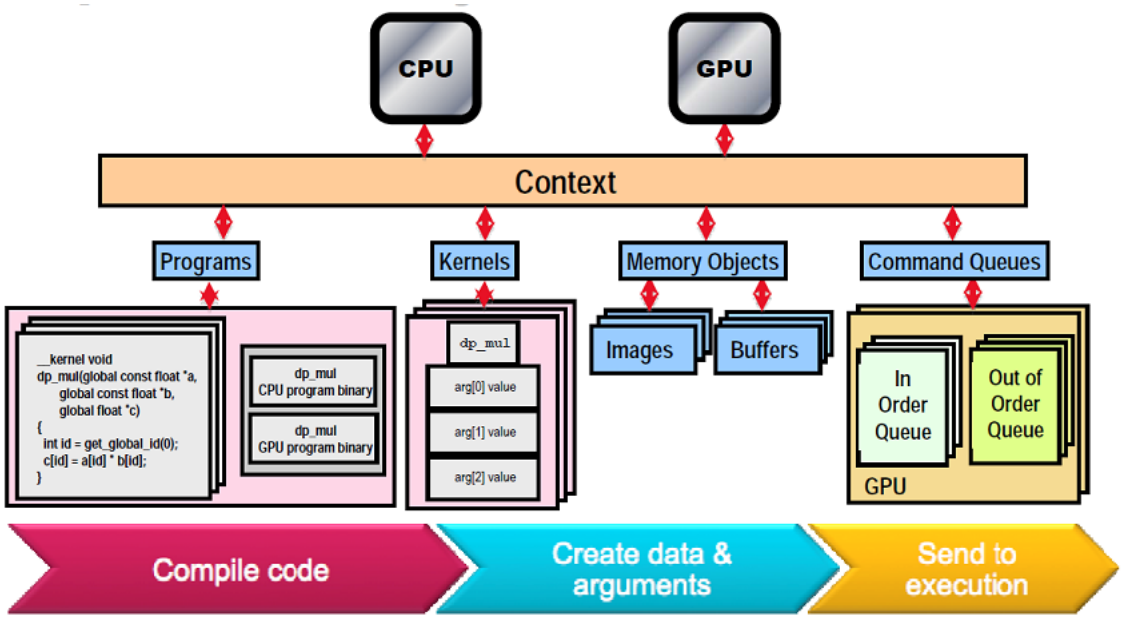
vlastní nástroje pro překlad. Odpadá tak zdlouhavé učení proprietárního jazyku určeného výrobcem. Tato výhoda má ovšem další, velmi důležitý aspekt. OpenCL dokáže využít i více zařízení naráz, které navíc nemusí být od stejného výrobce a tyto zařízení ani nemusí být založeny na stejné architektuře. Lze tak sjednotit hardware pro jeden výpočetní úkon. Pokud se připojí další zařízení, bude nutné program znovu sestavit, ale ne přepsat.

- Standarizované zpracování vektorů

  OpenCL označuje matematický vektor jako datovou strukturu s názvem *výpočetní vektor*, která obsahuje prvky stejného datového typu. Během vektorové operace je operace provedena nad každým prvkem v jednom hodinovém cyklu. Téměř všechny moderní procesory jsou schopné zpracovat vektory, ovšem nemají unifikované instrukce mezi výrobci. OpenCL tento problém řeší při kompilaci programu, kdy se použijí vektorové instrukce na základě výrobce hardwaru.

- Paralelní zpracování

  OpenCL podporuje *paralelní programování*, kdy se narozdíl od vláken a procesů sdílící zdroje, přiřadí úloha několika výpočetním prvkům. Vypočetní prvky budou takto vykonávat jim přiřazenou úlohu ve stejný čas. Tyto úlohy se nazývají *kernely* a jsou to speciální funkce v OpenCL programu. Kernely jsou rozesílány do výpočetních zařízení C\C++ programem, který se nazývá *host*. Host spravuje připojená zařízení skrze tzv. *kontext*. Ten je ukázán spolu s dalšími prvky OpenCL na obrázku 4.3.



Obrázek 4.3: Ilustrace funkčnosti OpenCL. Převzato a upraveno z webu[4].

[4] http://www.notebookcheck.net/fileadmin/_migrated/pics/OpenCL_Arbeitsablauf_Khronos_Group_04.png

Kapitola 5

Návrh a implementace

Tato kapitola se věnuje vlastnímu návrhu a implementaci algoritmu Pathtracing. Výsledkem je demonstrační program výkonnosti výpočtu na procesoru a výpočtu akcelerovaného na grafické kartě. Scény v programu jsou interaktivní a je možné měnit mezi dvěma scénami. Dále je možné ovládat kameru pro pohyb ve scéně a měnit pozici, barvu i intenzitu světla. Program je kompatibilní pouze s linuxovými distribucemi obsahujícími potřebné knihovny.

5.1 Zobrazení programu a tvorba okna

Pro vytvoření okna, do kterého se bude výsledek algoritmu renderovat, bylo původně vybráno API SDL2. Tato knihovna slibovala vysokou přenositelnot stejně jako schopnost obsluhovat vstup myši a klávesnice. Dalším důvodem výběru SDL2 byla dobrá dostupnost dokumentace. Bohužel se v průběhu implementace ukázalo, že vysoká náročnost výpočtu Pathtracing algoritmu značně zpomaluje funkce tohoto API, takže mnohdy nebylo ani možné okno programu vypnout. Dalším limitujícím faktorem také bylo předání informací o okně a jím vytvořeném OpenGL kontextu do nastavení OpenCL.

Po výše popsaných implementačních obtížích bylo vybráno API GLFW4.1. V této práci je využito především pro vyvoření okna, které má pro demonstrační účely rozlišení 512x512 pixelů. Okno poté vytvoří kontext pro OpenGL 4.2, ve kterém se bude pracovat. GLFW nakonec registruje tři *callback* funkce, které se starají o obsluhu vstupů. Callback funkce se předají jako argument jiné funkci, ve které jsou poté v určitý moment zavolány. Pro účely práce byly použity tyto callback funkce:

- `keyCallback` Stará se o obsluhu klávesových vstupů programu. Popis ovládání je v příloze.
- `mouseCallback` Stará se o obsluhu vstupu myši. Posunem myši se lze rozhlížet bez změny pozice po scéně.
- `windowResizeCallback` Tato funkce neobsluhuje přímo žádný vstup, je však dobré ji zmínit. Pokud uživatel změní velikost okna, tato funkce se postará o realokaci potřebných bufferů.

Po každé z těchto funkcí je potřeba zavolat funkci `reInit`, která vyresetuje číslo aktuálního vzorku a scéna je znovu překreslena.

Jako způsob vykreslení samotného výsledku Pathtracing algoritmu bylo vybráno zapisování do OpenGL textury, která je následně vykreslena na čtveřici bodů, které se nachází v

rozích okna programu. Po vytvoření okna je tedy potřeba vytvořit všechny potřebné buffery pro OpenGL.

Pro body textury je vytvořen speciální OpenGL *VBO* buffer a pro indexy těchto bodů *EBO* buffer. Indexový buffer slouží pro uložení indexů bodů, aby se na ně dalo v budoucnu odkazovat při vykreslování a jeden bod tak mohl být využit při více vykresleních. Oba tyto buffery jsou vázany jedním *VAO* bufferem.

Ještě před inicializací OpenGL bufferů je vytvořen jednoduchý shader program pomocí třídy `Shader`. Program se skládá z jednoho Fragment shaderu a jednoho Vertex shaderu4.2.1. Tyto 2 shadery tvoří minimum pro jejich využití programovatelné pipeline a také jsou jediné z shaderů, které program v této práci potřebuje k vykreslení výsledku algoritmu Pathtracing. Vertex shader se stará o napozicování bodů a Fragment shader navzorkovává texturu mezi těmito body.

Jako poslední ze speciálních OpenGL datových struktur je generována vykreslovací 2D textura, která uchovává hodnoty jako čtveřici $[R, G, B, A]$ typu `GL_UNSIGNED_BYTE`, kde:

- $\boldsymbol{R}$ je červená složka barvy.
- $\boldsymbol{G}$ je zelená složka barvy.
- $\boldsymbol{B}$ je modrá složka barvy.
- $\boldsymbol{A}$ je alfa kanál pixelu, nebo také tzv. průhlednost.

Všechny výše zmíněné operace probíhají v souboru `display.cpp`, který také obsahuje hlavní smyčku programu. Té se věnuje kapitola 5.3.1.

5.2 Popis scény

Jako demonstrace algoritmu Pathtracing jsou připraveny scény typu *Cornell Box*, které se sestávají z několika stěn, dvou koulí vložených do scény a malého zdroje světla. Ve skutečnosti jsou stěny scény tvořeny koulemi o velkém průměru. Opticky se tak dosáhne vjemu rovného povrchu. Zjednodušuje se tak výpočet algoritmu, protože takhle scéna obsahuje pouze jeden typ primitiva a není potřeba vytvářet další datové struktury a funkce na testování průniku. Pro tento účel postačí pouze jedna funkce s názvem `intersect`, kterou je možné použít na všechny objekty scény. Tato funkce iteruje přes všechny objekty (koule) ve scéně a testuje je na protnutí s daným paprskem. Samotný test probíhá ve funkci struktury `Sphere` opět s názvem `intersect`. Proces nalezení bodu protnutí je popsán v podkapitole 2.8.2.

První scéna je klasická variace Cornell Box scény a obsahuje dvě koule s difúzním povrchem tak, jak měl originální render z roku 1984. Druhá scéna už více demonstruje možnosti algoritmu Pathtracing. Obsahuje také dvě koule na stejném místě jako předchozí scéna, avšak nyní s jiným materiálem. Levá koule má odrazivý povrch stejně jako zadní stěna místnosti a pravá koule má povrch skleněný. Tato scéna tak zahrnuje všechny možné typy povrchů, které lze algoritmem Pathtracing řešit. Zdrojem světla v obou scénách je opět koule s malým průměrem u stropu simulující zdroj světla. Veškeré informace o kouli a test na její protnutí obsahuje struktura `Sphere`. Ta si uchovává ve speciální vektorové proměnné `emission` barvu a její vyzařovanou intezitu.

5.3 Pathtracing algoritmus

Algoritmy založené na principu vrhání paprsků do scény a jejich sledování jsou ideálními kandidáty pro paralelní zpracování. Po přeložení a sestavení pomocí programu *Make* jsou vytvořeny dva spustitelné binární soubory označené jako `CPU_IBP_pathtracer` a `GPU_IBP_pathtracer`. `CPU_IBP_pathtracer` spouští Pathtracing na CPU a jedinou akcelerací je využití knihovny OpenMP pro paralelizaci vysílání paprsků. Naproti tomu je výpočet Pathtracingu v programu `GPU_IBP_pathtracer` akcelerovaný na grafické kartě. Tento program spouští inicializaci třídy `ConfigCL` pro obsluhu OpenCL a spouštění výpočtu v kernel programu. Tato třída je detailněji popsána v podkapitole 5.3.4.

Implementace algoritmu Pathtracing je v obou verzích programu totožná a jediným rozdílem je v akcelerované verzi odstranění rekurze a rozdělení celého algoritmu na více menších funkcí.

5.3.1 Hlavní smyčka programu

Jádro celého programu tvoří nekonečná smyčka, která vykonává základní funkce pro vykreslování. Tato smyčka je přerušena pouze stisken klávesy `ESC`, která vyvolá ukončení renderování a vypnutí programu.

Na začátku hlavní smyčky se volají funkce z OpenGL API pro vyčištění obrazovky a nastavení korektní velikosti OpenGL kontextu kvůli možné změně velikosti okna programu. První důležitou funkcí, která je ve smyčce volána, je `updateRender`. Funke slouží pro aktualizaci vykreslování a blíže se jí věnuje kapitola 5.3.2. Před samotným voláním této funke je do proměnné `timeStart` uložen aktuální čas. Aktuální čas se také uloží po skončení funkce `updateRender`, a to do proměnné `timeEnd`. Obě tyto proměnné jsou využity pro výpočet celkového času, po který funkce `updateRender` (a tím i algoritmus Pathtracing) běžela. Tento čas je spolu s celkovým časem, po který program běží, a číslem aktuálního vzorku Pathtracingu vypsán funkcí `printStats`. Funkce `printStats` se volá po skočení funkce `updateRender`, aby byly vypsány důležité statistiky pro uživatele. Výsledky z `updateRender` se poté zakreslí do textury funkcí `drawTexture`. V této funkci se pomocí OpenGL shader programu vykreslí čtveřice bodů, na které se vykreslí textura s výsledným renderem. Pokud tuto funkci zavolal program `CPU_IBP_pathtracer` je potřeba zkopírovat barevné hodnoty z proměnné `framebuffer` přímo do textury. v proměnné `framebuffer` jsou uloženy jednotlivé barvy pixelů jako datový typ `unsigned int`. Ve verzi akcelerované na GPU není přesunutí hodnot nutné, protože vypočítaná barva se ukládá přímo do sdílené textury mezi OpenGL a OpenCL. Návrh hlavní smyčky je vidět v pseudokódu 1

Algorithm 1 Hlavní smyčka programu

```
1: function renderLoop
2:     while not skonciProgram do
3:         ziskej vstup
4:         ziskej velikost okna
5:         nastav novou velikost okna
6:         renderStart = aktualni cas
7:         aktualizujRender
8:         renderEnd = aktualni cas
9:         vypis statistiky
10:        vykresli do textury
11:        prohod GL buffery
12:    end while
13:    uvolni pamet
14:    skonci program
15: end function
```

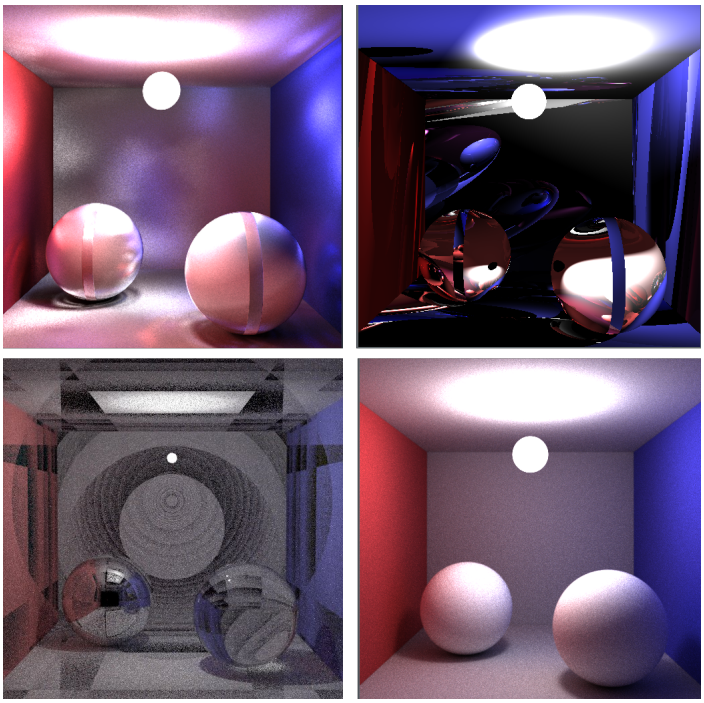
5.3.2 Generování paprsků

Chování této funkce je přímo závislé programu, který vyvolal její spuštění. Pokud se jedná o verzi pro CPU renderování dochází zde ke generování jednotlivých paprsků srkze každý jednotlivý pixel vykreslovací plochy. Ihned po získání koordinát [`x`,`y`], které reprezentují pozici pixelu, je inicializován Xorshift generátor náhodných čísel. Po inicializaci generátoru dojde k rozdělení vybraného pixelu na 2x2 subpixely s náhodnou pozicí. Ty jsou využity pro **vícenásobné vzorkování** barvy pixelu pomocí filtru **Tent filter**. Využití tohoto filtru zjemňuje ostré přechody ve scéně které se nazývají *aliasing*. Tent filter rozděluje jednotlivým vzorkům *váhu* z lineární b-spline funkce. Oblasti blíže středu pixelu je přidělena větší váha, která s horizontální a vertikální vzdáleností klesá. Takto vypočítané koordináty jsou použity pro nastavení směru paprsku do scény. Paprsku se jako počátek nastaví pozice kamery. Zinicializovaný paprsek se předá obarvovací funkci `radiance`, ve které je prováděn samotná algoritmus Pathtracing. Této funkci se věnuje kapitola 5.3.3

V prvotní fázi programu byl jako generátor náhodných čísel pro celý program vybrán linuxový generátor `erand48`. Jedná se o *lineární kongruentní generátor*2.5 náhodných čísel, který vrací čísla typu `double` používající 48 bitovou celočíselnou aritmetiku. Jednoduchost tohoto algoritmu je však zároveň překážkou pro jeho využití při Monte Carlo integraci. Jako lineární kongruentní generátor nabízí relativně malou periodu, a to i při použití 48 bitových čísel. Další nevýhodou je návratový typ `double`. Z důvodu lepší přenositelnosti OpenCL části programu se v celém programu veškerá čísla s plovoucí desetinnou čárkou ukládají jako datový typ `float`, čili čísla s pouze jednoduchou přesností. Využít typ `double` v OpenCL kernelu možné, pouze pokud výrobce zahrnul rozšíření *cl_khr_fp64* do svých ovladačů.

Z těchto důvodů je v práci naimplementován generátor Xorshift, který je využit napříč celým programem pro generování náhodných čísel. Jak je uvedeno v kapitole 2.5, generování náhodných čísel je kritickou částí Pathtracingu a Monte Carlo integrace. Nekvalitní generátor, špatně zvolené konstanty nebo typová koverze (v tomto případě `double` na `float`) mohou zapříčinit nechtěné chování celého algoritmu. Výsledné rendery se neblíží skutečnému vzhledu scény nebo ji dokonce přetvoří v abstraktní umění. Vliv generování náhodných čí-

sel je prezentován v obrázku 5.1. Levý horní obrázek byl vytvořen se špatně generovanou množinou náhodných čísel. V pravém horním obrázeku byly náhodná čísla zaměněna za konstantní hodnotu. Levý spodní obrázek ukazuje chybu při převádění z datového typu `double` na datový typ `float`. Konkrétní chyba v tomto případě je menší přesnost typu `float`. Teprve pravý spodní obrázek má správně nastavený generátor náhodných čísel.



Obrázek 5.1: Vliv generátoru náhodných čísel na výsledný render.

5.3.3 Obarvení pixelů

Bezpochyby nejzajímavější částí implementace této práce je funkce `radiance`, ve které dochází k výpočtu barvy jednotlivých pixelů obrazovky. Jedná se také o funkci, ve které je implementována většina pojmů a konceptů z kapitoly 2. Pseudokód 2 popisuje náplň funkce `radiance`. Pro urychlení výpočtu obsahuje funkce explicitní vzorkování světel. Osvětlení povrchu se v Pathtracing algoritmu získává náhodným vysíláním paprsků (viz. 2.9). Pravděpodobnost že se tímto postupem podaří strefit zdroj světla (především malý zdroj), je velmi malá. Proto je výpočet osvětlení rozdělen do dvou částí, výpočet přímého osvětlení a výpočet nepřímého osvětlení. Tyto dvě složky nakonec dají výslednou barvu osvětleného objektu.

Algorithm 2 Obarvovací funkce `Radiance`.

Vstup: paprsek, hloubka

```
1: function RADIANCE
2:     indexObjektu = 0
3:     if not testProtnuti(papsek, indexObjektu) then
4:         return RGB(0.0)                    // Pokud nenastalo protnutí, vrať černou.
5:     end if
6:     ziskejVlastnostiPovrchu(indexObjektu)
7:     Russian Roulette pro ukončení
8:     if materialobjektu = DIFFUSE then
9:         Náhodně vyber směr na jednotkové polokouli
10:        Explicitně vzorkuj světla ve scéně
11:        return barvaobjektu + RADIANCE(paprsek s náhodným směrem)
12:    end if                                 // Perfektní odraz - úhel odrazu = úhel dopadu
13:    if materialobjektu = SPECULAR then
14:        return barvaobjektu + RADIANCE(odražený paprsek)
15:    end if
16:    if materialobjektu = REFRACTIVE then
17:        if Úplný odraz světla == TRUE then
18:            return barvaobjektu + RADIANCE(odražený parpsek)
19:        end if
20:        Vypočti pravdepodobnost odrazu a lomu podle Fresnellových rovnice
21:        if pravděbodobost == ODRAZIT then
22:            return barvaobjektu + RADIANCE(odražený parpsek)
23:        end if
24:        if pravděbodobost == LOMIT then
25:            return barvaobjektu + RADIANCE(lomený parpsek)
26:        end if
27:    end if
28: end function
```

5.3.4 OpenCL část programu

OpenCL část programu je logicky oddělena od zbytku programu a je implementována jako třída `ConfigCL`. Tato třída má na starosti inicializaci OpenCL kontextu, vytvoření kernel programu a potřebných OpenCL bufferů.

Je to zároveň jediné využití OOP[1] v této práci. Prvotní implementace této práce využívala OOP ve velké míře. Bohužel se toto programovací paradigma ukázalo býti spíše na škodu. Složitá volání metod jednotlivých tříd celý kód znepřehledňovala a dokonce zpomalovala. Bylo proto rozhodnuto kód předělat a rozdělit program na jednotlivé funkce, které se v určitých momentech využijí. Stav programu a důležité informace jsou uloženy jako globální proměnné, ke kterým se jednotlivým částem programu dává přístup. Tento jednoduchý přístup zvýšil přehlednost kódu programu a navýšil také rychlost jeho spuštění.

Následující podkapitoly se věnují jednotlivým důležitým částem třídy `ConfigCL`.

[1] Objektově orientované programování.

Inicializace OpenCL kontextu

První důležitou operací kterou je potřeba provést je inicializace OpenCL kontextu na hostitelském zařízení (procesor na kterém je spuštěn hlavní program). Tato část je prováděna v metodě třídy `ConfigCL` s názvem `InitOpenCL`. Jako první je potřeba vyhledat všechny *OpenCL platformy* a veškerá jejich *OpenCL zařízení* (anglicky device). Mezi těmito zařízeními se vybere první, které podporuje OpenCL rozšíření dovolující sdílení bufferů mezi OpenCL a OpenGL s označením `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR`. Ve většině případů je tímto způsobem vybrána grafická karta která toto rožšíření podporuje ale je možné, že se vybere i jiný typ zařízení. Tento způsob výběru byl zvolen z důvodu lepší přenositelnosti a možnosti testování na více zařízeních.

Dále třída `ConfigCL` obsahuje metodu `AllocateBuffers`, ve které se potřebné OpenCL buffery alokují a plní hodnotami. V této funkci především vytvoří sdílená paměť mez OpenGL a OpenCL. Tuto sdílenou část představuje OpenGL textura, ze které se vytvoří OpenCL buffer s datovým typem `cl::ImageGL`. Odpadá tak potřeba neustále po každém výpočtu kopírovat data z CPU na GPU a zpět, čímž se výrazně zrychlí výpočetní čas programu.

Spuštění kernel programu

Poslední metodou, která stojí za zmínku, je metoda `RunKernel` pouštící výpočetní kernel na vybraném OpenCL zařízení. Nejprve provede nastavení potřebných argumentů, které se mohli pomocí uživatelovi interakce změnit. Takto jsou aktualizovány OpenCL buffery obsahující kameru a právě zobrazovaná scéna.

Prvním nastaveným argumentem je hodnota aktuálního zpracovávaného vzorku Pathtracingu. Protože průměrování hodnot Pathtracing algoritmu probíhá přímo v kernel programu, je potřeba tento argument v každém volání aktualizovat. Dalším argumentem předávaným argumentem je pole náhodně vygenerovaných celých čísel, kde velikost pole odpovídá rozměrům obrazovky. Každému zpracovávanému pixelu odpovídá jedna hodnota v poli náhodných čísel, kterou použije jako počáteční hodnotu (anglicky *seeed*) pro Xorshift generátor náhodných čísel. Tento generátoru byl poupraven a implemtován také přímo do kernel programu. Protože OpenCL nemá přímo implementovaný generátor náhodných čísel, který by mohl posloužit jako počáteční hodnota Xorshift generátoru, je potřeba tyto hodnoty generovat na hostitelkém zařízení a přenést je do kernel programu. Poslední operací, která je potřeba provést před samotným spuštění kernel programu, je zamčení OpenGL textury kvůli výlučnému přístupu. Jakmile je textura uzamčena, je možné spustit kernely pro jednotlivé pixely obrazovky. Po jejich skončení se OpeGL textura opět odemče aby ji bylo možné vykreslit na obrazovku.

Samotný kernel program je implementován stejně jako CPU verze až na pár drobných změn, které bylo potřeba provést z důvodu limitace jazyka OpenCL. Např. jednotlivé matematické operace pro práci s vektory bylo potřeba převést na samostatné funkce. V CPU verzi vystupují funkce struktury `Vec`. Jak již bylo zmíněno v úvodu kapitoly 5.3, bylo potřeba převést funkci `radiance` z rekurzivní funkce na funkci iterativní. Dále byly některé části přesunuty do samostatných celků jako např. funkce `sampleLights`, která řeší přímé osvětlení bodu. Po skončení funkce `radiance` jsou její výsledky rovnou zapsány do textury, která je následně vykreslena.

Kapitola 6

Testování

Tato kapitola pojednává o testování výsledného programu a dále srovnává jeho výsledky s existujícími řešeními.

6.1 Testování časové a výpočetní náročnosti

Je potřeba otestovat výkon výsledné implementace programu, aby bylo vůbec možné zhodnotit jeho kvalitu. Testování probíhalo na dvou připravených scénach Cornell Boxu, které jsou popsány v části 5.2. Pro prvotní testování bylo rozlišení obrazovky nastavano na 512x512 pixelů a délka paprsku na hodnotu 6. Hodnoty se začaly měřit po vykreslení min. 128 vzorků. Odstraní se tím možná nepřesnost a chyby při měření. Výsledky tohoto testování jsou v tabulce 6.1.

Tabulka 6.1: Průměrné naměřené hodnoty pro rozlišení 512x512

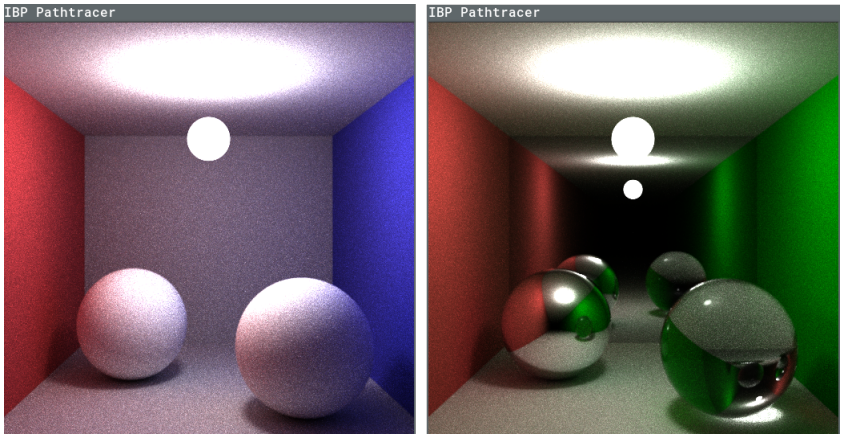
Sestava	Zařízení	Čas 1 vzorku v ms	Vzorky/sekunda	Pamětová náročnost v MB
1	CPU	2485.256	0.40	24
	GPU	315.227	3.17	96
2	CPU	720.355	13.82	40
	GPU	16.435	60.84	144

V předchozí tabulce záměrně chybí zmínka o použité scéně. Experimentálně bylo totiž zjištěno, že změna materiálu objektů měla pro toto rozlišení na vykreslovací čas zanedbatelný vliv. Bude však ještě nutné prověřit tento fakt na vyšším rozlišení. Výsledky programu po 256 vzorcích jsou vidět na obrázku 6.1.

Jako vyšší rozlišení bylo obecně rozšířené rozlišení *FullHD* 1920x1080, to dává celkem 2 073 600 paprsků. Pro srovnání, při rozlišení 512x512 je vysláno 252 144 paprsků. Což je zhruba 700% nárust. Sestava 1 už bohužel nedostačuje svým výkonem na tak náročnou operaci, proto není zaneseno do tabulek 6.2 a 6.3.

Tabulka 6.2: Průměrné naměřené hodnoty pro rozlišení 1920x1080 – první scéna

Sestava	Zařízení	Čas 1 vzorku v ms	Vzorky/sekunda	Pamětová náročnost v MB
2	CPU	5320.120	0.19	88
	GPU	77.456	12.91	192



Obrázek 6.1: Cornell Box scény vykreslené s 256 vzorky na pixel.

Tabulka 6.3: Průměrné naměřené hodnoty pro rozlišení 1920x1080 – druhá scéna

Sestava	Zařízení	Čas 1 vzorku v ms	Vzorky/sekunda	Pamětová náročnost v MB
2	CPU	5112.282	0.20	88
	GPU	78.822	12.69	192

Až tento test odhalil, že scéna 2 obsahující objekty s odrazivými a lomivými materiály se vykreslila rychleji než scéna 1 obsahující pouze difúzní objekty. V GPU implementaci je rozdíl opět zanedbatelný, ovšem v CPU verzi je rozdíl průměrného času pro jeden snímek cca 200 ms ve prospěch scény 2. Protože CPU verze běží už tak znatelně pomaleji, je i v tomto případě rozdíl rychlosti víceméně také zanedbatelný.

Dále je z naměřených hodnot patrná větší paměťová náročnost akcelerované verze programu, která je dána vyšší režií při použití OpenCL API a nutností alokovat pro toto API potřebné buffery.

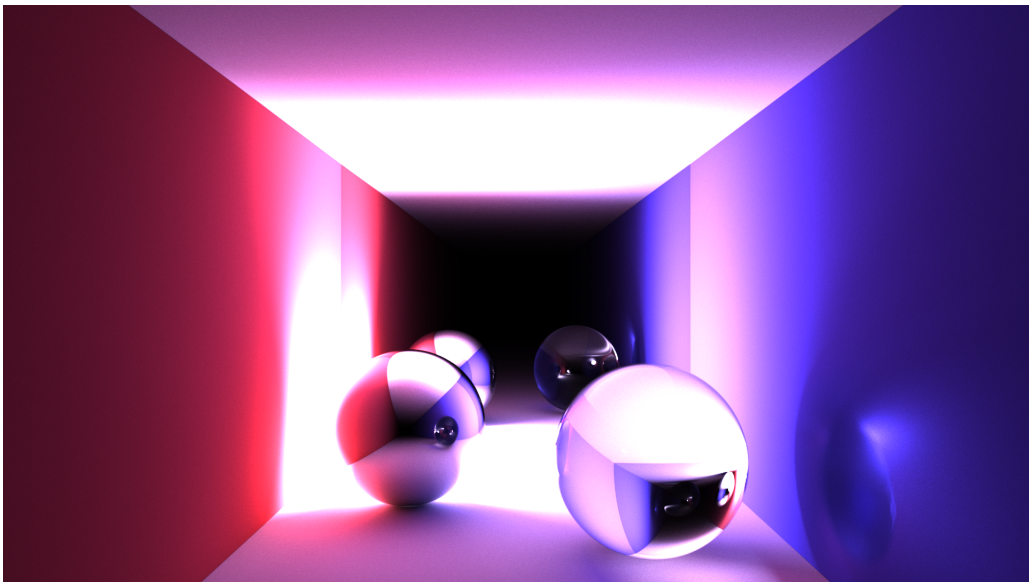
6.2 Srovnání se stávajícími řešeními

Pro objektivní porovnání bylo potřeba vybrat program vykreslující podobnou scénu při stejném nebo podobném rozlišení. Jako první se nabízí *WebGL Path Tracing* z části 3.4. Jeho implementace v jazyce JavaScript však nedovolila přesné změření výpočetního času. Není tedy možné objektivně porovnat toto řešení s výslednou implementací v této práci.

Následně bylo zvoleno řešení *SmallptGPU2* popsané v části 3.2. Scéna v tomto programu je stejný Cornell Box jako je využit v této práci. Bohužel se v tomto řešení podařilo změřit pouze výkon na CPU, a tak je porovnána pouze tato implementace. Jak ve vidět v tabulce 3.3, *SmallptGPU2* má na sestavě 1 stabilnější a výrazně lepší výsledky než implementovaný program. Stejně tak na sestavě 2 je patrný rozdíl v rychlosti, ovšem zde už ne tak drastický. To vše dává podnět k lepší optimalizaci a vylepšení stávající implementace např. pomocí akceleračních datových struktur.

Pro možnost srovnání výkonu implementace akcelerované na grafické kartě na sestavě 2 bylo vybráno řešení *Sfera* z podkapitoly 3.3. Tato hra sice obsahuje složitější scénu, avšak jak je vidět v tabulce 3.4, implementace této práce dosáhla stejného výsledku na sestavě 2 při akceleraci na GPU.

Ačkoliv při porovnání se *SmallptGPU2* má tato práce poněkud horší výsledky, při přepočtu doby výpočtu jednoho snímku na snímky za sekundu (FPS) dosáhne 60 FPS, což je velmi dobrý výsledek. Výsledný program také velmi rychle konverguje, a zbavuje se tak počátečního šumu. Výsledný program tedy dokáže vykreslit scénu poměrně zdatně. Příkladem jeho možností je obrázek 6.2.



Obrázek 6.2: Reprezentace možností výsledné implementace algoritmu Pathracing.

Kapitola 7

Závěr

Cílem této bakalářské práce bylo nastudování metody Pathtracing a její implementace, která je akcelerovaná na grafické kartě. Nejprve bylo potřeba nastudovat jednotlivé metody založené na principu sledování paprsku a seznámit se s již existujícími implementacemi. Nastudování teorie a především analýza ostatních řešení pomohly detailněji pochopit problematiku metod globálního osvětlení a realistického vykreslování. Existující řešení také ukázaly určité způsoby jak překonat již dlouho známé problémy, se kterými se tyto metody mohou potýkat.

Dále bylo potřeba nastudovat dostupné možnosti a technologie zabývající se akcelerovanými paralelními výpočty, které jsou v dnešní době velmi populární a hojně využívané např. v bioinformate, finančních analýzách nebo při výpočtech proudění tekutin. Teoreticky se dá říci, že použití akcelerace na grafické kartě se dá najít všude, kde je zapotřebí rychlost výpočtu a vývojáři jsou ochotní naučit se a aplikovat nové paralelní programovací paradigma. V tomto odvětví nejvíce převládá technologie CUDA od společnosti NVIDIA avšak pro tuto práci bylo vybráno OpenCL, především proto, že se jedná o otevřený standard a nezávisí na konkrétním hardware.

V neposlední řade bylo potřeba nastudovat API OpenGL a jeho programovatelné shadery pro zobrazování vypočtených vzorků algoritmu Pathtracing. Ikdyž je tato API v práci využita pouze pro vykreslení výsledků, pochopení její funkčnosti bylo potřeba pro korektní interoperabilitu mezi zobrazovacím OpenGL a výpočetním OpenCL.

Výsledkem práce je program schopný zobrazovat předem vytvořené scény v téměř fotorealistické kvalitě. Rychlost konvergence algoritmu a rychlost jeho výpočtu, především na grafické kartě, dovolují získat nezašuměné výsledky po pár vteřinách. V rámci lepší interaktivity uživatele s programem je výsledná implementace doplňena o možnost interagovat se scénou a uživatel si tak sám může prohlédnou a analyzovat různé efekty, které metoda Pathtracing a tato práce nabízí.

Experimenty s programem vykazovali velmi dobré výsledky, především na moderním grafickém hardware bylo možné dosáhnout ideální rychlosti 60 snímků za sekundu. Je tedy prokazatelné, že jednoduché scény je možné zvládnout vykreslovat v reálném čase s přijatelnou kvalitou. Právě tuto vlastnost využila naplno výše zmíněná hra Sfera pro interaktivní zábavu. K aplikaci na složitější scény bude ovšem potřeba provést ještě mnoho výzkumu a vyvinout rychlejší běžně dostupný hardware.

Možné rozšíření práce spočívá především v přidání dalších primitiv, se kterými by mohl algoritmus pracovat. V kombinaci s možností nahrávat modely nebo celé scény ve standarizovaném formátu by pomohlo program, a tím i algoritmus Pathtracing, lépe otestovat a získat tak zajímavější data a výsledné snímky pro analýzu kvality tohoto algoritmu. V sou-

vislosti s tímto rozšířením by bylo možné implementovat různé akcelerační datové struktury, které by umožnily snížit výpočetní nároky programu a opět jej přiblížily k vykreslování v reálném čase. V neposlední řade je potřeba zmínit možnost rozšířit tuto práci o implementaci metody obousměrného Pathtracingu (Bidirectional Pathtracing). Tato technika kombinuje sledování paprsků od kamery do scény z metody Pathtracing spolu se sledováním paprsků od zdroje světla do scény. Při stejném výpočetním čase na snímek by se s menším počtem vzorků podařilo dosáhnout vyšší kvality výsledného snímku.

S každým rokem přichází výrobci grafických karet s novými výrobními technologiemi které dosahují čím dál vyššího výkonu. Pokud se tento výkon zkombinuje s paralelním programovacím paradigmatem, bude možné dosáhnout velmi dobrých výsledků u jinak výpočetně náročných operací.

Literatura

[1] The Graphics Codex [online]. 2016-03-08 [cit. 2016-04-20].
URL http://graphicscodex.com/index.php

[2] Introduction to Parallel Programming video lecture series – Part 05 "OpenMP for Domain Decomposition": Intel Developer Zone [online]. 2016-04-10 [cit. 2016-05-10].
URL https://software.intel.com/en-us/videos/introduction-to-parallel-programming-video-lecture-series-part-05-openmp-for-domain

[3] Monte Carlo Methods in Practice: Scratchapixel 2.0 [online]. 2016-01-10 [cit. 2016-04-22].
URL http://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice

[4] APPEL, A.: Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring) [online], New York, NY, USA: ACM, 1968, s. 37–45, doi:10.1145/1468075.1468082.
URL http://doi.acm.org/10.1145/1468075.1468082

[5] DUTRÉ, P.; BEKAERT, P.; BALA, K.: *Advanced Global Illumination*. Natick, Mass.: AK Peters Ltd., 2003, ISBN 1-56881-177-2.

[6] KAJIYA, J. T.: The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86 [online], New York, NY, USA: ACM, 1986, s. 143–150, doi:10.1145/15922.15902, ISBN 0-89791-196-2.
URL http://doi.acm.org/10.1145/15922.15902

[7] MARSAGLIA, G.: Xorshift RNGs. Technická zpráva, The Florida State University, 2003.

[8] SCARPINO, M.: *OpenCL in Action*. Shelter Island, NY: Manning Publications Co., 2012, ISBN 978-1617290176.

[9] SELLERS, G.; Jr., R. S. W.; HAEMEL, N.: *OpenGL superbible: comprehensive tutorial and reference*. Addison-Wesley Professional, 7 vydání, 2015, ISBN 978-0672337475.

[10] SHIRLEY, P.; MORLEY, R. K.: *Realistic Ray Tracing*. AK Peters Ltd., 2003, ISBN 1-56881-198-5.

Přílohy

Seznam příloh

Příloha A

Obsah CD

Obsah CD a popis jednotlivých částí

- `doc` – složka obsahující dokumentaci k této práci `dokumentace.pdf`.
- `video` – složka obsahující video reprezentující výsledky práce.
- `src` – složka obsahující zdrojové kódy programu v jazyce C++.
- `shaders` – složka obsahující soubory s kódem pro fragment a vertex shader.
- `kernels` – složka obsahující soubor se zdrojovým kódem kernel programu.
- `obj` – složka pro objektové soubory při překladu programu.
- `Makefile` – soubor s direktivy pro program *make* k sestavení programu.

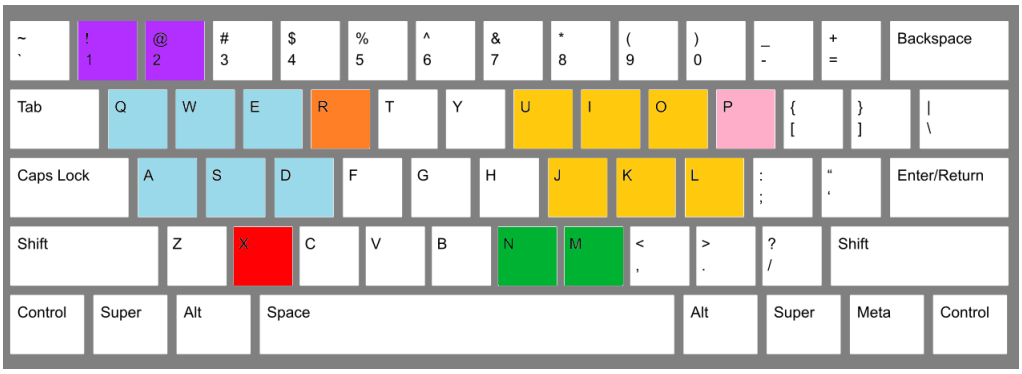
Příloha B

Manuál

B.1 Sestavení programu

Pro sestavení programu je přiložen soubor `Makefile`. Po jeho spuštění se vytvoří dva spustitelné soubory, `CPU_IBP_pathtracer` a `GPU_IBP_pathtracer`. První zmíněný soubor využívá k výpočtu pouze CPU, druhý výpočet akceleruje na grafické kartě pomocí OpenCL. Po spuštění jednoho z nich se objeví okno, do kterého bude algoritmus Pathtracing vykreslovat. Do terminálového okna se pak vypisují statistiky programu.

B.2 Ovládání programu



Obrázek B.1: Rozložení ovládání programu.

Ovládání programu zahrnuje klávesnici i myš. Myší je možné se po scéně rozhlížet a klávesnicí ovzládát různé objekty ve scéně. Ovládání pomocí klávesnice je následující:

- W, S, A, D – Pohyb kamery vpřed, vzad, vlevo a vpravo.
- Q, E – Pohyb kamery nahoru a dolů.
- R – Reset pozice kamery
- I, K, J, L – Pohyb světla vpřed, vzad, vlevo a vpravo
- U, O – Pohyb světla nahoru a dolů

- M, N – Zvýšení a snížení intenzity světla
- X – Zamknutí pohybu myší
- P – Náhodně změní barvu světla
- 1, 2 – Změna scény