

# Path Tracing na GPU

Karel Tomanec  
tomanka4@fel.cvut.cz

Závěrečná zpráva k projektu z předmětu *B4M39GPU*  
zimní semestr 2021/2022

**Abstrakt**—Cílem tohoto projektu je implementace metody sledování cest na CPU i na GPU s využitím akcelerační datové struktury a následné porovnání rychlosti obou implementací.

## I. TEORIE

V reálném světě se existence objektů výrazně projevuje při interakci se světlem. Objekty nejsou osvětleny pouze na částech, které jsou viditelné ze zdroje světla, ale také „nepřímo“ na částech odvrácených od zdroje světla a to odražením paprsků od jiných objektů.

Ray tracing je zobrazovací metoda, která používá zpětné sledování paprsku a umožňuje vytvořit globální osvětlení ve scéně, které simuluje reálné optické jevy jako jsou například odraz, lom světla, půjčování barev, kaustiky atd.

### A. Zobrazovací rovnice

Globální charakter vztahů mezi objekty popisuje zobrazovací rovnice.

Zobrazovací rovnice v úhlovém tvaru:

$$L(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{H(\mathbf{x})} L(r(\mathbf{x}, \omega_i), -\omega_i) f_r(\mathbf{x}, \omega_o, \omega_i) \cos(\theta_i) d\omega_i \quad (1)$$

kde

- $L(\mathbf{x}, \omega)$  je příchozí radiance v bodě  $\mathbf{x}$  ve směru  $\omega$
- $L_e(\mathbf{x}, \omega)$  je emitovaná radiance z bodu  $\mathbf{x}$  a pro daný směr  $\omega$
- $r(\mathbf{x}, \omega)$  je funkce vržení paprsku z bodu  $\mathbf{x}$  ve směru  $\omega$  vracející bod v prostoru
- $f_r(\mathbf{x}, \omega_o, \omega_i)$  je BRDF funkce v daném bodě pro příchozí a odchozí směr
- $\theta_i$  je úhel mezi normálou a příchozím paprskem
- $H(\mathbf{x})$  představuje všechny směry jednotkové hemisféry se středem v bodě  $\mathbf{x}$

Rovnice v tomto tvaru popisuje přenos světla ve scéně a její řešení poskytuje rovnovážný stav rozložení světla ve scéně (*energy balance*). Řešení této rovnice analyticky je ovšem v obecném případě nemožné, kvůli neznámým

radiancím  $L$  na obou stranách rovnice a proto je nutné řešit tuto rovnici approximací řešení analytického.

### B. Path tracing

Path tracing je nestranná metoda používající Monte Carlo integraci pro řešení zobrazovací rovnice.

Algoritmus postupně generuje cesty paprsku z kamery, které jsou následně rozptýleny ve scéně formou „náhodné procházky“. Výsledkem jsou příchozí radiance pro daný pixel, které jsou standardně zprůměrovány. V obecném případě je třeba vržení několika tisíců takových paprsků pro vytvoření kvalitního obrázku. Algoritmus používá metodu Monte Carlo pro odhad řešení integrálu v zobrazovací rovnici pomocí náhodně vygenerovaného směru a jeho hustoty pravděpodobnosti. Pseudokód tohoto algoritmu se nachází pod tímto odstavcem.

---

#### Algorithm 1 GetRadianceLi( $x, \omega$ )

---

```

 $L \leftarrow (0, 0, 0)$ 
 $\beta \leftarrow (1, 1, 1)$ 
loop
     $hit \leftarrow \text{NearestIntersection}(x, \omega)$ 
    if  $hit.intersect == \text{false}$  then
        return  $L + \beta \times \text{BackgroundRadiance}(x, \omega)$ 
    end if
    if  $IsOnLightSource(hit)$  then
         $L \leftarrow L + \beta \times Le(hit.pos, -\omega)$ 
    end if
     $\rho \leftarrow \text{Reflectance}(hit.pos, -w)$ 
    if  $\text{Rand}() < \rho$  then
         $\omega_i \leftarrow \text{SampleDir}(hit)$ 
         $\beta \leftarrow \beta \times f(hit.pos, \omega_i, -\omega) \times \text{Dot}(hit.n, \omega_i)$ 
         $\beta \leftarrow \beta / (\rho \times p(\omega_i))$ 
         $x \leftarrow hit.pos$ 
         $w \leftarrow \omega_i$ 
    else
        break
    end if
end loop
return  $L$ 

```

---

Path Tracing na rozdíl od standardního distribuovaného ray tracingu vrhá vždy pouze jeden sekundární paprsek, čímž nedochází k „explozi paprsků“ v nižších patrech rekurze.

Na vstupu funkce pro zjištění radiance, prostřednictvím path tracingu, je bod  $x$ , který představuje počátek paprsku a směr  $\omega$  představující jeho směr.

Algoritmus nejprve najde pro daný paprsek nejbližší průsečík se scénou. Pokud paprsek neprotne žádný objekt, funkce vrátí radianci mapy okolí, která představuje nekonečně vzdálený zdroj světla obklopující celou scénu.

Dále je potřeba zkontrolovat, zda je materiál objektu emisivní a představuje tedy implicitní zdroj světla. Pokud ano, přičteme k výsledné radianci příspěvek vyzářené radiance.

Následně je zjištěna odrazivost materiálu (*albedo*), podle které určíme pravděpodobnost přežití cesty porovnáním s náhodně vygenerovaným číslem (*russian roulette*). Tímto můžeme zamezit pokračováním rekurze, při které by byl příspěvek radiance minimální. Použití odrazivosti jako pravděpodobnosti přežití cesty dává smysl, protože jestli plocha odráží daný poměr energie, tak pokračujeme pouze s pravděpodobností rovnou tomuto poměru.

Pokud pokračujeme v rekurzi, tak je potřeba vygenerovat další paprsku směr. Tento směr vybíráme vzorkováním BRDF, které je popsáno v následující podkapitole.

V dalším kroku je potřeba upravit hodnotu  $\beta$ , která určuje pravděpodobnost jednotlivých příspěvků radiance. Tato hodnota je nejprve vynásobena funkcí  $f$  daného materiálu, představující BRDF, skalárním součinem normály povrchu a nově vygenerovaného směru. Dále je podělena součinem odrazivosti  $\rho$  a pravděpodobnostní hustoty nového směru  $p(\omega_i)$ .

Takto pokračujeme znova v nově vygenerovaném směru, dokud není cesta ukončena prostřednictvím *russian roulette*.

### C. Vzorkování Phongova BRDF

V algoritmu Path Tracing je potřeba vybrat nový směr po odrazu s povrchem objektu. Předpokládáme, že paprsek dopadá na povrch s fyzikálně věrohodnou Phongovou BRDF [2]:

$$f_r^{Phong}(\mathbf{x}, \omega_o, \omega_i) = \frac{\rho_d}{\pi} + \frac{n+2}{2\pi} \rho_s \cos^n(\theta_r) \quad (2)$$

kde

- $\rho_d$  je difúzní odrazivost materiálu
- $\rho_s$  je spekulární odrazivost materiálu
- $n$  je míra lesklosti materiálu
- $\cos(\theta_r) = \omega_o \cdot \omega_r$

- $\omega_r = 2(\omega_i \cdot \mathbf{n})\mathbf{n} - \omega_i$
- $\rho_d + \rho_s \leq 1$  (zákon zachování energie)

BRDF je matematický popis odrazivých vlastností povrchu a určuje hustotu pravděpodobnosti, že foton dopadající na povrch ze směru  $\omega_i$  bude odražen ve směru  $\omega_o$ .

Při vzorkování BRDF je potřeba nejprve vybrat druh interakce neboli komponentu BRDF (difúzní odraz, lesklý odraz, lom, absorpcie atd.). U fyzikálně korektní Phongova BRDF vygenerujeme náhodnou hodnotu v intervalu  $[0, 1]$  a poté ji porovnáváme s odrazovostmi materiálu. Pokud je hodnota menší než  $\rho_d$ , vybrali jsme difúzní odraz, pokud je hodnota menší než  $\rho_s + \rho_d$ , vybrali jsme spekulární (lesklý) odraz. V jiném případě dochází k absorpci.

Po výběru interakce se daná komponenta vzorkuje pomocí dvou náhodně vygenerovaných čísel  $r_1$  a  $r_2$ .

Vzorkování difúzního odrazu s bázovým vektorem ve směru normály povrchu a s pravděpodobnostní hustotou  $p(\theta) = \frac{\cos(\theta)}{\pi}$ , kde  $\theta$  je úhel mezi normálou povrchu a vygenerovaným sekundárním paprskem  $\omega_i$ :

$$\begin{aligned} x &= \cos(2\pi r_1)\sqrt{1 - r_2} \\ y &= \sin(2\pi r_1)\sqrt{1 - r_2} \\ z &= \sqrt{r_2} \end{aligned} \quad (3)$$

Vzorkování lesklého odrazu s bázovým vektorem ve směru normály povrchu a s pravděpodobnostní hustotou  $p(\theta) = \frac{n+1}{\pi} \cos^n(\theta)$ , kde  $\theta$  je úhel mezi ideálně zrcadlově odraženým  $\omega_o$  a vygenerovaným sekundárním paprskem  $\omega_i$ :

$$\begin{aligned} x &= \cos(2\pi r_1)\sqrt{1 - r_2^{\frac{2}{n+1}}} \\ y &= \sin(2\pi r_1)\sqrt{1 - r_2^{\frac{2}{n+1}}} \\ z &= r_2^{\frac{1}{n+1}} \end{aligned} \quad (4)$$

### D. Hierarchie obálek

*Bounding volume hierarchy*, neboli hierarchie obálek, je akcelerační struktura používaná standardně pro urychlení počítání průsečíků s objekty v Path Tracingu.

Jedná se o stromovou strukturu nad množinou geometrických primitiv. Primitiva jsou uložena v listech tohoto stromu, které nad nimi tvoří obálku. Vnitřní uzly stromu zapouzdřují obálky jednotlivých podstromů a tvoří tak větší obálku.

Při průchodu touto strukturou v Path Tracingu využíváme skutečnosti, že pokud paprsek mine danou obálku, tak mine všechna primitiva zapouzdřená v příslušném podstromu a nemusíme tak s nimi počítat průsečíky.

## II. POPIS IMPLEMENTACE NA CPU

Jako první byla vytvořena implementace metody sledování cest na CPU pro snadnější odladění a zjednodušení následné GPU implementace.

Implementace se skládá z několika částí:

- **ray.h** reprezentace paprsku
- **camera.h** reprezentace kamery, generování paprsku, pohyb
- **sphere.h** reprezentace sféry, výpočet průsečíku s paprskem
- **aabb.h** reprezentace AABB, výpočet průsečíku s paprskem
- **bvh.h** reprezentace hierarchie obálek, průchod strukturou
- **integrator.h** algoritmus sledování cest, renderovací cyklus
- **hitable.h** záznam o průsečíku se scénou
- **vector3.h** reprezentace 3D vektorů a operace s nimi
- **material.h** třída popisující materiály, výpočet BRDF, vzorkování BRDF
- **utility.h** pomocné funkce
- **image.h** reprezentace 2D obrázku
- **environment\_map.h** reprezentace mapy okolí

Jádrem implementace je **integrator.h**, který se stará o výpočet zobrazovací rovnice pomocí sledování cest. Funkce `Render` prochází jednotlivé pixely renderovaného obrázku dvěma for cykly. Třetí for cyklus iteruje přes počet vzorků pro pixel, uvnitř kterého se vygeneruje paprsek a spustí se funkce `Trace`, která paprsek vrhne do scény a provádí samotný algoritmus sledování cest, jak je uvedeno v pseudokódu. Implementovaný algoritmus je vytvořen pro scény, ve kterých se nachází pouze implicitní zdroje osvětlení.

Výpočet funkce `Trace` je nezávislý pro každý pixel, což vede na možnost paralelizace. For cyklus, který iteruje přes sloupce obrázku je tedy zparalelizovaný pomocí knihovny OpenMP.

Hierarchie obálek (BVH) je postavena metodou *top-down*. Dělící dimenze se postupně střídají (*round robin*) a primitiva jsou rozdělena na dvě části podle mediánu. Průchod hierarchií je *stack-based* přepsaný do nerekurzivní varianty.

## III. POPIS IMPLEMENTACE NA GPU

Paralelní řešení na GPU využívá nezávislého výpočtu funkce `Trace` pro každý pixel. Funkce `Render` je implementována jako CUDA kernel, který pro každý pixel generuje paprsek a volá funkci `Trace`.

Velikost bloku byla zvolena na  $8 \times 8$ . Tato velikost byla zvolena tak, aby byla násobkem 32, což je velikost warpu na současných architekturách. Druhým aspektem

při volbě velikosti bloku bylo vytvořit blok dostatečně malý, aby práce vykonaná jednotlivými bloky skončila v podobném čase, protože kdyby některé vlákna v jednom bloku prováděly výpočet mnohem déle, než v ostatních blocích, tak by to mohlo mít velmi negativní vliv na celkovou rychlosť renderingu.

Jednotlivá primitiva jsou vytvořená v poli na straně CPU a je nad nimi vytvořena hierarchie obálek, která používá celočíselné indexování v poli, aby mohla být přesunuta na GPU, jako souvislý blok dat.

Pro potřeby GPU implementace bylo potřeba provést anotaci funkcí klíčovým slovem `_device_`, aby mohly být volány z kernelů. Bylo také potřeba nahradit knihovní funkce z CPU knihoven funkcemi z knihoven CUDA. Bylo také potřeba přepsat všechna rekursivní volání funkcí na iterativní.

Při výpočtu v algoritmu sledování cest je potřeba generování velkého množství náhodných čísel. Generování náhodných čísel v CUDA požaduje použití knihovny cuRAND. Náhodná čísla generována počítači jsou ve skutečnosti pseudonáhodná a proto je potřeba vytvoření stavu (`curandState`) pro každé vlákno na GPU. Jednotlivé stavy jsou uložené v poli a je spuštěn kernel, ve kterém si každé vlákno nainicializuje svůj stav. V kernelu `render` si každé vlákno zkopíruje svůj stav z globální paměti do paměti lokální a předá na něj ukazatel do funkce `Trace`, aby mohlo být dále použito pro vzorkování BRDF, ruskou ruletu atd.

### A. Ovládání

Jelikož je renderování scény mnohem rychlejší, než na CPU, tak je možné měnit pozici kamery při samotném výpočtu pomocí kláves W,A,S,D a pohybem myši. Při každém pohybu se resetuje renderování průměrováním jednotlivých vzorků v pixelech a výpočet je prováděn znova. Díky tomuto je možné pozorovat progresivní výpočet a přitom jednoduše měnit pozici kamery.

### B. Potíže při implementaci

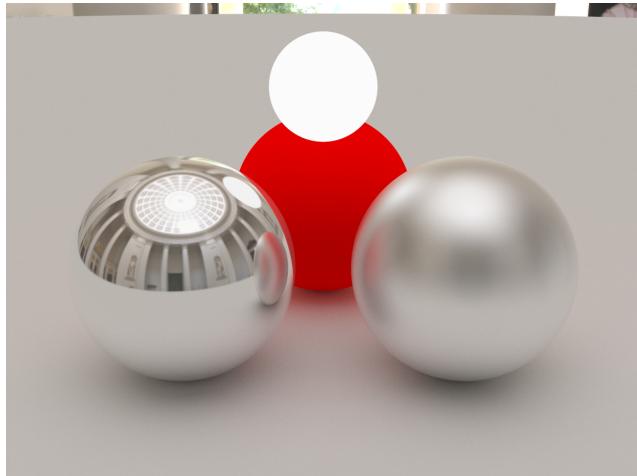
Při implementaci na GPU jsem se potýkal s několika potížemi.

Prvním problémem byl návrh jednotlivých tříd v CPU implementaci, které využívaly dědičnost a abstraktní třídy. Nakonec jsem si práci zjednodušil a spokojil jsem se s jedním druhem primitiv a další druhy primitiv přidám až při rozšiřování aplikace.

Druhým problémem byl debugging jednotlivých kernelů na GPU, k tomu jsem nakonec využil nástroj NVIDIA Nsight, který mi pomohl s odstraněním chyb v implementaci.

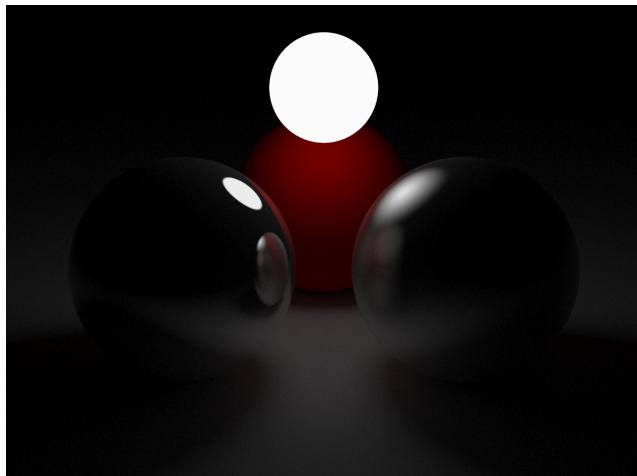
#### IV. VÝSLEDKY A MĚŘENÍ

Výsledky implementace generují pro velký počet vzorků na pohled krásné obrázky bez viditelného šumu, jak můžete vidět na obrázcích jednoduché scény s mapou okolí 1 a s jedním implicitním zdrojem osvětlení 2.



Obrázek 1. Jednoduchá scéna s mapou okolí.

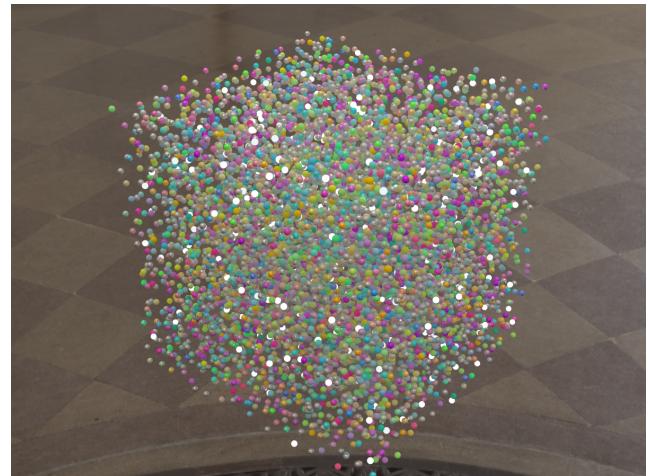
Pro účely měření rychlosti bylo potřeba vytvořit scénu s větším počtem primitiv. Byla proto vytvořena scéna obsahující 8 000 koulí náhodně uspořádaných v objemu krychle jak můžete vidět na obrázcích 3 a 4.



Obrázek 2. Jednoduchá scéna se světelným zdrojem.

Dále byla pro účely měření vytvořena ještě jedna složitější scéna, která obsahuje 64 000 koulí.

Měření bylo tedy provedeno celkově na třech scénách - jednoduchá scéna s 5 primitivy, která plně nevyužívá akcelerační struktury a dvou složitějších scén s parametry popsanými v předchozích odstavcích. Kamera byla vždy nastavena tak, aby snímalala všechna primitiva.



Obrázek 3. Složitá scéna.

První měření bylo provedeno na notebook s následujícími parametry:

- MS Windows 10 Home 64-bit
- Intel Core i7-11370H @ 3.30GHz
- 16.0 GB RAM
- NVIDIA GeForce RTX 3060 Laptop GPU
- Compute Capability 8.6
- CUDA version 11.5

Výsledky prvního měření můžeme vidět v následující tabulce:

Scéna	CPU[s]	GPU[s]
Simple	400.709	3.567
Complex1	823.233	11.715
Complex2	2404.017	41.725

Tabulka I

POROVNÁNÍ PRŮMĚRNÝCH ČASŮ VÝPOČTU PRVNÍHO MĚŘENÍ PRO 1 000 VZORKŮ NA PIXEL PŘI ROZLIŠENÍ 1920 × 1080 NA NOTEBOOKU.

Druhé měření bylo provedeno na stolním počítači s následujícími parametry:

- MS Windows 10 Education 64-bit
- Intel Core i9-10900X @ 3.70GHz
- 32.0 GB RAM
- NVIDIA GeForce GTX 1070 Ti
- Compute Capability 6.1
- CUDA version 11.4

Výsledky druhého měření můžeme vidět v následující tabulce:

Scéna	CPU[s]	GPU[s]
Simple	217.182	2.367
Complex1	428.641	9.524
Complex2	1319.429	33.115

Tabulka II

POROVNÁNÍ PRŮMĚRNÝCH ČASŮ VÝPOČTU PRVNÍHO MĚŘENÍ PRO 1 000 VZORKŮ NA PIXEL PŘI ROZLIŠENÍ 1920 × 1080 NA STOLNÍM PC.

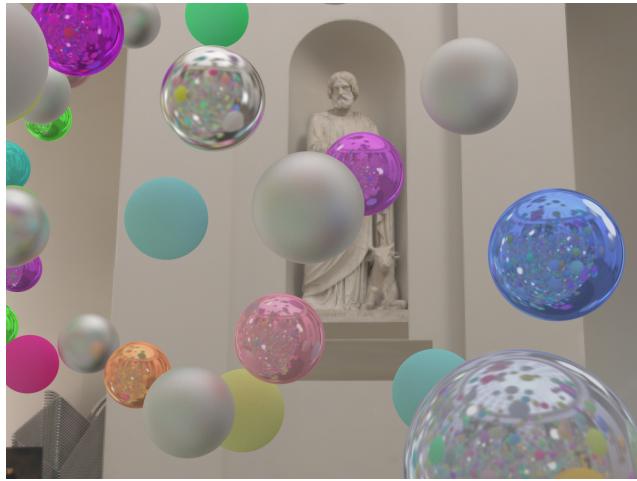
## REFERENCE

- [1] Pharr, Matt and Jakob, Wenzel and Humphreys, Greg - „Physically Based Rendering: From Theory to Implementation 3rd Edition“ (2016), Morgan Kaufmann Publishers Inc.
- [2] Lafortune, Eric & Willem, Yves. (1998). Using the Modied Phong Reflectance Model for Physically Based Rendering.

Z výsledků měření můžeme vidět značné zrychlení GPU implementace oproti implementaci na CPU ve všech scénách.

## V. ZÁVĚR

Cílem tohoto projektu byla implementace metody sledování cest na CPU a na GPU s využitím akcelerační struktury a následné porovnání rychlosti obou implementací. Toto zadání bylo splněno a byla vytvořena aplikace vhodná pro další rozšíření.



Obrázek 4. Detail složitější scény.

Důležitým důsledkem této práce je získání zkušenosti s implementací praktického problému na GPU s využitím knihovny CUDA. Díky tomuto projektu jsem si uvědomil několik omezení využití GPU oproti procesorové implementaci. Další užitečnou zkušeností bylo použití knihovny cuRAND, kterou jsme na cvičeních nepoužívali.

Aplikace může být dále rozšířitelná různými způsoby od větší optimalizace akcelerační struktury, at' už stavou nebo traverzací, rozšířením o jiné druhy primitiv, materiálů, explicitní světelné zdroje a jejich vzorkování atd. Bylo by také možné dosáhnout větší optimalizace již existujícího kódu, nebo další paralelizace jednotlivých částí aplikace.