Závěrečná zpráva - Raytracer s akcelerační strukturou a implementací na GPU

Jakub Votrubec
ČVUT FIT, NI-PG1 a NI-GPU 2024/25

1 Úvod

Tento dokument popisuje závěrečný výstup, v podobě semestrální práce, z předmětu NI-PG1 a NI-GPU. Náplní semestrální práce bylo vytvoření vlastního raytracering renderu, implementace akcelerační datové struktury pro zrychlení traverzace scény a následná implementace na GPU. Cílem bylo prokázat schopnost optimalizovat výkon raytraceru pomocí vhodné datové struktury a implementací na GPU.

Semestrální práce byla zpracována v jazyce C++, standard C++17, a využívá externí knihovny pouze pro načítání scén a konfigurace ve formátu `json`. Pro implementaci na GPU byl použit jazyk CUDA, který umožňuje efektivní paralelní zpracování dat na NVIDIA grafické kartě.

2 Definice problému

Raytracing je technika v počítačové grafice, která simuluje šíření světla pro generování digitálních obrazů s vysokou vizuální věrností. Oproti rychlejším metodám, jako je scanline rendering, je ray tracing výpočetně náročnější, ale umožňuje realistické zobrazení optických jevů. Původně se používal hlavně pro statické obrázky a filmové vizuální efekty, kde nevadila delší doba renderování. Od roku 2018 je však díky hardwarové akceleraci možné využívat ray tracing i v reálném čase, například ve videohrách, často v kombinaci s tradičním rasterizačním vykreslováním. Ray tracing lze využít i pro simulaci zvukových vln, což umožňuje realističtější prostorový zvuk.

Funguje tak, že sleduje cestu paprsku z imaginárního oka skrz každý pixel obrazovky a vypočítává barvu objektu, který je v daném směru viditelný. Ray tracing předpokládá, že paprsek vždy protne zobrazovací rovinu. Po dosažení maximálního počtu odrazů nebo určité vzdálenosti bez průsečíku se výpočet pro daný paprsek ukončí a hodnota pixelu se aktualizuje. Pseudokód raytracingu jsem přiložil zde 1.

Algorithm 1 Sekvenční raytracing

```
O ← objekty ve scéně
L ← světla ve scéně
d_max ← maximální hloubka zanoření
function TRACE_RAY(Paprsek r, Hloubka d)
    P ← nejbližší průsečík r a o ∈ O
    if P neexistuje then
        return barva_pozadi
    end if
    c ← černá (počáteční barva)
    for all l ∈ L do
        if l je vidět z r.origin then
            c_l ← výpočet barevného příspěvku z l a P
            c ← c + c_l
        end if
    end for
    if d < d_max then
        c_r ← TRACE_RAY(odražený paprsek, d + 1)
        c_t ← TRACE_RAY(lomený paprsek, d + 1)
        c ← +c_r + c_t
    end if
    return c
end function
```

3 Struktura raytraceru

Implementoval jsem základní raytracer pro vykreslování scén složených z trojúhelníků. Raytracer nabízí následující funkce, mezi nimiž lze přepínat pomocí konfigurace:

- Stíny pomocí stínových paprsků.
- Podpora plošného světla prostřednictvím náhodného vzorkování trojúhelníků.
- Vykreslení scény s různými osvětlovacími modely (distance shading, čistě difuzní osvětlení, Phongův a Blinn-Phongův osvětlovací model).
- Vykreslení scény s různými stínovacími modely (flat shading, smooth shading – Phongův stínací model).
- Podpora odrazu a lomu světla.
- Backface culling.
- Vykreslení scény s více vzorky na pixel (fuzzysampling), viz *Použité metody nad rámec cvičení* 3.1.

Raytracer je navržen tak, aby byl modulární a snadno rozšiřitelný. Hlavní třída `Raytracer` obsahuje metody pro vykreslování a ukládání výsledného obrazu. Scéna je reprezentována jako seznam trojúhelníků, které jsou načteny z `.obj` souboru.

Raytracer načítá scénu pomocí knihovny `tinyobjloader` a používá knihovnu `nlohmann/json` pro načítání konfigurace. Výsledná scéna je vykreslena do formátu `.ppm`.

3.1 Použité metody nad rámec cvičení NI-PG1

Při renderování scény jsem narazil na problém s velkým množstvím šumu ve výsledném obrazu, proto jsem se rozhodl implementovat vícenásobné vzorkování, které jsem nazval *fuzzysampling*.

Fuzzysampling spočívá v tom, že pro každý pixel se vygeneruje několik paprsků s mírně odlišným směrem. Jejich výsledky se následně průměrují, čímž se dosáhne hladšího obrazu a sníží se šum. Odchylky směru paprsku jsou generovány náhodně v rámci pixelu, což umožňuje zachovat detaily a zároveň eliminovat šum.

V rámci předmětu NI-GPU jsem implementoval raytracer na GPU pomocí jazyka CUDA. GPU verze raytraceru využívá stejnou datovou strukturu jako CPU verze, ale místo sekvenčního zpracování paprsků na CPU je využito paralelní zpracování na GPU. Každý paprsek je zpracován v samostatném vlákně, což umožňuje efektivní využití výpočetního výkonu GPU. Více viz *Implementace na GPU* 5.

4 Akcelerační datová struktura

Implementoval jsem dvě varianty akcelerační datové struktury octree:

- základní varianta octree,
- parametrická varianta podle článku *An Efficient Parametric Algorithm for Octree Traversal* od Revelles et al.

4.1 Stavba a traverzace

Octree je struktura, která dělí prostor na menší buňky, což umožňuje rychlejší vyhledávání kolizí mezi paprsky a trojúhelníky. Každý uzel octree obsahuje bounding box, který definuje prostor, který uzel pokrývá, a seznam trojúhelníků, které se v tomto prostoru nacházejí.

4.1.1 Základní octree

Základní octree je implementován jako rekurzivní struktura, která dělí prostor rekurzivně na osm dalších uzlů (oktetů), dokud není dosaženo maximální hloubky nebo počtu trojúhelníků v uzlu.

Traverzace probíhá tak, že pro každý paprsek se nejprve zjistí, zda protíná bounding box aktuálního uzlu. Pokud ano, poračuje se dál na jeho oktety, pokud ne, traverzace končí. Je-li oktet již list stromu (neobsahuje žádné vlastní oktety), traverzace končí a jeho trojúhelníky jsou přidány k výsledku. Traverzace pokračuje rekurzivně, dokud se neprohledá celý strom. Prochází se vždy všechny oktety uzlu.

4.1.2 Parametrický octree

Parametrický octree je vylepšená verze, která využívá parametrickou traverzaci. Tato metoda umožňuje efektivnější prohledávání prostoru tím, že využívá parametrické rovnice pro určení průsečíků paprsku s hranicemi uzlů octree (tzv. "slab"metoda).

Nejprve se vypočítají průsečíky paprsku s hranicemi uzlu, z nich se určí "čas", kdy paprsek vstoupí a opustí daný uzel na jednotlivých osách. Pokud paprsek opustil uzel na nějaké ose dříve, nežli vstoupil na všech osách - minul bounding box uzlu a tato větev traverzace se uzavírá.

Pokud paprsek vstoupil, tak na základě průsečíku vstupu se vypočítá první oktet, do kterého paprsek vstoupí. Ten se následně přidá do fronty pro další zpracování. Následně se postupně projdou a přidají všechny oktety, do kterých paprsek vstoupí. Procházení všech dalších oktetů probíhá na základě konečného automatu, který určuje, do kterého oktetu paprsek vstoupí na základě jeho směru a aktuálního oktetu.

Optimalizace oproti základnímu octree spočívá v tom, že se neprocházejí všechny oktety, ale pouze ty, do kterých paprsek skutečně vstoupí. Zároveň výpočet velikostí oktetů je založen jen na půlení velikosti rodičovského uzlu, což umožňuje efektivní a výpočetně nenáročné dělení prostoru.

Parametrickou traverzaci se mi bohužel nepodařilo implementovat, více viz *Problémy při implementaci*.

4.2 Problémy při implementaci

Při implementaci stavby octree jsem narazil na problémy s přesností výpočtů velikosti oktetů. Kvůli chybě u datových typů s plovoucí desteinou čárkou se ne vždy přiřadily všechny trojúhelníky rodičovského uzlu do jeho oktetů, což vedlo k chybám při traverzaci. Tento problém jsem vyřešil nafouknutím bounding boxů o toleranci *epsilon*.

Bohužel se mi nepodařilo správně implementovat a zprovoznit parametrický octree. Narazil na problémy při traverzaci. Traverzace nenalezne všechny trojúhelníky a scéna je poté vyrenderována s dírami, jak švýcarský sýr. Zkoušel jsem, jestli je problém opět v přesnosti výpočtů s čísly s plovoucí desetinnou čárkou, ale nepodařilo se mi ho vyřešit. Proto jsem se rozhodl parametrický octree nevyužít a v měření výkonu použít pouze základní octree.

5 Implementace na GPU

Implementace na GPU byla provedena pomocí jazyka CUDA. Hlavní myšlenkou bylo přesunout výpočet kolizí paprsků s trojúhelníky na GPU, což umožňuje paralelní zpracování velkého množství paprsků najednou.

GPU verze raytraceru využívá stejnou strukturu dat jako CPU verze, ale místo sekvenčního zpracování paprsků na CPU se využívá paralelní zpracování na GPU. Každý paprsek je zpracován v samostatném vlákně, což umožňuje efektivní využití výpočetního výkonu GPU.

5.1 Implementace CPU vs GPU

Při implementaci na GPU bylo nutné provést několik úprav oproti CPU verzi. Bylo potřeba odstranit závislost na standardní knihovně C++, alokovat paměť dostupnou pro GPU a přesunout data i nastavení rendereru do této alokované paměti. Struktura rendereru musela být změněna, protože funkce označené jako `__global__` nesmějí být členskými funkcemi tříd. Proto byly všechny GPU funkce přesunuty mimo třídu `Renderer` a funkce používané jak při CPU, tak GPU renderování byly zpřístupněny oběma přístupům. Dále bylo nutné některé funkce upravit tak, aby byly použitelné jak na CPU, tak na GPU, což bylo řešeno pomocí podmíněného překladu s direktivou `#ifdef __CUDA_ARCH__`.

Na GPU není implentována akcelerační struktura octree, ale pouze základní traverzace trojúhelníků. To znamená, že na GPU se nevyužívá žádná akcelerační datová struktura pro zrychlení traverzace scény. Všechny trojúhelníky jsou uloženy v jednom poli a pro každý paprsek se provádí sekvenční traverzace všech trojúhelníků.

Tato implementace je sice méně efektivní než CPU verze s octree, ale umožňuje využít paralelní zpracování na GPU a dosáhnout tak vyššího výkonu při renderování scény.

Dále v GPU verzi není dostupná funkcionalita *fuzzysampling*.

Implementaci na GPU lze nalést ve větvi `NI-GPU` v tomto repozitáři.

5.2 Problémy při implementaci na GPU

Při implementaci na GPU jsem narazil na problémy s tím, že funkce označené jako `__global__` nesmějí být členskými funkcemi tříd. To znamenalo, že jsem musel přesunout všechny GPU funkce mimo třídu `Renderer` a přepsat je tak, aby byly použitelné jak na CPU, tak na GPU.

Dále jsem se potýkal s nečekaným problémem, co se týče generování náhodných čísel na GPU. Funkce `rand()` není dostupná v CUDA, a tak jsem musel implementovat vlastní generátor náhodných čísel. Použil jsem jednoduchý lineární kongruentní generátor, který je dostatečný pro účely raytraceru.

6 Testování a měření výkonu

6.1 Nastavení měření

Pro měření výkonu byla použita scéna `CornellBox-Sphere.obj` s následující konfigurací:

Tabulka 1: Použité nastavení rendereru

Parametr	Hodnota
Výsledné rozlišení	800×800
Max. zanoření paprsku	10
Vzorků plošného světla na trojúhelník	50
Osvětlovací model	Blinn-Phong
Stínovací model	Smooth
Backface culling	Ano
Fuzzysampling	Ne

Tabulka 2: Použité nastavení struktury octree

Parametr	Hodnota
Max. počet trojúhelníků na uzel	16
Max. hloubka uzlů	10

6.2 Výsledky měření

Měření jsem prováděl na svém stroji *Lenovo Legion* s následujícími parametry:

- Procesor: AMD Ryzen 5 5600H; 3,3 GHz
- Grafická karta: NVIDIA GeForce RTX 3060 Laptop GPU; 6 GiB VRAM
- RAM: 16 GB
- Block size 512 (počet vláken v blocku)

Na závěr jsem také měření provádel na serveru *cluster.fit.cvut.cz*, který má následující parametry:

- Grafická karta: NVIDIA GeForce RTX 4070 Ti; 12 GiB VRAM
- Block size 256 (počet vláken v blocku)

Čas kopírování paměti na GPU je čas potřebný pro kopírování dat z hostitelské paměti (CPU) do sdílené paměti dostupné na zařízení (GPU). Tento čas je důležitý pro měření výkonu, protože ovlivňuje celkový čas renderování.

Celkový čas renderování je celkový čas potřebný pro vykreslení scény, včetně času kopírování paměti na GPU a času výpočtu kolizí.

Tabulka 3: Porovnání výkonu raytraceru

Metrika	Bez ADS	Octree	GPU Legion	GPU cluster
Čas kopírování paměti na GPU [s]	-	-	0.36s	1.91s
Celkový čas renderování [s]	3h 24m 12.4s	3m 33.9s	2m 44.3s	7.37s
Počet kolizí paprsek-trojúhelník	2.46×10^{11}	2.81×10^{9}	-	-
Průměrný počet kolizí na paprsek	383 665	4 395	-	-
Čas výpočtu kolizí [s]	8 032.8	121.5	-	-
Průměrná doba výpočtu kolizí na paprsek [ms]	12.55	0.19	-	-

Tabulka 4: Statistiky struktury octree

Statistika	Hodnota
Počet uzlů	1 584
Počet listů	1 300
Průměrná hloubka listů	5.22
Max. trojúhelníků v listu	34
Průměr trojúhelníků v listu	7.03
Čas stavby octree [s]	0.04
Počet prohledaných uzlů	1 138 825 603
Čas strávený traverzací	4m 43.943s
Průměrný počet trojúhelníků vrácených z traverzace	33.7

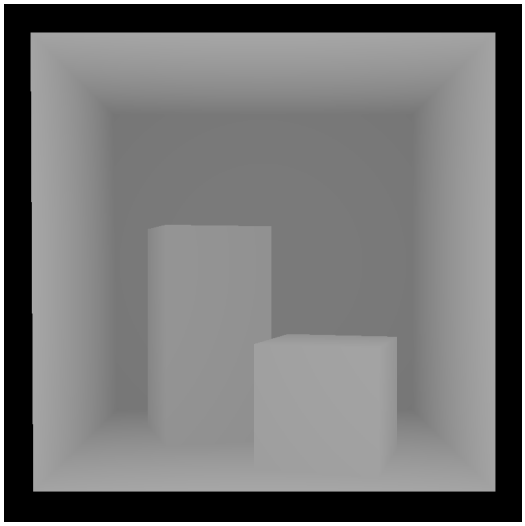
6.3 Zátěžový test

Pro zátěžový test jsem použil scénu `CornellBox-Sphere.obj` se stejnou konfigurací, jako při předchozím měření, až na rozlišení a tudíž počet počítaných paprsků.

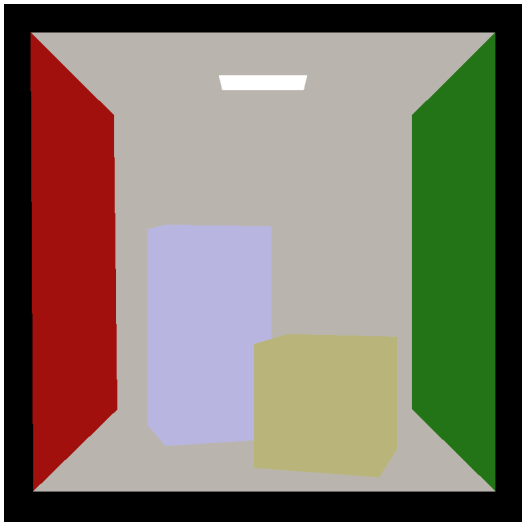
Tabulka 5: Zátěžový tet

GPU	Rozlišení	Čas kopírování paměti na GPU [s]	Čas renderování [s]
cluster.fit.cvut.cz	2000x2000	1.41s	31.52s
cluster.fit.cvut.cz	10000x10000	5.54s	10m 45.97s

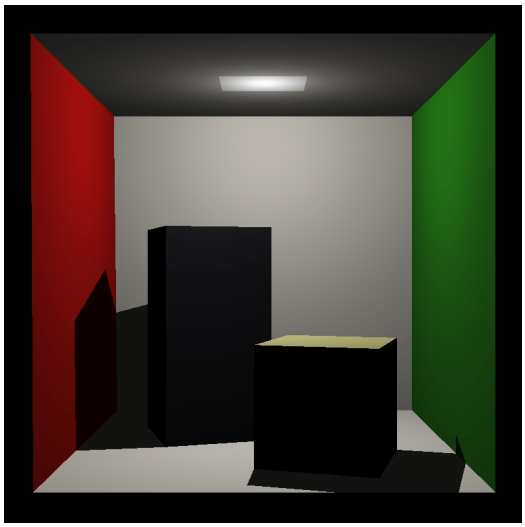
7 Ukázky výstupů



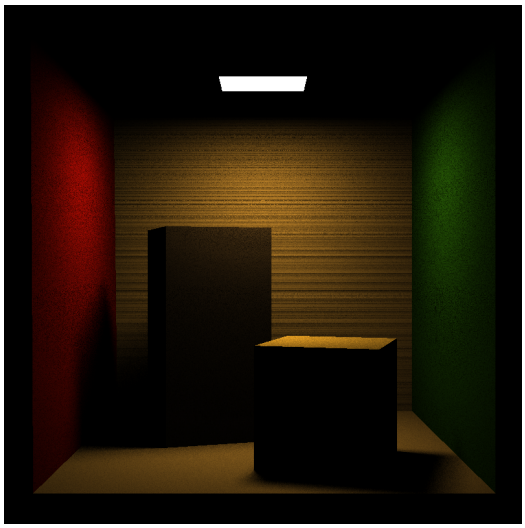
Obrázek 1: Ukázka vykreslení scény s distance shadingem.



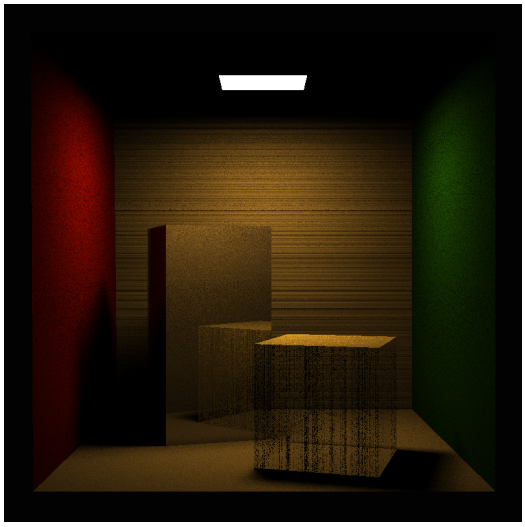
Obrázek 2: Ukázka vykreslení scény s difúzním osvětlením.



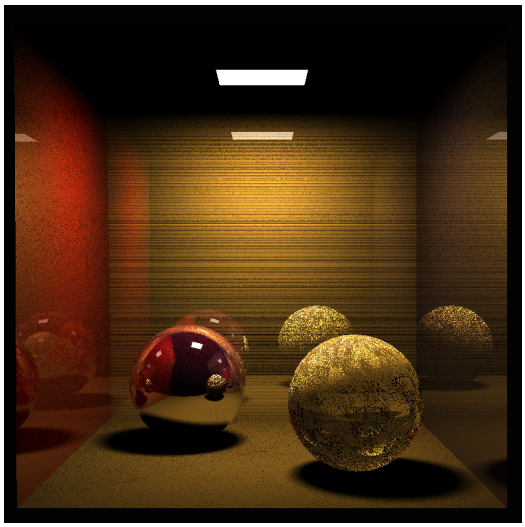
Obrázek 3: Ukázka vykreslení scény s Blinn-Phong osvětlením a stíny.



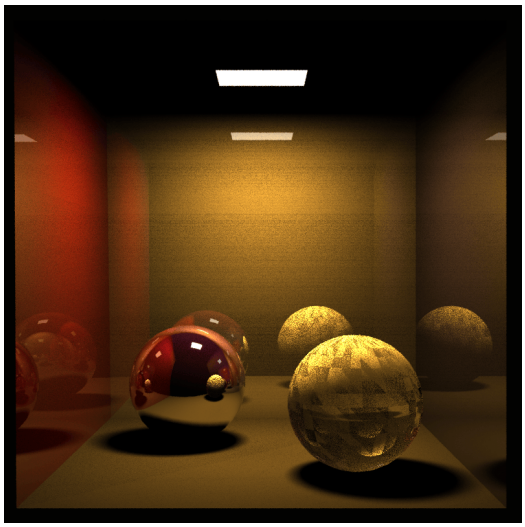
Obrázek 4: Ukázka vykreslení scény s plošným světlem.



Obrázek 5: Ukázka vykreslení scény s odrazem a lomem světla.



Obrázek 6: Ukázka vykreslení scény s odrazem a lomem světla na zrcadlových stěnách.



Obrázek 7: Ukázka vykreslení scény s fuzzysamplingem (10 vzorků na pixel).

8 Implementace

Veškerá implementace je dostupná na fakultním GitLabu a mém GitHubu.

Více informací o kompilaci, konfiguraci a spuštění naleznete v souboru README samotného repozitáře.

9 Závěr

Tato semestrální práce prokázala schopnost optimalizovat výkon raytraceru pomocí akcelerační datové struktury octree a implementace na GPU. Implementace základního octree vedla k významnému zrychlení traverzace scény a snížení počtu kolizí mezi paprsky a trojúhelníky. Parametrickou variantu octree se mi bohužel nepodařilo implementovat, což je škoda, protože by mohla přinést další zlepšení výkonu. implementace na GPU umožnila využít paralelní zpracování a výrazně zrychlit renderování scény. Výsledky měření ukázaly, že GPU verze raytraceru je schopna vykreslit scénu mnohem rychleji než CPU verze s octree.

Další rozšíření raytraceru by mohlo zahrnovat implementaci dalších akceleračních struktur, jako jsou BVH (Bounding Volume Hierarchy) nebo kd-stromy, které by mohly dále zlepšit výkon při renderování složitějších scén.

10 Použitá literatura

- Přednášky a materiály z předmětu NI-PG1.
- Přednášky a materiály z předmětu NI-GPU.
- *An Efficient Parametric Algorithm for Octree Traversal* - Revelles et al.
- `tinyobjloader` - Knihovna pro načítání `.obj` souborů.
- `nlohmann/json` - Knihovna pro práci s JSON v C++.
- Dokumentace k C++ standardu a knihovnám.
- Dokumentace ke CUDA knihovně.