Fakultet tehničkih nauka

Univerzitet u Novom Sadu

Računarski sistemi visokih performansi

Ubrzavanje algoritma za praćenje zraka u računarskoj grafici

Nikola Kušlaković E2 121/2025

30.01.2026.

Sažetak

Ovaj rad se bavi implementacijom i optimizacijom algoritma praćenja putanje zraka (path tracing) na različitim platformama. Razvijene su tri implementacije: sekvencijalna na procesoru, paralelna na procesoru koristeći OpenMP, i paralelna na grafičkoj kartici koristeći CUDA. Glavni fokus je bio na dizajnu fleksibilnog, prenosivog rešenja gde isti kod može da se prevede i izvršava na CPU i GPU hardveru bez žrtvovanja performansi. Testiranja pokazuju da je GPU implementacija najbrža (4 ms za 10 uzoraka), praćena OpenMP implementacijom (657 ms za iste parametre), dok je sekvencijalna implementacija najsporija (4.79 s). Rezultati demonstriraju efikasnost paralelizacije za probleme kao što je praćenje zraka.

Sadrzaj

Uvod

Opis problema

Praćenje zraka (eng. *ray tracing*) je tehnika renderovanja u računarskoj grafici koja simulira fizičko ponašanje svetlosti. Umesto da projektuje poligone na ekran (kao kod rasterizacije), ovaj algoritam prati putanju pojedinačnih zraka svetlosti od kamere (oka posmatrača) kroz svaki piksel na ekranu, simulirajući njihovu interakciju sa objektima u sceni (refleksiju, refrakciju, senke itd.). U literaturi termin *ray tracing* se često koristi za bilo koju tehniku koja simulira fizičko ponašanje svetlosti, a ne samo onu koja prati putanju zraka.

U istoriji razvoja ove tehnike, ključan je Whitted-style ray tracing [1], koji je prvi omogućio realistične refleksije i prelamanja svetlosti. Međutim, on je bio deterministički i nije mogao verno da prikaže meke senke, indirektno osvetljenje ili globalno osvetljenje.

U ovom radu, fokus je bio na metodi praćenja putanje zraka (eng. *path tracing*), modernijoj varijanti koja potpada pod Monte Carlo algoritme za renderovanje. Za razliku od Whitted-ovog modela, praćenje putanje zraka koristi nasumično uzorkovanje mnogo zraka kako bi rešio jednačinu renderovanja. On prati putanje zraka koji se nasumično odbijaju od površina, čime se postiže fotorealizam i prirodno globalno osvetljenje (eng. *global illumination*).

Algoritam praćenja zraka se danas najčešće koristi u:

- Filmskoj industriji i animaciji - Za postizanje vrhunskog fotorealizma gde vreme renderovanja nije kritično (eng. *offline rendering*).
- Video igrama - Iako su grafičke kartice danas jako performantne, ray tracing je još uvijek relativno spor, pa je zbog toga danas korišćena mnoštvo optimizacija kako bi se postigla optimalna performansa. Najčešće se koriste hibridne metode renderovanja koje kombiniraju ray tracing i rasterizaciju kako bi se postigla optimalna performansa.
- Arhitektonskoj vizuelizaciji i 3D modelovanju - Za precizan prikaz materijala i svetlosti u prostoru, kao i za modeliranje i vizuelizaciju prostora.

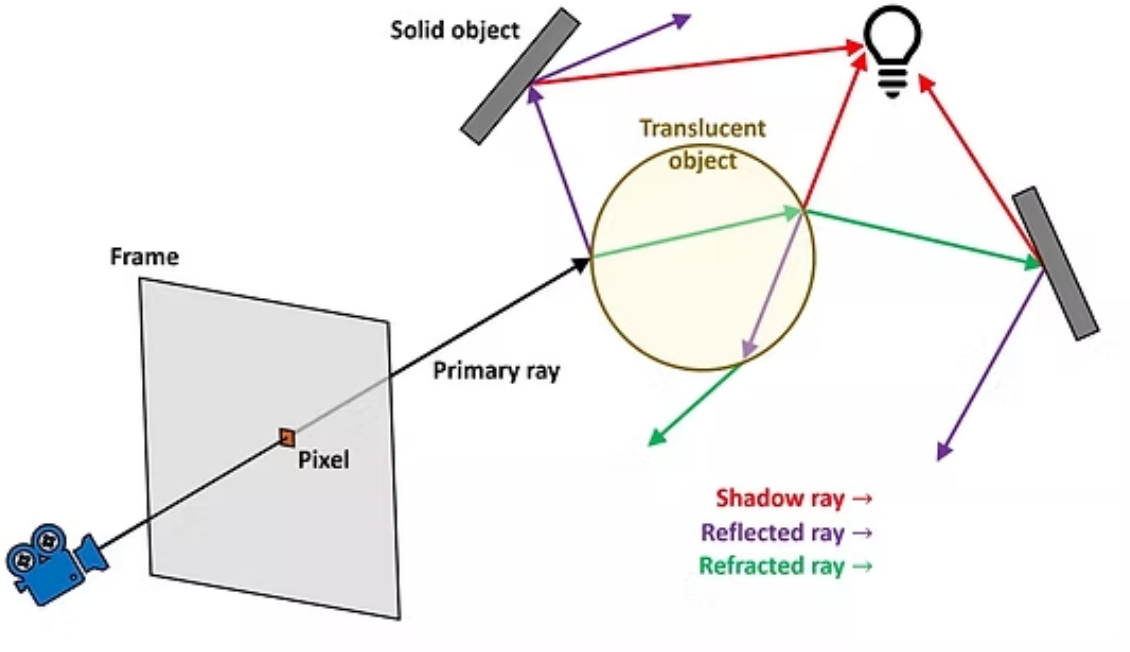
Glavni razlog zašto je praćenje putanje zraka sporo je ogromna količina računanja. Da bi se dobila čista slika bez šuma, potrebno je ispaliti stotine ili hiljade zraka po svakom pikselu. Svaki zrak mora da proveri koliziju sa hiljadama (ili milionima) poligona u sceni. Čak i uz optimizacije poput bolje pretrage prostora koristeći hijerarhiju obuhvatnih zapremina (eng. *Bounding Volume Hierarchy*) [2], proces zahteva masivnu procesorsku snagu, zbog čega je idealan kandidat za ubrzanje. Ovaj problem se može svrstati i u smešno jednostavne probleme [3] za ubrzanje, jer je praćenje jednog zraka nezavisno od ostalih zrakova.

Cilj

Cilj ovog rada je implementacija algoritma praćenja putanje zraka na procesoru i grafičkoj kartici koristeći CUDA i OpenMP radne okvire, kao i poređenje performansi između implementacija. Osim toga, fokus je bio i na dizajnu softverskog rešenja gde je u obzir uzeta prenosivost kako bi se postigla maksimalna fleksibilnost i olakšana podrška za budući razvoj.

Teorijske osnove

Na slici 1 je prikazan algoritam praćenja putanje zraka. Za svaki piksel na ekranu se ispaljuje najmanje jedan zrak, ali najčešće više zraka. Svaki zrak se kreće kroz prostor sve dok ne naiđe na neku površinu ili ne izađe iz scene. Ako zrak naiđe na neku površinu, računa se kolizija zraka sa površinom i određuje se novi zrak koji se širi u prostoru. Ovaj proces se ponavlja dok zrak ne izađe iz scene, dok ne dostigne maksimalan broj odbijanja ili ne bude absorbovan. Objekti u sceni imaju različite materijale koji imaju različite osobine, kao što su refleksija, prelamanje, apsorpcija, itd. Ukoliko zrak pogodi objekat koji je providan, dolazi do prelamanja zraka usled čega mogu da nastanu dva zraka: jedan koji se širi u prostoru i jedan koji se reflektuje. Različite implementacije ovog algoritma različito definišu šta sve može da se desi sa zrakom nakon što je pogodio objekat i to ponašanje se najčešće definiše materijalom samog objekta.



Slika 1: Algoritam praćenja putanje zraka za jedan piksel

Zrak

Zrak možemo da definišemo matematički kao parametarsku funkciju:

$$\mathbf{p}(t) = \mathbf{p_0} + t\mathbf{d} \quad (1)$$

gde je $\mathbf{p_0}$ početna tačka zraka, $\mathbf{d}$ je smer zraka (vektor), a $t \in [0, \infty)$ je parametar koji određuje poziciju zraka na putanji. Menjanjem t se dobija bilo koja tačka na putanji zraka.

Presek zraka i sfere

Sfera je jedan od najjednostavnijih geometrijskih tela, pa je zbog toga sfera bila korišćena kao test objekat. Njena jednačina je:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$$

gde je (x_0, y_0, z_0) centar sfere, a r je poluprečnik sfere. Ako imamo neku proizvoljnu tačku (x, y, z) onda može da bude u sledećem odnosu sa sferom:

- $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 < r^2$ - tačka je unutar sfere
- $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$ - tačka je na površini sfere
- $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 > r^2$ - tačka je van sfere

Pretpostavimo sada da je tačka (x, y, z) presek zraka i sfere. Tada možemo da napišemo jednačinu zraka kao:

$$\mathbf{p}(t) = (x, y, z) = (p_x, p_y, p_z) + t(d_x, d_y, d_z)$$

odnosno jednačinu sfere (1) možemo da napišemo kao skalarni proizvod:

$$(x - x_0, y - y_0, z - z_0) \cdot (x - x_0, y - y_0, z - z_0) = r^2$$

tj.

$$(\mathbf{p}(t) - \mathbf{c_0}) \cdot (\mathbf{p}(t) - \mathbf{c_0}) = r^2$$

gde je $\mathbf{c_0}$ centar sfere. Korišćenjem jednačine zraka (1) i zamenom $\mathbf{p}(t)$ dobija se:

$$(\mathbf{p_0} + t\mathbf{d} - \mathbf{c_0}) \cdot (\mathbf{p_0} + t\mathbf{d} - \mathbf{c_0}) = r^2$$

$$(t\mathbf{d} + (\mathbf{p_0} - \mathbf{c_0})) \cdot (t\mathbf{d} + (\mathbf{p_0} - \mathbf{c_0})) = r^2$$

Daljim izvođenjem dobija se kvadratna jednačina:

$$t^2\mathbf{d} \cdot \mathbf{d} + 2t\mathbf{d} \cdot (\mathbf{p_0} - \mathbf{c_0}) + (\mathbf{p_0} - \mathbf{c_0}) \cdot (\mathbf{p_0} - \mathbf{c_0}) - r^2 = 0 \quad (2)$$

U ovoj jednačini jedino je nepoznat parametar t, čijim rešavanjem se dobija tačka preseka. Radi lakšeg računanja, može da se uvede zamena:

$$a = \mathbf{d} \cdot \mathbf{d}$$

$$b = 2\mathbf{d} \cdot (\mathbf{p_0} - \mathbf{c_0})$$

$$c = (\mathbf{p_0} - \mathbf{c_0}) \cdot (\mathbf{p_0} - \mathbf{c_0}) - r^2$$

Tada jednačina (2) postaje:

$$at^2 + bt + c = 0$$

i njena dva rešenja su:

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

Odavde zaključujemo da zrak i sfera mogu da budu u sledećim odnosima:

- $b^2 - 4ac < 0$ - zrak i sfera nemaju presek
- $b^2 - 4ac = 0$ - zrak i sfera se dodiruju u jednoj tački
- $b^2 - 4ac > 0$ - zrak i sfera se seku u dve tačke

Ova izvedena formula i definicija parametara a, b i c su korišćene u implementaciji zbog lakšeg računanja i bolje preglednosti koda.

Materijali

Materijali su ključne komponente u sceni koje određuju ponašanje svetlosti na površini objekta. Oni određuju koliko će svetlost biti reflektovana, prelamana, apsorbovana ili transmitovana. Materijali su definisani kao funkcije koje primaju zrak i vraćaju novi zrak i intenzitet svetlosti. U ovom projektu su implementirana dva materijala:

- Lambertov materijal (idealni mat materijal) [4] - Rasipa (difuzno reflektuje) svetlost podjednako u svim pravcima, bez obzira na ugao posmatranja.
- Metal - reflektuje ulazni zrak pod određenim uglom i intenzitetom. Definisan i parametar hrapavosti (*fuzz*) koji određuje koliko će se izlazni zrak razlikovati od ulaznog. Što je hrapavost veća, to su refleksije “mutnije”.

Implementacija

Korišćen softver i hardver

Hardver koji je korišćen za testiranje je:

- CPU: AMD Ryzen 7 7700, 8 jezgara, 16 niti
- GPU: NVIDIA GeForce RTX 3070, 8 GB VRAM
- RAM: 32 GB
- OS: Windows 11, pokretano u WSL 2.6.3 (Ubuntu 22.04.05 LTS)

Softver i alati korišćeni za razvoj su:

- CUDA 11.5
- OpenMP 4.5
- GCC 11.4.0
- CMake 3.22.1
- GLM 0.9.9.8

GLM je biblioteka za matematiku koja je korišćena za vektorske i matrične operacije. Često je korišćena u programiranju grafike i fizike, jer implementira tipove kao što su `vec3`, `vec4`, `mat4` itd. [5]

Struktura projekta

Projekat je organizovan u tri glavna modula:

- **CORE** modul - Osnovni paket koji sadrži sve ključne komponente algoritma za praćenje zraka (kamera, zraci, sfere, materijali, svet itd.). Glavni izazov implementacije bio je dizajnirati ovaj paket kao višeplatformski, tako da isti kod može da se prevodi i izvršava i na CPU i na GPU hardveru.
- **CPU** modul - CPU implementacija koja koristi OpenMP za paralelizaciju ili sekvencijalno izvršava. Ovaj modul integriše `CORE` paket i implementira glavni program za CPU izvršavanje.
- **CUDA** modul - Paralelna GPU implementacija koristeći CUDA. Ovaj modul takođe koristi CORE modul, ali ga prevodi kao CUDA kod i izvršava na grafičkoj kartici.

Dizajn CORE modula

Pošto je bilo potrebno napraviti CORE modul višeplatformskim, bilo je potrebno pisati metode i klase na pametan način tako da one mogu da se prevedu i izvršavaju i na procesoru i na grafičkoj kartici. Bitna ograničenja su bila seledeća:

1. Izbegavanje korišćenja rekurzije zbog relativno plitkog steka koji je dostupan nitima na grafičkoj kartici
2. Izbegavanje korišćenja polimorfizma i virtualnih metoda jer ovo najčešće nije dovoljno dobro podržano i optimizovano da se izvršava na grafičkoj kartici
3. Izbegavanje korišćenja `std::vector`, `std::map` i `std::shared_ptr` sličnih kolekcija iz standardne C++ biblioteke pošto ih standardna biblioteka CUDA ne implementira
4. Iako CUDA zvanično podržava generisanje pseudo nasumičnih brojeva kroz svoju `cuRAND` biblioteku, odlučeno je da se implementira sopstveni generator nasumičnih brojeva koji bi bio kompatibilan i sa CPU i sa GPU izvršavanjem.

Ograničenje 1 je prevaziđeno prevođenjem rekurzije u iterativno izvršavanje, dok je ograničenje 2 rešeno korišćenjem ugrađenog tipa `union`. Što se tiče ograničenja 3, korišćeni su čisti pokazivači na objekte i

njihovi tipovi su bili fiksni i poznati u vremenu prevođenja.

Generator nasumičnih brojeva je implementiran kroz nekriptografsku hash funkciju koja je brza i jednostavna za implementaciju [6]. Ova metoda koristi stanje generatora koje se sastoji od četiri 32-bitna cela broja. Svaki put kada se zatraži novi nasumični broj, stanje se ažurira korišćenjem i vraća se novi nasumični broj. Generator je primao četiri parametra kako bi generisao nasumičan broj u intervalu $[0, 1)$. Ova četiri parametra su bila koordinata piksela (i, j), broj uzorka (*sample*) i trenutni broj odbijanja zraka (*bounce*). Na ovaj način, svaki piksel je imao svoj jedinstveni niz nasumičnih brojeva koji su bili nezavisni od ostalih piksela.

GLM biblioteka je kompatibilna sa CUDA radnim okvirom, tako da je moguće prevesti je korišćenjem NVCC prevodioca (NVIDIA CUDA Compiler). Kompatibilnost sa CUDA okvirom urađen je kroz sledeću `hpp` datoteku prikazanom na isečku koda 1. Svaka klasa iz CORE modula koja je htela da koristi GLM biblioteku je morala da uključi prvo ovu datoteku. Osim toga, ključne reči `__host__`, `__device__`, `__global__` i `__forceinline__` su definisane kako bi se omogućila kompatibilnost sa CUDA okvirom. Ukoliko se koristi neki prevodilac koji nije NVCC, ovi makroi se svode na prazne makroe tako da ne utiču na kod.

```
// cuda_compat.hpp
#pragma once

// CUDA support: include cuda.h first to define CUDA_VERSION for GLM
#ifdef __CUDACC__
#include <cuda.h>
#include <cuda_runtime.h>

// Define GLM settings for CUDA compatibility
#ifndef GLM_FORCE_CUDA
#define GLM_FORCE_CUDA
#endif
#ifndef GLM_FORCE_PURE
#define GLM_FORCE_PURE
#endif

#else
// Non-CUDA builds: provide compatible macros
#define __device__
#define __host__
#define __global__
#define __forceinline__ inline
#endif

#include <glm/glm.hpp>
```

Isečak koda 1: GLM CUDA kompatibilnost

Na isečku koda 2 prikazana je metoda koja se bavi renderovanjem slike i pripada klasi `Camera` iz CORE modula. Ovo je metoda sadrži tri ugnježdene petlje koje prolaze kroz svaki piksel slike i ispaljuju više zraka po pikselu (za broj uzorkovanje). Svaki zrak se zatim prati kroz scenu pozivom metode `traceRay`. Ova metoda je paralelizovana korišćenjem OpenMP okvira, ali samo kada se koristi CPU implementacija. Ovo je postignuto korišćenjem `#ifdef _OPENMP` direktive koja proverava da li je OpenMP podržan. Ukoliko nije, kod se prevodi bez OpenMP direktiva.

```
__host__
void Camera::render(const World& world, color* pixel_buffer) const {
#ifdef _OPENMP
    #pragma omp parallel for collapse(2)
#endif
    for (int j = 0; j < image_height; j++) {
        for (int i = 0; i < image_width; i++) {
            color pixel_color(0.0f, 0.0f, 0.0f);

            for (int s = 0; s < msaa_samples; s++) {
                Ray ray = getRay(i, j, s);
                RNGState state(i, j, s, 0);

                pixel_color += traceRay(ray, &world, state);
            }

            pixel_buffer[j * image_width + i] = pixel_color / float(msaa_samples);
        }
    }
}
```

Isečak koda 2: Glavna metoda za renderovanje slike u klasi Camera

Da bi se omogućilo izvršavanje na grafičkoj kartici, ova metoda je prepisana kao CUDA kernel i prikazana je na isečku koda 3. Ovaj kernel koristi blokove i niti za paralelno izvršavanje na grafičkoj kartici. Svaka nit obrađuje jedan piksel slike, a unutar svake niti se ispaljuju više zraka po pikselu. Rezultat se zatim upisuje u `pixel_buffer`. Sve ostale metode koje kernel poziva (`getRay` i `traceRay`) nije bilo potrebno ponovo prepisivati već su te metode bile označene sa ključnom rečju `__device__` čime su mogle da budu prevedene NVCC prevodiocem. Ova metoda je takođe bila zagrađena makorom, tako da nije ulazila u prevođenje ukoliko se modul ne prevodi sa NVCC prevodiocem.

```cpp
#ifdef __CUDACC__
__global__ void renderKernel(Camera* camera, World* world, color* pixel_buffer,
  int image_width, int image_height, int msaa_samples) {

  int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= image_width || j >= image_height)
        return;

    int pixel_index = j * image_width + i;
    color pixel_color(0.0f);

    for (int sample = 0; sample < msaa_samples; sample++) {
        Ray ray = camera->getRay(i, j, sample);
        RNGState state(i, j, sample, 0);
        pixel_color += camera->traceRay(ray, world, state);
    }

    pixel_color /= static_cast<float>(msaa_samples);

    pixel_buffer[pixel_index] = pixel_color;
}
#endif
```

Isečak koda 3: CUDA kernel za renderovanje slike u klasi Camera

Proces prevođenja i izvršavanja

Ceo tok prevođenja bio je ostvaren uz pomoć alata CMake. Svaki modul je imao svoju `CMakeLists.txt` datoteku koja je definisala sve potrebne informacije za prevođenje. Na primer, CUDA modul je imao `CMakeLists.txt` datoteku koja je definisala sve potrebne konfiguracije za NVCC prevodilac. U korenu projekta nalazio se glavna `CMakeLists.txt` datoteka koja je pozivala sve pod datoteke. Ovo je omogućilo fleksibilnost u prevođenju i izvršavanju projekta. Ukoliko sistem ne poseduje podršku za OpenMP ili CUDA okvir, projekat se prevodi bez njih. Proizvod procesa prevođenja su tri izvršne datoteke:

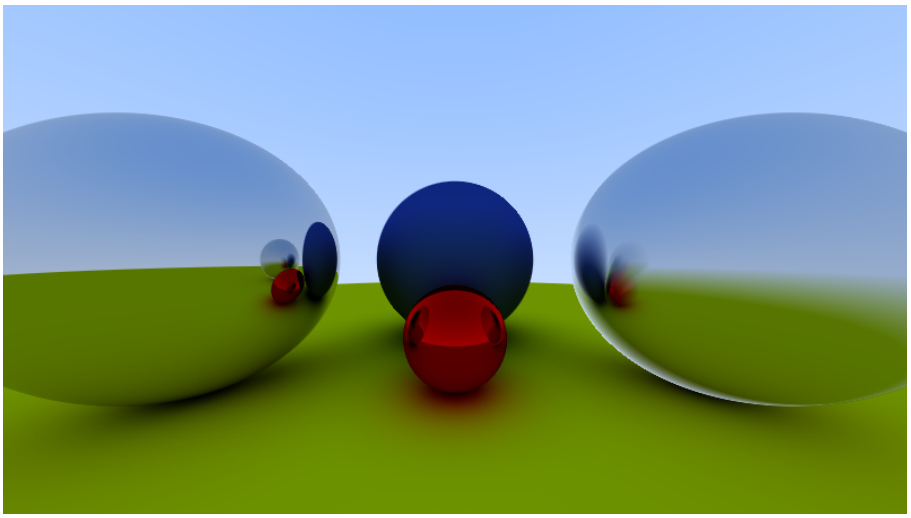
- `raytracer-seq` - Sekvencijalna izvršna datoteka za CPU izvršavanje
- `raytracer-omp` - Paralelna izvršna datoteka za CPU izvršavanje koristeći OpenMP
- `raytracer-cuda` - Paralelna izvršna datoteka za GPU izvršavanje koristeći CUDA radni okvir

Rezultati i diskusija

Ispitano je kako različiti parametri izvršavanja utiču na kvalitet dobijene slike i na vreme izvršavanja programa. Parametri koji su ispitani su:

- Rezolucija slike
- Broj zraka po pikselu (parametar `samples`)
- Maksimalan broj odbijanja zraka (parametar `max_bounce`)

Scena kojom su sprovođeni testovi je prikazana na slici 2. Sastoji se od pet sfere. Podloga na kojoj se nalaze četiri vidljive sfere je zapravo takođe sfera (zeleno) sa velikim poluprečnikom. Leva i crvena sfera u sredini su sačinjene od metalnog materijala bez hrapavosti, dok je sfera desno ima nešto veći parametar hrapavosti. Sfera u centru je sačinjena od Lambertovog materijala (plavo). Vidimo refleksiju na svim sferama koje su sačinjene od metalnog materijala. Takođe, sve sfere bacaju senku na podlogu.



Slika 2: Scena za testiranje

U tabelama 1 do 6 su prikazani rezultati izvršavanja programa za određenu kombinaciju parametara. Rezolucija je uvek bila fiksna i iznosi 800x450 piksela, dok su ostali parametri varirani. Kod GPU implementacije, veličina bloka je bila 16x16 niti.

CPU (sekvencijalno)

Tabela 1: Uticaj broja zraka po pikselu na vreme izvršavanja na procesoru (sekvencijalno)

Rezolucija slike	Broj zraka po pikselu	Max broj odbijanja zraka	Vreme izvršavanja
800x450	10	16	4.79 s
800x450	100	16	48.71 s
800x450	1000	16	—*
800x450	10000	16	—*

*Nije mereno zbog prevelikog vremena izvršavanja

Tabela 2: Uticaj maksimalnog broja odbijanja zraka na vreme izvršavanja na procesoru (sekvencijalno)

Rezolucija slike	Broj zraka po pikselu	Max broj odbijanja zraka	Vreme izvršavanja
800x450	100	4	44.67 s
800x450	100	8	47.97 s
800x450	100	16	48.71 s
800x450	100	32	49.60 s
800x450	100	64	48.89 s

CPU (OpenMP, 16 niti)

Tabela 3: Uticaj broja zraka po pikselu na vreme izvršavanja na procesoru (OpenMP, 16 niti)

Rezolucija slike	Broj zraka po pikselu	Max broj odbijanja zraka	Vreme izvršavanja
800x450	10	16	657 ms
800x450	100	16	6.84 s
800x450	1000	16	68.54 s
800x450	10000	16	683.98 s

Tabela 4: Uticaj maksimalnog broja odbijanja zraka na vreme izvršavanja na procesoru (OpenMP, 16 niti)

Rezolucija slike	Broj zraka po pikselu	Max broj odbijanja zraka	Vreme izvršavanja
800x450	100	4	5.86 s
800x450	100	8	6.97 s
800x450	100	16	6.84 s
800x450	100	32	7.18 s
800x450	100	64	6.94 s

GPU (CUDA)

Tabela 5: Uticaj broja zraka po pikselu na vreme izvršavanja na grafičkoj kartici (CUDA)

Rezolucija slike	Broj zraka po pikselu	Max broj odbijanja zraka	Vreme izvršavanja
800x450	10	16	4 ms
800x450	100	16	46 ms
800x450	1000	16	390 ms
800x450	10000	16	3.94 s

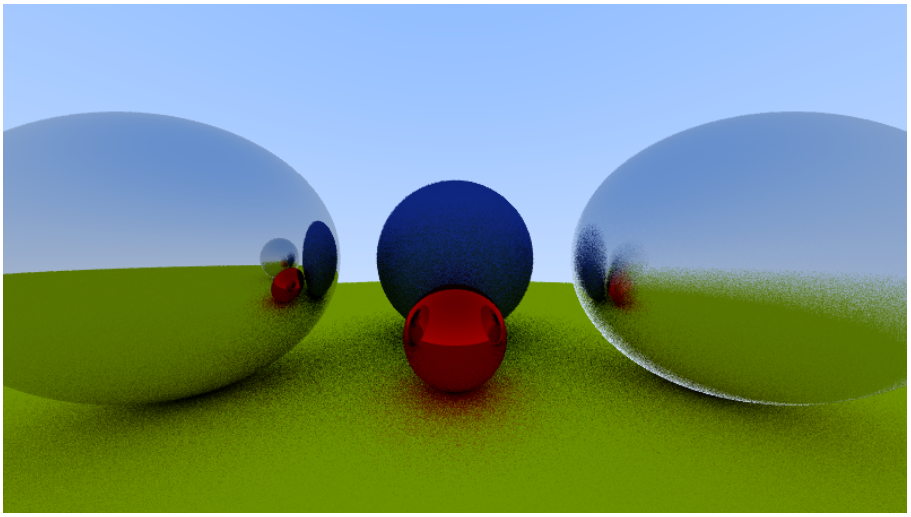
Tabela 6: Uticaj maksimalnog broja odbijanja zraka na vreme izvršavanja na grafičkoj kartici (CUDA)

Rezolucija slike	Broj zraka po pikselu	Max broj odbijanja zraka	Vreme izvršavanja
800x450	100	4	30 ms

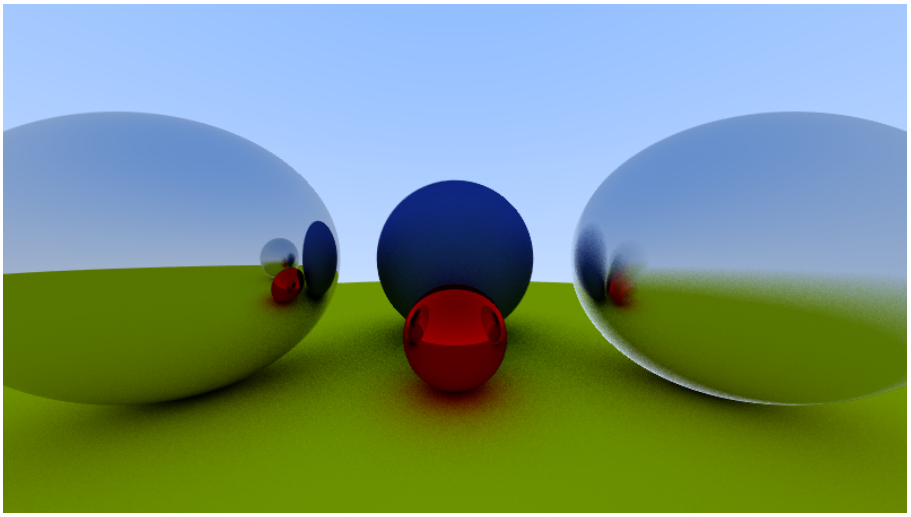
Rezolucija slike	Broj zraka po pikselu	Max broj odbijanja zraka	Vreme izvršavanja
800x450	100	8	37 ms
800x450	100	16	46 ms
800x450	100	32	42 ms
800x450	100	64	43 ms

Kvalitet slika

Slike od 3 do 6 pokazuju kako se menja kvalitet kada se menja broj zraka po pikselu (*samples*).



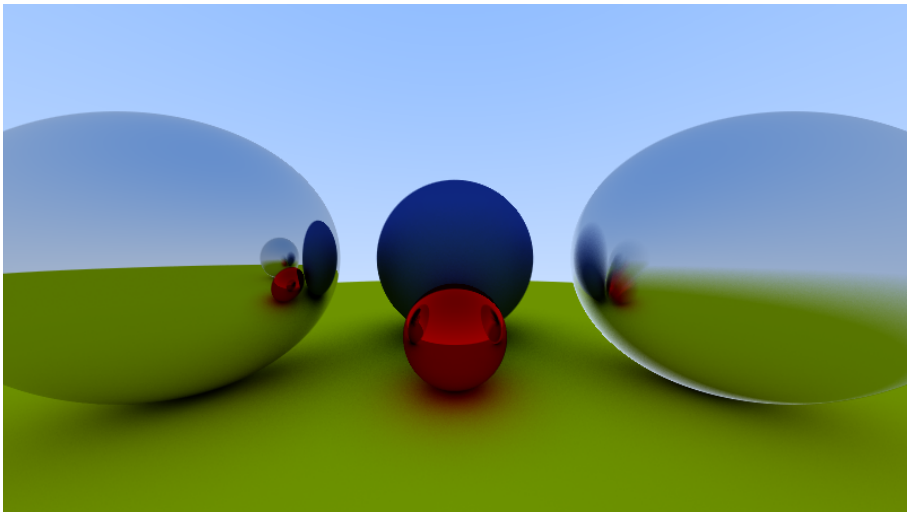
Slika 3: samples=10



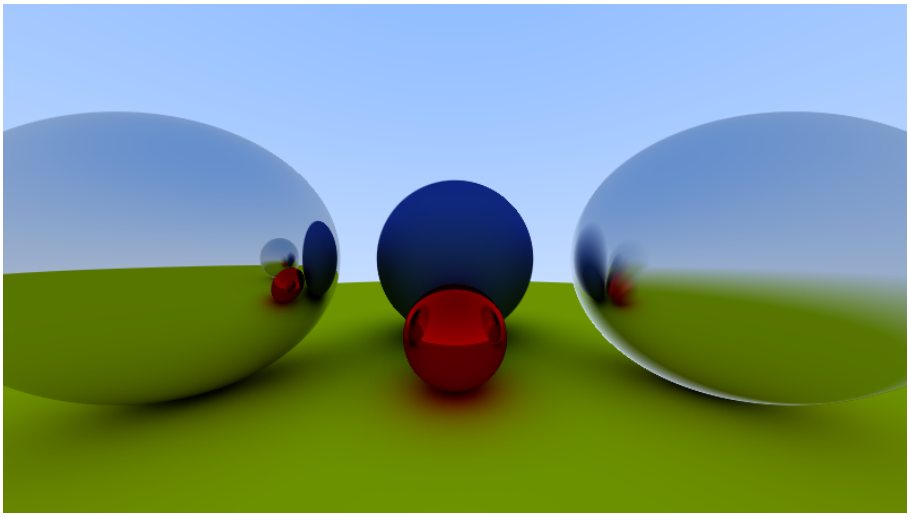
Slika 4: samples=100

Diskusija rezultata

Vidimo da je sekvencijalna CPU implementacija bila izuzetno spora (red veličine desetine sekundi), paralelna CPU implementacija (koristeći OpenMP) bila znatno brža, a GPU implementacija (koristeći CUDA okvir) je bila najbrža. Kod GPU implementacije, pri vremenima izvršavanja kraćim od 33



Slika 5: samples=1000



Slika 6: samples=10000

ms, moguće je postići renderovanje u realnom vremenu sa učestalošću od 30 sličica u sekundi (eng. *Frames Per Second*). Ipak, vizuelni kvalitet tako generisanih slika je nezadovoljavajući, što je direktna posledica redukovanog broja uzoraka po pikselu i ograničenog broja odbijanja zraka radi postizanja visokih performansi. Kada je broj zraka po pikselu veći od 1000, kvalitet slika je veoma dobar i šum se značajno smanjuje.

Zaključak

Rad na ovom projektu je pokazao da je moguće napisati višeplatformski kod koji se može prevesti i izvršavati i na procesoru i na grafičkoj kartici, a da se pri tome ne žrtvuju performanse. Ovime je značajno umanjeno dupliranje koda. Kao što je bilo i očekivano, sekvencijalna CPU implementacija je bila najsporija, paralelna CPU implementacija (koristeći OpenMP) bila znatno brža, a GPU implementacija (koristeći CUDA okvir) je bila najbrža.

Dalji razvoj može da se fokusira na razvoju novih materijala i objekata koji mogu da se koriste u sceni. Osim toga, implementacija metode kao što je hijerarhija obuhvatnih zapremina (eng. *Bounding Volume Hierarchy*) može poslužiti kao dodatna optimizacija za ubrzanje programa. Ovde bi izazov bio implementacija stabla koja je bez problema radi i na procesoru i na grafičkoj kartici.

Dodatak

Ovaj dokument je pisan u `Markdown` formatu, uz ručno definisanje LaTeX stila i komandi. Preveden je u `PDF` format korišćenjem `Pandoc` alata.

`GitHub` repozitorijum projekta koji sadrži izvorni kod i ovaj dokument: https://github.com/nkusla/accel-raytracer

Literatura

[1] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.
[2] "Bounding volume hierarchy." [Online]. Available: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy
[3] "Embarrassingly parallel." [Online]. Available: https://en.wikipedia.org/wiki/Embarrassingly_parallel
[4] "Lambertian reflectance." [Online]. Available: https://en.wikipedia.org/wiki/Lambertian_reflectance
[5] "GLM - opengl mathematics." [Online]. Available: https://glm.g-truc.net/
[6] C. Wellons, "Hash prospector." 2018. Available: https://github.com/skeeto/hash-prospector
[7] S. H. Peter Shirley Trevor David Black, "Ray tracing in one weekend." [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html
[8] "Overview of the ray-tracing rendering technique." [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/ray-tracing-rendering-technique-overview.html