UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Marko S. Spasić

Paralelizacija tehnike iscrtavanja praćenjem zraka na platformi Nvidia CUDA

master rad

Beograd, 2022.

Mentor:

dr Ivan Čukić,
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Saša Malkov, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Vesna Marinković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami, tati i sestri

Naslov master rada: Paralelizacija tehnike iscrtavanja praćenjem zraka na platformi Nvidia CUDA

Rezime: Tehnika praćenja zraka je pristup iscrtavanju trodimenzione scene u računarskoj grafici. Računski je zahtevnija od iscrtavanja scene tehnikom rasterizacije, ali slike dobijene iscrtavanjem scene tehnikom praćenja zraka mogu imati visok stepen foto-realističnosti i biti nerazlučive za ljudsko oko od slika foto-aparata visoke rezolucije. Tehnika praćenja zraka je pogodna za paralelizaciju jer je računanje boje svakog piksela nezavisno i može se izračunati paralelno. Ova činjenica otvara mogućnosti za ubrzavanje iscrtavanja tehnikom praćenja zraka na višeprocesorskim sistemima. Grafičke jedinice su se do nedavno koristile isključivo za ubrzavanje tehnike rasterizacije u grafičkim aplikacijama, ali se u poslednjih desetak godina sve više koriste za ubrzavanje i drugih algoritama pogodnih za paralelizaciju. Platforma CUDA, razvijena od strane kompanije Nvidia, stavlja na raspolaganje programski interfejs za kontrolu grafičke jedinice i olakšava pisanje paralelnog koda koji će se izvršavati na hiljadama jezgara. U ovom radu implementirana je tehnika praćenja zraka tako da može da se izvršava na jednom jezgru centralne procesorske jedinice ili na platformi CUDA. Implementacija na jednom jezgru služi kao referentna tačka za poređenje dobijenog ubrzanja. Rezultati dobijeni u ovom radu pokazuju da platforma CUDA značajno može ubrzati iscrtavanje slike tehnikom praćenja zraka.

Ključne reči: rej trejsing, CUDA, paralelizacija, računarska grafika, C++

Sadržaj

Glava 1

Uvod

Tehnika praćenja zraka iscrtava foto-realistične slike trodimenzionih scena aproksimacijom odbijanja zraka po sceni iz izvora svetlosti do sočiva kamere. Pogodna je za paralelizaciju jer se boje svakog piksela na slici koja se iscrtava mogu nezavisno izračunati. Cilj rada je da prikaže jedan način paralelizacije algoritma praćenja zraka na platformi CUDA i ubrzanje koje je moguće ostvariti paralelizacijom.

U poglavlju 2 opisane su matematičke osnove tehnike praćenja zraka i algoritam za iscrtavanje slike tehnikom praćenja zraka. Matematičke osnove tehnike praćenja zraka podrazumevaju način predstavljanja geometrijskih oblika, materijala i kamere u računaru, a opis algoritma podrazumeva prikaz procedure u formatu pseudokoda kojom se iscrtava slika scene tehnikom praćenja zraka.

Poglavlje 3 predstavlja uvod u platformu CUDA i opisuje namenu platforme CUDA, način prevođenja i pokretanja programa na platformi CUDA, model izračunavanja, model memorije i na kraju uspostavlja vezu između platforme CUDA i algoritma praćenja zraka.

U poglavlju 4 opisani su glavni moduli projekta i njihova namena. Implementacija tehnike praćenja zraka na jednoj niti procesora i implementacija na platformi CUDA dele većinu koda u projektu. Zbog toga je u poglavlju 4 prvo opisan kôd koji dele obe implementacije, a zatim delovi specifični za svaku implementaciju.

Poglavlje 5 je poslednje poglavlje i u njemu je prikazana iscrtana scena koja sadrži sve objekte i materijale opisane u poglavlju 2 i predstavljeno je ubrzanje koje se dobije paralelizacijom tehnike praćenja zraka na platformi CUDA. Na kraju poglavlja 5, diskutuju se moguća unapređenja, dalji rad i aktuelne tehnologije u oblasti iscrtavanja tehnikom praćenja zraka.

Glava 2

Uvod u tehniku praćenja zraka

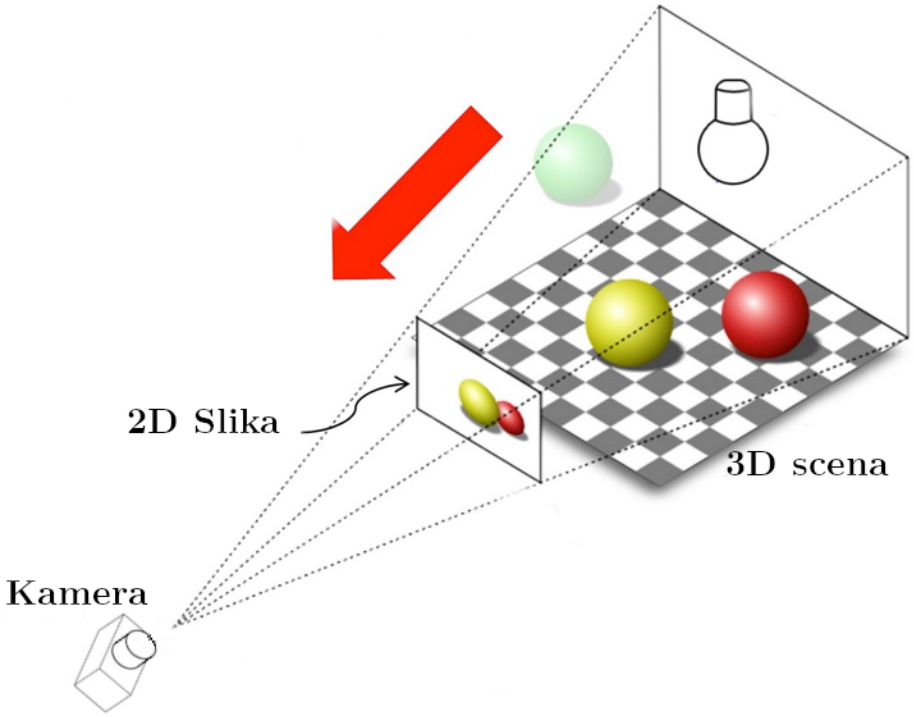
Tehnika iscrtavanja praćenjem zraka (eng. ray tracing) deo je oblasti računarske grafike. Računarska grafika se bavi generisanjem i obradom slika u računaru. Matematički je dobro zasnovana, a pored matematike u svojoj osnovi oslanja se još i na fiziku, optiku i percepciju.

U računarskoj grafici scena se sastoji od kamere, izvora svetlosti i objekata. Svi elementi scene obično su definisani korišćenjem geometrijskih primitiva. Zatim se tako definisani elementi scene matematičkim funkcijama transformišu u dvodimenzionalni niz piksela koji predstavlja pogled na tu scenu iz perspektive kamere.

2.1 Tipovi iscrtavanja u računarskoj grafici

Postoje dva glavna pristupa transformaciji scene iz matematičkih primitiva u sliku koji su poznati pod nazivima *rasterizacija* i *praćenje zraka*. Može se reći da se kod iscrtavanja slike tehnikom rasterizacije, elementi scene projektuju ka kameri, dok se prilikom iscrtavanja slike praćenjem zraka, iz kamere zraci puštaju ka sceni.

Na slici 2.1 ilustrovana je skica scene pri iscrtavanju procesom rasterizacije. Velika crvena strelica označava smer projekcije. Kod tehnike rasterizacije, scena se projektuje ka kameri. Kada se objekti koji se nalaze u vidnom polju kamere projektuju ka njoj, postojaće presek sa 2D slikom. Tačka preseka na 2D slici određuje piksel koji će biti obojen. Boja tog piksela računa se na osnovu atributa projektovanog objekta. Osvetljenost objekta se obično približno računa pomoću Fongovog ili Blin-Fongovog modela. Ovi modeli ne proizvode foto-realistično osvetljenje, ali je kvalitet osvetljenja dovoljno dobar za slučajeve korišćenja u kojima



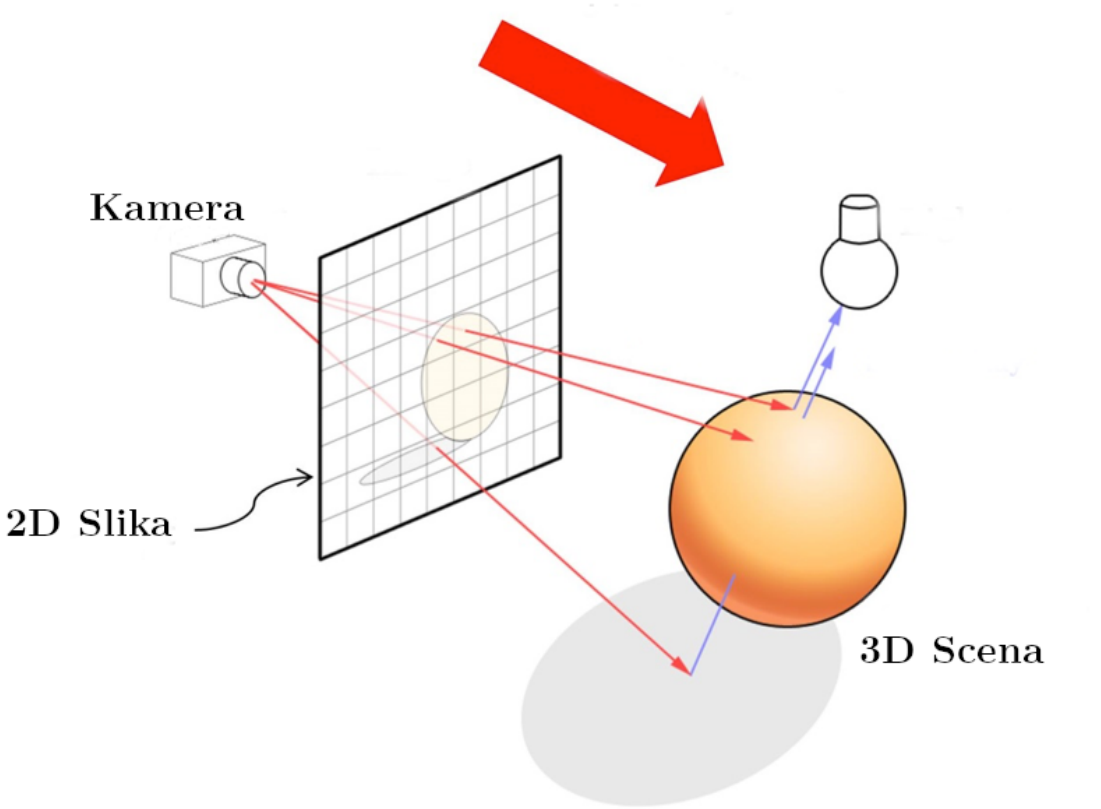
Slika 2.1: Skica scene pri iscrtavanju tehnikom rasterizacije [4].

se primenjuje. Aproksimacija osvetljenja Fongovim ili Blin-Fongovim modelom je računski efikasnija od praćenja zraka.

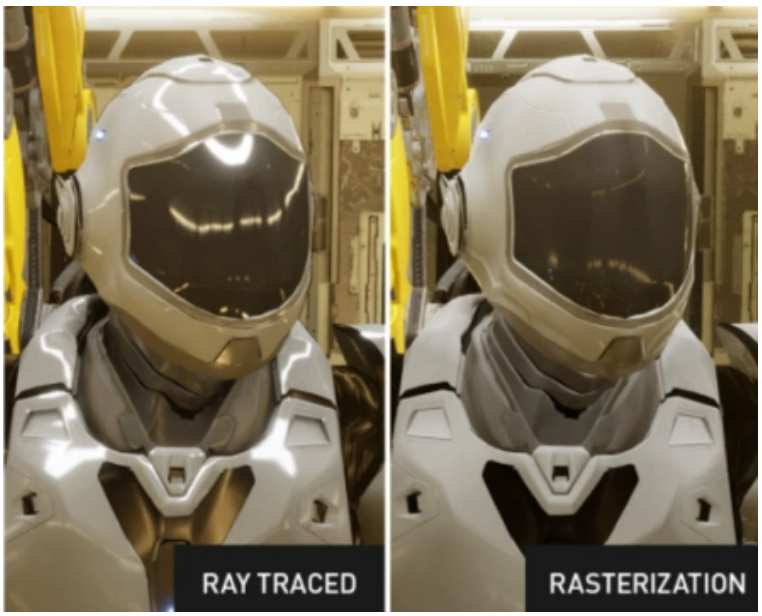
Kod praćenja zraka, kroz svaki piksel 2D slike pušta se zrak iz kamere ka sceni. Zrak se odbija od objekata na sceni dok ne dođe do izvora svetlosti. Na osnovu atributa objekata od kojih se zrak odbijao određuje se konačna boja piksela 2D slike kroz koji je zrak prošao. Na slici 2.2 ilustrovana je skica scene pri iscrtavanju tehnikom praćenja zraka. Velika crvena strelica ilustruje da se zraci puštaju od kamere ka sceni, za razliku od rasterizacije, gde se objekti scene projektuju ka kameri.

Tehnika praćenje zraka računa putanju zraka kroz scenu unazad, od kamere do izvora svetlosti. Materijal svakog objekta od kojeg se zrak odbio doprinosi konačnoj boji piksela kroz koji je zrak pušten. Tehnika praćenje zraka je, stoga, u opštem slučaju računski složenija od tehnike rasterizacije, ali proizvodi slike sa većim stepenom foto-realističnosti. Razlika u kvalitetu slike iscrtane scene dobijene tehnikom praćenja zraka i tehnikom rasterizacije ilustrovana je na slici 2.3.

Zbog računske zahtevnosti, tehnika praćenje zraka se do nedavno primenjivala isključivo u aplikacijama koje nisu radile u realnom vremenu. Međutim, napret-



Slika 2.2: Skica scene pri iscrtavanju tehnikom praćenja zraka [4].



Slika 2.3: Primer scene iscrtane tehnikom praćenja zraka (levo) i scene iscrtane tehnikom rasterizacije (desno) [17].

kom hardverskih komponenti koje su dizajnirane za grafička izračunavanja moguće je kombinovati tehnike rasterizacije i praćenja zraka. U nekim slučajevima, na dovoljno efikasnom hardveru, moguće je iscrtavati scenu praćenjem zraka i u realnom vremenu.

Značajan napredak u oblasti primene praćenja zraka u aplikacijama koje rade u realnom vremenu ostvaren je od strane kompanije Nvidia. Tehnologija RTX [16], ilustrovana na slici 2.4, omogućava kombinovanje tehnike rasterizacije i praćenja zraka tako što se veći deo scene iscrta pomoću rasterizacije, a praćenje zraka se primeni za iscrtavanje senki i reflektivnih površina. Ubrzanje koje se dobija pri iscrtavanju scene tehnikom praćenja zraka korišćenjem tehnologije RTX postignuto je kombinacijom hardverskih i softverskih rešenja koja obuhvataju čipove specijalizovane za izvršavanje matematičkih operacija koje se često koriste u tehnici praćenja zraka, specijalnih formata modela na sceni i programskog interfejsa niskog nivoa koji programerima pruža veću kontrolu nad samom platformom.

Slika 2.4: Primer scene iscrtane sa isključenom tehnologijom RTX (levo) i scene iscrtane sa uključenom tehnologijom RTX (desno) [19].

2.2 Tehnika praćenja zraka

U daljem delu glave sledi opis algoritma `praćenje_zraka` i glavnih elemenata scene. Algoritam i prateće strukture podataka biće izložene pristupom odozgo-nadole, dok će funkcije i strukture podataka koje se koriste u algoritmu praćenja zraka biti objašnjeni u narednim sekcijama. Ukoliko nije navedeno drugačije pret-

postavlja se da su vektori koji određuju elemente algoritma i struktura podataka iz prostora R^3.

```
1  pracenje_zraka(scena, kamera) -> Slika {
2    M = kamera.visina_platna
3    N = kamera.sirina_platna
4    slika = inicijalizuj_sliku(M, N)
5    for i = 0..M {
6       for j = 0..N {
7         zrak = pusti_zrak(kamera, piksel(i, j))
8         slika[i, j] = boja_zraka(zrak, scena)
9       }
10   }
11   return slika
12 }
```

Algoritam 2.1: Procedura praćenja zraka za iscrtavanje scene iz perspektive kamere

Vrednosti `kamera.visina_platna` i `kamera.sirina_platna` su pozitivni celi brojevi koji određuju broj piksela 2D slike na kojoj će scena biti iscrtana. Parametar `scena` sadrži sve objekte i osvetljenje na sceni koja će biti iscrtana, a parametar `kamera` vrednosti pozicije, smera pogleda, veličine 2D slike i udaljenosti 2D slike od kamere. Funkcija `inicijalizuj_sliku(M,N)` inicijalizuje strukturu podataka u kojoj će biti sačuvane boje piksela iscrtane scene iz perspektive kamere.

Nakon koraka inicijalizacije, iz kamere se ka sceni pušta zrak kroz svaki od piksela `(i, j)`. Parametri zraka određuju se na osnovu parametara kamere. Funkcija `pusti_zrak` zapravo ne računa propagiranje zraka kroz scenu, već samo vraća parametre koji određuju zrak koji je pušten kroz piksel `(i, j)`.

Funkcija `boja_zraka` dati `Zrak` propagira kroz scenu odbijajući ga od objekata sa kojima se zrak sudara sve dok zrak ne dođe do nekog izvora svetlosti. Tokom odbijanja, kumulativno se računa konačna boja na osnovu osobina materijala od kojih se zrak odbio i osobina svetla do kojeg je zrak stigao odbijajući se od objekata na sceni. Tako dobijena boja upisuje se na poziciju piksela `(i, j)` u strukturi podataka `Slika`.

Neka je k broj objekata na sceni. Funkcija `inicijalizuj_sliku` je složenosti $\mathcal{O}(1)$ jer samo inicijalizuje strukturu podataka u kojoj će se čuvati slika. Složenošću same procedure dominira poziv funkcije `boja_zraka` unutar dvostruke petlje. Neka je b maksimalan broj odbijanja zraka dok ne stigne do izvora svetlosti na datoj sceni i k broj objekata na samoj sceni. Složenost funkcije `boja_zraka` je onda

$\mathcal{O}(b \cdot f(k))$ gde je $f(k)$ složenost funkcije koja računa u kojoj tački zrak ima presek sa objektom na sceni i u kom smeru se zrak odbio od objekta sa kojim ima presek. Zrak može imati presek sa više objekata na sceni i zbog toga je neophodno da funkcija $f(k)$ pronađe presek koji je najbliži kameri. Dakle, ukupna složenost procedure `praćenje_zraka` je $\mathcal{O}(M \cdot N \cdot b \cdot f(k))$ za sliku dimenzija $M \times N$, k objekata i gornje ograničenje od b odbijanja zraka po sceni dok ne dođe do izvora svetlosti.

Činjenica koja ovaj algoritam čini pogodnim za paralelizaciju je to što je računanje boje svaka dva različita piksela `(i, j)`, `(i', j')` nezavisno. Takođe, funkcije `pusti_zrak` i `boja_zraka` mogu se bezbedno pozivati iz više niti istovremeno jer ne menjaju vrednosti prosleđenih argumenata, nemaju sporednih efekata i ne zavise od deljenog stanja čija se vrednost može promeniti tokom izvršavanja algoritma.

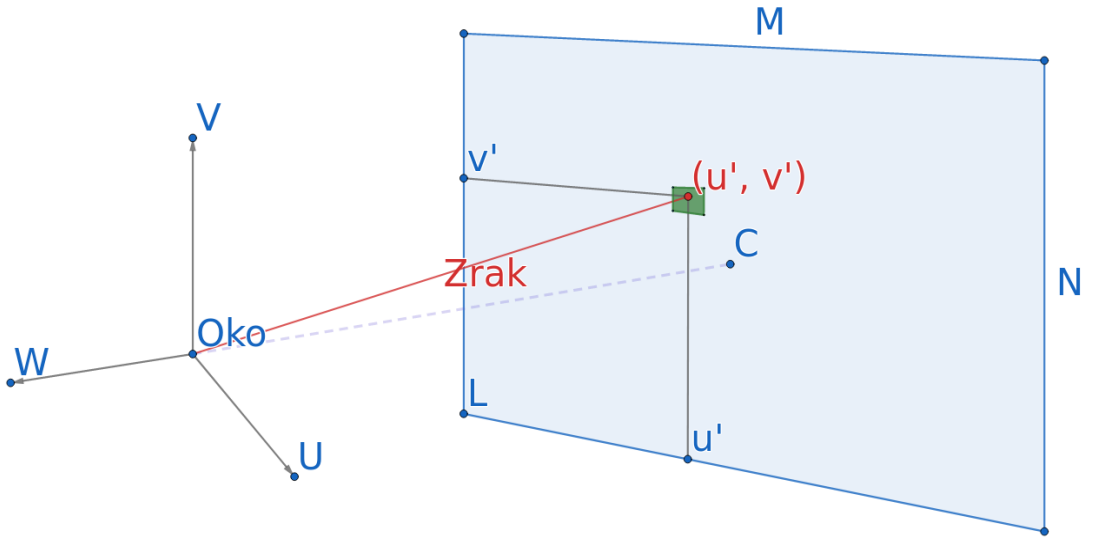
Paralelizacijom po pikselu, sa $M \times N$ paralelnih procesora, složenost algoritma bi bila $\mathcal{O}(b \cdot f(k))$ jer bi linije 7 i 8, algoritma 2.1, mogle da se izvrše paralelno, tako što bi se svakom procesoru dodelio jedan piksel `(i,j)` za koji bi računao boju zraka projektovanog kroz taj piksel.

2.3 Zrak i kamera

U ovom poglavlju biće date definicije `Kamere`, `Zraka` i funkcije `pusti_zrak`. U daljem tekstu, termin *prostor scene* (eng. scene space ili eng. world space) odnosi se na koordinatni sistem prostora u kojem se nalaze objekti na sceni. Kada se kaže da se neki objekat nalazi u prostoru scene na koordinatama (x, y, z), te koordinate su relativne koordinate u odnosu na koordinatni početak scene. Slično važi i za *prostor kamere* (eng. camera space): kada se kaže da se neki objekat nalazi na koordinatama (u, v, w) u prostoru kamere, to znači da je pozicija kamere koordinatni početak, a koordinate (u, v, w) su relativne koordinate u odnosu na taj koordinati početak.

Model kamere

Glavna uloga modela kamere je da odredi deo scene koji će biti iscrtan. Deo scene koji se nalazi u vidnom polju kamere naziva se *zapremina pogleda* (eng. viewing frustum).



Slika 2.5: Matematički elementi modela kamere.

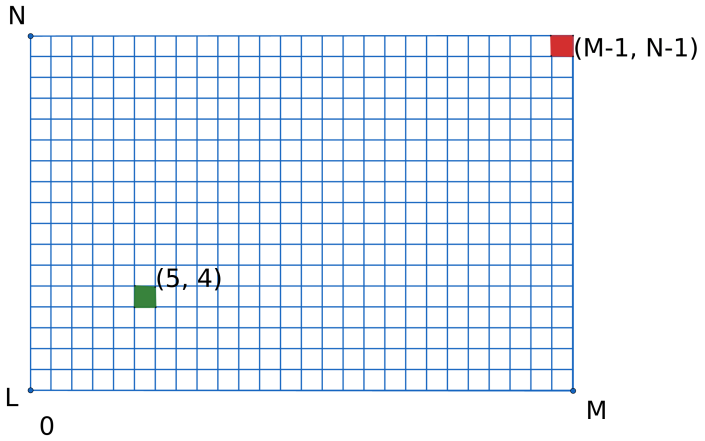
Na slici 2.5 ilustrovan je matematički model kamere. Tačka u kojoj se kamera nalazi naziva se *oko*. Orijentacija kamere poštuje pravilo desne ruke: V-osa pokazuje na gore, U-osa na desno, a kamera gleda niz negativan deo W ose. Vektori U, V, W određuju prostor kamere. Plavi pravougaonik predstavlja matricu piksela kroz koju se zraci puštaju ka sceni i obično se u literaturi naziva *platno* (eng. canvas). Dimenzije platna određene su vrednostima M i N. Deo scene koji će biti vidljiv iz tačke gledišta kamere nalazi se sa druge strane platna u odnosu na kameru i obuhvata svaki objekat pogođen zrakom puštenim iz tačke `Oko` kroz neki od piksela.

Tačka C je presek negativnog dela W ose koordinatnog sistema kamere sa platnom i uvek se nalazi na centru lokalnih koordinata u prostoru platna. Tačka sa koordinatama (u', v') nalazi se u prostoru platna i predstavlja presečnu tačku zraka i platna za zrak pušten iz tačke `Oko` ka sceni kroz piksel $(i, j) \in [0..M) \times [0..N)$, koji je na slici 2.5 predstavljen zelenim kvadratom. Tačka L je tačka u prostoru scene i određuje poziciju donjeg levog temena platna u odnosu na kameru.

Podrazumevano, kamera se nalazi u koordinatnom početku scene. Koordinatni vektori prostora kamere U, V, W se u tom slučaju poklapaju sa koordinatnim osama scene X, Y, Z. Ovako definisana kamera omogućava iscrtavanje samo onih scena u kojima se kamera nalazi u koordinatnom početku. Kamera je fiksna, a ukoliko je potrebno promeniti ugao gledanja ili rastojanje na kome se kamera

nalazi od scene, onda se cela scena transformiše tako da se ostvari željena tačka pogleda. Ovakav pristup nije intuitivan jer se svaka transformacija nad kamerom mora posmatrati kao inverzna transformacija nad scenom. Na primer, ukoliko je potrebno translirati kameru u poziciju $(1, 2, 3)$, onda se svi objekti na sceni transliraju za vektor $(-1, -2, -3)$ kako bi se postigao željeni efekat. Stoga se u praksi kamera obično zadaje preko vrednosti koje su ljudima lakše za poimanje, kao što su tačka na sceni u koju kamera gleda, vektor na gore i pozicija kamere na sceni. Na osnovu ovih vrednosti se onda izračunaju koordinatne ose kamere i pozicija platna na sceni u odnosu na kameru. Ovakav pristup više odgovara ljudskom razumevanju pozicioniranja kamere na sceni i slikanja scene. Scena je statična, a kamera se pomera u okviru scene.

Na slici 2.6 prikazana je matrica piksela na platnu dimenzija $M \times N$. Pozicije su diskretne i numerisane kao polja u matrici sa početkom u donjem levom uglu. Tačka L je donje levo teme piksela na poziciji $(0, 0)$ dok je piksel u gornjem desnom uglu u matrici piksela na poziciji $(M - 1, N - 1)$.

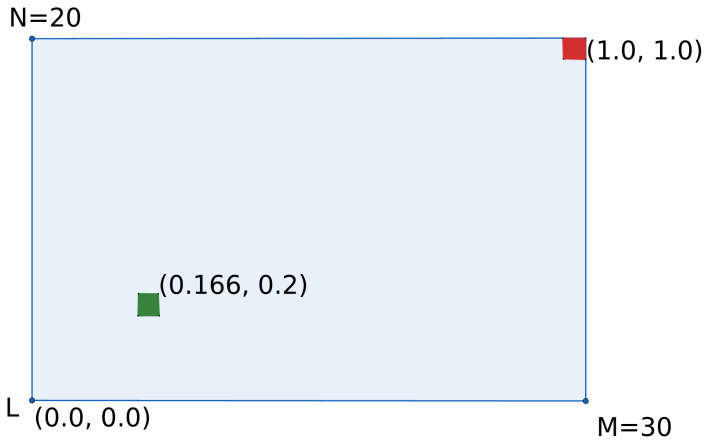


Slika 2.6: Matrica piksela na platnu dimenzija $M \times N$.

Na slici 2.7 prikazan je prostor platna. Koordinate svake tačke unutar prostora platna su u opsegu $[0, 1] \times [0, 1]$. Vrednost $(0.166, 0.2)$ iz prostora platna, preslikava se na piksel $(5, 4)$ sa slike 2.6.

U opštem slučaju, koordinate piksela (i, j), za platno dimenzija $M \times N$, u prostoru platna imaju vrednost $(u', v') = (\frac{i}{M-1}, \frac{j}{N-1})$

Model kamere koji ovaj rad implementira ima dodatne parametre koji nisu navedeni u ovom poglavlju radi jednostavnosti uvoda u algoritam praćenja zraka.



Slika 2.7: Vrednost piksela $(5, 4)$ sa slike 2.6 u prostoru platna dimenzije 30×20.

Parametri koji nisu spomenuti u ovom poglavlju biće objašnjeni u glavi 4.

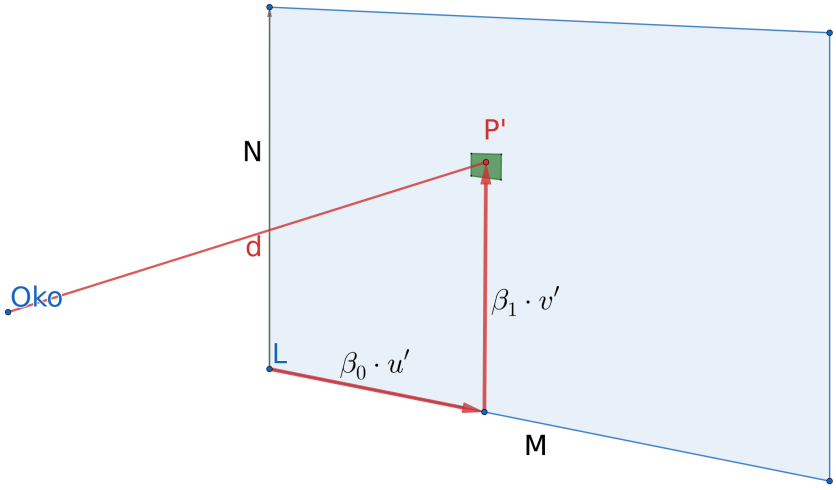
Model zraka

U tehnici praćenja zraka, zrak aproksimira putanju fotona od izvora svetlosti do kamere. Zrak se pušta od kamere ka sceni, kroz piksele na platnu, ne od izvora svetlosti do kamere. Zato se kaže da se putanja zraka kroz scenu izračunava unazad, od kamere ka izvoru svetlosti. Alternativni pristup bi bio da se zraci puštaju iz izvora svetlosti ka sceni i da se računa odbijanje zraka od objekata scene sve dok kroz svaki od piksela platna do kamere ne prođe barem jedan zrak. Taj pristup je daleko složeniji za izračunavanje i neefikasan jer većina zraka koji se odbijaju po sceni ne stigne do kamere.

```
1 pusti_zrak(kamera, piksel) -> Zrak {
2   u',v' = (piksel.i/(M-1), (piksel.j/(N-1))
3   B0 = (M, 0.0, 0.0)
4   B1 = (0.0, N, 0.0)
5   zrak.o = kamera.Oko
6   zrak.d = kamera.L + B0 * u'+ B1 * v'- kamera.Oko
7   return zrak
8 }
```

Algoritam 2.2: Funkcija za puštanje zraka iz kamere.

Takođe, broj zraka koje bi bilo potrebno generisati, kako bi barem jedan prošao kroz svaki od piksela platna, bio bi daleko veći nego pri puštanju zraka kroz svaki piksel platna i praćenja putanje od kamere do izvora svetlosti. *Zrak* je definisan formulom $r(t) = o + t \cdot d$, gde je $o \in \mathbb{R}^3$ početna tačka zraka, a $d \in \mathbb{R}^3$ vektor smera u kojem se zrak kreće. Vrednost $t \in \mathbb{R}$ određuje tačku u prostoru na datom zraku r. Početna tačka zraka o je pozicija kamere u svetu. Funkcija `pusti_zrak`, data u algoritmu 2.2, računa smer d zraka puštenog iz kamere za `piksel(i, j)`.



Slika 2.8: Ilustracija formule za računanje smera zraka d iz tačke Oko.

Na slici 2.8 ilustrovani su svi vektori koji učestvuju u računanju zraka. Vektor smera može se izračunati kao $d = P' - Kamera.Oko$, gde je tačka P' preslikana tačka $P = (u', v')$ iz prostora platna u prostor scene.

Algoritam 2.2 je uprošćena verzija stvarne implementacije koja dodavanjem elementa nasumičnosti pri puštanju zraka bolje aproksimira stvarno ponašanje zraka svetlosti u prirodi. U glavi 4 biće naglašeno na koji način je implementacija dopunjena.

2.4 Materijal

U ovom poglavlju biće objašnjeno kako se boja zraka menja pri sudaranju sa objektom na sceni. Implementacija tehnike praćenja zraka, u ovom radu, atribute objekta koji definišu interakciju svetla sa tim objektom naziva *materijal*. U daljem tekstu, kada se spominju zrak i materijal, te definicije se odnose na modele zraka i materijala u implementaciji tehnike praćenja zraka u ovom radu, nikada na ponašanje fotona i materijala u stvarnom svetu.

Materijal može *emitovati* (eng. emit) ili *rasipati* (eng. scatter) zrake. Emitovanje i rasipanje zraka su transformacije koje se primenjuju na zrak kada se sudari sa nekim objektom na sceni. Kada zrak stigne do objekta na sceni koji je izvor svetlosti, onda se na zrak primeni transformacija emitovanja zraka. Kada zrak stigne do objekta na sceni koji ne emituje svetlo, onda se na zrak primeni transformacija rasipanja zraka. Ove transformacije primenjuju se unutar funkcije `boja_zraka` iz algoritma 2.1.

Materijal može biti deo geometrije objekta ili nezavisan od geometrije objekta. Na primer, interakcija zraka sa sferom nekog materijala može se razlikovati od interakcije zraka sa kockom istog tog materijala. U ovom radu i većini programa za 3D iscrtavanja, materijal i geometrija objekta su odvojeni, tako da se jedan materijal može dodeliti objektima različitih geometrija.

Tipovi materijala

Egzaktna simulacija interakcije fotona sa materijalom u stvarnom svetu bi bila računski previše složena da bi bila primenljiva kao rešenje za grafičke aplikacije. Zbog toga se, kao i sa zrakom, koristi aproksimacija čija se preciznost može parametarski podešavati u zavisnosti od željenog kvaliteta slike.

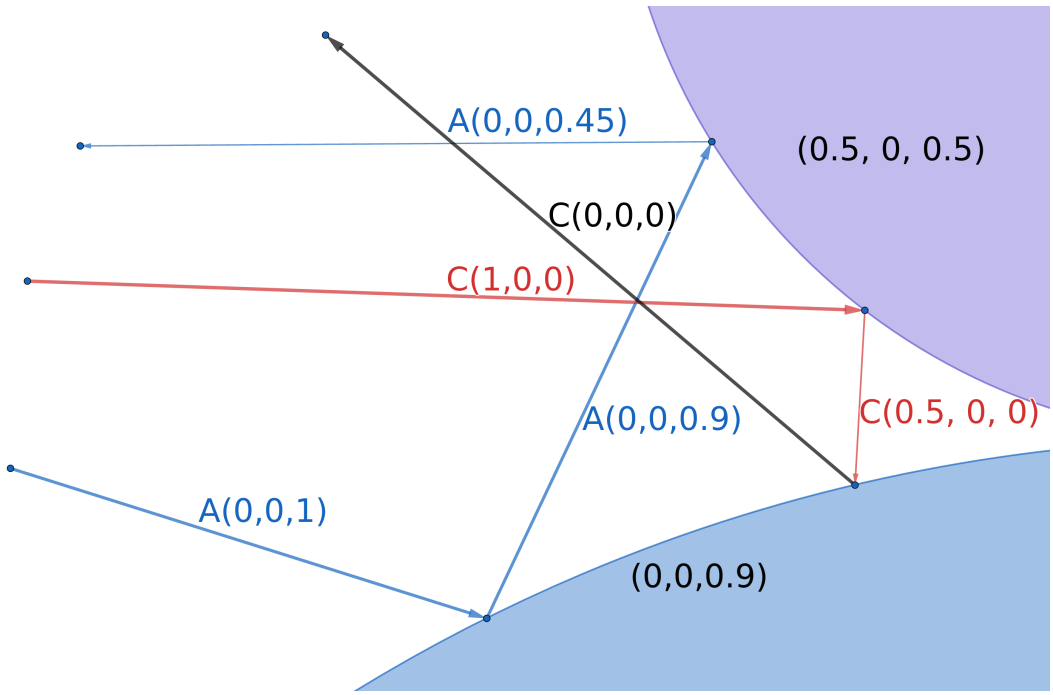
Implementacija tehnike praćenja zraka u ovom radu razlikuje četiri vrste materijala:

- difuzni (eng. diffuse)
- metalni (eng. metal)
- providni (dijelektrični (eng. dielectric))
- izotropični (eng. isotropic)

Kada se detektuje presek zraka i objekta, na osnovu tipa materijala objekta sa kojim se zrak sudario primenjuje se transformacija rasipanja zraka za taj materijal. Na ovaj način se modeluje interakcija zraka sa objektom na sceni. Ova lista daleko je od potpune i napredni programi za iscrtavanje implementiraju bogatiji skup materijala objekta. Međutim, modelovanje svakog od mogućih tipova materijala prevazilazi opseg ovog rada.

Difuzni materijal

Difuzni ili *mat* materijali ne emituju svetlo, već zrake koje dolaze iz okruženja moduliraju sopstvenom bojom. Smer reflektovanja zraka od difuznog materijala je nasumičan. Kada zrak ima presek sa objektom difuznog materijala on se može reflektovati ili potpuno apsorbovati ako je materijal tamne boje. Na primer, ako se zrak koji nosi samo crvenu boju odbije od objekta difuznog materijala čija je boja mešavina crvene i zelene, samo će crvena boja biti reflektovana.

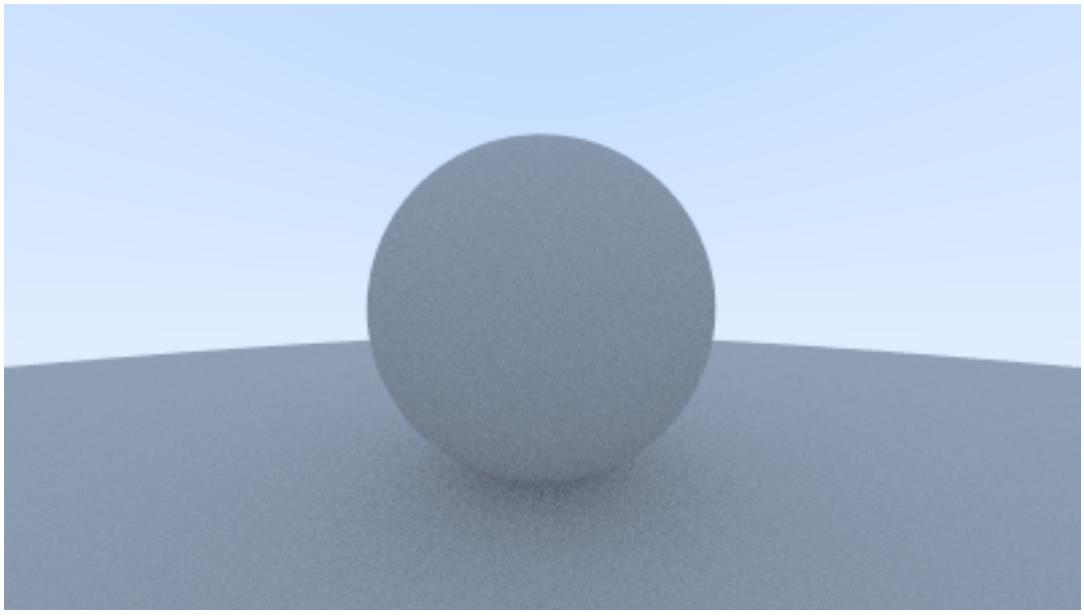


Slika 2.9: Ilustracija interakcije zraka sa objektima difuznog materijala.

Na slici 2.9 ilustrovana je interakcija zraka sa objektima difuznog materijala. Pogled na scenu je presek dve lopte, a zrak je predstavljen kao vektor sa smerom kretanja od izvora svetlosti ka sceni radi lakše ilustracije računanja boje zraka. U implementaciji praćenja zraka, zrak se pušta od kamere ka sceni i odbija se

od objekata na sceni dok ne stigne do izvora svetlosti. Zrak A emitovan je iz izvora svetlosti plave boje, a zrak C dolazi iz izvora svetlosti crvene boje. Uređena trojka vrednosti pored naziva svakog zraka predstavlja boju zraka u RGB formatu. Donjoj sferi je dodeljen difuzni materijal čiste plave boje, a gornja sfera ima difuzni materijal ljubičaste boje koja se dobija kao mešavina plave i crvene.

Zrak A odbija se od donje sfere i prilikom tog odbijanja rasipa se plava boja zraka jer je objekat od koga se zrak odbija plave boje. Ljubičasti objekat reflektuje plavu i crvenu boju. Kada se plavi zrak A odbije od ljubičastog objekta, jedina boja koja se može reflektovati je plava. Intenzitet plave boje zraka A pre odbijanja od ljubičaste sfere je 0.9. Ljubičasta sfera reflektuje 50% plave boje dolazećeg zraka. Kada se zrak A, sa intenzitetom plave boje u vrednosti od 0.9, sudari sa objektom koji odbija samo 50% plave boje dolazećeg zraka, odbijeni zrak imaće intenzitet plave boje u vrednosti od 0.45. Kada se zrak $C(1, 0, 0)$ odbije od ljubičaste sfere nastaje zrak $C(0.5, 0, 0)$ koji će biti apsorbovan od strane plave sfere jer plava sfera odbija samo plavu komponentu zraka. Putanje zraka na slici su od izvora svetlosti do kamere, zbog toga postoji zrak $C(0, 0, 0)$ koji je crne boje, jer do kamere mora stići informacija da u preseku zraka $C(0.5, 0, 0)$ i plave sfere postoji senka.



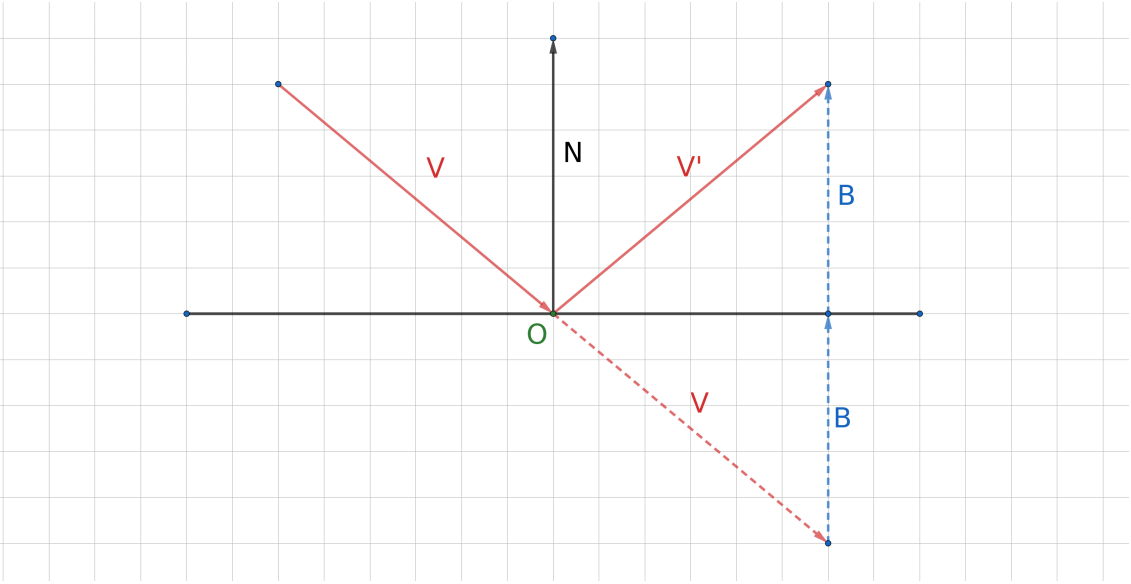
Slika 2.10: Iscrtana sfera difuznog materijala [21][Slika 8].

Na slici 2.10 iscrtane su dve sfere plave boje, difuznog materijala. Druga sfera nalazi se ispod centralne i većeg je poluprečnika, zato izgleda kao da centralna

sfera stoji na podlozi. U podnožju centralne sfere može se primetiti blaga senka nastala apsorbovanjem dela intenziteta boje zraka koji su se odbijali između dve sfere u tim delovima scene.

Metal

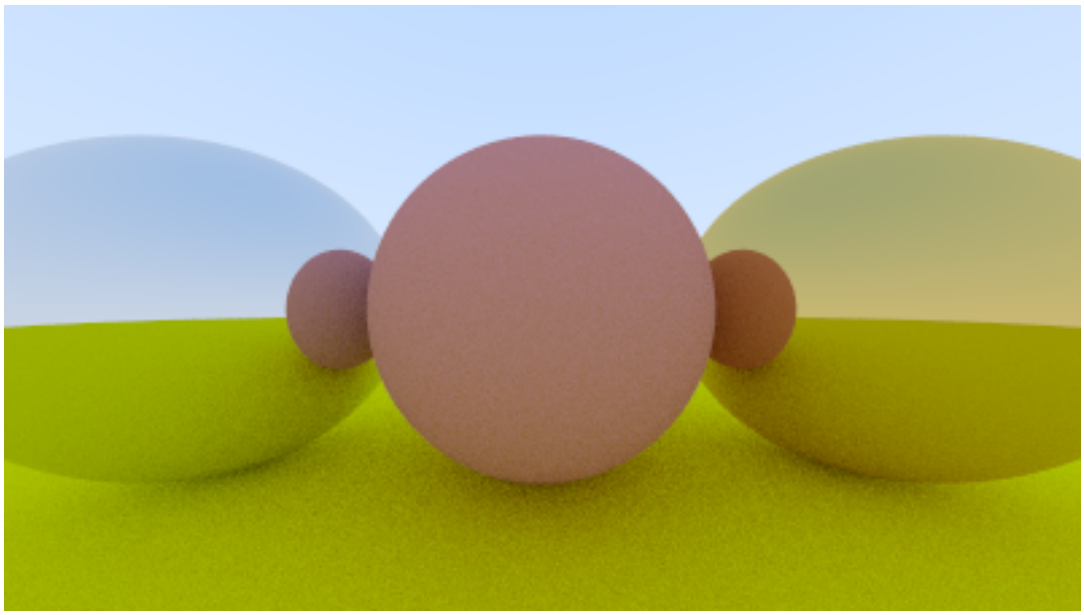
Metal je materijal koji modeluje interakciju svetla sa reflektivnom površinom. Kada se zrak sudari sa objektom kojem je dodeljen materijal metala, odbija se u zavisnosti od ugla padanja, a ne nasumično kao što je to bio slučaj kod difuznog materijala. Ovakav način odbijanja zraka proizvodi efekat materijala koji se presijava kada svetlo pada na njega.



Slika 2.11: Računanje vektora refleksije za vektor V oko normale N u tački preseka O zraka i površine objekta.

Vektor odbojnog zraka od objekta kojem je dodeljen materijal metala biće vektor refleksije oko vektora normale nad površinom objekta u tački preseka sa zrakom. Na slici 2.11 vektor V predstavlja smer upadnog zraka. Tačka O je tačka preseka objekta i zraka, a vektor N vektor normale u toj tački. Slika ilustruje računanje vektora smera odbijenog zraka $V' = V - 2(V \cdot N)N$, gde je $B = (V \cdot N)N$, po formuli refleksije. Računanje refleksije je česta operacija u računarskoj grafici, stoga na većini platformi za ubrzavanje grafičkih izračunavanja postoji kao intrinzična funkcija [7].

Računanje vektora smera odbijenog zraka od objekta metalnog materijala pri iscrtavanju proizvodi efekat glatkog metala. Na slici sfere sa leve i desne strane



Slika 2.12: Primer scene sa refleksijom zraka od objekata metalnog materijala [21][Slika 11].

imaju materijal metala. Sfera u sredini i podloga imaju difuzni materijal. Efekat glatkog presijavanja postignut je računanjem vektora refleksije bez elemenata nasumičnosti.

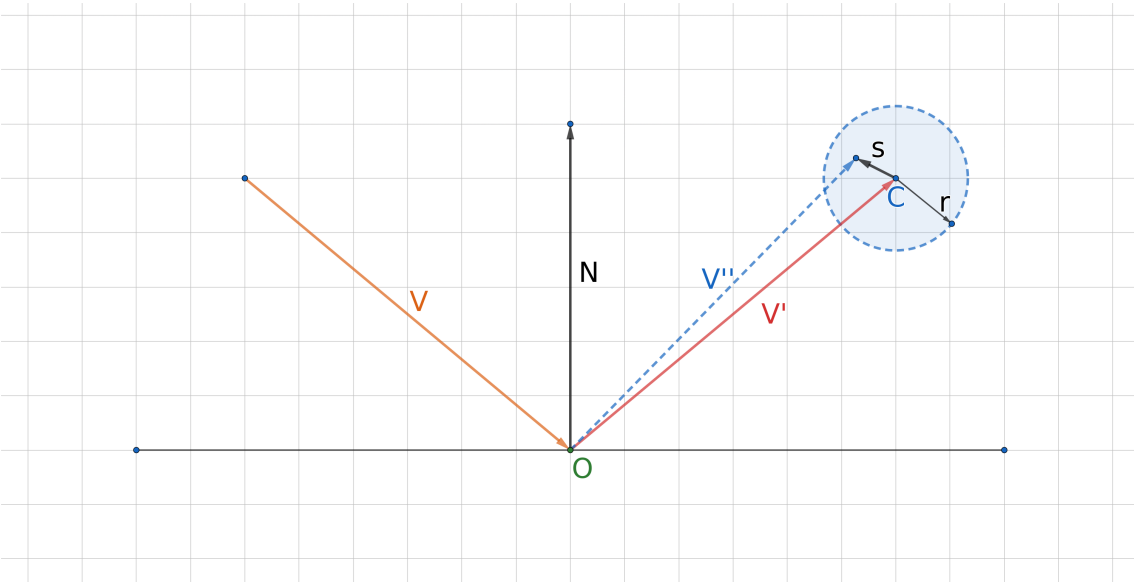
Dodavanjem elementa nasumičnosti prilikom računanja vektora refleksije može se proizvesti efekat nesavršenosti i hrapavosti materijala. Što je veći faktor nasumičnosti, to će efekat hrapavosti biti izražajniji.

Vektor V', sa slike 2.13 dobija se refleksijom vektora V oko vektora normale N. Da bi se dobio konačan odbijeni vektor zraka V'' od objekta metalnog materijala, smer vektora V' se izmešta za neki mali vektor pomeraja s i izračunava se preko formule:

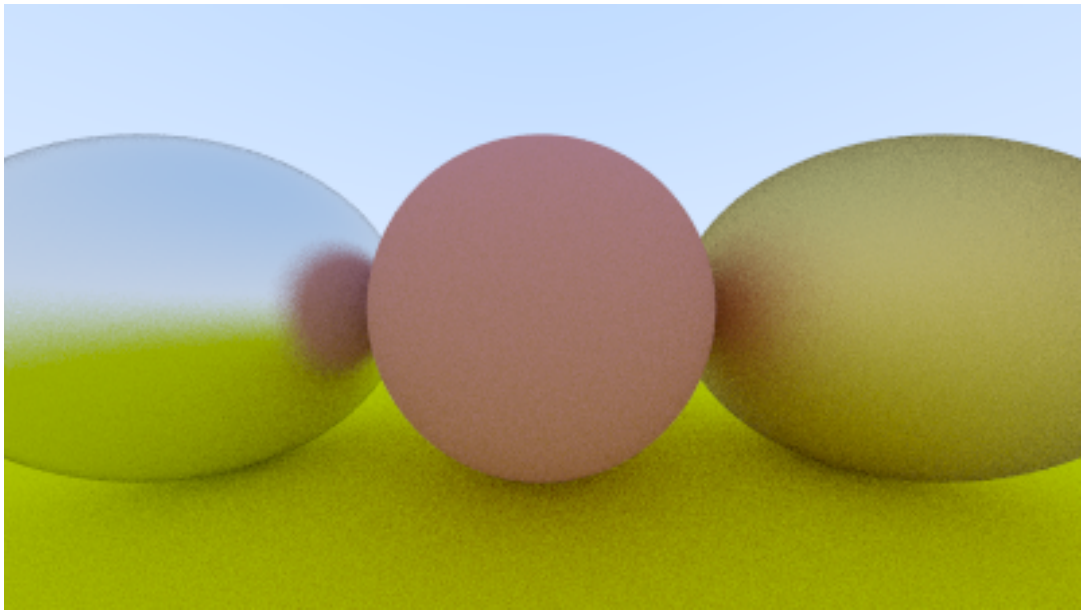
$$V' = V - 2(V \cdot N)N$$

$$V'' = V' + s, s \in S(C; r)$$

gde je s vektor sa početkom u tački C, proizvoljnog smera, dužine najviše r. Dužina r određuje faktor hrapavosti materijala tako što će za veću vrednost r efekat hrapavosti biti jači. Efekat hrapavosti sfera na slici 2.14 dobija se za $r > 0$. Materijal će izgledati mutnije jer se vektori reflektuju sa većim stepenom nepravilnosti što je vrednost r veća. U slučaju da je $r = 0$, metal će izgledati glatko kao na slici 2.12.



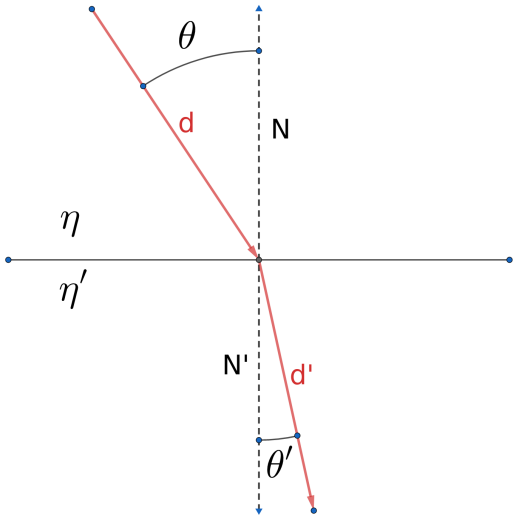
Slika 2.13: Nasumični pomeraj reflektovanog vektora V' oko normale N za vektor s radi postizanja efekta zamućenosti.



Slika 2.14: Primer scene sa efektom refleksije zraka od objekata metalnog materijala sa faktorom hrapavosti [21][Slika 12].

Providni materijal

Providni materijali transformišu zrak svetlosti propuštajući ga kroz površinu objekta.



Slika 2.15: Vektor smera zraka d i refraktovani vektor smera zraka d'.

Efekat koji nastaje kada zrak prelazi iz jednog prostora u drugi, pri čemu su prostori različite gustine u fizici se naziva *refrakcija* i opisan je *Snelovim* (eng. Snell) zakonom:

$$\eta \cdot \sin\theta = \eta' \cdot \sin\theta' \tag{2.1}$$

gde su η i η' sa slike 2.15 redom faktori refrakcije prostora u kojem se zrak trenutno nalazi i prostora u koji zrak prelazi, a θ i θ' redom uglovi koje ulazni zrak i izlazni zrak zaklapaju sa vektorima normala N i N'. Na primer, ako zrak iz vazduha prelazi u staklo, onda je $\eta = 1.0$, a $\eta' = 1.5$ gde je η faktor refrakcije vazduha, a η' faktor refrakcije stakla. Vektor d je smer zraka, a vektori N i N' vektori normale u tački preseka zraka i objekta sa slike 2.15. Kako bi se izračunao smer vektora d' izlaznog zraka potrebno je rešiti jednačinu (2.1) po θ'

$$\sin\theta' = \frac{\eta}{\eta'} \cdot \sin\theta. \tag{2.2}$$

Vektor d' može se zapisati kao zbir

$$d' = d'_{\top} + d'_{||} \tag{2.3}$$

gde je $d'_{\top}$ vektor normalan na N', a $d'_{||}$ vektor paralelan sa N'. Kada se jednačina (2.3) reši po $d'_{\top}$ i $d'_{||}$ dobije se:

$$d'_{\perp} = \frac{\eta}{\eta'}(d + N\cos\theta)$$

$$d'_{||} = -N\sqrt{1 - |d'_{\perp}|^2}.$$

Sa ograničenjem da su d i N jedinični vektori, $\cos\theta$ može se zapisati kao $\cos\theta = -d \cdot N$ čime se dobija izraz

$$d'_{||} = \frac{\eta}{\eta'}(d + (-d \cdot N)N).$$

Smer refraktovanog zraka d' sada se može izračunati na osnovu vektora smera ulaznog zraka d, normale N u tački preseka ulaznog zraka sa objektom i faktora refrakcije η i η'.

U slučaju kada zrak prelazi iz materijala sa većim faktorom refrakcije u materijal manjeg faktora refrakcije, tada postojanje rešenja Snelovg zakona zavisi od vrednosti θ. Na primer, ako zrak prelazi iz stakla ($\eta = 1.5$) u vazduh ($\eta' = 1.0$) tada jednačina (2.2) ima oblik:

$$\sin\theta' = \frac{1.5}{1.0} \cdot \sin\theta \tag{2.4}$$

Uglovi θ i θ' su uvek iz intervala $[0, \pi)$. Na primer, ako je ugao $\theta = \frac{\pi}{2}$, jednačina (2.4) ima oblik $\sin\theta' = 1.5$ i tada ne postoji rešenje po θ iz intervala $[0, \pi)$.

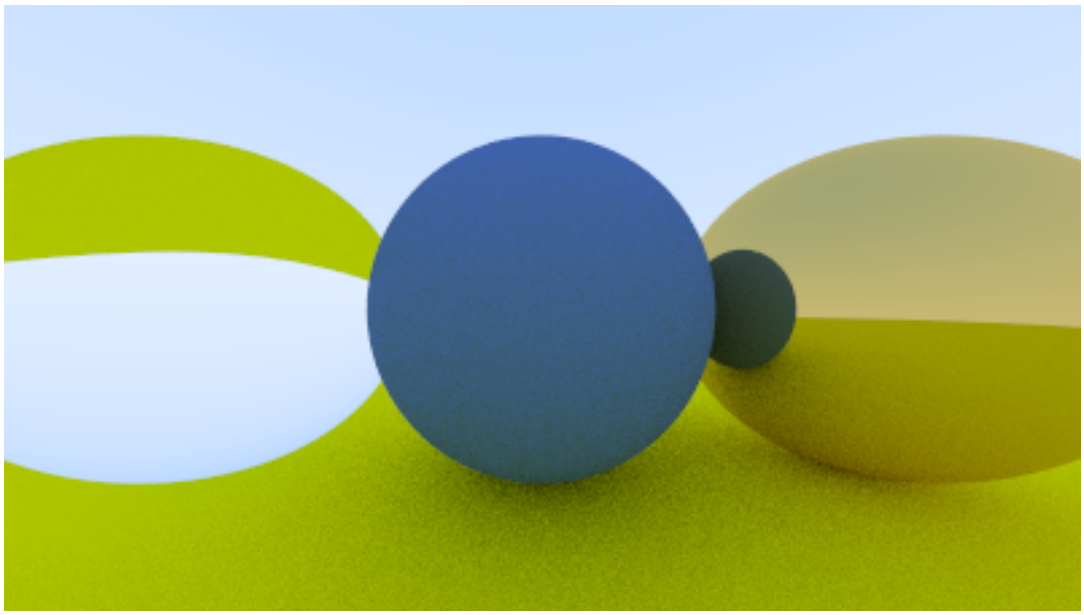
Kod pravog stakla reflektivnost zavisi i od ugla padanja zraka. Zbog toga se pod određenim uglovima staklo ponaša kao ogledalo. Jednačina po kojoj se menja smera zraka prilikom prolaska kroz pravo staklo je izuzetno komplikovana, te se zbog toga koristi Šlikova (eng. Christophe Schlick) aproksimacija:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos(\theta))^5 \tag{2.5}$$

gde je

$$R_0 = \left(\frac{\eta - \eta'}{\eta + \eta'}\right)^2.$$

Računanje vektora refrakcije deli se onda na dva slučaja. Ako postoji rešenje Snelovog zakona za date η, η', d ili je $R(\theta) \geq random(0, 1)$, onda se zrak refraktuje, u suprotnom zrak se reflektuje. Element nasumičnosti koristi se kako bi se bolje aproksimiralo ponašanje zraka u prirodi. Implementacija računanja smera zraka



Slika 2.16: Ilustracija efekta refrakcije zraka na sferi providnog materijala (levo) [21][Slika 15].

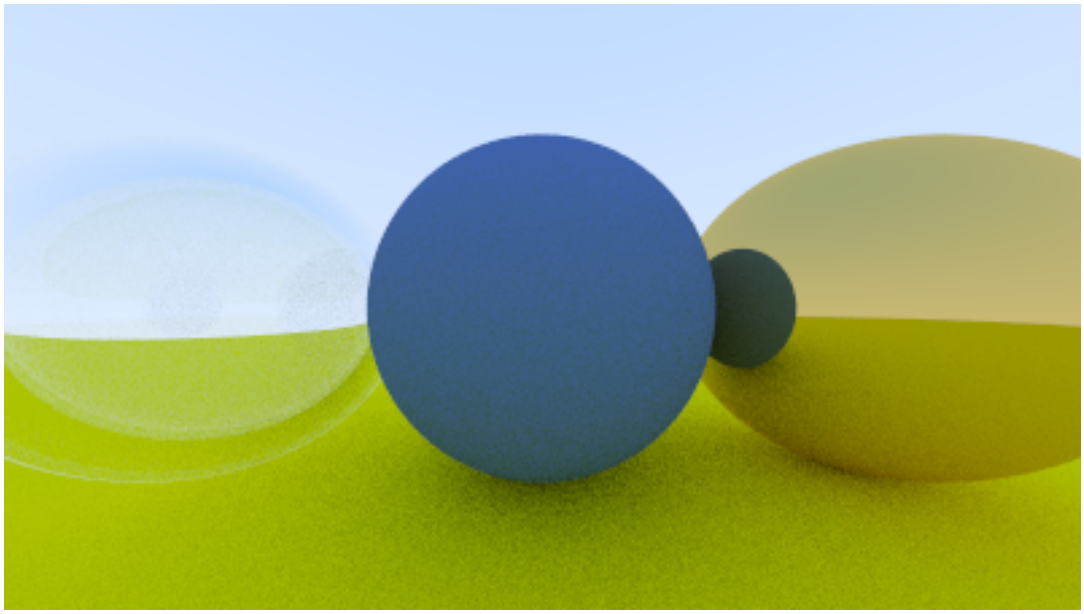
u slučaju providnog materijala je data u poglavlju 4, a na slici 2.16 ilustrovan je primer scene na kojoj je leva sfera od providnog materijala, srednja od difuznog, a desna od metalnog. Zraci pušteni iz kamere koji pogađaju gornji deo leve sfere refraktuju se ka zemlji, jer vektori normala na gornjem delu sfere pokazuju ka nebu i ka kameri, dok se zraci koji pogađaju sferu od sredine pa naniže refraktuju ka nebu jer vektori normala na time delovima sfere pokazuju ka kameri i ka zemlji.

Na slici 2.17 može se videti interesantan efekat postavljanja faktora refrakcije na negativnu vrednost za sferu sa leve strane. U tom slučaju sve normale na površini sfere pokazuju ka unutra i zbog toga leva sfera izgleda kao providni mehur.

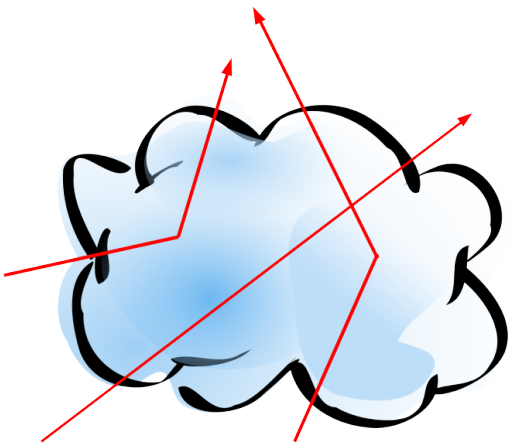
Izotropični materijal

Izotropične zapremine (eng. isotropic volume) su delovi prostora u kojima se zrak može nasumično odbiti u proizvoljnom smeru. Mogu modelovati ponašanje zraka pri prolasku kroz dim ili maglu.

Kod difuznih, providnih i metalnih materijala zrak se sudara sa površinom objekta i na osnovu njegove površine izračunava se smer odbijanja zraka, dok se kod izotropičnih zapremina ponašanje zraka izračunava na osnovu njegove interakcije sa unutrašnjošću zapremine objekta. Na slici 2.18 ilustrovan je prolazak



Slika 2.17: Ilustracija efekta negativnog faktora refrakcije na sferi providnog materijala (levo) [21][Slika 16].

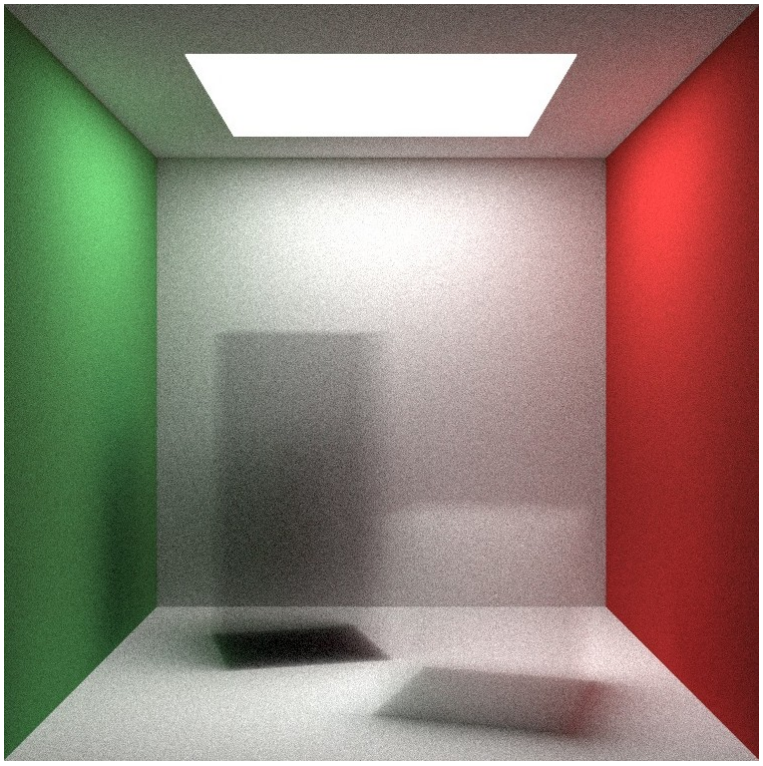


Slika 2.18: Ilustracija nasumičnosti odbijanja zraka pri prolasku kroz izotropičnu zapreminu (oblak).

zraka kroz unutrašnjost objekta koji predstavlja oblak dima i na tom putu može se odbiti u proizvoljnom smeru. Radi pojednostavljivanja izračunavanja interakcije zraka sa unutrašnjošću objekta i uniformnosti sa izračunavanjem interakcije zraka sa ostalim materijalima, unutrašnjost objekta se zamenjuje sa površi čije je postojanje u datom trenutku uslovljeno nekom funkcijom verovatnoće koja ima oblik:

$$p = C \cdot \Delta L \tag{2.6}$$

U formuli (2.6), p je verovatnoća odbijanja zraka u nasumičnom smeru u dužini ΔL kojom se prostire kroz zapreminu i zavisi od faktora gustine zapremine C. Što je faktor gustine zapremine veći, veća je i verovatnoća da će zrak u nekom trenutku promeniti smer. Takođe, što zrak duže putuje kroz zapreminu, ΔL ima veću vrednost pa je i verovatnoća veća da će se zrak odbiti dok putuje kroz zapreminu. Rešavanjem jednačine (2.6) može se odrediti tačno mesto unutar objekta na kojem se dogodila promena smera zraka.



Slika 2.19: Iscrtana scena sa dve kutije izotropičnog materijala [22][Slika 21].

Slika 2.19 ilustruje izgled objekata izotropičnog materijala. Beli kvadrat na plafonu predstavlja izvor svetlosti, a u sredini sobe nalaze se dve kutije izotropič-

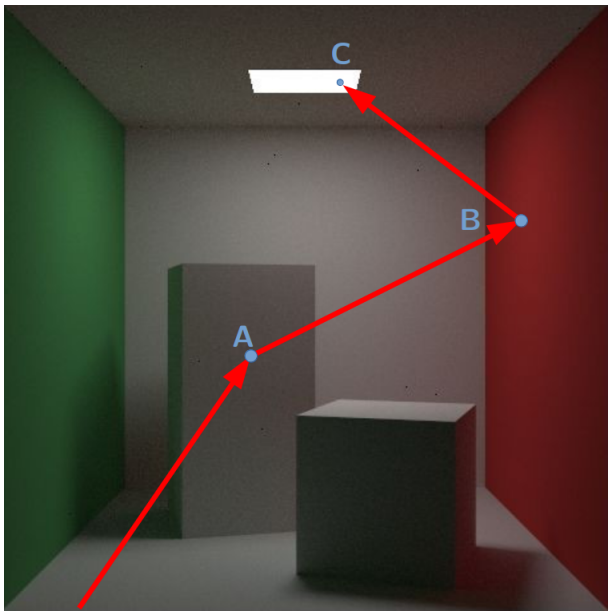
nog materijala koje se prilikom interakcije sa zrakom ponašaju kao oblak ili dim. Na podu se može videti senka svake kutije, a u isto vreme kutije izotropičnog materijala se provide. Deo scene koji se nalazi iza objekta izotropičnog materijala je mutniji nego kod providnih materijala jer se smer zraka može nasumično promeniti prilikom prolaska kroz izotropičan materijal.

Ovako definisan izotropičan materijal može se dodeliti bilo kom objektu, uz ograničenje da je oblik konveksan jer bi kod nekonveksnih oblika zrak mogao da prelazi između objekta i prostora u kome se objekat nalazi što bi znatno otežalo računanje verovatnoće po formuli (2.6). Izotropičan materijal može se dodeliti i nekonveksnom objektu ako se objekat podeli na više manjih, konveksnih objekata i svakom od njih se dodeli isti izotropičan materijal.

Izvor svetlosti

Komercijalni softveri za iscrtavanje tehnikom praćenja zraka implementiraju više tipova izvora svetlosti, svaki sa specifičnim osobinama kako bi bolje modelovali ponašanje svetla u različitim okruženjima.

U ovom radu implementirana je jedna vrsta izvora svetlosti koja se nalazi na konačnoj razdaljini od scene i emituje zrake svetlosti iz svoje celokupne površine. Objektu koji emituje svetlo dodeljuje se posebna vrsta materijala koja zaustavlja propagiranje zraka kroz scenu i dodeljuje mu boju koju emituje.



Slika 2.20: Ilustracija odbijanja zraka po sceni dok ne stigne do izvora svetlosti.

Na slici 2.20 ilustrovan je proces propagiranja zraka puštenog iz kamere koji se najpre odbije od kutije u tački A, zatim se odbije od zida u tački B da bi na kraju stigao do izvora svetlosti u tački C. Kada zrak stigne do izvora svetlosti tada se zaustavlja njegovo propagiranje po sceni i zraku se dodeljuje boja izvora svetlosti. Boja emitovanog svetla koja će biti dodeljena zraku izračunava se na osnovu boje koju materijal objekta ima u tački preseka zraka i njegove površine. To može biti jedna boja, na primer bela kao na sceni sa slike 2.20, ali može biti i tekstura. Jedina razlika između objekta koji jeste izvor svetlosti i objekta koji nije izvor svetlosti je što se propagiranje zraka zaustavlja kada zrak stigne do objekta koji je izvor svetlosti na sceni. Postoji i hibridna vrsta objekata koji u isto vreme emituju zrak svetlosti i odbijaju dolazeće zrake, ali se oni ređe koriste.

2.5 Boja zraka

Ovo poglavlje objašnjava kako funkcija `boja_zraka` propagira zrak kroz scenu dok ne dođe do izvora svetlosti i zatim na osnovu materijala svih objekata od kojih se zrak odbio izračunava njegovu boju. Funkcija se poziva u algoritmu 2.1 kako bi se odredila boja piksela kroz koji je zrak propušten.

```
1  boja_zraka(zrak, scena) -> boja_piksela {
2    pogodak = scena.pogodjena(zrak)
3    if (!pogodak) {
4      return scena.pozadina()
5    }
6
7    e = pogodak.materijal.emituje();
8    odbijanje = pogodak.materijal.odbij_zrak(zrak)
9    if (!odbijanje) {
10     return e
11   }
12
13   a = odbijanje.boja
14   z = odbijanje.zrak
15   return e + a * boja_zraka(z, scena)
16 }
```

Algoritam 2.3: Funkcija koja određuje boju zraka puštenog ka sceni.

U funkciji `boja_zraka` u algoritmu 2.3 parametar `zrak` sadrži podatke o smeru zraka i njegovoj početnoj tački dok parametar `scena` sadrži sve objekte i dodatne

podatke o sceni. Povratna vrednost funkcije je boja koja će biti upisana u piksel konačne slike kroz koji je `zrak` pušten. Boja piksela je vektor sa tri komponente iz intervala $[0, 1]$. Komponente vektora određuju redom vrednost crvene, zelene i plave boje. Kombinacijom intenziteta ove tri boje dobija se širok spektar boja koji se naziva model boja RGB (crvena, zelena, plava) [23].

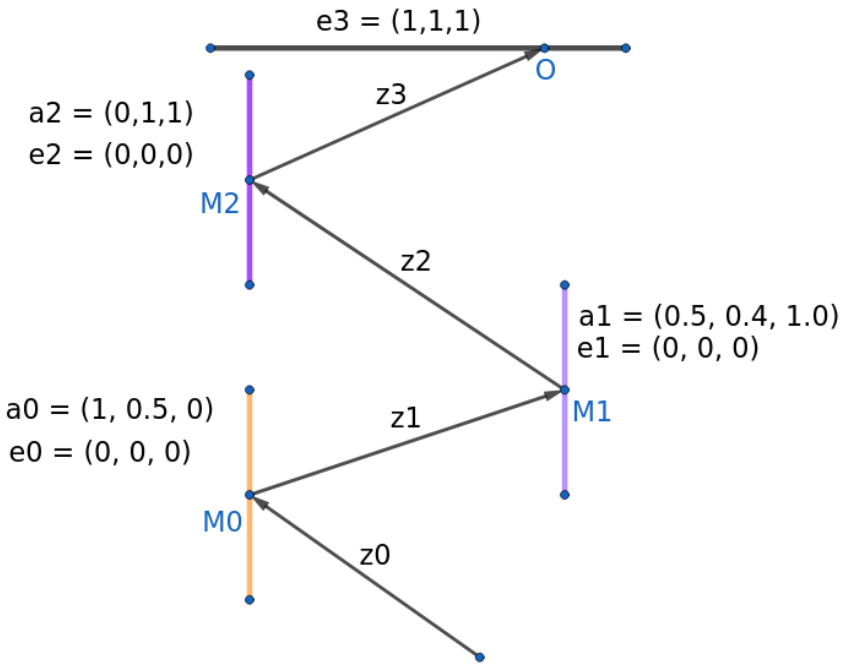
U liniji 2 algoritma 2.3, poziv funkcije `scena.pogodjena` pronalazi objekat na sceni koji je najbliži početnoj tački zraka, takav da taj objekat i zrak imaju presek; u stvari važno je da presečna tačka bude najbliža početku zraka. Pronalaženje objekta koji zrak pogađa se efikasno izračunava pomoću *prostornih struktura podataka* (eng. spatial data structure) koje pametno particionišu prostor na delove za koje je se u nekoliko koraka može izračunati da li zrak kroz njih prolazi. Prostor je rekurzivno particionisan tako da je složenost pronalaženja objekata koji zrak pogađa najčešće $O(\log(n))$, a u najgorem slučaju iznosi $O(n)$, gde je n broj objekata na sceni. Presek zraka i objekta na sceni izračunava se samo za one objekte koji se nalaze u podprostoru kroz koji zrak prolazi. Povratna vrednost ove funkcije je struktura koja sadrži sve podatke o pogotku koji će biti potrebni za izračunavanje boje i odbijanja zraka.

U liniji 3 algoritma 2.3, proverava se da li zrak pogađa neki objekat na sceni. Ako pogotka nije bilo, u liniji 4 algoritma 2.3 vraća se boja pozadine scene. Ako je scena zatvoreni prostor, onda se ovaj slučaj nikada ne dešava jer će zrak sigurno pogoditi neki objekat na sceni, a u slučaju otvorenog prostora obično se konstruiše objekat koji obuhvata scenu sa svih strana na koju se nalepi tekstura pozadine. Ako zrak pogodi pozadinu, onda se propagiranje zraka zaustavlja.

Podaci o materijalu pogođenog objekta upisani su u polje `pogodak.materijal`. Ako je objekat izvor svetlosti, u liniji 7 algoritma 2.3 poziv funkcije `emituje` vraća boju koju objekat emituje, u suprotnom objekat nije izvor svetlosti i funkcija `emituje` vraća crnu boju. U liniji 8 algoritma 2.3, poziv funkcije `odbij_zrak` izračunava smer zraka nakon interakcije sa pogođenim objektom. Specifičnosti interakcije zraka sa različitim vrstama materijala objašnjene su u poglavlju 2.4.

Propagiranje zraka kroz scenu nakon interakcije sa objektom se zaustavlja ako je objekat izvor svetlosti ili ako je objekat od materijala koji upija svetlost. Ako je objekat izvor svetlosti, onda je emitovao zrak koji je do njega stigao i tada se emitovana boja upisuje u promenljivu `e`. Ako objekat nije izvor svetlosti i nije bilo odbijanja, to znači da je materijal potpuno upio zrak i tada promenljiva `e` ima vrednost crne boje. U linijama 13 i 14 algoritma 2.3, promenljiva `a` je boja zraka

nakon odbijanja, a promenljiva `z` zrak nakon interakcije sa materijalom objekta. Funkcija `boja_zraka` se zatim poziva rekurzivno dok odbijeni zrak `z` ne stigne do izvora svetlosti.



Slika 2.21: Skica odbijanja zraka po sceni dok ne stigne do izvora svetlosti.

Na slici 2.21 ilustrovano je propagiranje zraka kroz scenu. Na sceni se nalaze tri objekta `M0`, `M1` i `M2` različitih materijala. Na scenu je pušten zrak `z0` koji dopire do izvora svetlosti u tački `O` odbijajući se redom od objekata `M0`, `M1` i `M2`. Vrednosti `e0`, `e1`, `e2` i `a0`, `a1`, `a2` su vrednosti promenljivih `e` i `a` iz funkcije `boja_zraka` redom za nulti, prvi i drugi rekurzivni poziv. Vrednosti emitovane svetlosti `e0`, `e1` i `e2`, iz objekata `M0`, `M1` i `M2` su nula vektori, koji predstavljaju crnu boju, jer nijedan od objekata nije izvor svetlosti. Poslednji objekat sa kojim se zrak sudara u tački `O` je izvor bele svetlosti i zbog toga je `e3 = (1,1,1)`.

Za zrak `z0`, izraz u liniji 15 u algoritmu 2.3 imaće oblik:

```
(0,0,0) + (1, 0.5, 0) * boja_zraka(z1, scena)
```

Kada se rekurzivni pozivi razmotaju do izvora svetlosti u tački O, ceo izraz se može zapisati kao

```
e0 + a0 * (e1 + a1 * (e2 + a2 * e3)).
```

U poslednjem koraku rekurzivni poziv za zrak `z3` vraća vrednost emitovanog svetla `e3` u liniji 10, iz izvora svetlosti u tački `O`. Svi `e0`, `e1`, `e2` su bili nula vektori jer nijedan od objekata do tačke `O` nije emitovao svetlo. Na kraju, konačna vrednost boje piksela za dati `zrak` i `scenu` biće `(0, 0.2, 0)`.

U algoritmu 2.3 izostavljeni su implementacioni detalji radi jednostavnosti prikaza glavnih delova algoritma. Implementacija data u poglavlju 4 proširuje algoritam 2.3 tako što dodaje ograničenje dubine rekurzivnog poziva jer se zrak u trenutnoj definiciji funkcije `boja_zraka`, u teoriji, može beskonačno puta odbijati po sceni ako na svom putu nema nijedan izvor svetlosti ili pozadinu scene. Takođe, zbog ograničenosti veličine steka svake niti na platformi Nvidia CUDA, implementacija algoritma 2.3 je za ovu platformu iterativna, gde broj iteracija odgovara ograničenju dubine rekurzivnih poziva.

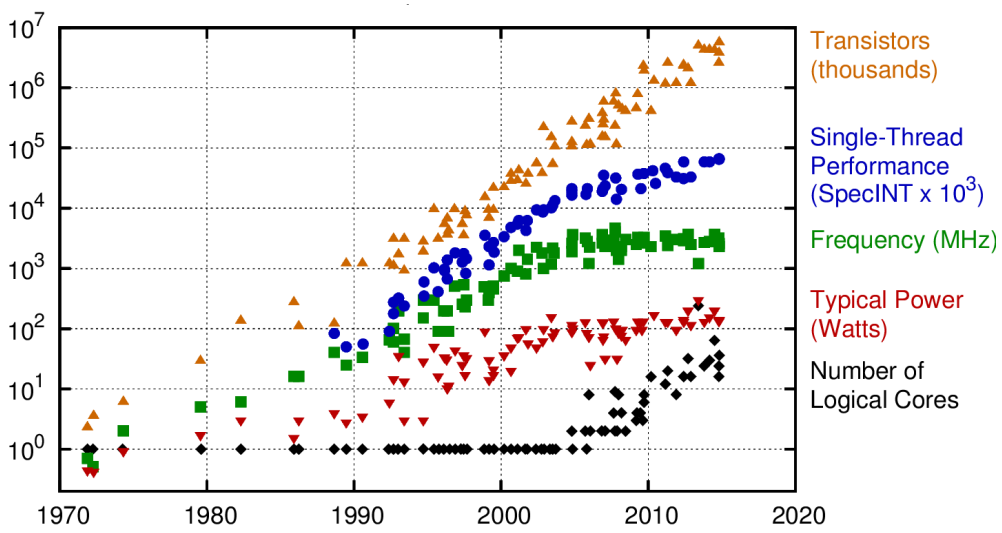
Glava 3

Platforma Nvidia CUDA

U ovom poglavlju biće opisan osnovni princip dizajna i namene platforme CUDA. Poglavlje 3.1, o heterogenom izračunavanju, se fokusira na širu sliku paralelnog izračunavanja i mesto platforme CUDA u modernom računarskom sistemu. Sledeće poglavlje 3.2 predstavlja programski interfejs i model kompilacije programa pisanih za platformu CUDA i predstavlja pogleda na platformu iz perspektive korisnika. Poglavlja 3.3 i 3.4 na logičkom nivou opisuju model izračunavanja i model memorije platforme CUDA, a poslednje poglavlje 3.5 opisuje konkretan pristup paralelizaciji tehnike praćenja zraka na platformi CUDA.

3.1 Heterogeno izračunavanje

Računari su prvobitno imali samo jednu centralnu procesorsku jedinicu (eng. **c**entral **p**rocessing **u**nit (CPU)), u daljem tekstu procesor, koja je najčešće bila opšte namene. Poboljšanje performansi računarskih sistema je decenijama dolazilo od povećanja frekvencije, broja tranzistora i poboljšanja dizajna samog procesora koji je imao jedno jezgro za izvršavanje instrukcija.

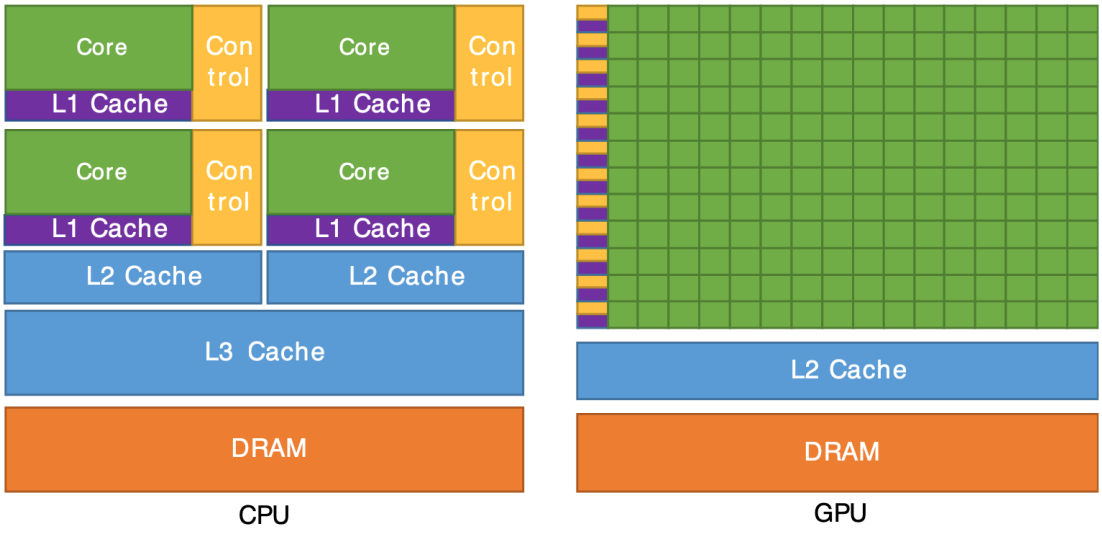


Slika 3.1: Četrdeset godina razvoja mikroprocesora [9].

Slika 3.1 prikazuje trend rasta glavnih pokazatelja performansi mikroprocesora. Kada je trend rasta brzine jednog jezgra počeo da usporava, proizvođači mikroprocesora su preusmerili napore u pakovanje više logičkih jezgara u jedan mikroprocesor. Početkom 2008. godine na tržištu počinje da se pojavljuje sve više procesora sa više jezgara opšte namene koja mogu izvršavati instrukcije paralelno.

Pored centralne procesorske jedinice, moderni sistemi imaju i grafičku procesorsku jedinicu (eng. **g**raphics **p**rocessing **u**nit (GPU)), u daljem tekstu grafička jedinica, koja može biti integrisana u sam procesor ili odvojena komponenta računarskog sistema sa kojom procesor komunicira preko magistrale.

Na slici 3.2 ilustrovana je razlika u arhitekturi jezgara procesora i jezgara grafičkih jedinica. Na levom delu slike 3.2 nalazi se dijagram arhitekture centralne procesorske jedinice (CPU) na kojem zeleni pravougaonik sadrži aritmetičko-logičku jedinicu jezgra procesora, a žuti pravougaonik predstavlja kontrolnu jedinicu. Na desnom delu slike 3.2 nalazi se dijagram arhitekture grafičke jedinice (GPU) na kojem žuti pravougaonik predstavlja kontrolnu jedinicu, a zeleni pravougaonik

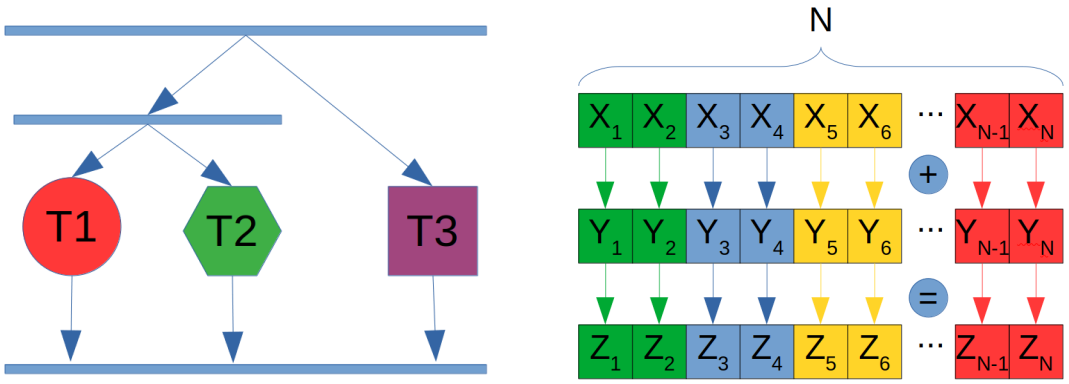


Slika 3.2: Arhitektura procesora (CPU) i grafičke jedinice (GPU) [14][Slika 1].

jedinicu izračunavanja (eng. compute unit). Na desnom dijagramu na slici 3.2 kontrolne jedinice i jedinice izračunavanja poređane su u redove. U svakom redu se nalazi jedna kontrolna jedinica koja upravlja svim jedinicama izračunavanja iz tog reda. Jedna kontrolna jedinica u procesoru upravlja jednom aritmetičko-logičkom jedinicom, a kod grafičke jedinice jedna kontrolna jedinica upravlja sa više jedinica izračunavanja.

Jezgra procesora su „teška" i dizajnirana su za izračunavanja složenih logika kontrole toka programa i izvršavanja sekvencijalnih programa. Jezgra grafičkih jedinica su „lagana" i optimizovana su za paralelno izvršavanje istog skupa instrukcija, sa jednostavnom kontrolom toka programa, nad velikim skupom podataka, sa fokusom na visok stepen protočnosti [2]. Najnovije generacije procesora imaju desetine jezgara [5], dok grafičke jedinice imaju na hiljade [18]. Grafičke jedinice su se uglavnom koristile za ubrzavanje iscrtavanja tehnikom rasterizacije, ali se ubrzo uvidelo da se mogu koristiti i za izračunavanja opšte namene koja su pogodna za paralelizaciju.

Prema načinu obrade podataka paralelizacija se grubo može podeliti na *paralelizam zadataka* (eng. task parallelism) i *paralelizam podataka* (eng. data parallelism) čiji su dijagrami načina obrade podataka ilustrovani na slici 3.3. Kod paralelizma zadataka više različitih zadataka izvršava se istovremeno, dok se kod paralelizacije podataka jedna operacija istovremeno primenjuje nad istim podacima.



(a) Paralelizam zadataka. (b) Paralelizam podataka.

Slika 3.3: Konceptualni dijagrami paralelizma zadataka i paralelizma podataka.

Paralelizam zadataka je istovremeno izvršavanje više različitih zadataka nekog programa. Na slici 3.3(a) zadaci T1, T2, T3 su različiti zadaci koji se u sistemu izvršavaju istovremeno. Na primer, program za obradu teksta može istovremeno izvršavati proveru pravopisa, iscrtavanje teksta, predloge rečenica i iscrtavanje grafičkog korisničkog interfejsa. Svaki od ovih zadataka može biti dodeljen jednoj niti. Skaliranjem broja niti ne skaliraju se performanse ovako dizajniranog sistema ukoliko broj niti prevazilazi broj zadataka. Takođe, ukoliko nekom od zadataka treba više vremena da izvrši neku operaciju, ne postoji način da se ona ubrza dodavanjem još jezgara u sistem bez izmene dizajna programa.

Kod paralelizma podataka jedna operacija se izvršava paralelno nad nekim skupom podataka. Postoji više načina da se podaci podele između paralelnih jezgara. Na slici 3.3(b) ilustrovana je jedna podela pri paralelizaciji izračunavanja zbira dva vektora $Z = X + Y$. Svakoj niti dodeli se blok podataka dužine N/k gde je N broj komponenti vektora, a k broj niti. Blokovi različitih boja izvršavaju se paralelno. Ako su vektori dužine N i na raspolaganju je N niti procesora, onda se sabiranje dva vektora dužine N može izvršiti u složenosti $O(1)$.

Arhitektura procesora sa slike 3.2 je pogodnija za izvršavanje paralelizma zadataka jer svako jezgro procesora ima sopstvenu kontrolnu jedinicu i može paralelno izvršavati različite zadatke. Sa druge strane, arhitektura grafičke jedinice sa slike 3.2 je pogodnija za paralelizam podataka jer ima više jedinica izračunavanja koje mogu paralelno izvršavati istu operaciju, na primer operaciju sabiranja, nad većim skupom podataka.

Dok su procesori dizajnirani za primenu nad širokim skupom problema i imaju zadovoljavajuće performanse u obe vrste paralelizma, grafičke jedinice daleko

nadmašuju performanse procesora u zadacima koji su pogodni za paralelizaciju podataka. Da bi se maksimalno iskoristile mogućnosti računarskih sistema, aplikacije pored procesora koriste i grafičke jedinice. Istovremeno korišćenje ne samo više jezgara jednog mikroprocesora, već i više različitih arhitektura mikroprocesora naziva se *heterogeno izračunavanje*.

Jedan primer programa koji maksimalno koristi resurse procesora i grafičke jedinice su video igre. Da bi se iscrtalo stanje igrice u nekom trenutku potrebno je da sistem reaguje na ulaz korisnika, ažurira stanja fizike sveta, sistema veštačke inteligencije, stanja entiteta, događaja i onda izda naredbe grafičkoj jedinici za iscrtavanje sveta. Za dobar doživljaj, igrica ove operacije mora da ponovi barem 60 puta u sekundi. Da bi se postigle željene performanse, izračunavanja se raspoređuju po jezgrima procesora i grafičke jedinice. Deo resursa grafičke jedinice koristi se za iscrtavanje scene, a ostatak se koristi za ubrzavanje izračunavanja potrebnih ostatku sistema.

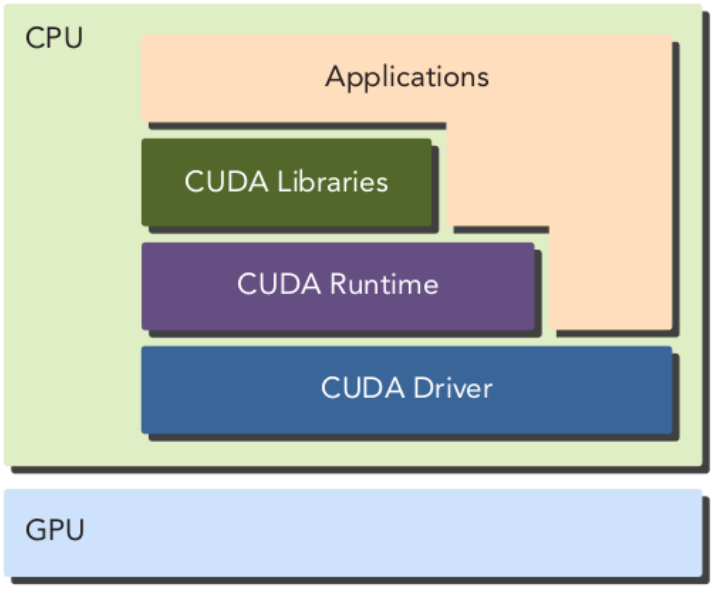
Veliki broj današnjih dostignuća u oblastima veštačke inteligencije, računarske grafike, naučnog izračunavanja, medicini i bilo kojoj oblasti koja se oslanja na izračunavanja visokih performansi (eng. **h**igh-**p**erformance **c**omputing - HPC) postignuta su zahvaljujući napretku hardverskih komponenti za paralelna izračunavanja i korišćenjem heterogenog izračunavanja.

3.2 Nvidia CUDA

CUDA (eng. **c**ompute **u**nifed **d**evice **a**rhitecture) je platforma za heterogeno izračunavanje opšte namene koja stavljajući na raspolaganje programski interfejs (eng. application programming interface (API)) za kontrolu grafičke jedinice i komunikaciju sa ostatkom sistema olakšava pisanje aplikacija koje koriste paralelno izračunavanje. Platforma CUDA ima slojevitu arhitekturu koja omogućava visok stepen kontrole nad grafičkom jedinicom uz interfejs niskog nivoa i produktivnost pomoću biblioteka koje implementiraju najčešće korišćene algoritme iz oblasti naučnog i grafičkog izračunavanja i programskog interfejsa visokog nivoa. Sadrži bogat skup alata za ispitivanje performansi koda, pronalaženje grešaka, kompiliranje, detaljnu dokumentaciju kao i literaturu sa mnoštvom primera koji demonstriraju različite mogućnosti platforme.

Programski interfejs

Platforma CUDA se može direktno koristiti u aplikacijama pisanim u programskim jezicima C, C++, Fortran i Python pomoću jezičkih proširenja specifičnih za platformu CUDA. Izračunavanja na grafičkoj jedinici mogu se pokrenuti korišćenjem biblioteke ili programskog interfejsa višeg i nižeg nivoa u zavisnosti od potreba aplikacije.



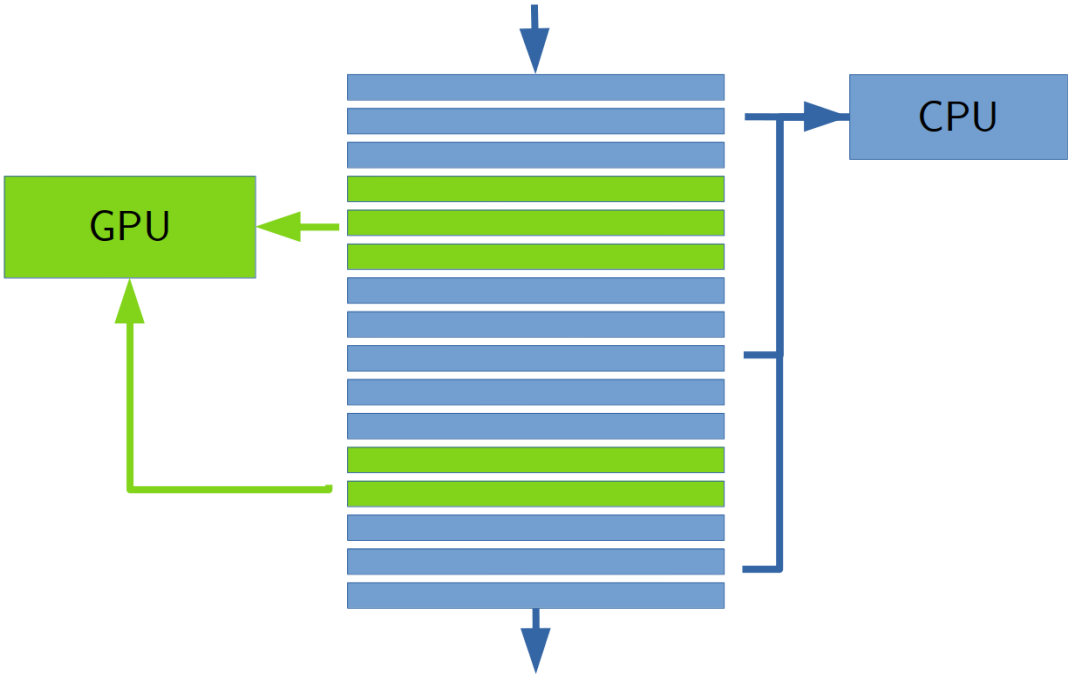
Slika 3.4: Dijagram nivoa programskog interfejsa platforme CUDA [2].

Na slici 3.4 ilustrovana je slojevitost interfejsa za programiranje grafičke jedinice u kome je svaki sloj platforme CUDA implementiran preko nižeg sloja. Na najvišem nivou nalazi se standardna biblioteka CUDA koja implementira često korišćene funkcionalnosti platforme sa ciljem lakoće korišćenja. Dva nivoa programskog interfejsa preko kojih se može upravljati resursima grafičke jedinice i kontrolisati izvršavanje programa su CUDA *drajverski* API (eng. CUDA driver API) i CUDA *izvršni* API (eng. CUDA runtime API). Drajverski programski interfejs platforme CUDA je nižeg nivoa i komplikovaniji za korišćenje, ali nudi veću kontrolu nad grafičkom jedinicom, dok je izvršni programski interfejs višeg nivoa i implementiran je preko drajverskog programskog interfejsa [2].

Slojevitost programskog interfejsa omogućava programerima da odaberu adekvatan nivo apstrakcije za rešavanje problema. Viši slojevi su lakši za korišćenje, ali pružaju manji stepen kontrole nad resursima grafičke jedinice, dok su niži slojevi složeniji za korišćenje, ali pružaju veći stepen kontrole.

Izvršavanje

Ideja heterogenog izračunavanja nije da zameni izračunavanje na procesoru već da rasporedi izračunavanja na arhitekturu na kojoj se mogu najefikasnije izvršiti.



Slika 3.5: Raspoređivanje izračunavanja na odgovarajuću arhitekturu [2].

Ako su podaci male veličine, složene logike sa paralelizmom na niskom nivou, onda je CPU bolji izbor za obradu takve vrste podataka jer je dizajniran da efikasno izvršava kompleksnu logiku kontrole toka i paralelizma na nivou instrukcija. Sa druge strane, ukoliko je potrebno obraditi ogromne količine podataka i ako je izračunavanje moguće paralelizovati na hiljadama jezgara, onda je GPU bolji izbor zbog velikog broja jezgara dizajniranih da efikasno izvršavaju operacije nad velikim skupom podataka paralelno. Na primer, komplikovana logika parsiranja datoteke koja sadrži izvorni kôd nekog programskog jezika efikasnije se izvršava na procesoru, a primena filtera na milione slika se najefikasnije može izvršiti na grafičkoj jedinici zbog pogodnosti paralelizacije množenja velikog broja matrica na koje se primena filtera svodi.

Na slici 3.5 ilustrovan je tok izvršavanja programa pisanih za platformu CUDA. Svaki pravougaonik na sredini predstavlja jednu logičku celinu programa. Delovi obojeni u plavo su glavni delovi programa koji će se izvršiti na procesoru, a delovi obojeni u zeleno su paralelizovani delovi koji će se izvršiti na grafičkoj jedinici.

Glavni deo programa može izdati i više naredbi za izračunavanje grafičkoj jedinici. Ti zadaci biće poređani u red za čekanje i izvršavaće se redom kojim su zadati. Program se sada izvršava na više arhitektura istovremeno tako što su paralelna izračunavanja prepuštena grafičkoj jedinici.

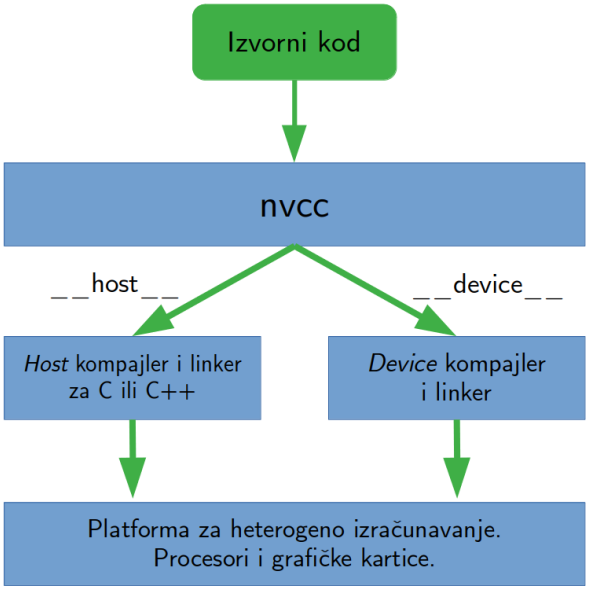
Aplikacije pisane za platformu CUDA podeljene su na delove koji se izvršavaju na procesoru i delove koji se izvršavaju na grafičkoj jedinici. Pomoću platforme CUDA aplikacija komunicira sa grafičkom jedinicom kao sa servisom koji omogućava skaliranje paralelnih zadataka. Podelu na deo koji se izvršava na procesoru i deo koji se izvršava na grafičkoj jedinici radi sam programer kroz pisanje programa. Platforma CUDA samo olakšava skaliranje i implementaciju određenih delova paralelnih izračunavanja, ali ih ne izvodi automatski.

Kompilacija

Integralni deo platforme CUDA je kompajler `nvcc` (eng. Nvidia CUDA compiler) koji izvorni kôd datog programa kompilira u izvršivu datoteku. Model kompilacije programa pisanih za platformu CUDA razlikuje se od standardnog modela kompilacije programa po tome što kompajler `nvcc` izvorni kôd programa deli na kôd koji je pisan za grafičku jedinicu i kôd koji je pisan za sistem čiji je grafička jedinica sastavni deo.

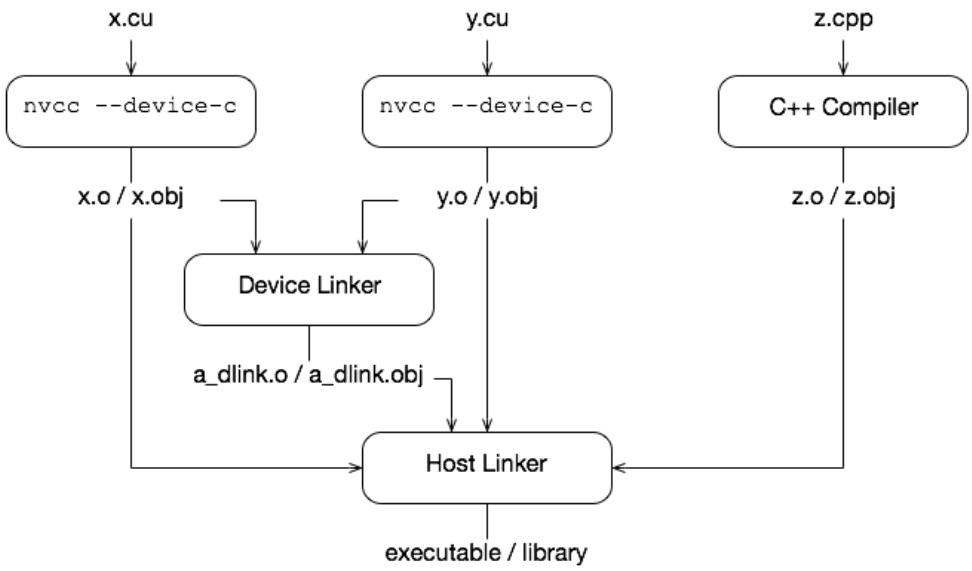
Na platformi CUDA grafička jedinica naziva se *uređaj* (eng. device), a glavni procesor koji kontroliše rad grafičke jedinice naziva se *matični procesor* (eng. host). Kôd koji se izvršava na procesoru naziva se *matični kôd* (eng. host code), a memorija kojoj procesor ima direktan pristup kao što su registri, keš memorija procesora i radna memorija naziva se *matična memorija* (eng. host memory). Kôd koji se izvršava na grafičkoj jedinici naziva se *kôd uređaja* (eng. device code), a memorija alocirana pomoću programskog interfejsa platforme CUDA i memorija kojoj grafička jedinica ima direktan pristup naziva se *memorija uređaja* (eng. device memory). Definicije i deklaracije funkcija dopunjuju se specijalnim anotacijama kako bi se kompajleru naglasilo za koji deo sistema se kôd kompilira. Anotacija `__device__` označava funkciju koja će se izvršiti na grafičkoj jedinici, a anotacija `__host__` označava funkciju koja će se izvršiti na procesoru. Ukoliko funkcija nije označena, podrazumevano ima anotaciju `__host__`.

Na slici 3.6 ilustrovani su glavni delovi toka kompilacije izvornog koda koji kompajler `nvcc` deli na matični kôd napisan u matičnom programskom jeziku i kôd za uređaj napisan u proširenoj verziji matičnog programskog jezika za plat-



Slika 3.6: Podela kompilacije izvornog koda za platformu CUDA od strane kompajlera nvcc [8][slika 2-3].

formu CUDA. Matični kôd prepušta se kompatibilnom kompajleru na sistemu, na primer alatima `gcc` ili `clang`, a ostatak koda za grafičku jedinicu kompajler `nvcc` kompilira i linkuje zasebno. Na kraju se oba dela programa linkuju u izvršivu datoteku linkerom sa sistema na kom je program kompiliran.



Slika 3.7: Dijagram kompilacije programa pisanog u više datoteka za platformu CUDA [15][slika 4].

Programi za platformu CUDA mogu biti napisani u više datoteka. Izvorni kôd za platformu CUDA ima ekstenziju `.cu`. Na primer, neka je program napisan u datotekama `x.cu`, `y.cu` i `z.cpp`. U datotekama `x.cu` i `y.cu` nalazi se kôd napisan za grafičku jedinicu dok se u datoteci `z.cpp` nalazi kôd napisan za procesor. Datoteke sa ekstenzijom `.cu` često sadrže i matični kôd koji sadrži pozive funkcija koje se izvršavaju na grafičkoj jedinici koje će biti detaljnije objašnjene u glavi 3.3. Nakon pokretanja komande za kompilaciju programa `nvcc x.cu y.cu z.cpp`, kompilacija se razdvaja po datotekama kao na slici 3.7. Kompajler za uređaj kompilira datoteke `x.cu` i `y.cu`, a matični kompajler kompilira datoteku `z.cpp`. Izlaz iz prvog koraka kompilacije su objektne datoteke `x.o`, `y.o`, `z.o` redom za datoteke sa izvornim kodom `x.cu`, `y.cu` i `z.cpp`. Objektne datoteke za grafičku jedinicu linkuju se pomoću linkera za uređaj i rezultat je `a_dlink.o` datoteka. Na kraju, sve objektne datoteke linkuju se pomoću matičnog linkera u izvršivu datoteku.

Kompajler `nvcc` razdvaja kompilaciju programa za procesor i grafičku jedinicu kako bi se funkcije kompilirale za odgovarajuću arhitekturu i dodale instrukcije na mestima na kojima matični kôd poziva kôd za uređaj. Razdvajanjem kompilacije, platforma CUDA može koristiti kompajler sa sistema za kompilaciju koda za procesor.

3.3 Model izračunavanja

Programi pisani za platformu CUDA se istovremeno izvršavaju na grafičkoj jedinici i procesoru. CUDA drajver upravlja paralelizacijom, redosledom izvršavanja i dodelom resursa apstrahovanjem fizičkog uređaja u virtuelnu mrežu jezgara, a procesor može izdati naredbu za započinjanje izračunavanja na grafičkoj jedinici pozivom *kernel* (eng. kernel) funkcije.

Kernel poziv

Izvršavanje programa uvek počinje na procesoru od funkcije `main`, a paralelna izračunavanja na grafičkoj jedinici se pokreću pozivom posebnog tipa funkcije koja se naziva kernel. Definicija kernel funkcija označava se anotacijom `__global__` koja služi kao indikacija kompajleru da je kôd ove funkcije potrebno kompilirati za grafičku jedinicu i da je na mestu njenog poziva potrebno ubaciti posebne naredbe i pozive programskog interfejsa platforme CUDA koje pripremaju i pokreću kernel.

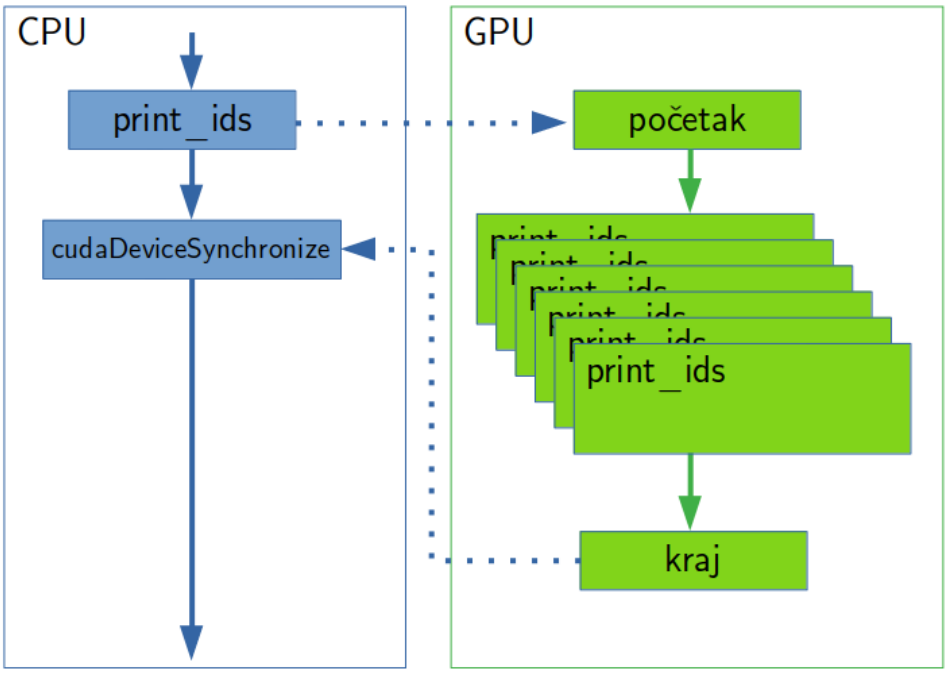
Kernel funkcije mogu se pozivati samo unutar matičnog koda i sintaksno imaju sličan oblik pozivu običnih funkcija, sa dodatkom posebnih zagrada `<<<...>>>` nakon naziva funkcije, a pre operatora poziva `()`. Vrednosti koje se prosleđuju unutar zagrada `<<<...>>>` su parametri koji određuju stepen paralelizacije kernel funkcije i biće detaljnije objašnjeni kasnije.

```
1  __global__
2  void print_ids() {
3    printf("(B:(%d, %d), T:(%d, %d))\n",
4           blockIdx.x, blockIdx.y,
5           threadIdx.x, threadIdx.y);
6  }
7
8  int main() {
9    dim3 blocks_dim(3, 2);
10   dim3 threads_dim(5, 3);
11   print_ids<<<blocks_dim, threads_dim>>>();
12   cudaDeviceSynchronize();
13   return 0;
14 }
```

Program 3.1: Primer poziva kernel funkcije.

U programu 3.1 prikazan je primer poziva kernel funkcije `print_ids` koja će se izvršiti na grafičkoj jedinici. Funkcija `main`, programa 3.1 započinje izvršavanje na procesoru i u trenutku poziva funkcije `main` grafička jedinica nije aktivna. Kada kontrola toka programa stigne do linije 11 u kojoj se nalazi poziv kernel funkcije `print_ids`, CUDA drajver instancira poziv kernel funkcije na grafičkoj jedinici, a izvršavanje funkcije `main` se nastavlja ne čekajući kraj izvršavanja funkcije `print_ids`. U liniji 12, programa 3.1, funkcija `cudaDeviceSynchronize()` blokira dalje izvršavanje na procesoru dok se ne završe sve instancirane kernel funkcije. Pozivi kernel funkcija su podrazumevano asinhroni, što znači da pozivalac funkcije ne čeka kraj izvršavanja kernel funkcije. Kada ne bi postojao poziv funkcije `cudaDeviceSynchronize()` u liniji 12 programa 3.1, glavni deo programa bi se sigurno završio pre kernel funkcije `print_ids`.

Na slici 3.8 ilustrovan je tok komunikacije između procesora i grafičke jedinice prilikom poziva kernel funkcije `print_ids` i funkcije CUDA drajvera. Plave strelice predstavljaju tok izvršavanja programa na procesoru, a zelene tok izvršavanja na grafičkoj jedinici. Strelice sa isprekidanom linijom predstavljaju operacije CUDA



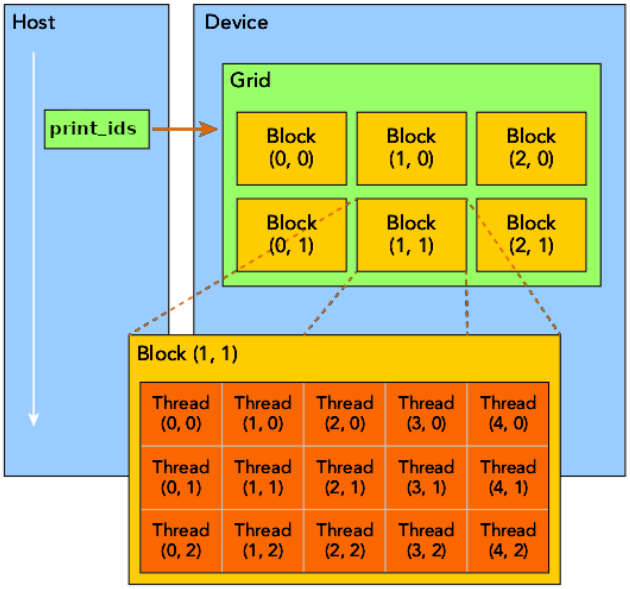
Slika 3.8: Dijagram toka izvršavanja poziva funkcije print_ids iz programa 3.1.

drajvera koji posreduje komunikaciji između procesora i grafičke jedinice. Poziv kernel funkcije `print_ids`, CUDA drajver instancira na grafičkoj jedinici i započinje njeno paralelno izvršavanje koje je na slici 3.8 predstavljeno sa više zelenih pravougaonika sa nazivom `print_ids`. Dok se instance kernel funkcije izvršavaju na grafičkoj jedinici, na matičnom procesoru bi se sve operacije koje se nalaze između poziva funkcije `print_ids` i poziva funkcije `cudaDeviceSynchronize` izvršavale nezavisno od grafičke jedinice. Poziv funkcije `cudaDeviceSynchronize` blokira izvršavanje niti matičnog procesora sa koje je pozvana dok ne stigne signal od CUDA drajvera da su završene operacije na grafičkoj jedinici, nakon čega se izvršavanje na matičnom procesoru može nastaviti.

CUDA drajver pozive kernel funkcija izvršava u redosledu u kojem su izdate sa procesora. Ukoliko postoji dovoljno resursa hardvera, CUDA drajver će sve pozvane kernel funkcije instancirati na fizička jezgra grafičke jedinice, u suprotnom, drajver će sačekati da dovoljno resursa hardvera postane slobodno kako bi započeo izvršavanje sledeće kernel funkcije.

Blokovi i niti

Fizička jezgra grafičke jedinice apstrahovana su u virtuelnu mrežu *blokova* (eng. block) i *niti* (eng. threads) koji čine osnovne gradivne elemente paralelizma platforme. Mreže niti i blokova mogu imati oblik niza, matrice ili kvadra. CUDA drajver korišćenjem argumenata prosleđenih u zagradama `<<<...>>>` automatski paralelizuje poziv kernel funkcije `print_ids` tako što na svaku nit, u svakom bloku, instancira po jedan poziv kernel funkcije `print_ids`. Prvi argument u zagradama `<<<blocks_dim, threads_dim>>>` određuje dimenzije mreže blokova, a drugi argument određuje dimenzije mreže niti unutar svakog bloka.



Slika 3.9: Struktura virtuelnih blokova i niti na platformi CUDA [2][slika 2-4].

Na slici 3.9, plavi pravougaonik sa oznakom `Host` predstavlja sekvencijalni tok izvršavanja programa na procesoru, dok plavi pravougaonik sa oznakom `Device` predstavlja grafičku kraticu na kojoj će se paralelizovati poziv kernel funkcije `print_ids`. U liniji 9, programa 3.1, promenljivom `blocks_dim` definisane su dimenzije mreže blokova, a u liniji 10, promenljivom `threads_dim` dimenzije mreže niti unutar svakog bloka. Na slici 3.9 zeleni pravougaonik sa oznakom `Grid` sadrži mrežu blokova dimenzije 3×2 kao što je definisano promenljivom `blocks_dim`, a unutar svakog bloka nalazi se mreža niti dimenzija 5×3 kao što je definisano promenljivom `threads_dim`. Na grafičkoj jedinici biće instancirano ukupno $3 \cdot 2 \cdot 5 \cdot 3 = 90$ paralelnih poziva kernel funkcije `print_ids` jer se poziv kernel funkcije instancira na svakoj niti unutar svakog bloka.

Blok i nit na kojoj se kernel funkcija izvršava mogu se jedinstveno identifikovati unutar tela same kernel funkcije pomoću specijalnih konstanti `blockIdx` za blok i `threadIdx` za nit unutar bloka. Pozicija bloka unutar mreže određena je sa `x` i `y` koordinatama konstante `blockIdx`. Slično, pozicija niti unutar bloka određena je sa `x` i `y` koordinatama konstante `threadIdx`. Na primer, na slici 3.9, u pozivu funkcije `print_ids`, unutar bloka `(1,1)` i niti `(4,2)`, `x` i `y` polja konstanti `blockIdx` i `threadIdx` imaće vrednosti:

```
blockIdx.x = 1, blockIdx.y = 1, threadIdx.x = 4, threadIdx.y = 2.
```

Specijalne konstantne `blockIdx` i `threadIdx` imaju i treću koordinatu `z` čija je vrednost 0 u programu 3.1 jer mreže blokova i niti imaju oblik matrice. U slučaju da mreža ima oblik kvadra, onda će vrednost koordinate `z` u specijalnoj konstanti predstavljati treću dimenziju mreže, ali u implementaciji tehnike praćenja zraka u ovom radu neće biti potrebna.

Dakle, funkcija `print_ids` iz programa 3.1 na standarni izlaz ispisuje blok i nit nad kojom je instancirana. Funkcija `printf`, pozvana unutar funkcije `print_ids`, pripada CUDA biblioteci. Grafička jedinica nema standardni izlaz, ali programski interfejs platforme CUDA podržava poziv funkcije `printf` tako što njen izlaz čuva u internom baferu i u slučaju događaja koji bafer prazni, drajver platforme CUDA sadržaj bafera prebaci u radnu memoriju, pa zatim na standardni izlaz programa.

Arhitektura grafičke jedinice kompanije Nvidia je bazirana na nizu *više-nitnih protočnih multiprocesora* (eng. multithreaded streaming multiprocessor). Kada se iz matičnog koda pozove kernel funkcija, blokovi se dodele multiprocesorima na grafičkoj jedinici. Blokovi mogu biti dodeljeni multiprocesoru u proizvoljnom redosledu i niti iz različitih blokova ne mogu se međusobno sinhronizovati. Niti unutar bloka izvršavaju se paralelno i u zavisnosti od broja multiprocesora grafičke jedinice više blokova se takođe mogu izvršavati paralelno. Kada sve niti u jednom bloku završe izvršavanje novi blokovi se mogu pokrenuti na slobodnim multiprocesorima. Jedan multiprocesor može konkurentno izvršavati više blokova u zavisnosti od resursa grafičke jedinice koji su potrebni svakom bloku. Broj niti unutar bloka ograničen je arhitekturom grafičke jedinice. Niti unutar bloka grupisane su u *pakovanje* (eng. warp) od 16 ili 32 niti u zavisnosti od arhitekture grafičke jedinice. Sve niti u jednom pakovanju izvršavaju istu instrukciju. Ako kontrola toka unutar pakovanja divergira na dve grane, na primer u naredbi `if`, onda pakovanje niti izvršava obe grane, a ostale niti u pakovanju se pauziraju dok se grane ne

izvrše. Divergencija grana se može pojaviti samo kod niti iz istog pakovanja jer se različita pakovanja izvršavaju nezavisno bez obzira na razliku u kontroli toka.

Skaliranje

Paralelizaciju izračunavanja na grafičkoj jedinici radi platforma CUDA, a na korisniku je da odredi stepen paralelizma specificiranjem dimenzija blokova i niti nad kojim će se izvršavati kernel funkcija. Određivanje optimalnih dimenzija mreže blokova i niti nije trivijalno, jer se niti alociraju i oslobađaju po blokovima. Nit bloka postaje slobodna tek kada se računanja na svakoj od niti unutar bloka, kome nit pripada, završe.

Kada broj instanciranih poziva prevazilazi broj dostupnih fizičkih jezgara, CUDA drajver pravi redosled po kojem će se niti izvršavati na fizičkim jezgrima. Instancirani pozivi kernel funkcije se iz perspektive korisnika izvršavaju istovremeno dok stepen stvarnog paralelizma na samom hardveru zavisi od fizičkog broja jezgara. Virtuelizacija jezgara u vidu blokova i niti omogućava efikasno skaliranje prelaskom na hardver većih mogućnosti. Pokretanjem programa na hardveru sa većim brojem jezgara CUDA drajver instancira veći broj paralelnih izvršavanja kernel funkcije i time smanjuje ukupno vreme izvršavanja.

Ukoliko se program ne oslanja na specifičnost arhitekture uređaja kako bi ostvario neku operaciju, ovako dizajniran paralelizam omogućava lako skaliranje prelaskom na bolji hardver. Takođe, platforma CUDA omogućava paralelizaciju kernel funkcija nad više uređaja različitih arhitektura, tako da se program može izvršavati na sistemu sa jednim procesorom više grafičkih jedinica. Tema paralelizacije nad klasterom grafičkih jedinica prevazilazi opseg ovog master rada, ali je svakako nešto što je značajna tema u oblasti paralelnog izračunavanja.

3.4 Model memorije

Memorija heterogenih aplikacija deli se na matičnu memoriju i memoriju uređaja. Matična memorija se odnosi na registre i keš memoriju procesora, glavnu memoriju i disk računarskog sistema. Memorija uređaja pripada adresnom prostoru grafičke jedinice. Dva glavna aspekta modela memorije kod platforme CUDA su interfejs za upravljanje memorijom iz matičnog koda i nadogradnje programskog jezika za upravljanje pristupom memoriji iz koda uređaja. Matični kôd može alocirati memoriju na grafičkoj jedinici i vršiti transfer podataka iz glavne memorije u memoriju grafičke jedinice i obratno. Kôd za uređaj može do određene mere kontrolisati u kojoj vrsti memorije se čuvaju podaci kako bi se ubrzala izračunavanja njihovim efikasnijem rasporedom u memoriji.

Hijerarhija video memorije

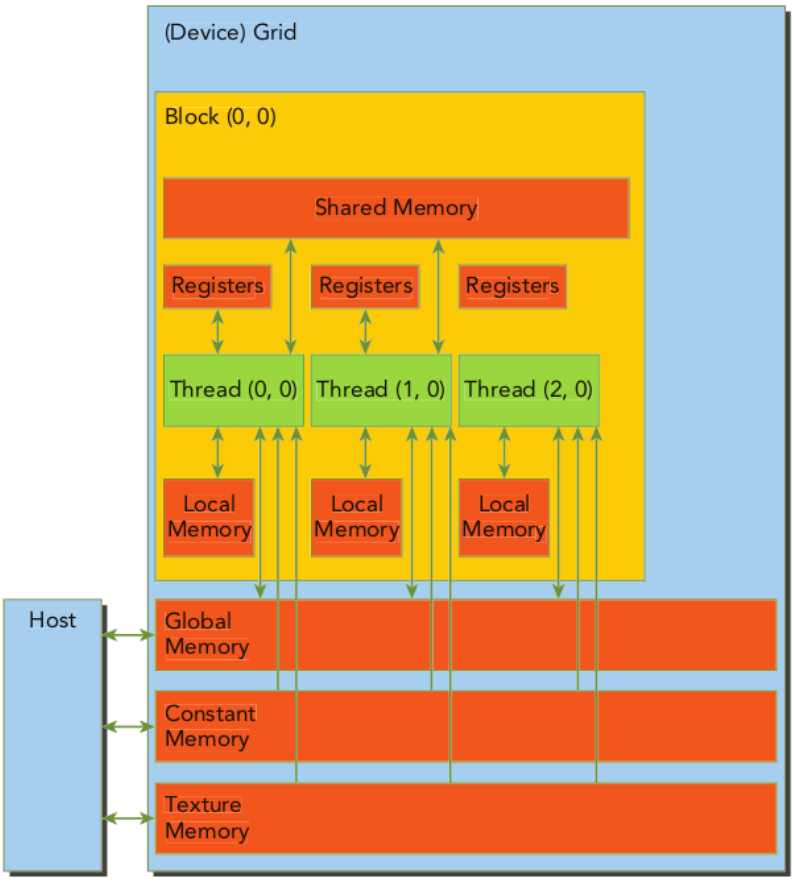
Grafičke jedinice imaju radnu memoriju koja se naziva i *video memorija* (eng. video memory). Kapacitet video memorije zavisi od same grafičke jedinice, a danas je u opsegu od nekoliko gigabajta do nekoliko desetina gigabajta. Za razliku od radne memorije računara koja je iz perspektive korisnika jedan veliki blok memorije, video memorija je podeljena u nekoliko tipova memorije posebne namene.

Aplikacije generalno ne pristupaju proizvoljnim delovima memorije u nekom trenutku u vremenu već prate princip lokalnosti po kome pristup memoriji može biti lokalan u prostoru ili lokalan u vremenu. Ako aplikacija pristupi nekoj lokaciji, onda princip lokalnosti kaže da je velika verovatnoća da će joj uskoro pristupiti ponovo. Lokalnost u prostoru znači da ukoliko aplikacija pristupi nekoj memorijskoj lokaciji, verovatno će pristupiti i nekoj susednoj memorijskog lokaciji. Poštujući ta dva principa, memorija na platformi CUDA je organizovana u hijerarhiju po brzini pristupa i stavlja na raspolaganje tipove memorije koje mogu iskoristiti princip lokalnosti kako bi ubrzale pristup potrebnim podacima i time skratile vreme izvršavanja.

Na računarskom sistemu procesor najbrže pristupa podacima koji se nalaze u registrima i keš memoriji. Za red veličine sporije pristupa podacima koji se nalaze u radnoj memoriji u odnosu na podatke u registrima i keš memoriji, a najsporiji je pristup podacima koji se nalaze na disku. Svakim nivoom udaljavanja od procesora, povećava se kapacitet memorije, ali se smanjuje brzina. Programer može kontrolisati da li se podaci čuvaju na disku ili u glavnoj memoriji, ali ne može

direktno kontrolisati da li će se podaci naći u kešu ili registrima. Koji podatak će se naći u registru određuje kompajler, a keš sistem procesora je automatski i programer nema kontrolu nad njim.

Platforma CUDA na raspolaganje stavlja hijerarhiju programibilne memorije koja se sastoji od registara (eng. registers), deljene memorije (eng. shared memory), lokalne memorije (eng. local memory), konstantne memorije (eng. constant memory), teksturne memorije (eng. texture memory) i globalne memorije (eng. global memory). Različiti tipovi memorije na uređaju mogu biti implementirani identično i imati isti odziv i protočnost, ali imati različite nivoe pristupa i odgovornosti na logičkom nivou.



Slika 3.10: Logički dijagram strukture video memorije na platformi CUDA [2][slika 4-2].

Na slici 3.10 ilustrovana je podela video memorije grafičke jedinice na platformi CUDA. Plavi pravougaonik sa nazivom `Device` je memorija grafičke jedinice, a plavi pravougaonik sa oznakom `Host` radna memorija računara. Svaki crveni

pravougaonik označava jednu vrstu memorije, a strelice između crvenih pravougaonika smer prenošenja podataka između njih. Prikaz strukture memorije je logički i znači da različiti tipovi memorije mogu nalaziti na istom fizičkom delu grafičke jedinice. Zeleni pravougaonici predstavljaju logičke niti, a ne fizička jezgra na grafičkoj jedinici.

Svaka nit ima registre koji pripadaju samo toj niti i na slici 3.10 su predstavljeni crvenim pravougaonikom sa oznakom `Registers`. U registrima se čuvaju lokalne promenljive i elementi lokalnih nizova pod uslovom da ih ima manje nego registara. Nit ne može pristupiti registrima i lokalnoj memoriji druge niti. Životni vek registara je vezan za životni vek kernel funkcije i kada se kernel funkcija završi ne može se pristupiti sadržaju registara te kernel funkcije. U registrima se pored lokalnih promenljivih mogu čuvati i često korišćene vrednosti od strane kernel funkcije. Kao i na procesoru, registri su najbrži oblik memorije i nema ih mnogo. Broj registara koje svaka nit ima zavisi od arhitekture grafičke jedinice. Na primer, arhitektura Fermi ima 63 registra, a arhitektura Kepler ima 255 registra.

Ako nema dovoljno registara, vrednosti koje bi mogle da se čuvaju u registru čuvaju se u lokalnoj memoriji, na slici 3.10 predstavljenoj kao crveni pravougaonik sa nazivom `Local Memory`. Lokalna i globalna memorija se na grafičkoj jedinici nalaze u istom tipu fizičke memorije, ali logički svaka nit ima sopstvenu lokalnu memoriju dok je globalna memorija zajednička za sve niti. To znači da je pristup lokalnoj memoriji podjednako neefikasan kao i pristup globalnoj memoriji.

Deljena memorija je zajednička memorija za sve niti unutar jednog bloka. Fizički se nalazi na samom čipu blizu jezgara i zbog toga toga ima veći protok i manji odziv od lokalne i globalne memorije. Njena svrha je slična L1 keš memoriji na procesoru, ali se razlikuje od keš memorije procesora po mogućnosti da programer odredi životni vek podataka koji su u njoj sačuvani. Vrednosti unutar niti se mogu eksplicitno smestiti u deljenu memoriju korišćenjem anotacije `__shared__` ispred deklaracija lokalne promenljive. Pristup deljenoj memoriji nije automatski sinhronizovan, stoga je neophodno osigurati da ne dođe do istovremenog čitanja i pisanja u istu memorijsku lokaciju od strane više niti. Životni vek deljene memorije je vezan za kernel, ne za funkciju koja deklariše deljenu memoriju.

Konstantna memorija je oblik programibilne keš memorije ograničenog kapaciteta i na većini arhitektura njen kapacitet iznosi 64KB za ceo uređaj. Statički je deklarisana pomoću anotacije `__constant__` i zajednička je za sve kernel funkcije unutar jedinice prevođenja u kojoj je deklarisana. Inicijalizuje se pozivom funkcije

programskog interfejsa platforme CUDA i kernel funkcije mogu samo čitati njen sadržaj. Obično se koristi za konstante koje pri računanju koristi jedan blok niti. Najveći dobitak performansi se dobija kada jedan blok niti u isto vreme čita istu memorijsku lokaciju u konstantnoj memoriji.

Teksturna memorija je takođe poseban oblik keš memorije sa podrškom za česte operacije filtriranja u računarskoj grafici. Optimizovana je za lokalnost pristupa za podatke koji imaju oblik dvodimenzionih slika. Na primer, kada nit pristupa lokaciji (i, j) dvodimenzione teksture, onda verovatno neka druga nit iz iste grupe pristupa nekoj od lokacija $(i \pm 1, j \pm 1), (i, j \pm 1), (i \pm 1, j)$ i u tom slučaju će čitanje biti brže ukoliko se tekstura nalazi u teksturnoj memoriji.

Globalna memorija je memorija najvećeg kapaciteta opšte namene. Ukoliko nije jednostavno ili moguće smestiti podatke u neku od prethodno navedenih tipova memorije, onda se oni stavljaju u globalnu memoriju. Životni vek globalne memorije je vezan za životni vek programa, a po opsegu je dostupna svim aktivnim kernel pozivima na sistemu. Može biti statički ili dinamički alocirana. Statički alocirana globalna memorija biće svaka globalna promenljiva u programu označena anotacijom `__device__`. Memorija se može alocirati i dinamički preko programskog interfejsa sličnog interfejsu standardne biblioteke programskog jezika C.

Pored programibilne memorije, grafičke jedinice imaju i automatsku keš memoriju koja se sastoji od L1 keša, L2 keša, keša konstantne memorije i keša memorije tekstura. Svako jezgro ima L1 keš i jedan L2 keš koji dele sva jezgra. Na procesoru se čitanja i pisanja mogu sačuvati u keš memoriji, a na grafičkoj jedinici mogu se keširati samo čitanja. Svako jezgro ima keš memoriju za čuvanje rezultata čitanja iz konstantne memorije i keš memoriju za čuvanje rezultata čitanja iz memorije za teksture.

Grafička jedinica ima bogatiju hijerarhiju memorije od procesora jer pristupi memoriji tokom izračunavanja na grafičkoj jedinici prate principe lokalnosti i zbog toga se različiti tipovi memorije i njihove karakteristike mogu iskoristiti kako bi se značajno poboljšale performanse programa.

Programski interfejs za upravljanje globalnom memorijom

Iz kernel funkcije nije moguće pristupiti vrednosti koja se nalazi u memorijskom adresnom prostoru tekućeg procesa, niti je iz funkcije koja se izvršava na procesoru moguće pristupiti vrednostima u globalnoj memoriji grafičke jedinice. Zbog toga se programski interfejs za upravljanje globalnom memorijom sastoji od skupa funkcija za alokaciju globalne memorije na grafičkoj jedinici i skupom funkcija za transfer podataka između radne memorije procesa i video memorije grafičke jedinice.

Za svaku od funkcija standardne biblioteke programskog jezika C `malloc`, `free`, `memcpy` i `memset` postoji odgovarajuća funkcija na platformi CUDA `cudaMalloc`, `cudaFree`, `cudaMemcpy`, `cudaMemset` za upravljanje globalnom memorijom grafičke jedinice i kopiranjem podataka između radne memorije procesa i video memorije grafičke jedinice.

Globalna memorija na grafičkoj jedinici alocira se pomoću funkcije

```
cudaError_t cudaMalloc(void** dev_ptr, size_t size);
```

Parametar `dev_ptr` je adresa promenljive u matičnoj memoriji na kojoj će biti upisana početna adresa alociranog bloka iz globalne memorije, a parametar `size` određuje broj bajtova koji će biti alociran. Povratna vrednost funkcije `cudaMalloc` ima tip `enum cudaError_t` koji može imati neku od unapred definisanih vrednosti koje predstavljaju moguće greške prilikom poziva funkcije programskog interfejsa platforme CUDA. Povratna vrednost `cudaSuccess=0` znači da se funkcija uspešno završila i da je u `*dev_ptr` upisana početna adresa memorije veličine `size` bajtova. Memorija je uvek adekvatno poravnata za svaki primitivan tip.

Memorija alocirana pozivom funkcije `cudaMalloc` oslobađa se funkcijom

```
cudaError_t cudaFree(void *dev_ptr);
```

Parametru `dev_ptr` prosleđuje se adresa globalne memorije dobijena pozivom funkcije `cudaMalloc`, slično kao što se funkciji `free` prosleđuje memorijska adresa dobijena pozivom funkcije `malloc` ili `calloc`. U slučaju da se funkciji `cudaFree` prosledi vrednost koja nije dobijena nekom od funkcija za alokaciju globalne memorije, povratna vrednost funkcije `cudaFree` biće kôd greške. Povratna vrednost funkcije `cudaFree` biće greška ako se prosledi adresa već oslobođene memorije.

Kopiranje podataka između matične memorije i memorije uređaja izvodi se funkcijom:

```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count,
                       enum cudaMemcpyKind kind);
```

Parametar `dst` je memorijska lokacija na koju će `count` bajtova biti kopirano sa memorijske lokacije `src`. Četvrti parametar `kind` određuje smer kopiranja podataka između matične memorije i memorije uređaja i može imati vrednosti

- `cudaMemcpyHostToHost` - kopira sadržaj iz matične memorije u matičnu memoriju,
- `cudaMemcpyHostToDevice` - kopira sadržaj iz matične memorije u memoriju uređaja,
- `cudaMemcpyDeviceToHost` - kopira sadržaj iz memorije uređaja u matičnu memoriju,
- `cudaMemcpyDeviceToDevice` - kopira sadržaj iz memorije uređaja u memoriju uređaja.

Memorijske lokacije koje se proslede kao argumenti moraju biti iz odgovarajuće memorije i poklapati se sa smerom kopiranja određenog parametrom `kind`. U slučaju da se adrese prosleđene parametrima `dst` i `src` ne poklapaju sa smerom kopiranja podataka određenim parametrom `kind`, funkcija `cudaMemcpy` neće izvršiti operaciju kopiranja i povratna vrednost će biti kôd greške.

Globalna memorija alocirana pozivom funkcije `cudaMalloc` nije inicijalizovana i sadržaće vrednosti koje se u trenutku alokacije nađu u alociranom memorijskom bloku. Bajtovi memorijskog bloka u globalnoj memoriji grafičke jedinice mogu se inicijalizovati pozivom funkcije

```
cudaError_t cudaMemset(void *dev_ptr, int value, size_t count);
```

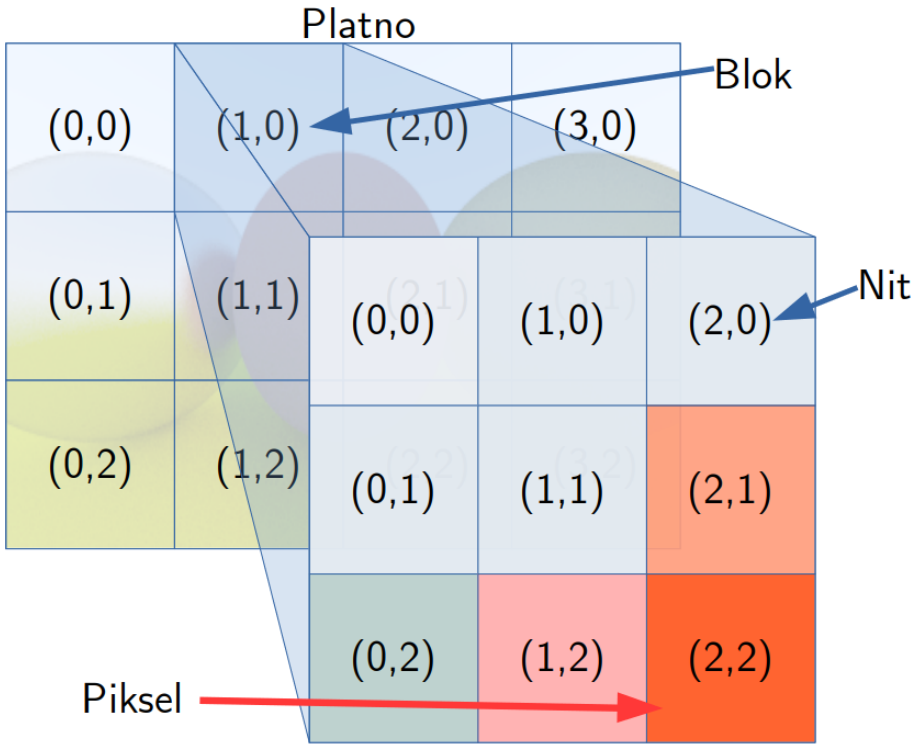
Parametar `dev_ptr` je pokazivač na početak memorijskog bloka u globalnoj memoriji grafičke jedinice, parametar `value` vrednost na koju će svaki bajt biti inicijalizovan, a parametar `count` određuje broj bajtova koji će biti inicijalizovani počevši od memorijske adrese `dev_ptr`. Funkciju `cudaMemset` je najefikasnije koristiti ukoliko je potrebno izvršiti jednostavnu inicijalizaciju, najčešće na vrednost 0 ili -1, a za kompleksne inicijalizacije efikasnije je koristiti kernel funkcije.

Komunikacija i kopiranje podataka između procesora i grafičke jedinice nije jeftino i zbog toga se u aplikacijama teži ka minimalnom broju poziva kernel funkcija i funkcija programskog interfejsa platforme CUDA.

3.5 CUDA i praćenje zraka

U prethodnim poglavljima predstavljene su glavne osobine platforme CUDA i interfejs za programiranje koji olakšava implementaciju paralelnih algoritama na grafičkoj jedinici, a u poglavlju 2.2 objašnjeno je zašto je tehnika praćena zraka algoritmom 2.1 pogodna za paralelizaciju. U ovom poglavlju biće predstavljen konkretan pristup paralelizaciji algoritma 2.1, na platformi CUDA.

Platno na kome se iscrtava slika scene je po svojoj strukturi dvodimenziona mreža piksela. Dvodimenziona mreža blokova i niti može se preslikati na platno tako što se svakoj niti dodeli jedan piksel. Računanje boje svaka dva različita piksela `(i, j)`, `(i', j')` je nezavisno i može se izvršiti paralelno. Dakle, jedan pristup paralelizaciji tehnike iscrtavanja praćenjem zraka je da svaka nit u jednom trenutku računa boju jednog piksela na platnu.



Slika 3.11: Podela piksela slike za iscrtavanje na blokove i niti.

Na slici 3.11 ilustrovana je podela iscrtavanja slike scene na blokove i niti. Platno se podeli na blokove, gde svaki blok sadrži grupu piksela, dok svaka nit računa boju jednog piksela iz bloka. Ilustrovana podela je logička, dok stepen fizičke paralelizacije zavisi od raspoloživog broja jezgara na uređaju.

```
1  __global__
2  void pracenje_zraka_kernel(scena, kamera, slika) {
3    int i = threadIdx.y + blockIdx.y * blockDim.y;
4    int j = threadIdx.x + blockIdx.x * blockDim.x;
5    int M = kamera.visina_platna, N = kamera.sirina_platna;
6    if (i < M && j < N) {
7      zrak = pusti_zrak(kamera, piksel(i, j));
8      slika[i, j] = boja_zraka(zrak, scena);
9    }
10 }
11
12 pracenje_zraka(scena, kamera) -> Slika {
13   M = kamera.visina_platna;
14   N = kamera.sirina_platna;
15   slika = inicijalizuj_sliku(M, N);
16   niti = odredi_broj_niti(M, N);
17   blokovi = odredi_broj_blokova(M, N);
18   pracenje_zraka_kernel<<<blokovi, niti>>>(scena, kamera, slika);
19   return slika;
20 }
```

Algoritam 3.2: Procedura praćenja zraka na platformi CUDA.

Algoritam 3.2 je paralelizovana verzija sekvencijalnog algoritma 2.1. Detalji alokacije, upravljanja memorijom i kopiranja podataka u odgovarajući memorijski adresni prostor su izostavljeni radi jednostavnosti prikaza paralelizacije, a biće objašnjeni u poglavlju 4.4.

Matična funkcija `pracenje_zraka` algoritma 3.2 poziva se sa procesora i iscrtava scenu paralelizacijom tehnike praćenja zraka na grafičkoj jedinici. Povratna vrednost funkcije `pracenje_zraka` algoritma 3.2 je slika scene prosleđene preko parametra `scena`, iz perspektive kamere određene parametrom `kamera`.

U linijama 16 i 17, algoritma 3.2, računaju se redom dimenzije mreže niti, pozivom funkcije `odredi_broj_niti` i dimenzije mreže blokova, pozivom funkcije `odredi_broj_blokova`. Dimenzije mreža blokova i niti određuju se na osnovu dimenzije platna na kojem će slika scene biti iscrtana. Određivanje optimalnih dimenzija blokova i niti nije jednostavno. U slučaju da su blokovi preveliki može

se desiti da par niti blokira ostatak niti iz bloka jer se zrak koji im je dodeljen za računanje odbija po sceni više puta nego zraci dodeljeni preostalim nitima u bloku. Sa druge strane, ukoliko su dimenzije blokova i niti premale, performanse neće biti optimalne i zato se prilikom određivanja dimenzija blokova i niti računaju dimenzije slike, a često i osobine grafičke jedinice.

U algoritmu 3.2, boju zraka računa kernel funkcija `pracenje_zraka_kernel`. Umesto dvostruke petlje `for`, kao u algoritmu 2.1, pozicija piksela u slici računa se na osnovu identifikatora bloka i niti na kojoj je kernel funkcija instancirana. Na primer, neka su dimenzije platna `M = 9, N = 12` i neka su blokovi dimenzije `4x3`, a niti `3x3`, kao na slici 3.11. Piksel na koji pokazuje crvena strelica na slici 3.11, nalazi se u bloku `(1,0)` i niti `(2,2)`. Zbog toga će vrednosti izraza u linijama 3 i 4, algoritma 3.2, biti `(j, i) = (2 + 0 * 3, 2 + 1 * 4) = (2, 5)`, što je tačna pozicija piksela na platnu.

Provera u liniji 6 algoritma 3.2 je neophodna kako ne bi došlo do pristupa nedozvoljenom segmentu memorije u slučaju da blokovi na granici imaju više niti nego što ima piksela na tom delu platna. U liniji 7 algoritma 3.2 pozivom funkcije `pusti_zrak` računa se smer vektora i početna tačka zraka iz piksela `(i, j)`, a u liniji 8, pozivom funkcije `boja_zraka` računa se boja koju će imati piksel u slici na poziciji `(i, j)`. Funkcije `pusti_zrak` i `boja_zraka` iz algoritma 3.2 su funkcije za uređaj jer se pozivaju iz kernel funkcije i obe se izvršavaju na grafičkoj jedinici.

Linija 18 algoritma 3.2 je ključna promena u odnosu na sekvencijalni algoritam 2.1, jer se jednim pozivom kernel funkcije paralelizuje praćenje zraka. U sekvencijalnom algoritmu 2.1 kroz piksele se iterira preko dvostruke petlje `for`, dok se u paralelnom algoritmu 3.2 poziv kernel funkcije instancira za svaki piksel, a pozicija piksela na platnu računa se pomoću jedinstvenih identifikatora bloka i niti. Na taj način, omogućeno je paralelno izračunavanje boja piksela na platnu uz minimalnu modifikaciju sekvencijalnog algoritma.

Algoritam 3.2 nije jedini pristup paralelizaciji praćenja zraka. Zbog jednostavnosti objašnjenja, algoritam je ilustrovan nad slučajem kada se jednoj niti dodeljuje jedan piksel. U praksi, efikasnije je da jedna nit računa više piksela ili da se napravi red piksela kao red zadataka koje niti preuzimaju i računaju. Paralelizacija i ubrzavanje tehnike praćenja zraka je aktivna oblast izučavanja sa mnoštvom pristupa i izazova u kojoj svake godine dolazi do novih otkrića i poboljšanja na nivou hardvera i softvera.

Glava 4

Implementacija

U ovom poglavlju prikazan je ključni deo projekta koji implementira tehniku praćenja zraka na jednoj niti matičnog procesora i na platformi CUDA. Poglavlje 4.1 opisuje najvažnije module projekat, glavne ideje svakog modula i svrhu kojoj modul služi u projektu. U poglavlju 4.2 predstavljena je ulazna tačka u program koju dele implementacije na jednoj niti matičnog procesora i na platformi CUDA. Poglavlje 4.3 opisuje glavni deo i specifičnosti implementacije na jednoj niti, a poglavlje 4.4 opisuje glavni deo i specifičnosti implementacije na platformi CUDA. Poglavlja 4.3 i 4.4 su relativno kraća od poglavlja 4.1 i 4.2 jer obe implementacije dele većinu koda u projektu.

4.1 Projekat

Tehnika praćenja zraka u ovom master radu implementirana je u programskom jeziku C++ jer on poseduje svojstva koja olakšavaju pisanje operacija niskog nivoa neophodnih za efikasnu implementaciju tehnike praćenja zraka i potpuno je podržan na platformi CUDA. Programski interfejs platforme CUDA implementiran je u programskom jeziku C i zbog toga ga je moguće direktno koristiti iz koda pisanog u programskom jeziku C++. Projekat trenutno podržava samo operativni sistem Linuks, ali se može proširiti da podržava svaki operativni sistem na kojem postoji podrška za platformu CUDA. Na primer, da bi projekat radio i na operativnom sistemu Vindouz (eng. Windows) dovoljno je dopuniti modul za upravljanje memorijom sistemskim pozivima za alokaciju memorije na ovom operativnom sistemu jer ostatak projekta zavisi samo od platforme CUDA i standardne biblioteke programskog jezika C++.

Jedina biblioteka koju projekat koristi, van programskog interfejsa platforme CUDA i standardne biblioteke programskog jezika C++, je `stb_image`. Biblioteka `stb_image` pripada skupu biblioteka `stb` otvorenog koda, pod licencom javnog dobra, koja implementira često potrebne operacije koje se koriste prilikom programiranja aplikacija koje koriste računarsku grafiku i igrica. U projektu se koristi za snimanje slika na disk i učitavanja slika raznih formata sa diska zato što svaki format slike zahteva adekvatno parsiranje binarnih datoteka u kojima je slika upisana što prevazilazi opseg ovog master rada.

`rad`	Tekst master rada
`rt`	Implementacija projekta
`test`	Testovi modula iz direktorijuma `rt`
`resources`	Slike, teksture i iscrtane scene
`modules`	Moduli CMake sistema
`CMakeLists.txt`	Glavna datoteka cmake sistema
`README.md`	Opis projekta

Tabela 4.1: Datoteke u korenom direktorijumu projekta

U tabeli 4.1 nabrojane su glavne datoteke koje se nalaze u korenom direktorijumu projekta i njihovi kratki opisi. Implementacija tehnike praćenja zraka za platformu CUDA i implementacija na jednoj niti glavnog procesora nalazi se u direktorijumu `rt`. Testovi glavnih modula iz direktorijuma `rt` nalaze se u direktorijumu `test`. Aplikacija tokom izvršavanja učitava teksture iz direktorijuma `resources`, ukoliko se tekstura ne nalazi u tom direktorijumu neće biti učitana.

Projekat se prevodi pomoću sistema `CMake` i sadrži datoteku `CMakeLists.txt` koja se nalazi u korenom direktorijumu projekta. Sistem `CMake` pronalazi biblioteke platforme CUDA pomoću modula `modules/FindCUDALibs.cmake`, a verzija platforme CUDA, verzija matičnog kompajlera i opcije kompilacije mogu se postaviti direktno u datoteci `CMakeLists.txt` ili proslediti prilikom pokretanja programa `cmake` iz terminala.

U tabeli 4.2 navedeni su, abecednim redom, glavni moduli i kratki opisi funkcionalnosti koje svaki modul implementira, a u daljem tekstu je svaki modul detaljnije objašnjen. U modulima `rt_world`, `rt_material`, `rt_ray` i `rt_camera` nalazi se većina implementacije tehnike praćenja zraka, a moduli `rt_common`, `rt_math`, `rt_memory` i `rt_runner` mogu biti moduli opšte namene i nisu usko vezani za tehniku praćenja zraka. Modul `rt_cuda` sadrži apstrakcije za olakšavanje rada sa programskim interfejsom niskog nivoa platforme CUDA i nije deo platforme. Ako

`rt_camera`	Svojstva i funkcije kamere
`rt_common`	Funkcionalnosti zajedničke za više modula
`rt_cuda`	Apstrakcije nad platformom CUDA
`rt_material`	Materijali objekata i interakcija zraka sa materijalom
`rt_math`	Matematičke operacije nad vektorima
`rt_memory`	Upravljanje matičnom memorijom i memorijom uređaja
`rt_ray`	Definicija zraka i propagiranje kroz scenu
`rt_runner`	Inicijalizacija, pokretanje i obrada rezultata praćenja zraka
`rt_world`	Definicije objekata i scene

Tabela 4.2: Kratak opis modula projekta.

nije navedeno drugačije, svaka funkcija je označena `__host__` i `__device__` anotacijom jer implementacije tehnike praćenja zraka na procesoru i grafičkoj jedinici dele većinu koda.

`u8, u16, u32, u64`	`uint8_t, uint16_t, uint32_t, uint64_t`
`umem, usize`	`uintptr_t, size_t`
`s8, s16, s32, s64`	`int8_t, int16_t, int32_t, int64_t`
`f32, f64, f128`	`float, double, long double`
`b8, b32`	`bool, int32_t`

Tabela 4.3: Imena primitivnih tipova u implementaciji.

Da bi se izbegle greške pri računanju zbog različitosti platformi projekat koristi primitivne tipove tačno određene veličine iz zaglavlja standardne biblioteke `cstdint` i daje im kraća imena koja su prikazana u tabeli 4.3 tako što je svaki tip imenovan po šablonu `{u|s|f|b}{8|16|32|64}`. Na primer, na jednoj platformi `long` može biti veličine 4 bajta, a na drugoj 8 bajtova što može dovesti do neočekivanih odsecanja vrednosti prelaskom na platformu na kojoj je tip `long` manje veličine. Tipovi koji predstavljaju neoznačene cele brojeve su `u8`, `u16`, `u32`, `u64`, a označene `s8`, `s16`, `s32`, `s64` i redom su preciznosti 8, 16, 32 i 64 bita. Po standardnu IEEE754, tip `float` je preciznosti 32 bita, tip `double` je preciznosti 64 bita, a tip `long double` preciznosti 128 bitova, zbog toga su i kraći nazivi redom `f32`, `f64`, `f128`.

U kodu koji implementira tehniku praćenja zraka, polimorfizam se postiže pomoću *označenih unija* (eng. tagged unions). Označena unija je struktura koja sadrži uniju i jedno polje `type` koje određuje tip vrednosti koji je u datom trenutku sačuvan u uniji. U standardnoj biblioteci programskog jezika C++ postoji struktura `variant` koja implementira semantiku označene unije.

```
1  struct Material {
2    enum class Type {
3      None = 0,
4      Lambertian,
5      Metal,
6      Dielectric,
7      Isotropic,
8      DiffuseLight,
9    };
10   Type type;
11   union {
12     Lambertian lambertian;
13     Metal metal;
14     Dieletric dielectric;
15     Isotropic isotropic;
16     DiffuseLight diffuse_light;
17   };
18 }
19 static_assert(std::is_trivially_copyable_v<Material>
20        && std::is_trivially_destructible_v<Material>);
21
22 rgb Material::emitted(...) {
23   switch (type) {
24   case Material::Type::Lambertian: { /* Kod */ }break;
25   case Material::Type::Metal: {/* Kod */ } break;
26   // ostatak materijala ...
27   case Type::None: { SHOULD_NOT_REACH(msg); } break;
28   }
29 }
```

Listing 4.1: Primer označene unije kao strukture `Material`.

Na primer, struktura `Material` iz listinga 4.1 predstavlja materijal objekta koji može biti neki od tipova navedenih u enumeratoru `Type`. Polje `type`, u liniji 10 listinga 4.1 određuje koji materijal se nalazi u uniji. Članska funkcija `emitted`, definisana u liniji 22 listinga 4.1, računa odgovarajuću povratnu vrednost emitovanog svetla na osnovu vrednosti polja `type`. Kôd koji koristi člansku funkciju `emitted` se ne oslanja na konkretan tip materijala, slično kao što se prilikom poziva virtuelne funkcije preko pokazivača pozivalac ne oslanja na dinamički tip objekta. Statičke provere u linijama 19 i 20 listinga 4.1 osiguravaju da struktura `Material` uvek može trivijalno da se kopira i uništi jer se kôd koji efikasno

manipuliše objektima na sceni i kopiranjem podataka između matične memorije i memorije uređaja oslanja na ove dve osobine. Na primer, nijedan tip koji se nalazi u uniji ne može imati virtuelnu funkciju jer bi prilikom kopiranja niza objekata iz matične memorije u memoriju uređaja pokazivač na tabelu virtuelnih funkcija imao vrednost adrese iz matične memorije. Prilikom kopiranja objekata tabela virtuelnih funkcija ne bi bila kopirana u memoriju uređaja, niti bi pokazivači u tabeli bili adekvatno prepravljeni. Izvršavanje programa bi se prekinulo u slučaju da kôd na uređaju pozove virtuelnu funkciju objekta iz matične memorije jer bi ovim pozivom kôd na uređaju pokušao da pristupi matičnoj memoriji.

Efikasnost označene unije dolazi do izražaja u slučajevima kada je potrebno alocirati i iterirati kroz veliki niz objekata koji mogu biti različitog tipa i kada su tipovi koji se nalaze u uniji približno iste veličine i nad njima se izvodi ista operacija, na primer struktura `Material` i funkcija `emitted` iz listinga 4.1. U tom slučaju, veća je šansa da će se sledeći objekat u nizu naći u keš memoriji ukoliko se kroz objekte iterira redom jer se u memoriji objekti nalaze jedan do drugog. Takođe, ukoliko unija nema previše tipova, verovatno je da će i kôd funkcije koja se primenjuje na elemente biti u keš memoriji, pa će i njeno izvršavanje biti brže.

Korišćenje označene unije umesto nasleđivanja drastično olakšava kopiranje podataka između matične memorije i memorije uređaja, jer se niz objekata koji sadrži označenu uniju može prekopirati jednim pozivom funkcije `cudaMemcpy`. Sa druge strane, niz objekata čiji su tipovi deo hijerarhije klasa morao bi da se kopira objekat po objekat ili da se objekti pojedinačno konstruišu u memoriji uređaja što je daleko komplikovanije od niza objekata označenih unija.

rt_math

Modul `rt_math` implementira matematičke operacije na način na koji od korisnika modula sakriva arhitekturu na kojoj se operacija izvršava i tako omogućava deljenje većine koda za matematička izračunavanja između matičnog procesora i grafičke jedinice. To postiže definisanjem platformski nezavisnog interfejsa, a u implementaciji koristi činjenicu da kompajler `nvcc` odvojeno kompilira kôd za matični procesor i kôd za uređaj što omogućava razdvajanje implementacija matematičkih operacija između arhitektura korišćenjem pretprocesorskih direktiva.

U tehnici praćenja zraka većina matematičkih operacija izračunava se nad vektorima. U modulu `rt_math` implementirane su strukture koje predstavljaju vektore dimenzije 2, 3 i 4. Biće predstavljen kôd samo za vektore dimenzije 3 jer

je najčešće korišćen, a kôd za vektore dimenzija 2 i 4 je identičan do na dimenziju vektora.

```
1  struct v3 {
2    f32 e0;
3    f32 e1;
4    f32 e2;
5    f32 x() const { return e0; }
6    f32 y() const { return e1; }
7    f32 z() const { return e2; }
8    f32& x() { return e0; }
9    f32& y() { return e1; }
10   f32& z() { return e2; }
11 };
```

Listing 4.2: Struktura trodimenzionog vektora.

Struktura `v3` iz listinga 4.2 sadrži tri vrednosti `e0`, `e1`, `e2` koje predstavljaju redom nultu, prvu i drugu komponentu trodimenzionog vektora. Članske funkcije `x()`, `y()` i `z()` koriste se kada vektor predstavlja tačku u prostoru ili smer kretanja. Analogno funkcijama `x()`, `y()` i `z()`, struktura `v3` ima i članske funkcije `r()`, `g()`, `b()` koje se koriste u kontekstu kada se vektorom predstavlja boja i članske funkcije `u()`, `v()`, `w()` za pozicije na teksturi ili pri preslikavanju vektora u drugi prostor.

```
1  v3 &operator+=(v3 &a, v3 b) {
2    a.e0 += b.e0;
3    a.e1 += b.e1;
4    a.e2 += b.e2;
5    return a;
6  }
7
8  v3 operator+(v3 a, v3 b) {
9    a += b;
10   return a;
11 }
```

Listing 4.3: Vektorske operacije.

Nad strukturom `v3` iz listinga 4.2 definisani su operatori koordinatnog sabiranja, oduzimanja, množenja, deljenja. U listingu 4.3 prikazana je definicija operatora `+=` koji kao parametre ima dva vektora gde je prvi prosleđen po referenci, a drugi po vrednosti. Prosleđivanje prvog parametra po referenci je neophodno kako bi se sabiranje izvršilo u mestu, a ne nad kopijom. Operator sabiranja `+` iz

listinga 4.3 definiše se pomoću operatora sabiranja u mestu +=. Na sličan način definišu se i operacije oduzimanja, množenja i deljenja, tako što se znak + zameni odgovarajućim simbolom operacije.

```
1  // rt_math.h
2  __host__ __device__
3  f32 powf(f32 x, f32 y);
4
5  // rt_math.cu
6  __host__ __device__
7  f32 powf(f32 x, f32 y) {
8  #if __CUDA_ARCH__
9    return __powf(x, y);
10 #else
11   return std::pow(x, y);
12 #endif
13 }
```

Listing 4.4: Definicija funkcije stepenovanja.

U listingu 4.4 je na primeru funkcije stepenovanja `powf` prikazan obrazac po kojem su definisane matematičke funkcije u modulu `rt_math`. Zaglavlje `rt_math` deklariše prototip funkcije `powf`, a definicija se nalazi u `rt_math.cu` datoteci. Definicija funkcije `powf` u svom telu sadrži naredbu uslovnog prevođena, koja u slučaju kada se funkcija kompilira za platformu CUDA koristi intrinzičnu funkciju `__powf` platforme, a u suprotnom kada se funkcija kompilira za matični procesor koristi funkciju standardne biblioteke programskog jezika C++ `std::pow`. Kada kompajler `nvcc` kompilira funkciju za uređaj, preprocesorska konstanta `__CUDA__ARCH__` je definisana i njena vrednost predstavlja verziju arhitekture za koju se funkcija kompilira, a kada kompajler `nvcc` kompilira funkciju za matični procesor, preprocesorska konstanta `__CUDA__ARCH__` nije definisana. Dakle, nakon koraka preprocesiranja telo funkcije `powf` sadržaće poziv `__powf` ako se funkcija kompilira za uređaj ili poziv `std::pow` ako se kompilira za matični procesor.

Dve donje crte `__` u prefiksu imena funkcije programskog interfejsa platforme CUDA znače da je to intrinzična funkcija koja će ukoliko je to moguće biti zamenjena instrukcijom koja izvršava istu matematičku operaciju na arhitekturi uređaja za koju se kôd kompilira. Nisu sve matematičke funkcije platforme CUDA intrinzične, neke su implementirane kao funkcije matematičke biblioteke platforme.

```
1  // rt_math.h
2  struct RandomState {
3    union {
4      curandState_t *cuda_state;
5      std::mt19937_64 *generator;
6    };
7  };
8
9  // rt_math.cu
10 __device__ __host__
11 f32 uniform(RandomState state, f32 a, f32 b) {
12 #if __CUDA_ARCH__
13   f32 v = curand_uniform(state.cuda_state);
14   return (b - a) * v + a;
15 #else
16   std::uniform_real_distribution<f32> dist(a, b);
17   f32 v = dist(*state.generator);
18   return v;
19 #endif
```

Listing 4.5: Implementacija funkcije uniformne raspodele za matični procesor i uređaj.

Matični kôd i kôd za uređaj imaju različite implementacije generisanja nasumičnih vrednosti. Na listingu 4.5 prikazan je primer implementacije funkcije za generisanje nasumičnih vrednosti iz uniformne raspodele u opsegu `[a, b]`. Kôd za uređaj koristi funkciju `curand_uniform` programskog interfejsa platforme CUDA i stanje `curandState_t`, a matični kôd koristi `std::uniform_real_distribution` i generator `std::mt19937_64`.

Inicijalizacija objekta `curandState_t` i `std::mt19937_64` nije jeftina i zato se izvodi jednom na početku izvršavanja niti, tako što svaka nit uređaja inicijalizuje objekat lokalan za nit tipa `curandState_t`, a svaka nit matičnog procesora objekat lokalan za nit tipa `std::mt19937_64`. Stanje na osnovu kojeg se generišu nasumične vrednosti prenosi se funkcijama pomoću objekta `RandomState`. Na taj način se ne razlikuje interfejs funkcija za generisanje nasumičnih vrednosti između matičnog koda i koda uređaja.

rt_camera

Tehnika praćenja zraka koristi sintetički model kamere čije su matematičke osnove opisane u poglavlju 2.3, a u ovom poglavlju biće opisane dodatne karakteristike i način konstrukcije modela kamere koji je jednostavniji za korišćenje. Parametri kamere su podesivi i mogu se postaviti pre početka iscrtavanja scene.

```
1  struct Camera {
2    v3 origin;
3    v3 lower_left_corner;
4    v3 horizontal;
5    v3 vertical;
6    v3 w, u, v;
7    f32 lens_radius;
8    f32 time0;
9    f32 time1;
10
11   Camera();
12   Camera(const CameraParams &params);
13 };
```

Listing 4.6: Definicija modela kamere.

Struktura `Camera` u listingu 4.6 sadrži sva polja koja definišu model kamere. Polje `origin` je pozicija kamere na sceni, polje `lower_left_corner` je pozicija donjeg levog ugla platna u odnosu na poziciju kamere, polja `horizontal` i `vertical` su redom širina i visina platna, a polja `u`, `v` i `w` su redom vektor na desno, vektor na gore i vektor pogleda sa slike 2.5. Parametri kamere koji nisu opisani u poglavlju 2.3 su `lens_radius` koji određuje stepen zakrivljenja sočiva kamere i parametri `time0` i `time1` koji omogućavaju iscrtavanje objekata u pokretu definisanjem vremenskog intervala `[time0, time1]` u kom će kamera periodično slikati scenu.

Kamera definisana u listingu 4.6 ima podrazumevan konstruktor, deklarisan u liniji 11 i dodatni konstruktor definisan u liniji 12 koji inicijalizuje kameru na osnovu prosleđenih vrednosti parametara kamere u strukturi `CameraParams`. Podrazumevani konstruktor kamere inicijalizuje polja strukture `Camera` iz listinga 4.6 tako da se kamera nalazi u koordinatnom početku i gleda niz negativan deo `Z`-ose scene.

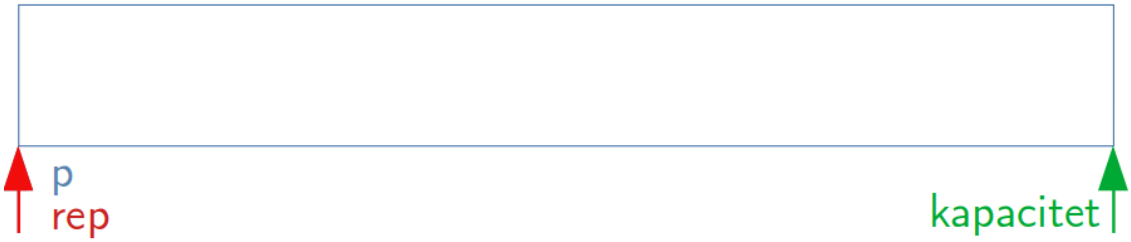
```
1  struct CameraParams {
2    f32 aspect_ratio = 16.0 / 9.0;
3    i32 image_width = 400;
4    i32 image_height = i32(image_width / aspect_ratio);
5    v3 lookfrom;
6    v3 lookat;
7    v3 vup;
8    f32 vfov;
9    f32 aperture;
10   f32 focus_dist;
11   rgb background;
12   i32 max_ray_depth = 5;
13   i32 samples_per_pixel = 1;
14 };
```

Listing 4.7: Parametri za inicijalizaciju kamere.

Ponekad je potrebno fino podesiti model kamere, ali to nije uvek jednostavno izvesti menjanjem polja strukture `Camera` iz listinga 4.6. Na primer, podesiti da se kamera nalazi u proizvoljnoj poziciji `A` i da gleda u tačku `B` zahteva vektorski račun da bi se polja `u, v, w` strukture `Camera` iz listinga postavile na odgovarajuće vrednosti. Konstruktor `Camera(const CameraParams&)` izvodi račun potreban da se polja strukture `Camera` iz listinga 4.6 inicijalizuju tako da određuju model kamere definisan poljima strukture `CameraParams` iz listinga 4.7. Polje `aspect_ratio` određuje odnos širine platna, definisan preko polja `image_width` i visine platna definisan poljem `image_height`. Vektor `lookfrom` određuje poziciju na kojoj se kamera nalazi na sceni, vektor `lookat` tačku na sceni u koju kamera gleda, a vektor `vup` određuje vektor na gore. Jednostavnije je definisati kameru preko vektora `lookfrom`, `lookat` i `vup` i prepustiti konstruktoru strukture `Camera` da izračuna vektore `v,u,w` iz listinga 4.6. Polje `vfov` određuje ugao gledanja koji kamera zahvata, polje `aperature` poluprečnik otvora sočiva kamere koje kontroliše opseg tačke fokusa na sceni na rastojanju od kamere koje je definisano poljem `fosuc_dist`. Polje `background` određuje boju pozadine scene, a polje `max_ray_depth` maksimalan broj puta koji zrak može da se odbije od objekata na sceni dok ne stigne do izvora svetlosti. Za implementaciju tehnike *anti-alijasinga* (eng. anti-aliasing) polje `samples_per_pixel` određuje koliko će zraka biti pušteno iz jednog piksela. Konačna boja piksela biće prosečna vrednost boje svih zraka puštenih kroz taj piksel.

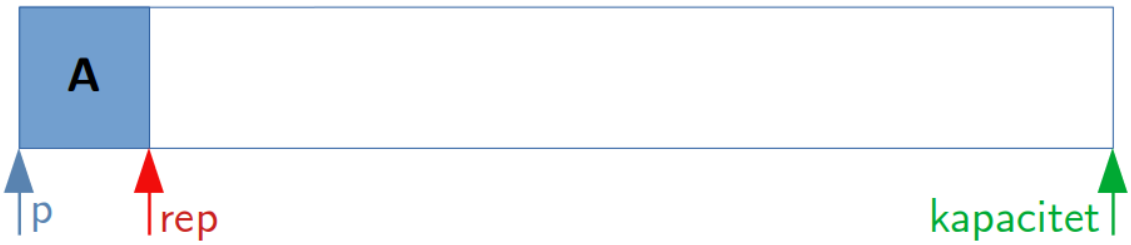
rt_memory

Modul `rt_memory` upravlja matičnom memorijom i memorijom uređaja na način koji omogućava matičnom kodu i kodu uređaja da identično pristupaju objektima koji se nalaze u memoriji. Identičan interfejs za pristup objektima postiže se korišćenjem relativnog adresiranja i strategije alokacije memorije koja omogućava jednostavno kopiranje grupe objekata između matične memorije i memorije uređaja.



Slika 4.1: Stanje prazne memorijske arene.

Memorijska arena (eng. memory arena) upravlja alokacijom memorije i prebacivanjem podataka između matične memorije i memorije uređaja. Na početku izvršavanja aplikacije memorijska arena alocira jedan blok memorije dovoljno velikog kapaciteta da sačuva sve objekte koji će biti potrebni u aplikaciji. Na slici 4.1 ilustrovano je stanje tek inicijalizovane memorijske arene u kojoj još uvek nema alociranih objekata. Plavi pravougaonik predstavlja blok memorije u kojem će arena alocirati objekte, pokazivač `p` čuva adresu početka alociranog memorijskog bloka, pokazivač `rep` pokazuje na kraj trenutno zauzetog prostora, a pokazivač `kapacitet` pokazuje na kraj memorijskog bloka i služi kao graničnik koji sprečava da se alokacija izvrši van memorijskog bloka.



Slika 4.2: Stanje memorijske arene nakon alociranja objekta A.

Na slici 4.2 ilustrovano je stanje arene nakon alokacije objekta A. Pokazivači `p` i `kapacitet` ostaju nepromenjeni, a pokazivač `rep` je pomeren tako da pokazuje na prvi slobodan bajt od početka memorijskog bloka.

Slika 4.3: Stanje memorijske arene nakon alociranja objekata A i B.

U slučaju da memorijska lokacija na koju pokazuje `rep` nije adekvatno poravnata za alokaciju novog objekta, neophodno je pomeriti `rep` na prvu sledeću memorijsku lokaciju koja je adekvatno poravnata za tip objekta koji se alocira. Na slici 4.3 prikazano je stanje arene nakon alokacije objekta B za koji memorijska lokacija na koju je pokazivao `rep` na slici 4.2 nije bila adekvatno poravnata. Beli prostor između objekata A i B predstavlja prazan prostor koji ne pripada nijednom objektu već je nastao kao rezultat poravnanja pokazivača `rep` za alokaciju objekta B.

```
1  struct MemoryArena {
2    template<typename T, typename ...Args>
3    ArenaPtr<T> alloc(Args &&...args);
4    // ...
5  };
6  template<typename T> struct ArenaPtr {
7    friend struct MemoryArena;
8    T &operator*();
9    const T &operator*() const;
10   T *operator->();
11   const T *operator->() const;
12   // ...
13 private:
14   T *m_ptr = nullptr;
15 };
```

Listing 4.8: Interfejs za alokaciju objekata pomoću memorijske arene.

Objekat se u memorijskoj areni alocira pomoću šablonske funkcije `alloc` de-

klarisane u liniji 3 listinga 4.8. Parametar `T` šablona funkcije `alloc` određuje tip objekta koji će biti alociran, a parametri `Args...` tipove parametara konstruktora objekta `T` koji će biti pozvan nakon alokacije. Povratna vrednost funkcije `alloc` je objekat `ArenaPtr<T>` definisan u liniji 6 listinga 4.8 takav da polje `m_ptr` pokazuje na objekat koji je funkcija `alloc` alocirala i konstruisala. Razlikovanje običnog pokazivača `T*` od `ArenaPtr<T>` olakšava pisanje koda koji koristi memorijsku arenu smanjenjem kognitivnog opterećenja koje programer ima prilikom praćenja životnog veka i memorijskog prostora u kome je objekat sačuvan.

```
1 ...
2 MemoryArena m_host_memory_arena = MemoryArena::create_with_malloc(GB(2));
3 MemoryArena m_device_memory_arena = ;
4   MemoryArena::create_with_cuda_malloc(m_host_memory_arena.size());
5 ...
6 setup_scene(..., &m_host_memory_arena, ...);
7 ...
8 m_host_memory_arena.memcpy_to_device(&m_device_memory_arena);
9 ...
```

Listing 4.9: Primer sinhronizacije podataka objekata između matične arene i arena na uređaju.

Na listingu 4.9 prikazan je postupak korišćenja memorijskih arena u pripremi scene i prebacivanja svih potrebnih podataka u memoriju uređaja. Na samom početku, aplikacija pravi jednu matičnu memorijsku arenu čiji se memorijski blok nalazi u matičnoj memoriji i koja je definisana u liniji 2 listinga 4.9 i jednu arenu za uređaj čiji se memorijski blok nalazi na uređaju i definisana je u liniji 3 listinga 4.9.

Poziv funkcije `create_with_malloc(GB(2))` u liniji 2 listinga 4.9 pravi memorijsku arenu čiji je memorijski blok alociran funkcijom `malloc` i čiji će kapacitet biti 2 gigabajta, a poziv funkcije `create_with_cuda_malloc(...)` u liniji 3 pravi memorijsku arenu čiji će memorijski blok biti alociran na uređaju pozivom funkcije `cudaMalloc` i koji će biti istog kapaciteta kao i memorijski blok arena `m_host_memory_arena`. Pozivom funkcije `setup_scene` u liniji 6 listinga 4.9, scena i objekti koje scena sadrži inicijalizuju se na procesoru i alociraju u matičnoj memorijskoj areni `m_host_memory_arena` jer je procesor efikasniji od grafičke jedinice u izvršavanju operacija potrebnih za inicijalizaciju scene.

Kada su inicijalizovani svi sistemi i objekti celokupna matična memorijska arena kopira se na uređaj u liniji 8 listinga 4.9 pozivom funkcije `memcpy_to_device` koja iz matične memorijske arene `m_host_memory_arena` kopira ceo memorijski blok u memorijsku arenu na uređaju `m_device_memory_arena` jednim pozivom funkcije `cudaMemcpy`. Matični kôd može pristupati objektima alociranim u matičnoj memorijskoj areni `m_host_memory_arena` pomoću objekta `ArenaPtr<T>`, ali kôd na uređaju ne može koristiti `ArenaPtr<T>` koji je nastao kao rezultat poziva funkcije `alloc` iz listinga 4.8 jer polje `m_ptr` pokazuje na memorijsku lokaciju u matičnoj memoriji. Da bi se ovaj problem rešio, referenca na objekat se ne čuva kao direktan pokazivač `ArenaPtr<T>`, već se referenca na objekat čuva kao relativna pozicija od početka memorijskog bloka u kojem je objekat alociran.

Greška koja može nastati prilikom korišćenja `ArenaPtr<T>` je da objekat koji se čuva u areni, koji će biti kopiran iz matične memorije u memoriju uređaja, sačuva `ArenaPtr<T>` na drugi objekat u matičnoj memoriji. U tom slučaju, polje `m_ptr` iz listinga 4.8 imaće vrednost memorijske lokacije iz matične memorije i dereferenciranjem polja `m_ptr` u članskoj funkciji `operator*` iz koda uređaja će dovesti do nedozvoljenog pristupa memoriji i prekida programa. Srećom, ovakav tip greške je lako uočiti i ukloniti jer se alatom `gdb` može pronaći linija koda u kojoj se nedozvoljeni pristup dogodio.

```
1  template<typename T> struct Handle {
2    // ...
3  private:
4    umem m_offset = 0;
5  };
6  struct MemoryArena {
7    template<typename T> Handle<T> get_handle(ArenaPtr<T> ptr) const;
8    template<typename T> ArenaPtr<T> get_ptr(Handle<T> handle) const;
9    // ...
10 };
11 struct Texture {
12   Handle<u8[]> pixels;
13   // ...
14 };
```

Listing 4.10: Interfejs arene za adresno nezavisno referisanje objekata.

Šablonska struktura `Handle<T>`, definisana na početku listinga 4.10, predstavlja relativnu adresu objekta šablonskog tipa `T` alociranog u memorijskoj areni.

Polje `m_offset` u liniji 4 listinga 4.10 je relativna adresa objekta u odnosu na početak memorijskog bloka arene. Relativno rastojanje od početka memorijskog bloka `m_offset` je razlika memorijske adrese na kojoj se objekat nalazi i adrese početka memorijskog bloka arene. Dakle, struktura `ArenaPtr<T>` predstavlja stvarnu adresu objekta u memoriji, a `Handle<T>` relativnu adresu objekta u memorijskoj areni. Šablonska funkcija `get_handle` deklarisana u liniji 7 na listingu 4.10 konvertuje stvarnu adresu objekta u relativnu adresu objekta unutar memorijske arene, a šablonska funkcija deklarisana u liniji 8 konvertuje relativnu adresu objekta u memorijskoj areni u njegovu stvarnu adresu u memoriji.

Objekat `Handle<T>` ne može se dereferencirati, ali se može kopirati i proslediti funkciji `get_handle`. Na primer, struktura `Texture` iz listinga 4.10 pomoću objekta `Handle<u8[]> pixels` čuva poziciju u memorijskoj areni na kojoj se nalazi niz piksela teksture. Kada se objekat `Texture` kopira između matične memorije i memorije uređaja neće se desiti da objekat na uređaju sačuva pokazivač na objekat u matičnoj memoriji. Pokazivač na niz piksela može se dobiti pozivom šablonske funkcije `get_ptr` iz listinga 4.10.

```
1  template<typename T> struct Handle<T[]> {
2    // ...
3  private:
4    u32 m_offset;
5    i32 m_size;
6  };
7  template<typename T> struct ArenaPtr<T[]> {
8    // ...
9    T& operator[](i32 index);
10   const T& operator[](i32 index) const;
11 private:
12   i32 m_size;
13   T * m_elements;
14 };
15 struct MemoryArena {
16   template<typename T> ArenaPtr<T[]> alloc_array(usize n);
17   template<typename T> Handle<T[]> get_handle(ArenaPtr<T[]> ptr) const;
18   template<typename T> ArenaPtr<T[]> get_ptr(Handle<T[]> handle) const;
19 };
```

Listing 4.11: Interfejs memorijske arene za nizove.

Memorijska arena ima zaseban interfejs za alokaciju i čuvanje referenci na nizove, jer je za svaki niz potrebno čuvati i koliko elemenata se u njemu nalazi. Za rad sa nizovima koriste se šablonske specijalizacije struktura `Handle<T[]>` i `ArenaPtr<T[]>` iz listinga 4.10. Struktura `Handle<T[]>` pored relativne početne pozicije elemenata u areni sačuvane u polju `m_offset`, čuva i broj elemenata u nizu u polju `m_size` i ima istu semantiku kao i struktura `Handle<T>`. Struktura `ArenaPtr<T[]>` pored pokazivača na elemente `m_elements`, takođe čuva broj elemenata u nizu i umesto operatora dereferenciranja, koje ima struktura `ArenaPtr<t>`, ima operator indeksiranja `[]` za pristup elementima niza. Na listingu 4.11 u liniji 16 nalazi se deklaracija šablonske funkcije za alociranje niza od `n` elemenata tipa `T`, a u liniji 17 i 18 deklarisane su funkcije koje virtuelnu adresu niza koja se nalazi u `ArenaPtr<T[]>` pretvaraju u relativnu koja se čuva u `Handle<T[]>` i obratno.

Dakle, u modulu za upravljanje memorijom struktura `ArenaPtr<T>` predstavlja stvarnu adresu objekta u memoriji, a struktura `Handle<T>` relativnu adresu objekta u memorijskoj areni. Objekti čuvaju reference na druge objekte pomoću strukture `Handle<T>` da bi kopiranje podataka između matične memorije i memorije uređaja bilo bezbedno, a tokom izvršavanja funkcija na objekte se referiše pomoću strukture `ArenaPtr<T>`. Moguće je konvertovati objekat tipa `Handle<T>` u objekat tipa `ArenaPtr<T>` pomoću funkcije `get_ptr`, a obrnuto pomoću funkcije `get_handle`. Slično, moguće je konvertovati objekat tipa `Handle<T[]>` u objekat tipa `ArenaPtr<T[]>` pomoću preopterećene verzije funkcije `get_ptr`, a obrnuto pomoću preopterećene verzije funkcije `get_handle`.

rt_ray

Modul `rt_ray` implementira funkciju `boja_zraka` iz algoritma 2.3 i prateće strukture podataka. U poglavlju 2.5 funkcija za računanje boje zraka u algoritmu 2.3 je data u rekurzivnom obliku koji je lakši za čitanje i razumevanje, ali nedovoljno efikasan u brzini izvršavanja.

```
1  rgb ray_color(Ray r, World *world, i32 depth,
2            RaytraceState *state) {
3     RayHit ray_hit;
4     RayScatter ray_scatter;
5     rgb attenuation = rgb::white();
6     rgb result = rgb::black();
7     for (i32 i = 0; i < depth; ++i) {
8        if (world->hit(&ray_hit, &r, 0.001, infinity, state)) {
9           rgb emitted = ray_hit.texture->emitted(&ray_hit, state);
10          result += emitted * attenuation;
11          b32 scattered = ray_hit.texture->scatter(
12            &ray_scatter, &r, &ray_hit, state);
13          if (scattered) {
14            attenuation *= ray_scatter.attenuation;
15            r = ray_scatter.ray;
16          } else {
17            break;
18          }
19       } else {
20          result += attenuation * world->background;
21          break;
22       }
23    }
24    return result;
25 }
```

Listing 4.12: Iterativna implementacija funkcije za računanje boje zraka.

Takođe, zbog ograničenosti veličine steka niti na platformi CUDA nije moguće dostići dovoljnu dubinu rekurzivnog poziva funkcije `boja_zraka` iz algoritma 2.3 kako bi se izračunala boja svakog zraka. Zbog toga je implementacija funkcije za određivanje boje zraka iterativna.

U algoritmu 2.3 u poslednjoj naredbi `return` boja zraka računa se kao

```
e + a * boja_zraka(z, scena)
```
(4.1)

gde je `e` promenljiva `emitted` iz listinga 4.12, promenljiva `a` je promenljiva `ray_scatter.attenuation` iz listinga 4.12, a `z` je zrak transformisan interakcijom sa objektom i u listingu 4.12 odgovara promenljivoj `ray_scatter.ray`. Kada se izraz (4.1) razmota do dubine `n` dobije se izraz

```
e0 + a0 * (e1 + a1 * (e2 + a2 * (...en)))
```
(4.2)

u kojem indeksi promenljivih predstavljaju redni broj iteracije petlje `for` iz listinga 4.12, a vrednost `n` parametar `depth` funkcije `ray_color`. Međurezultati izraza (4.2) akumuliraju se u promenljivoj `result` koja će na kraju iteracije petlje `for` imati vrednost boje zraka `r`. Promenljiva `ray_hit` iz listinga 4.12 čuva podatke o tački preseka zraka i objekta potrebnih za računanje boje, a promenljiva `ray_scatter` zrak nakon njegove interakcije sa objektom. Promenljiva `attenuation` je promenljiva `a` iz algoritma 2.3, a promenljiva `result` se koristi za akumulaciju rezultata.

Petlja `for` u listingu 4.12 ponavlja odbijanje zraka po sceni dok zrak ne dođe do izvora svetlosti ili se ne dostigne maksimalni broj odbijanja `depth`. Poziv funkcije `world->hit` računa da li se zrak `r` sudara sa nekim objektom na sceni i ukoliko se sudara povratna vrednost funkcije `hit` biće `true`, a u promenljivoj `ray_hit` biće upisani podaci o objektu potrebni za izračunavanje boje. Ako se zrak ne sudara ni sa jednim objektom na sceni, onda se u konačan rezultat u liniji 20 na listingu 4.12 dodaje boja pozadine scene `world->background` i zaustavlja odbijanje zraka po sceni izlaskom iz petlje `for`.

Ako se zrak sudario sa nekim objektom na sceni sledeći korak je izračunati boju koju taj objekat emituje pozivom funkcije `emitted` u liniji 9 listinga 4.12 nad materijalom sa kojim se zrak sudario. Emitovana boja sačuvana u promenljivoj `emitted` dodaje se na konačan rezultat i nakon toga se u liniji 11 listinga 4.12 poziva funkcija `scatter` koja računa promenu smera zraka `r` prilikom interakcija sa materijalom `texture` i rezultat izračunavanja upisuje u polje `ray_scatter`. Ako zrak nije upijen, onda će vrednost promenljive `scattered` u liniji 11 listinga 4.12 biti `true` i tada se promenljiva `attenuation` u liniji 14 množi sa izmenjenom bojom transformisanog zraka `ray_scatter.attenuation`. U trenutku poziva funkcije `ray_color` parametar `r` sadrži podatke o zraku puštenom ka sceni. Pozivima funkcija unutar petlje `for` u listingu 4.12 se zrak uvek prosleđuje preko promenljive `r`. Zbog toga je u liniji 15 na listingu 4.12 potrebno ažurirati promenljivu `r` na transformisan zrak `ray_scatter.ray`. Ako je zrak bio upijen, promenljiva `scattered` će imati vrednost `false` i ulaskom u granu `else` u liniji 16 listinga 4.12 zaustavlja se odbijanje zraka po sceni.

rt_material

U modulu `rt_material` implementirane su funkcije `emitted` i `scatter` koje se pozivaju u funkciji `ray_color` iz listinga 4.12. Dizajn modula omogućava dodavanje novih materijala definisanjem interakcije sa zrakom preko funkcija `emitted` i `scatter`. Materijal se može dodeliti bilo kom obliku, nezavisno od njegove geometrije.

```
1  struct Material {
2    enum class Type {
3      None = 0,
4      Lambertian,
5      Metal,
6      Dielectric,
7      Isotropic,
8      DiffuseLight,
9    };
10   struct Lambertian { /*...*/ };
11   struct Metal { /*...*/ };
12   struct Dielectric { /*...*/ };
13   struct Isotropic { /*...*/ }
14   struct DiffuseLight { /*...*/ };
15   Type type;
16   union {
17     Lambertian lambertian;
18     Metal metal;
19     Dielectric dielectric;
20     Isotropic isotropic;
21     DiffuseLight diffuse_light;
22   };
23   // ...
24   b32 scatter(RayScatter *scatter_result, const Ray *r,
25               const RayHit *rec, RaytraceState *rc);
26   rgb emitted(const RayHit *ray_cast, RaytraceState *rc);
27 };
```

Listing 4.13: Definicija strukture `Material`.

Struktura `Material` iz listinga 4.13 sadrži materijale opisane u poglavlju 2.4. Funkcija `scatter` definiše kako se zrak transformiše prilikom interakcije sa materijalom, a funkcija `emitted` definiše računanje boje svetlosti koju objekat koji je izvor svetlosti emituje. Novi materijal definiše se dodavanjem naziva u enumerator

`Type` iz listinga 4.13, strukture koje sadrži podatke koje opisuju taj materijal i implementacijom interakcije zraka sa novim materijalom u funkcijama `scatter` i `emitted`.

```
1  b32 Material::scatter(RayScatter *scatter_result, const Ray *r,
2                        const RayHit *rec, RaytraceState *rc) {
3    switch (type) {
4    case Material::Type::Lambertian: { /*...*/ } break;
5    case Material::Type::Metal: { /*...*/ } break;
6    case Material::Type::Dielectric: { /*...*/ } break;
7    case Material::Type::Isotropic: {
8      scatter_result->ray = Ray(rec->p,
9          v3::sample_from_sphere(rc->random_state), r->time);
10     scatter_result->attenuation = isotropic.texture.value(
11         rec->texture_coordinates, rec->p, rc);
12     return true;
13   } break;
14   case Material::Type::DiffuseLight: { /*...*/ } break;
15   case Type::None: { /*...*/ } break;
16   default: {
17     SHOULD_NOT_REACH("Type not implemented!");
18   }
19   }
20   return false;
21 }
```

Listing 4.14: Funkcija za računanje interakcije zraka sa materijalom.

U listingu 4.14 prikazan je deo implementacije funkcije `scatter`, a funkcija `emitted` je već bila prikazana u listingu 4.1. Preko naredbe `switch` funkcija `scatter` implementira interakciju zraka sa različitim materijalima na osnovu tipa materijala `type`. Interakcija sa novim materijalom dodaje se kao novi slučaj naredbe `switch`. U listingu 4.14 prikazana je implementacija interakcije zraka sa izotropičnim materijalom unutar grane `case Material::Type::Isotropic` u liniji 7. Funkcija `scatter` iz listinga 4.14 u promenljivu `scatter_result.ray` upisuje zrak koji je transformisan interakcijom sa materijalom, a u promenljivu `scatter_result.attenuation` boju zraka nakon interakcije sa materijalom.

rt_world

U modulu `rt_world` implementirani su objekti koji se mogu postaviti na scenu, strukture podataka koje čuvaju sve elemente scene i strukture podataka za ubrzavanje detekcije preseka zraka sa objektima na sceni.

```
1  struct Hittable {
2    enum class Type {
3      None,
4      Box,
5      BVHNode,
6      ConstantMedium,
7      Array,
8      // ...
9    };
10   Type type;
11   union {
12     Box box;
13     BVHTree node;
14     Sphere sphere;
15     ConstantMedium constant_medium;
16     // ...
17   };
18   b32 hit(RayHit *rec, const Ray *r,
19       f32 t_min, f32 t_max, RaytraceState *rc);
20
21   b32 bounding_box(AABB *output_box,
22       f32 time0, f32 time1, MemoryArena *arena) const;
23 };
```

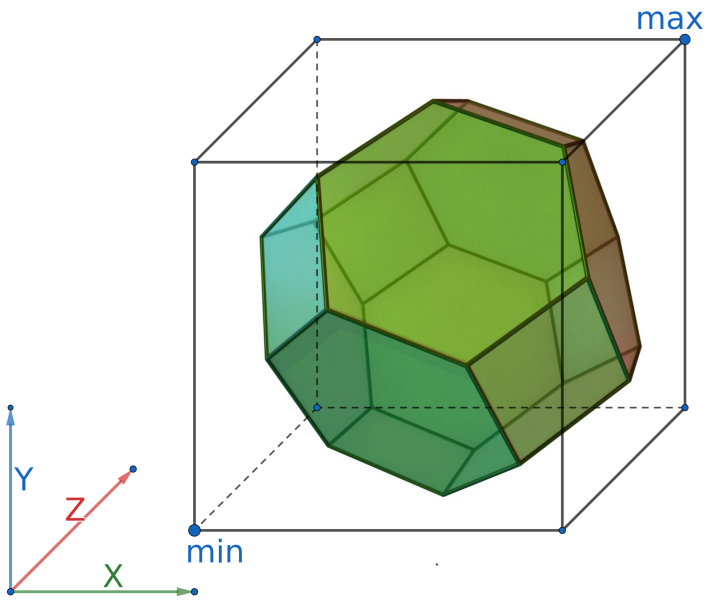
Listing 4.15: Definicija strukture `Hittable`.

U listingu 4.15 dat je deo definicije strukture `Hittable` koja čini osnovni gradivni element scene. Struktura `Hittable` iz listinga 4.15 je označena unija svih objekata koje zrak može pogoditi, ali ne nužno svih objekata koji se mogu iscrtati. Članska funkcija `hit` iz listinga 4.15 računa da li postoji presek zraka `r` sa objektom u vremenskom periodu `[t_min, t_max]` i ako postoji vraća vrednost `true` i relevantne podatke materijala objekta i tačke preseka upisuje na memorijsku lokaciju `rec`, a u suprotnom vraća `false`. Kada je potrebno napraviti novi objekat koji se može postaviti na scenu dovoljno je napraviti strukturu koja sadrži sve podatke o tom tipu objekta, dodati je u uniju iz listinga 4.15, dodati polje u enumeratoru

`Type` iz listinga 4.15 i implementirati funkciju `hit` koja računa presek zraka `r` sa tim objektom.

```
1  struct AABB {
2    math::v3 min;
3    math::v3 max;
4    // ...
5  };
6
7  b32 aabb_hit(const AABB &aabb, const rt::Ray &r, f32 t_min, f32 t_max);
8  AABB aabb_surrounding_box(AABB box0, AABB box1);
```

Listing 4.16: Definicija strukture `AABB`.



Slika 4.4: Ilustracija kutije AABB koja sadrži oktahedron.

Za ubrzavanje detekcije kolizije zraka i objekta koriste se *stabla graničnih opsega* (eng. bounding volume tree) koja rekurzivno dele prostor na *osno poravnate kutije* (eng. **a**xis **a**ligned **b**ounded **b**ox) i koriste se zbog efikasnosti računanja preseka zraka i neke od stranica kutije. Definicija strukture osno poravnate kutije `AABB` data je u listingu 4.16 i sastoji se od dve dijagonalne tačke kocke `min` i `max`. Funkcija `aabb_hit` iz listinga 4.16 vraća `true` ako zrak `r` ima presek sa kutijom `aabb` u vremenskom intervalu `[t_min, t_max]`, a funkcija `aabb_surrounding_box` konstruiše kutiju minimalne zapremine koja sadrži kutije `box0` i `box1`. Detekcija kolizije zraka sa nekim objektom na sceni se ubrzava korišćenjem stabla kocki `AABB`

konstruisanim nad scenom gde su listovi individualni objekti scene, a unutrašnji čvorovi su kocke `AABB` minimalne zapremine koje sadrže sve objekte na sceni koji su u stablu direktni potomci tih unutrašnjih čvorova. Koren stabla je kocka `AABB` koja obuhvata celu scenu. Članska funkcija `bounding_box` iz listinga 4.15 konstruiše kutiju `AABB` minimalne zapremine koja sadrži objekat nad kojim je pozvana i koristi se prilikom konstrukcije stabla graničnih opsega nad scenom koje članska funkcija `hit` koristi kako bi se ubrzala detekcija kolizije zraka i objekta na sceni.

Stabla graničnih opsega smanjuju broj objekata za koje se vrši provera da li postoji presek zraka i objekta tako što ograničavaju proveru preseka na objekte koji se nalaze u osno poravnatoj kutiji kroz koju zrak prolazi. Provera da li zrak prolazi kroz neku osno poravnatu kutiju je efikasnija od provere da li zrak ima presek sa nekim objektom. Za svaki čvor u stablu graničnih opsega važi da ako zrak ima presek sa osno poravnatom kutijom tog čvora, onda možda ima presek i sa objektom koji se nalazi u direktnom potomku tog čvora. U suprotnom, ako zrak nema presek sa osno poravnatom kutijom čvora stabla graničnih opsega, onda nema presek ni sa jednim potomkom u tom stablu jer svaka osno poravnata kutija u stablu graničnih vrednosti sadrži sve njene potomke.

```
1 struct World {
2   Camera camera;
3   Scene scene;
4   // ...
5   b32 hit(RayHit *rec, const Ray *r,
6       f32 t_min, f32 t_max, RaytraceState *rc);
7 };
```

Listing 4.17: Definicija strukture `World`.

Na slici 4.4 ilustrovana je kutija `AABB` određena tačkama `min` i `max`, konstruisana oko dodekahedrona. Stranice kutije poravnate su sa koordinatnim osama `X`, `Y` i `Z` da bi testiranje preseka zraka i neke od stranica kutije bilo efikasno. U konkretnom slučaju sa slike 4.4 efikasnije je prvo izračunati da li zrak seče neku od stranica kutije i ako seče izračunati da li seče i neku od stranica dodekahedrona, nego za svaki zrak računati da li seče neku od stranica dodekahedrona jer većina zraka koji su pušteni ka sceni nemaju presek sa većinom objekata na sceni.

Na listingu 4.17 prikazana je definicija strukture `World` koja sadrži kameru, određenu poljem `camera` i scenu određenu poljem `scene` u kojoj se nalazi svi objekti na sceni. Članska funkcija `hit` iz listinga 4.17 poziva se u liniji 8 na listingu 4.12 i računa da li zrak `r` pogađa neki od objekata na sceni u vremenskom intervalu `[t_min, t_max]` i ako pogađa vraća `true` i rezultat pogotka upisuje u `rec`, a u suprotnom vraća `false`.

4.2 Ulazna tačka programa

U ovom poglavlju biće prikazana ulazna tačka programa koju dele implementacije tehnika praćenja zraka na jednoj niti matičnog procesora i platformi CUDA. Program je struktuisan pomoću bazne klase `RayTracer` koju različite implementacije nasleđuju i modifikuju.

```
1  // main.cu
2  #include "rt_runner.h"
3  int main(int argc, const char **argv) {
4    rt::RayTracer::run(argc, argv);
5    return 0;
6  }
7  // rt_runner.cu
8  void RayTracer::run(int argc, const char **argv) {
9    // ...
10   parse_command_line_arguments(argc, argv);
11   RayTracer *ray_tracer = nullptr;
12   if (m_runtime_arg.runtime_type.value() == CPU_SINGLE_THREADED) {
13     ray_tracer = new SingleCoreCpuRayTracer();
14   } else if (m_runtime_arg.runtime_type.value() == CUDA_PARALLEL) {
15     ray_tracer = new CudaRayTracer();
16   } else {
17     fprintf(stderr, "unsupported ray-tracer type\n");
18     std::exit(EXIT_FAILURE);
19   }
20   if (!ray_tracer->initialize()) {
21     fprintf(stderr, "Failed to initialize\n");
22     std::exit(EXIT_FAILURE);
23   }
24   ray_tracer->setup_scene();
25   // ...
26   ray_tracer->render();
27 // ...
28 }
```

Listing 4.18: Ulazna tačka izvršavanja programa.

Funkcija `main` prikazana u listingu 4.18 poziva statičku člansku funkciju `run` koja inicijalizuje program, postavlja scenu i iscrtava je tehnikom praćenja zraka. Pomoću argumenata komandne linije moguće je odbrati vrstu implementacije tehnike praćenja zraka i podesiti parametre scene. Statička funkcija `run` imple-

mentira obrazac za projektovanje *šablonske metode* (eng. template method) koja poziva funkciju inicijalizacije `initialize` u liniji 20 na listingu 4.18, zatim postavlja scenu pozivom funkcije `setup_scene` u liniji 24 i iscrtava scenu pozivom funkcije `render` u liniji 26. Implementacije funkcija `initialize`, `setup_scene` i `render` koje će biti pozvane zavise od dinamičkog tipa objekta `RayTracer` u liniji 12 na listingu 4.18.

```
1 class RayTracer {
2 public:
3   static void run(int argc, const char **argv);
4 protected:
5   virtual b32 initialize();
6   virtual void setup_scene();
7   virtual void render() = 0;
8   // ...
9 };
```

Listing 4.19: Bazna klasa `RayTracer`.

Klasa `RayTracer` prikazana na listingu 4.19 predstavlja baznu klasu različitih implementacija i sadrži tri funkcije koje izvedena klasa treba da implementira. Članska funkcija `initialize` inicijalizuje program i vraća vrednost `true` ako je inicijalizacija uspešna, a u suprotnom vraća `false`. Na primer, u slučaju da je odabrana implementacija na platformi CUDA, a program ne može da detektuje grafičku jedinicu inicijalizacija neće uspeti. Članska funkcija `setup_scene` učitava resurse potrebne sceni, na primer teksture i slike, pravi objekte na sceni i konstruiše prateće strukture koje se koriste za ubrzavanje praćenja zraka. Različite implementacije dele veći deo koda funkcija `initialize` i `setup_scene`, zbog toga ove dve funkcije imaju podrazumevanu implementaciju koje izvedene klase mogu dopuniti po potrebi. Virtuelnu funkciju `render` definiše svaka implementacija tehnike praćenja zraka zasebno jer je algoritam iscrtavanja scene suštinski različit za jednu nit matičnog procesora i platformu CUDA. Pozivom funkcije `render` iscrtava se postavljena scena i rezultat iscrtavanja se upisuje na disk u formatu slike.

```
class SingleCoreCpuRayTracer : public RayTracer {
public:
  using RayTracer::RayTracer;
protected:
  b32 initialize() override;
  void render() override;
  // ...
};
class CudaRayTracer : public RayTracer {
public:
  using RayTracer::RayTracer;
protected:
  b32 initialize() override;
  void render() override;
  // ...
};
```

Listing 4.20: Klasa `SingleCoreCpuRayTracer` implementacije na jednoj niti procesora i klasa `CudaRayTracer` implementacije na platformi CUDA.

```
b32 rt::RayTracer::initialize() {
  // ...
  using namespace rt::memory;
  m_host_memory_arena = MemoryArena::create_with_os_mapped(
    compute_host_arena_size(&m_runtime_args));
  if (m_host_memory_arena.type() == MemoryBlockStorageType::None) {
    fprintf(stderr, "Failed to initialize MemoryArena\n");
    return false;
  }
  // ...
  return true;
}
b32 rt::SingleCoreCpuRayTracer::initialize() {
  if (!RayTracer::initialize()) {
    return false;
  }
  // ...
  return true;
}
```

Listing 4.21: Prikaz ponovnog korišćenja i dopune zajedničkog dela koda inicijalizacije programa.

Na listingu 4.20 prikazane su izvedene klase `SingleCoreCpuRayTracer` koja implementira tehniku praćenja zraka na jednoj niti i klasa `CudaRayTracer` koja implementira tehniku praćenja zraka na platformi CUDA. Za dodavanje nove implementacije, na primer na više niti na matičnom procesoru ili hibridne koja u isto vreme koristi i matični procesor i uređaj, dovoljno je definisati novu klasu koja nasleđuje klasu `RayTracer`, implementirati funkcije `initialize` i `render` i dodati je kao opciju u funkciji `run` iz listinga 4.18.

Na listingu 4.21 prikazan je deo podrazumevane implementacije funkcije `RayTrace::initialize` i funkcije `initialize` iz klase `SingleCoreCpuRayTracer` da bi se prikazao način na koji klase izvedene iz klase `RayTracer` mogu dopuniti podrazumevanu implementaciju neke od funkcija bazne klase. Na primer, deo inicijalizacije koju dele obe implementacije je pravljenje memorijske arene u matičnoj memoriji, zbog toga se memorijska arena inicijalizuje u liniji 4 na listingu 4.21. Ako pravljenje memorijske arene `m_host_memory_arena` ne uspe, onda se u liniji 8 na listingu 4.21 izlazi iz funkcije `initalize`. Funkcija `SingleCoreCpuRayTracer::initialize` u liniji 14 na listingu 4.21 prvo poziva funkciju `RayTracer::initialize` i ako je poziv uspešan nastavlja sa inicijalizacijom specifičnom za implementaciju tehnike praćenja zraka na jednoj niti matičnog procesora.

4.3 Implementacija na jednoj niti matičnog procesora

Tehnika praćenja zraka na jednoj niti matičnog procesora implementirana je radi poređenja ubrzanja koje se dobije paralelizacijom tehnike na platformi CUDA. U ovom poglavlju biće prikazana samo definicija funkcije za iscrtavanje jer implementacija klase `SingleCoreCpuRayTracer` koristi podrazumevanu implementaciju bazne klase `RayTracer`.

```
1  void rt::SingleCoreCpuRayTracer::render() {
2    // ...
3    for (i32 j = image_height - 1; j >= 0; --j) {
4      for (i32 i = 0; i < image_width; ++i) {
5        rgb color = v3(0.0);
6        for (int s = 0; s < samples_per_pixel; ++s) {
7          f32 u = f32(i + math::random::uniform(0.0, 1.0, /*..*/))
8              / (image_width - 1.0f);
9          f32 v = f32(j + math::random::uniform(0.0, 1.0, /*..*/))
10             / (image_height - 1.0f);
11         Ray r = cast_ray_from_camera(world->camera, v2(u, v), /*..*/);
12         color += ray_color(r, world.get(), max_ray_depth, /*..*/);
13       }
14       image.at(i, j) = color;
15     }
16   }
17   // ...
18   image.write_to_file(m_runtime_arg.image_output_path.value(),
19     m_runtime_arg.samples_per_pixel, /*..*/);
20 }
```

Listing 4.22: Implementacija funkcije `render` za izvršavanje tehnike praćenja zraka na jednoj niti matičnog procesora.

Implementacija članske funkcije `SingleCoreCpuRayTracer::render` deklarisane u listingu 4.20 je slična algoritmu 2.1. Na listingu 4.22 prikazan je glavni deo definicije funkcije `render` koja implementira tehniku praćenja zraka tako da se izvršava na jednoj niti matičnog procesora. Zbog jednostavnosti prikaza neki detalji implementacije su izostavljeni iz listinga 4.22. Petlja `for` u liniji 3 na listingu 4.22 iterira kroz indekse redova slike, a petlja `for` u liniji 4 kroz indekse

kolona slike. Boja piksela `(i, j)` čuva se u promenljivoj `color` i na početku petlje ima vrednost crne boje. Petlja `for` u liniji 6 na listingu 4.22 više puta pokreće računanje boje zraka koji se pušta kroz piksel `(i, j)`. Zrak se uvek malo pomeri sa svoje putanje kako bi se ublažio efekat oštrih ivica objekata prilikom iscrtavanja scene, a koordinate piksela `(i,j)` se skaliraju na opseg `[0,1]` da bi matematičke operacije nad vektorima u funkcijama `cast_ray_from_camera` i `ray_color` bile jednostavnije za implementaciju. Skaliranje koordinata piksela `(i, j)` i malo pomeranje zraka sa svoje putanje računa se u linijama 7 i 9 na listingu 4.22 tako što se koordinata piksela prvo izmesti za vrednost `uniform(0.0, 1.0)`, a zatim se skalira na opseg `[0,1]` deljenjem sa maksimalnom vrednošću koordinate. Pozivu funkcije `cast_ray_from_camera` se zatim prosleđuje skaliran piksel `(u, v)`. Broj uzoraka boje zraka `samples_per_pixel` se obično određuje eksperimentalno jer veća vrednost promenljive `samples_per_pixel` dovodi do većeg kvaliteta slike, ali i sporijeg iscrtavanja jer je potrebno izračunati sve uzorke. Zbog toga je potrebno pronaći optimalnu vrednost čijim daljim povećavanjem se ne dobija primetna razlika u kvalitetu slike. Dobijena boja zraka `color` upisuje se u sliku `image` u liniji 14 na listingu 4.22 i nakon toga se računa boja narednog piksela. Kada se izračunaju boje svih piksela, slika iscrtane scene upisuje se na disk u liniji 18 na putanji `image_output_path` i unutar funkcije `write_to_file` se akumulirane boje u slici `image` skaliraju na opseg boja piksela koji odgovara formatu slike.

Funkcija `render` sa listinga 4.22 je veoma slična algoritmu 2.1, a razlikuje se po tome što funkcija `render` skalira koordinate piksela na opseg `[0,1]` i implementira tehniku anti-alijasinga računanjem više vrednosti boje zraka za jedan piksel.

4.4 Implementacija na platformi Nvidia CUDA

Implementacija tehnike praćenja zraka na platformi CUDA deli većinu koda sa implementacijom tehnike praćenja zraka na matičnom procesoru. U ovom poglavlju biće prikazana inicijalizacija platforme CUDA, funkcija `render` i kernel funkcija za iscrtavanje scene tehnikom praćenja zraka.

```
1  b32 CudaRayTracer::initialize() {
2    //...
3    if (!RayTracer::initialize()) {
4      return false;
5    }
6    int version;
7    cudaRuntimeGetVersion(&version);
8    CUDA_CHECK_ERR("cuda runttime get version failed");
9    // ...
10   usize stack_size = KB(4);
11   cudaDeviceSetLimit(cudaLimitStackSize, stack_size);
12   CUDA_CHECK_ERR("cuda set limit failed");
13   // ...
14   return true;
15 }
```

Listing 4.23: Implementacija funkcije za inicijalizaciju uređaja platforme CUDA.

Funkcija `CudaRayTracer::initialize` iz listinga 4.23 inicijalizuje uređaj platforme CUDA. U liniji 3 na listingu 4.23 inicijalizuje se deo programa koji je zajednički za obe implementacije i nakon toga započinje inicijalizacija uređaja platforme CUDA pozivom funkcije `cudaRuntimeGetVersion` koja u promenljivu `version` upisuje verziju izvršnog okruženja platforme CUDA. Ako poziv funkcije `cudaRuntimeGetVersion` ne uspe, znači da drajver platforme CUDA nije uspeo da pronađe uređaj na sistemu. Poziv funkcije `CUDA_CHECK_ERR` u liniji 8 na listingu 4.23 proverava da li je došlo do greške pri pozivu funkcije programskog interfejsa platforme CUDA i ako jeste zaustavlja izvršavanje programa, a na standardni izlaz za greške ispisuje poruku. U liniji 11 na listingu 4.23 pozivom funkcije `cudaDeviceSetLimit` povećava se veličina steka svake niti na platformi CUDA jer je implementaciji neophodno više memorije za stek svake niti od podrazumevane veličine. U projektu postoji još operacija koje se izvode prilikom inicijalizacije platforme CUDA, ali su izostavljeni iz listinga 4.23 jer nisu od suštinskog značaja.

```
1  void CudaRayTracer::render() {
2    //...
3    m_device_memory_arena = MemoryArena::create_with_cuda_malloc(
4         m_host_memory_arena.size());
5    m_host_memory_arena.memcpy_to_device(&m_device_memory_arena);
6    //...
7    dim3 blocks_dim = compute_blocks_layout(&m_runtime_arg);
8    dim3 threads_dim = compute_threads_layout(&m_runtime_arg);
9    cuda_raytrace<<<blocks_dim, threads_dim>>>(
10      m_world_handle, m_image_handle,
11      m_device_memory_arena.pass_to_cuda_kernel(), /*...*/);
12   CUDA_CHECK_ERR("cuda_ray_trace");
13   CUDA_ASSERT(cudaDeviceSynchronize());
14
15   m_device_memory_arena.memcpy_to_host(&m_host_memory_arena);
16   CUDA_ASSERT(cudaDeviceSynchronize());
17
18   Image image = m_image_handle.unwrap(&m_host_memory_arena);
19   image.write_to_file(m_runtime_arg.image_output_path.value(),
20      camera->samples_per_pixel);
21 }
```

Listing 4.24: Implementacija funkcije iscrtavanja na platformi CUDA.

Na listingu 4.24 prikazana je definicija funkcije `render` koja iscrtava sliku scene tehnikom praćenja zraka na platformi CUDA. U liniji 3 na listingu 4.23 pravi se memorijska arena za uređaj koja će biti identičnog kapaciteta kao memorijska arena iz matične memorije u kojoj su alocirani svi objekti koji se nalaze na sceni. Sadržaj memorijske arene u matičnoj memoriji kopira se u memorijsku arenu na uređaju pozivom funkcije `memcpy_to_device` u liniji 5 na listingu 4.24. Dimenzije mreže blokova i mreže niti računaju se redom pozivom funkcija `compute_blocks_layout` i `compute_threads_layout` u linijama 7 i 8 na listingu 4.24. U liniji 9 na listingu 4.24 poziva se kernel funkcija `cuda_raytrace`, a u liniji 12 pomoću funkcije `CUDA_CHECK_ERR` se proverava da li je došlo do greške prilikom poziva kernel funkcije i ako jeste prekida se izvršavanje programa. U liniji 13 poziv funkcije `cudaDeviceSynchronize` blokira izvršavanje matične niti procesora u kojoj se poziv nalazi dok se kernel funkcija `cuda_raytrace` ne izvrši, a povratna vrednost se prosleđuje funkciji `CUDA_ASSERT` koja proverava da li je došlo do greške i ako jeste zaustavlja izvršavanje programa. Kada kernel funkcija završi iscrtavanje scene sadržaj arene uređaja kopira se iz memorije uređaja u memorijsku arenu u

matičnoj memoriji u liniji 15 na listingu 4.24. Na kraju, u liniji 19 na listingu 4.24 slika se upisuje na putanju `image_output_path` identično kao u funkciji `render` na listingu 4.22.

```
1  __global__
2  void cuda_raytrace(Handle<World> world_handle, ImageHandle image_handle,
3      MemoryArena device_arena, /*...*/) {
4    u32 i = threadIdx.x + blockIdx.x * blockDim.x;
5    u32 j = threadIdx.y + blockIdx.y * blockDim.y;
6    //...
7    ArenaPtr<World> world = device_arena.get_ptr(world_handle);
8    // ...
9    if (i < image_handle.width() && j < image_handle.height()) {
10     Image image = image_handle.unwrap(&device_arena);
11     rgb color = v3(0.0f);
12     for (int s = 0; s < samples_per_pixel; ++s) {
13       auto u = f32(i + math::random::uniform(0.0f, 1.0f, /*...*/))
14           / (image_width - 1.0f);
15       auto v = f32(j + math::random::uniform(0.0f, 1.0f, /*...*/))
16           / (image_height - 1.0f);
17       Ray r = cast_ray_from_camera(world->camera, v2(u, v), /*...*/);
18       color = color + ray_color(r, world.get(), max_ray_depth, /*...*/);
19     }
20     image.at(i, j) = color;
21   }
22 }
```

Listing 4.25: Kernel funkcija `cuda_raytrace`.

Na listingu 4.25 prikazana je definicija kernel funkcije `cuda_raytrace` koja se poziva u liniji 9 na listingu 4.24. Prvi parametar `world_handle` kernel funkcije `cuda_raytrace` predstavlja svet koji sadrži scenu, kameru i prateće strukture podataka potrebne prilikom iscrtavanja tehnikom praćenja zraka. Drugi parametar `image_handle` kernel funkcije `cuda_raytrace` je slika u koju se upisuje iscrtana scena. Treći parametar `device_arena` kernel funkcije `cuda_raytrace` je memorijska arena u kojoj su alocirani svi objekti scene i prateće strukture podataka potrebne prilikom iscrtavanja scene. U liniji 4 na listingu 4.25 se pomoću ugrađenih konstanti `threadIdx`, `blockIdx` i `blockDim` određuje broj kolone piksela na slici, a u liniji 5 broj reda piksela na slici. Kôd iz listinga 4.25 od linije 11 do linije 20 identičan je kodu iz listinga 4.22 od linije 5 do linije 14.

Glava 5

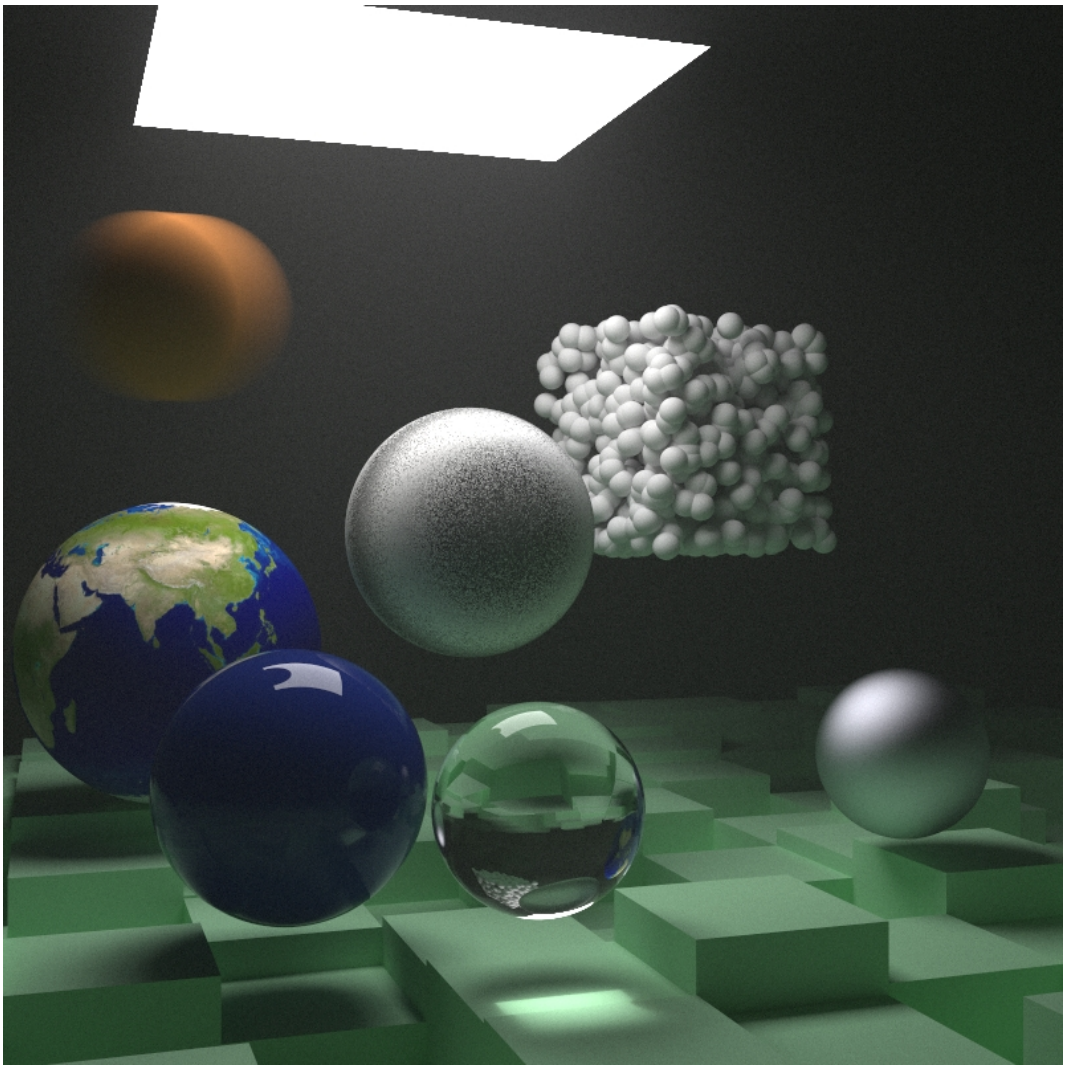
Uporedni prikaz rezultata

U ovom poglavlju predstavljeni su dobijeni rezultati, navedeni predlozi daljih unapređenja i predstavljena aktuelna dešavanja u oblasti iscrtavanja tehnikom praćenja zraka. Od aktuelnih dešavanja opisane su tehnologije koje se koriste za iscrtavanje tehnikom praćenja zraka i relevantna literatura ove oblasti.

5.1 Vreme izvršavanja

U daljem tekstu sledi slika scene koja je iscrtana tehnikom praćenja zraka i koja sadrži sve objekte i materijale opisane u poglavlju 2. Takođe, biće prikazan grafikon vremena koje je potrebno da se scena iscrta pomoću implementacije tehnike praćenja zraka na jednoj niti i na platformi CUDA i grafikon koji prikazuje ubrzanje dobijeno paralelizacijom na platformi CUDA. Na kraju poglavlja predložena su moguća unapređenja koja bi ubrzala iscrtavanje scene na platformi CUDA.

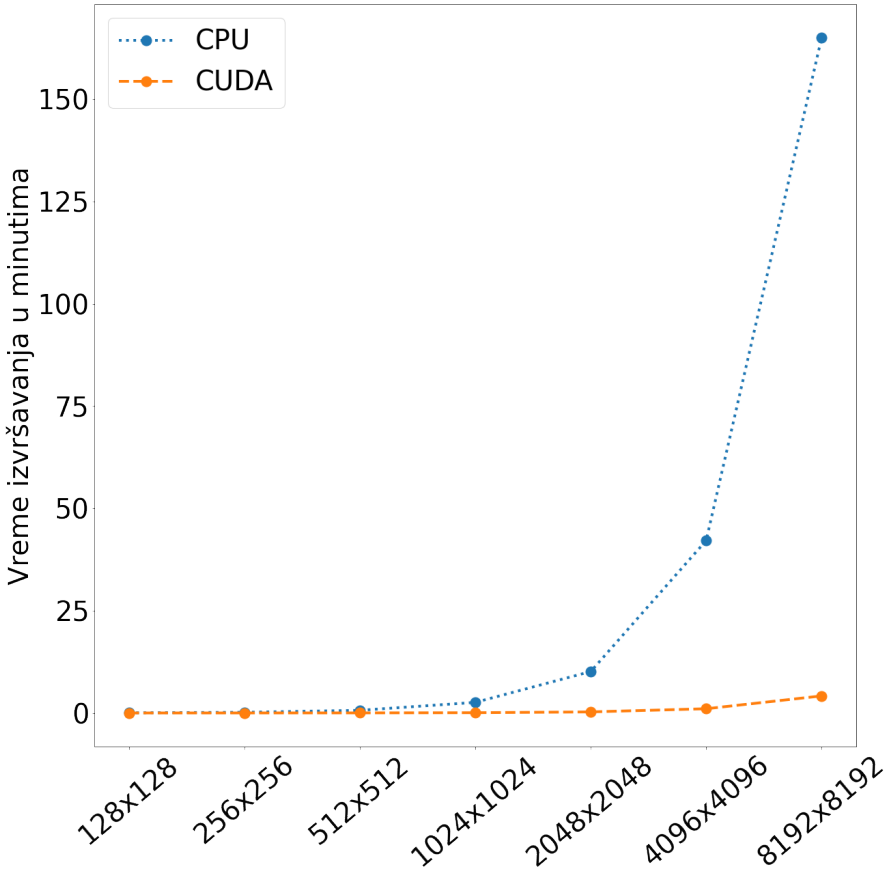
Vreme izvršavanja mereno je na računaru sa procesorom `AMD Ryzen 7 2700x` i grafičkom jedinicom `GeForce RTX 2060`. Program je preveden kompajlerom `nvcc` verzije `11.7.64` i matičnim kompajlerom `gcc` verzije `11.2` na operativnom sistemu `Ubuntu 20.04` sa uključenim optimizacijama `-O2`. Obično se mereno vreme izvršavanja prikazuje u formatu kutijica sa medijanom i percentilima, ali to u ovom slučaju nije neophodno jer algoritam ima stabilno vreme izvršavanja u više pokretanja nad istim ulazom.



Slika 5.1: Scena iscrtana pomoću tehnike praćenja zraka implementirane u ovom master radu [22][Slika 22].

Slika 5.1 predstavlja scenu iscrtanu tehnikom praćenja zraka implementiranom u ovom radu. Na sceni se nalaze sve vrste objekata i materijala opisanih u poglavlju 2. Sfera tamno narandžaste boje u gornjem levom delu slike je objekat u pokretu, zbog toga izgleda mutno, a ostale sfere na sceni su statične. Sfera koja izgleda kao planeta Zemlja ima difuzni materijal sa dodeljenom teksturom. Desno od planete Zemlje, na samoj sredini slike 5.1 nalazi se bela sfera, metalnog materijala sa visokim stepenom hrapavosti, a u donjem desnom delu slike 5.1 nalazi se još jedna sfera metalnog materijala, manjih dimenzija sa niskim stepenom hrapavosti. Ispred sfere koja izgleda kao planeta Zemlja nalazi se sfera izotropičnog materijala, plavkaste boje sa visokim stepenom gustinom, a desno od nje sfera providnog

materijala koja refraktuje svetlost koja na nju pada. U gornjem desnom delu slike 5.1 nalazi 1000 malih belih sfera poređanih u obliku kocke. Efekat maglovitosti scene dobijen je dodavanjem jedne velike sfere, izotropičnog materijala, dovoljno velikog poluprečnika da se cela scena nalazi unutar te sfere. Pod scene sastavljen je od kutija difuznog materijala zelene boje.

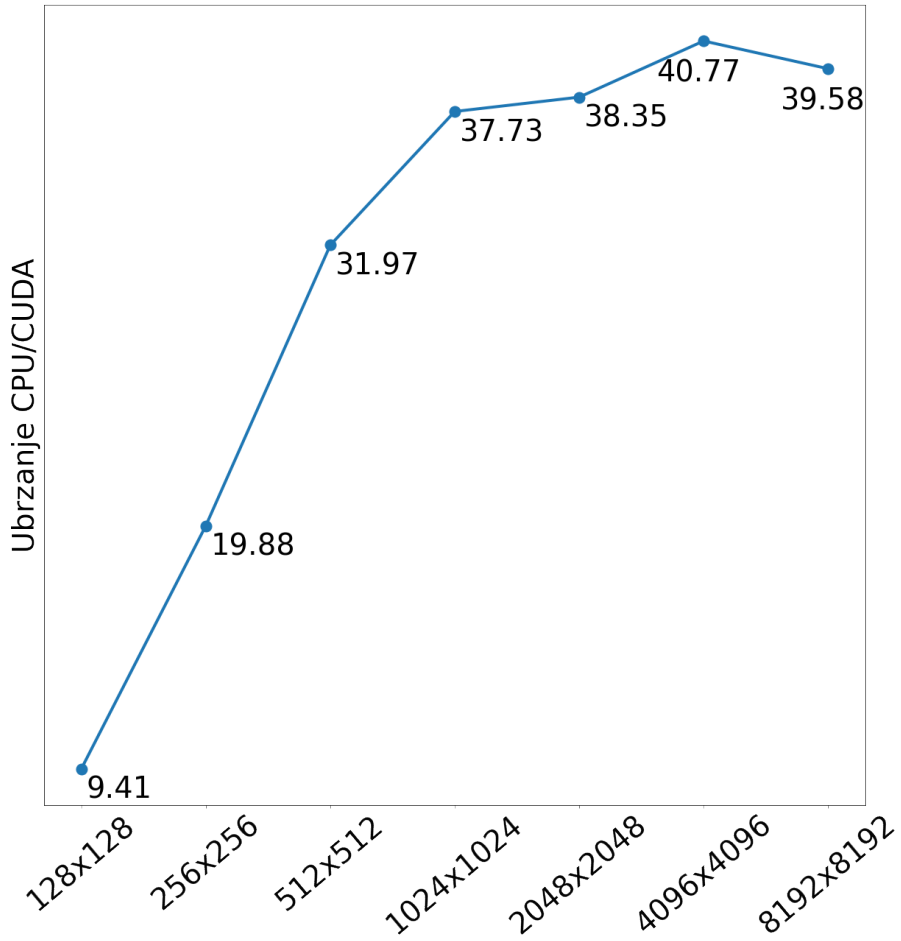


Slika 5.2: Ukupno vreme izvršavanja programa u minutima za implementaciju na matičnom procesoru i platformi CUDA.

Za scenu sa slike 5.1 mereno je vreme koje je potrebno da se scena iscrta u rezolucijama prikazanim na X-osi grafikona sa slike 5.2. Na Y-osi grafikona 5.2 nalazi se vreme izvršavanja programa u minutima od trenutka pokretanja do završetka. Za broj niti unutar bloka kernel poziva za iscrtavanje na platformi CUDA odabrana je vrednost `(16,16)` jer se pokazala kao optimalna od testiranih `(4,4)`, `(8,8)`, `(16,16)` i `(20,20)`. Broj blokova u mreži je postavljen na `(širina, visina)/(16,16)`.

Plava tačkasta linija predstavlja vreme izvršavanja na jednoj niti matičnog procesora, a narandžasta isprekidana linija vreme izvršavanja na platformi CUDA.

Kada je rezolucija slike mala i broj zraka koji će biti pušteni ka sceni je mali i tada nema velike razlike u vremenu izvršavanja. Međutim, sa povećanjem rezolucije slike povećava se broj zraka koji će biti pušteni ka sceni i vreme izvršavanja se drastično razlikuje.



Slika 5.3: Ubrzanje dobijeno paralelizacijom na platformi CUDA.

Na slici 5.3 prikazan je grafikon dobijenog ubrzanja izračunatog deljenjem izmerenog vremena izvršavanja na jednoj niti matičnog procesora sa vremenom izvršavanja na platformi CUDA. Na X-osi grafikona na slici 5.3 nalaze se rezolucije platna na kom je scena iscrtana, a na Y-osi dobijeno ubrzanje za datu rezoluciju.

Iz dobijenih rezultata merenja može se zaključiti da se paralelizacijom tehnike praćenja zraka na platformi CUDA, implementiranoj u ovom radu, može dobiti značajno ubrzanje iscrtavanja scene, ali postoji još prostora za poboljšanje. Dimenzije blokova i niti prilikom poziva kernel funkcije koja iscrtava scenu na platformi CUDA mogu uticati na vreme izvršavanja i možda postoji vrednost

za koju se program na platformi CUDA brže izvrši od testiranih `(4,4)`, `(8,8)`, `(16,16)` i `(20,20)`. U trenutnoj implementaciji na platformi CUDA teksture se uzorkuju identično kao i kod implementacije na jednoj niti matičnog procesora. Ako bi se uzorkovanje tekstura na platformi CUDA promenilo tako da koristi teksturnu memoriju grafičke jedinice i hardverski podržano uzorkovanje to bi ubrzalo iscrtavanje objekata sa nalepljenim teksturama. Svaki upis boje u kernel funkciji iz listinga 4.25 izvršava se direktno nad globalnom memorijom grafičke jedinice. Umesto direktnog upisivanja vrednosti u globalnu memoriju nakon što svaka nit izračuna boju piksela, bilo bi efikasnije privremeno čuvati boje piksela u deljenoj memoriji niti i tek kada se sve niti u jednom bloku izvrše, onda upisati sve izračunate boje piksela u globalnu memoriju grafičke jedinice. Objekti kojima se često pristupa kao što su osvetljenje i strukture podataka za ubrzavanje detekcije preseka zraka i objekta bi mogli da se čuvaju u konstantnoj memoriji kako bi niti grafičke jedinice mogle brže da im pristupe.

Paralelizacija na platformi CUDA u ovom radu deli većinu koda sa implementacijom na jednoj niti procesora. Posledica toga je gubitak na efikasnosti u vremenu izvršavanja za implementaciju na platformi CUDA. Niti na platformi CUDA izvršavaju se u pakovanju, obično od 16 ili 32 niti, što podrazumeva da sve niti u jednom pakovanju izvršavaju istu instrukciju. Ako kontrola toka jedne niti u pakovanju uđe u neku granu izvršavanja, sve ostale niti moraju sačekati na toj grani dok prva nit ne izvrši sve instrukcije u toj grani. Kada niti u jednom pakovanju računaju boje puštenih zraka često se dešava da jedan zrak bude upijen od strane objekta na sceni, a drugi zrak koji prati nit iz istog pakovanja bude odbijen od objekta. U tom slučaju dolazi do grananja između niti u jednom pakovanju i ostale niti u pakovanju moraju čekati dok nit čija se kontrola toka izdvojila grananjem ne završi sve instrukcije u datoj grani. Zbog toga se izbegavaju grananja u kodu koji se izvršava na platformi CUDA i u idealnom slučaju se izračunavanja implementiraju tako da nema grananja.

U implementaciji tehnike praćenja zraka nije moguće potpuno eliminisati grananje, ali se izračunavanja mogu osmisliti tako da se resursi raspoloživih multiprocesora grafičke jedinice maksimalno iskoriste. U tom slučaju bi se kôd implementacije tehnike praćenja zraka na platformi CUDA značajno razlikovao od implementacije na jednoj niti procesora, ali bi bio brži od implementacije predstavljene u ovom radu.

5.2 Tehnika praćenja zraka danas

Oblast iscrtavanja tehnikom praćenja zraka je aktivna oblast istraživanja u kojoj se prepliću oblasti matematike, fizike, dizajna hardvera, mašinskog učenja i matematičke optimizacije.

Trenutno najpopularnija tehnika iscrtavanja foto-realističnih slika je tehnika *praćanje putanja* (eng. path tracing). Zasniva se na radu [3] objavljenom 1984. godine u kojem je data ideja kako bi praćenje zraka moglo da se koristi u filmskoj industriji, što je do tada bilo nedostižno u oblasti računarske grafike. Dve godine kasnije, profesor Džim Kadžija (eng. Jim Kajiya) je u radu od sedam strana [6] povezao računarsku grafiku i fiziku algoritmom praćenja putanja na osnovu *jednačine iscrtavanja* (eng. rendering equation)

$$I(x, x') = g(x, x')\left[\epsilon(x, x') + \int_S \rho(x, x', x'')I(x', x'')dx''\right] \tag{5.1}$$

gde je

- $I(x, x')$ - intenzitet svetlosti od tačke x' do tačke x
- $g(x, x')$ - geometrija prostora od tačke x' do tačke x
- $\epsilon(x, x')$ - intenzitet emitovanog svetla od tačke x' do tačke x
- $\rho(x, x', x'')$ - intenzitet svetlosti od tačke x'' do tačke x dobijene rasipanjem svetla od površi u okolini tačke x'

Jednačina (5.1) je koncizna, ali je nije moguće rešiti direktno. Umesto direktnog rešenja, moguće je rešiti jednačinu (5.1) po putanjama individualnih zraka, a zatim aproksimirati osvetljenje scene. Sa dovoljnim brojem zraka pomoću jednačine (5.1) moguće je dobiti foto-realističnu sliku scene.

Zbog napretka u brzini hardvera tehnika praćenja zraka se koristi u video igrama u kombinaciji sa tehnikom rasterizacije. Novije verzije programskog interfejsa za iscrtavanje DirectX [10] i Vulkan [13] imaju mogućnosti praćenja zraka na delu scene ili iscrtavanja celokupne scene tehnikom praćenja zraka.

Kompanija Nvidia je za grafičke jedinice njihove proizvodnje napravila programski interfejs za iscrtavanje scena praćenjem zraka pod nazivom *optiks* (eng. OptiX) [12]. Jedan od interesantnih elemenata implementacije programskog interfejsa optiks je to što koristiti algoritme veštačke inteligencije i time drastično smanjuje vreme koje je potrebno za iscrtavanje slike scene [1].

Još jedna tehnologija koja ubrzava iscrtavanje slika pomoću algoritama veštačke inteligencije je *uzorkovanje zasnovano na dubokom učenju* (eng. deep learning super sampling - DLSS) [11]. Ubrzanje iscrtavanja dobijeno tehnologijom DLSS daje dovoljno prostora za iscrtavanje dela scene tehnikom praćenja zraka čime se poboljšava doživljaj scene i kvalitet iscrtane slike.

Postoje i implementacije tehnike praćenja zraka koje su otvorenog koda. Knjiga [20] je dobila mnoge nagrade i priznanja kao odlična literatura za učenje iscrtavanja tehnikom praćenja zraka. Knjiga vodi čitaoca korak po korak od matematičkih osnova do implementacije programa za praćenje zraka.

U narednoj deceniji očekuje se sve veća primena tehnike praćenja zraka u video igrama i aplikacijama koje rade u realnom vremenu zahvaljujući poboljšanju performansi grafičkih jedinica i razvoja softverskih tehnologija.

Glava 6

Zaključak

U ovom radu predstavljene su matematičke osnove tehnike praćenja zraka i opisani glavni elementi platforme CUDA. Tehnika praćenja zraka implementirana je tako da može da se izvršava na jednoj niti matičnog procesora i na platformi CUDA. Implementacija na platformi CUDA deli većinu koda sa implementacijom na jednoj niti matičnog procesora.

Rezultati dobijeni u ovom radu ukazuju da se paralelizacijom tehnike praćenja zraka na platformi CUDA može dobiti značajno ubrzanje u vremenu potrebnom za iscrtavanje scene. Platforma CUDA se može koristiti za ubrzanje i drugih algoritama pogodnih za paralelizaciju. U radu su korišćene samo osnovne funkcionalnosti platforme CUDA i postoji još mnogo prostora za poboljšanje performansi.

Dalja unapređenja bi podrazumevala korišćenje naprednih funkcionalnosti platforme CUDA, odabir efikasnih struktura podataka za detekciju preseka zraka i objekta, keširanje često korišćenih objekata i korišćenje intrinzičnih instrukcija platforme CUDA za ubrzavanje matematičkih izračunavanja. Maksimalna iskorišćenost resursa grafičke jedinice bi se postigla kada bi izračunavanja u tehnici praćenja zraka bila implementirana na način koji minimizuje grananja i svodi većinu operacija na jednostavna množenja matrica i vektora.

Navedena lista poboljšanja nije ni blizu potpune i ta činjenica čini oblast iscrtavanja tehnikom praćenja zraka neiscrpnim izvorom novih izazova u oblasti izučavanja hardvera, računarstva i softvera.

Bibliografija

[1] Chakravarty R. Alla Chaitanya, Anton Kaplanyan, Marco Salvi Christoph Schied, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *SIGGRAPH*, 2017.

[2] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*. 2014.

[3] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH*, pages 137–145, 1984.

[4] Hendric, the free encyclopedia. Rasterizacija i rej trejsing, 2018. [Online; preuzeto 06., 2022; `https://www.cadalyst.com/files/cadalyst/nodes/2018/40910/041918-HoH-4.png`].

[5] Intel. Intel processors, 2022. [Online; preuzeto 09., 2022; `https://www.intel.com/content/www/us/en/products/details/processors/core.html`].

[6] James T. Kajiya. The rendering equation. *SIGGRAPH*, pages 143–150, 1986.

[7] Khronos Group. OpenGL 4 Documentation, 2013. [Online; preuzeto 09., 2022; `https://registry.khronos.org/OpenGL-Refpages/gl4`].

[8] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach*. 2016.

[9] M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten. Later. 40 years of microprocessor trend data, 2022. [Online; preuzeto 09., 2022; `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`].

[10] Microsoft. Announcing microsoft directx raytracing!, 2018. [Online; preuzeto 09., 2022; `https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/`].

[11] Nvidia. Nvidia deep learning super sampling (dlss). [Online; preuzeto 09., 2022; `https://developer.nvidia.com/rtx/dlss`].

[12] Nvidia. Nvidia optix ray tracing engine. [Online; preuzeto 09., 2022; `https://developer.nvidia.com/rtx/ray-tracing/optix`].

[13] Nvidia. Nvidia vulkan raytracing, 2018. [Online; preuzeto 09., 2022; `https://developer.nvidia.com/blog/vulkan-raytracing/`].

[14] Nvidia. *CUDA C Programming Guide*. 2021.

[15] Nvidia. Nvidia cuda compiler documentation, 2022. [Online; preuzeto 09., 2022; `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html`].

[16] Nvidia. Nvidia rtx, 2022. [Online; preuzeto 06., 2022; `https://www.nvidia.com/en-eu/geforce/rtx/`].

[17] Nvidia. Rasterization and raytracing, 2022. [Online; preuzeto 06., 2022; `https://blogs.nvidia.com/blog/author/brian-caulfield/`].

[18] Nvidia. RTX 3000 Series, 2022. [Online; preuzeto 06., 2022; `https://www.nvidia.com/en-eu/geforce/graphics-cards/30-series/rtx-3060-3060ti/`].

[19] Par Quentin. Nvidia rtx : Top 5 des jeux qui utilisent bien le ray tracing (et 5 qui le font mal), 2022. [Online; preuzeto 09., 2022; `https://www.omgpu.com/`].

[20] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering, Third Edition*. 2016.

[21] Peter Shirley. Ray tracing in one weekend, December 2020. [Online; preuzeto 09., 2022; `https://raytracing.github.io/books/RayTracingInOneWeekend.html`].

[22] Peter Shirley. Ray tracing: The next week, December 2020. [Online; preuzeto 09., 2022; `https://raytracing.github.io/books/RayTracingTheNextWeek.html`].

[23] Wikipedia. RGB color model, 2022. [Online; preuzeto 09., 2022; `https://en.wikipedia.org/wiki/RGB_color_model`].

Biografija autora

Marko S. Spasić rođen je 20.06.1994. godine u Beogradu. Završio je osnovnu školu Vuk Karadžić u Sremčici i srednju Elektrotehničku školu Lola u Železniku.

Visoku strukovnu školu za informacione tehnologije Comtrade upisuje 2013. godine po završetku srednje škole. Na navedenoj strukovnoj školi stiče diplomu strukovnog inženjera informacionih tehnologija 2016. godine.

Smer Informatika na Matematičkom fakultetu Univerziteta u Beogradu upisuje 2016. godine. Na navedenom smeru je diplomirao 2020. godine sa prosečnom ocenom 9,60. Master studije upisuje na istom fakultetu odmah nakon diplomiranja.

U oktobru 2020. godine biva izabran za saradnika u nastavi na Matematičkom fakultetu. Drži vežbe iz kurseva „Računarska grafika" i „Veštačka inteligencija" na smeru Informatika.

Oblasti interesovanja uključuju računarsku grafiku, kompajlere i izračunavanje visokih performansi.