SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1670

POSTUPAK PRAĆENJA ZRAKE I PRAĆENJA PUTA

Davor Najev

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1670

POSTUPAK PRAĆENJA ZRAKE I PRAĆENJA PUTA

Davor Najev

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1670

Pristupnik:	**Davor Najev (0036543497)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Postupak praćenja zrake i praćenja puta**

Opis zadatka:

Proučiti postupke praćenje zrake (engl. ray tracing) i praćenje puta (engl. path tracing). Razraditi navedene postupke tako da se mogu usporediti ostvareni rezultati. Posebno obratiti pažnju na implementaciju tako da bude učinkovito ostvarena. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++ i grafičko programsko sučelje OpenGL. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

1. Uvod

Renderiranje fotorealistične slike je cilj od samih početaka računalne grafike. Kako bi postigli fotorealističnu sliku, potrebno je odabrati dobar model osvjetljavanja. Metode bacanja zrake (engl. *ray casting*), praćenja zrake (engl. *ray tracing*) i praćenja puta (engl. *path tracing*) su jako dobri kandidati za ovu svrhu.

U ovom radu ću objasniti osnovne ideje iza ovih algoritama, dati njihov osnovni matematički opis, dati konkretne primjere u programskom jeziku C++ i OpenGL-u te ih usporediti međusobno te s Phongovim osvjetljenjem kao predstavnikom lokalnog modela osvjetljenja.

2. Tehnike renderiranja

Modele osvjetljavanja možemo podijeliti u 3 skupine:

- Empirijski modeli
- Prijelazni modeli
- Analitički modeli

Empirijski modeli se temelje na iskustvu i estetskim aproksimacijama [1]. Rezultat toga je da u će u mnogo slučaja dati nezadovoljavajuće rezultate. Njihova prednost je što su relativno jednostavni pa je i njihov izračun relativno jednostavan, to omogućava brz izračun što znači visoke performance i renderiranje u stvarnom vremenu. Najpoznatiji ovakav model je Phongov model, a koriste ga konstatni, Gouradov i Phongov model sjenčanja.

Prijelazni modeli se temelje se na geometrijskim svojstvima svjetlosti. Uzimamo kao pretpostavku da se svjetlost širi pravocrtno, a osvjetljenje se ostvariva puštanjem zrake kako bismo odredili koji sve objekti i kako utječu na osvjetljenje određene točke. Ovakvi modeli također uključuju i zrcaljenje, sjene i prozirnost što bismo kod empirijskih morali ostvarivati dodatnim postupcima i to manje uspješno.

Analitički modeli se temelje na fizikalnim svojstvima svjetlosti. Ovakvi modeli su vrlo kompleksni jer barataju s mnogo fizikalnih veličina, ali daju jako dobre rezultate. Primjer ovakvog modela je Cook-Torrenceov model. Algoritam koji koristi ovakve modele je postupak isijavanja [1].

Tehnike bacanja zrake, praćenja zrake i praćenja puta pripadaju prijelaznim modelima.

Tehnika praćenja zrake i praćenja puta su tehnike globalnog modela osvjetljavanja. To znači da svaki objekt u sceni doprinosi osvjetljenju u nekoj točki za razliku od jednostavnih (lokalnih) modela osvjetljavanja. Rezultat toga su realističniji vizualni učinci koje je vrlo teško oponašati kod lokalnih metoda – trebali bismo koristiti "trikove" kako bi dobili približno dobar rezultat.

Obradu ovih tehnika ćemo započeti od najjednostavnijom od spomenutih tehnika – postupkom bacanja zrake.

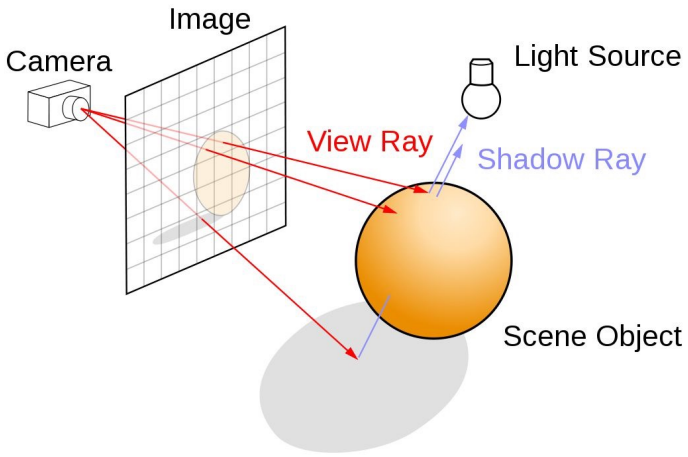
2.1. Postupak bacanja zrake

Tehnika bacanja zrake je najjednostavniji postupak globalnog osvjetljavanja i kao takav nije vrlo koristan već služi kao uvod u ostale tehnike, i to konkretno za tehniku praćenja zrake.

Znamo da je svjetlost dvojne prirode – očituje se kao titranje čestica i kao val. No, radi jednostavnosti ćemo tu činjenicu ignorirati i pretpostaviti da se svjetlosti širi pravocrtno. Imajući to na umu, izvor svjetlosti možemo gledati kao izvor beskonačno mnogo svjetlosnih zraka koje se šire jednoliko u svim smjerovima. Neke pogađaju objekte u sceni i reflektiraju se od njih, ponekad i više puta, a dio njih pogađa i oko promatrača. Kad se zraka reflektira o površinu, reflektirat će se ovisno o fizičkim svojstvima površine od koje se reflektira, ako je površina grublja, na primjer, zraka će imati nasumičnu komponentu pri računanju njenog novog vektora smjera što će također ovisiti i o boje i slično. Međutim, imamo probleme promatrajući svjetlost na ovakav način:

- Ne možemo pratiti beskonačno mnogo zraka, imamo ograničenu računalnu moć za izračun.
- Većina zraka koju svjetlosni izvor pušta neće završiti u oku promatrača – trošimo previše računalnih resursa na zrake koje neće završiti nigdje.

Stoga ćemo za početak još više pojednostavniti stvar. Recimo da smo pronašli neku zraku koja se reflektira od objekta do promatrača. Put te zrake možemo rekonstruirati obrnemo li njen put – pratimo zraku od promatrača do objekta. Postupak je konceptualno prikazan na slici 2.1.



Slika 2.1. Prikaz zraka korištenih kod popstupka bacanja zrake [2]

Na ravnini projekcije se nalazi naš zaslon proizvoljne visine i širine. Uzmimo za primjer zaslon rezolucije 1920×1080 piksela. Za svaki od piksela ćemo pustiti zraku od promatračevog oka koja prolazi kroz dotični piksel te ćemo odrediti prvi objekte kroz koje prolazi (to je već više od 2 milijuna zraka!).

Ako zraka ne probada nijedan objekt, onda ona pogađa kupole scene (engl. *skybox*) ili poprima neku preodređenu boju. Ako probada jedan ili više objekata, uzima se objekt bliži promatraču. Sljedeći korak je sjena. Bacaju se dodatne zrake iz sjecišta između zrake i objekta prema svakom izvoru svjetlosti u sceni (engl. *shadow rays*). Ako postoji objekt između sjecišta i izvora objekta, taj izvor ne doprinosti intenzitetu sjecišta, u suprotnom doprinosi, a doprinos se računa na temelju Phongova modela osvjetljavanja.

Prisjetimo se, Phongov model osvjetljavanja je lokalni model osvjetljavanja, a to znači da u izračunu ne uzima doprinos osvjetljenja od drugih objekta u sceni. Kako je ovo jedini doprinos osvjetljenja u nekoj točki, ovaj algoritam je također model lokalnog osvjetljenja. Vidjet ćemo da to neće biti slučaj kod kompliciranijih algoritama.

2.1.1. Matematički opis algoritma

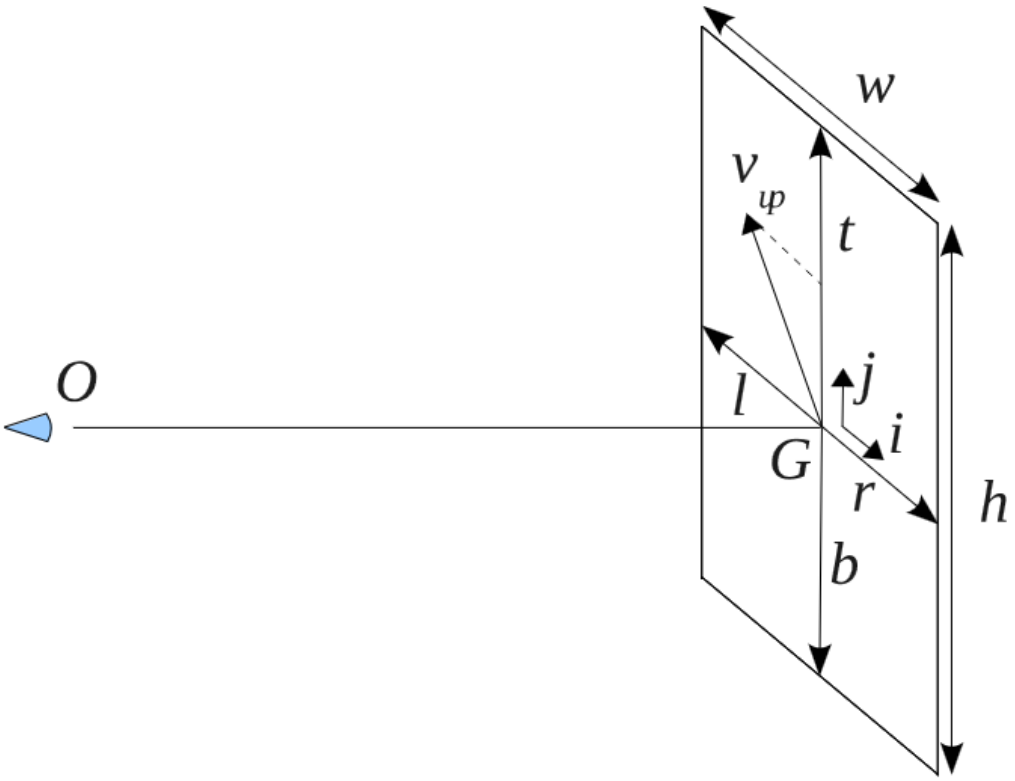
Neka je $\vec{t}(\lambda)$ pravac zadan dvije točke $\vec{T}_S$ i $\vec{T}_E$. Možemo opisati ovaj pravac s izrazom:

$$\vec{t}(\lambda) = \vec{T}_S + \lambda \cdot \vec{d},\ \lambda \in \mathbb{R} \tag{2.1}$$

gdje je $\vec{d}$ jedinični vektor smjera i on je zadan formulom:

$$\vec{d} = \frac{\vec{T}_E - \vec{T}_S}{||\vec{T}_E - \vec{T}_S||} \tag{2.2}$$

Uzmimo da je $\vec{T}_S$ očište kamere u sceni. Neka je ravnina projekcije određena parametrima l, r, t, b. Neka su w širina, i h visina prikaza u pikselima. Neka je $\vec{OG} = O - G$ vektor između očišta i ravnine. Neka su vektori $\vec{i}$ i $\vec{j}$ okomiti vektori koji određuju ravninu projekcije. Oni se mogu dobiti na više načina, a najlakše je normiranjem $\vec{(V)}_{right}$ i $\vec{(V)}_{up}$ vektora (uz pretpostavku da vektor $\vec{(V)}_{forward}$ leži na vektoru O - G) Ovi parametri mogu se vidjeti na slici 2.2.



Slika 2.2. Ravnina projekcije s parametrima [3]

Ideja je baciti zraku između očišta i svake točke ravnine projekcije, pratiti tu zraku te izračunati boju fragmenta ovisno gdje pogodi. No to ne možemo napraviti za beskonačno mnogo točaka pa ćemo morati nekako diskretizirati ravninu projekcije. Najrazumnija diskretizacija bi bila prema visini i širini prikaza – na takav način će svaki fragment odgovarati jednom pikselu na prikazu.

Možemo dobiti neku točku prikaza po sljedećoj formuli prema izvoru iz [3]:

$$\vec{T}_E = G + \vec{i} \cdot \left(-l + \frac{x}{w} \cdot (l + r)\right) + \vec{j} \cdot \left(-b + \frac{y}{h} \cdot (t + b)\right)$$

Ovu točku i točku očišta možemo iskoristiti za jednadžbu pravca zrake (formule (2.1) i (2.2)). Konstruirali smo zraku, sada je potrebno izračunati presjek između zrake i objekata scene kako bi izračunali osvjetljenje kako je ranije opisano. Potrebno je izračunati točku presjeka sa svakim objektom u sceni. Za svaku točku presjeka, potrebno je izračunati udaljenost između očišta i točke presjeka i konačno, kao točku koju je zraka pogodila se uzima najbliža točka očištu.

Sada trebamo baciti baciti zraku prema svakom svjetlu (engl. *shadow ray*). Ako nema nijedne prepreke između točke i svjetla, pribrojit ćemo doprinos tog svjetla pomoću Phongovog modela osvjetljavanja.

U nastavku su dane jednadžbe presjeka između najčešćih zrake i primitiva.

Presjek s ravninom Neka je ravnina zadano implicitno, $A \cdot x + B \cdot y + C \cdot z + D = 0$. Zapisat ćemo ju u normalnom obliku $\vec{n} \cdot \vec{t} + D = 0$ gdje je $\vec{t}$ neka točka ravnine. Kao točku ravnine ćemo unijeti jednadžbu pravca te iz toga izvući λ:

$$\lambda = \frac{-(\vec{n} \cdot \vec{T}_S + D)}{\vec{n} \cdot \vec{d}}$$

Moramo imati na umu specijalan slučaj kada je $\vec{n} \cdot \vec{d} = 0$, u tom slučaju su ravnina i zraka paralelni i nema presjeka. Ako je $\lambda < 0$, to znači da je ravnina iza točke i taj ćemo slučaj odbaciti. Ako je $\lambda = 0$, to znači da je početna točka u ravnini. Preostali slučaj je da je $\lambda > 0$ i možemo dobiti točku presjeka uvrštavanjem λ u (2.1).

Presjek sa sferom Jednadžba sfere sa središtem u točki C i polumjerom r:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

Neka je $\vec{t}$ neka točka na sferi. U tom slučaju gornju jednadžbu možemo zapisati kao:

$$(\vec{t} - \vec{C}) \cdot (\vec{t} - \vec{C}) = r^2$$

Uvrstimo li u ovu jednadžbu jednadžbu pravca i raspišemo dobijemo:

$$\lambda^2 \vec{d} \cdot \vec{d} + 2\lambda \vec{d} \cdot (\vec{T_S} - \vec{C}) + (\vec{T_S} - \vec{C}) \cdot (\vec{T_S} - \vec{C}) - r^2 = 0$$

Dobivena jednadžba je kvadratna jednadžba oblika $a\lambda^2 + b\lambda + c = 0$. Možemo uočiti da je član a jednak $\vec{d} \cdot \vec{d} = 1$ pa će rješenja kvadratne jednadžbe biti:

$$\lambda_{1,2} = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$$

Postoji nekoliko mogućnosti ovisno o rješenjima jednadžbe:

- λ_1 i λ_2 su dva pozitivna različita realna broja. U tom slučaju imamo 2 probodišta sa sferom koja su ispred promatrača. Probodišta dobijamo uvrštavanjem λ_1 i λ_2 u jednadžbu pravca, a bliže sjecište je ono koje dobijemo uvrštavanjem manjim od parametara.

- λ_1 i λ_2 su dva različita realna broja gdje je jedan pozitivan, a drugi negativan. Presjecište gdje je parametar pozitivan je ispred promatrača, a negativno je iza. Prema tome, promatrač se nalazi unutar sfere.

- λ_1 i λ_2 su jednaki i realni. To znači da postoji samo jedno presjecište, odnosno pravac je tangenta na sferu. Ako je $\lambda_1 = \lambda_2 = \lambda$ pozitivan, točka dirališta se nalazi ispred promatrača, a ako je negativan, onda je iza.

- λ_1 ili λ_2 je jednak 0 – promatrač se nalazi na sferi.

Presjek s trokutom Problem ćemo svesti na problem proboda pravca i ravnine.

Točke trokuta definiraju ravninu s kojom tražimo presjecište. Ako su točke trokuta A, B i C, normalu ravnine možemo izračunati vektorskim umnoškom vektora bridova trokuta: $\vec{n} = (\vec{B} - \vec{A}) \times (\vec{C} - \vec{A})$. Komponentu D možemo izračunati izvlačenjem iz normalnog oblika: $D = -\vec{n} \cdot \vec{t}$. Kao $\vec{t}$ možemo koristiti bilo koju točku trokuta, primjer s točkom A: $D = -\vec{n} \cdot \vec{A}$.

Nakon što smo našli točku presjeka, moramo provjeriti nalazi li se unutar trokuta ili

van. To možemo napraviti računanjem baricentričnih koordinata:

$$T = t1 \cdot \vec{A} + t2 \cdot \vec{B} + t3 \cdot \vec{C}$$

gdje je točka presjeka a $t1$, $t2$ i $t3$ baricentrične koordinate. Ako je $t1 > 0$, $t2 > 0$ i $t3 > 0$, točka je unutar trokuta, u suprotnom nije.

Za detaljniji izvod, pogledati [3].

2.2. Postupak praćenja zrake

Kod prijašnjeg algoritma smo napravili mnogo pojednostavljenja pa smo posljedično izgubili na kvaliteti osvjetljenja. Algoritmom praćenja zrake uzimamo u obzir neke fizikalne zakonitosti koje smo kod postupka bacanja zrake ignorirali što vodi do uvjerljivijeg krajnjeg rezultata.

Kada svjetlost udari u površinu, mogu nastati dvije dvije zrake: odbijena i lomljena. Lomljenu smo zraku u prijašnjem algoritmu ignorirali, ali u ovom nećemo nego ćemo ju uzeti u obzir u računu Phongova modela osvjetljavanja (uz odbijenu zraku). Moramo imati na umu i da se novonastale zrake mogu sudariti s objektom u sceni i time stvoriti dvije nove zrake. Možemo primijetiti da broj zraka eksponencijalno raste, odnosno maksimalan broj zraka je 2^n. To će osjetno utjecati na brzinu izvođenja algoritma.

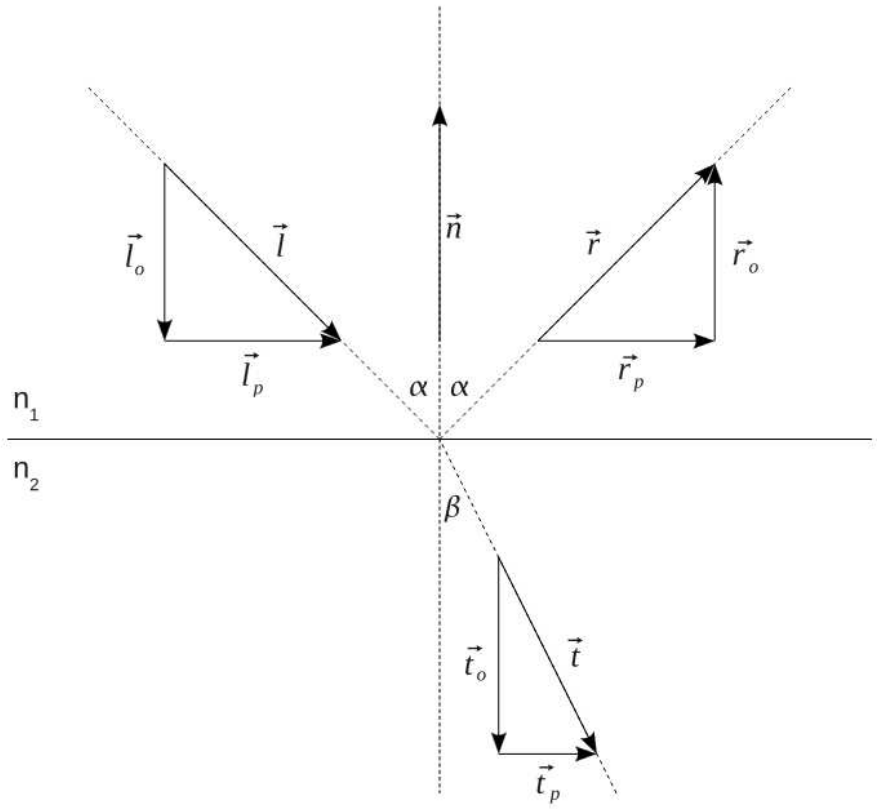
2.2.1. Matematički opis algoritma

Ovaj algoritam ćemo opisati proširivanjem algoritma bacanja zrake.

Neka je $\vec{T_S}$ točka koju je zraka pogodila algoritmom bacanja zrake. Sada ćemo, osim zraka prema svjetlu, baciti dvije nove zrake: odbijenu zraku i zraku loma.

Kao točku iz koje bacamo zraku ćemo uzeti $\vec{T_S}$. Na slici 2.2. možemo vidjeti odbijenu zraku i zraku loma.

Odbijena (reflektirana) zraka je, kao što samo ime govori, zraka nastala odbijanjem od površine objekta u točki sudara zrake s njime. Znamo da je kut između upadne zrake ($\vec{l}$) i normale ($\vec{n}$) jednak kutu između normale i odbijene zrake. Iz slike 2.2. možemo



Slika 2.3. Odbijena i lomljena zraka [3]

vidjeti da se zrake sastoje od dvije komponente: jedna je paralelna s ravninom poligona, a druga je paralelna s normalom. Označimo prvu komponentu upadne zrake s $\vec{l}_p$, a drugu s $\vec{l}_o$. Slično, označimo prvu komponentu reflektirane zrake s $\vec{r}_p$, a drugu s $\vec{r}_o$.

$$\vec{l}_o = -\vec{n} \cdot \cos(\alpha) = \vec{n} \cdot (\vec{l} \cdot \vec{n}),$$

$$\vec{l}_p = \vec{l} - \vec{l}_o$$

Sada je jednostavno izračunati komponente reflektirane zrake te i samu reflektiranu zraku:

$$r_o = -\vec{l}_o \qquad \vec{r}_p = \vec{l}_p$$

$$\vec{r} = \vec{r}_o + \vec{r}_p = -\vec{l}_o + \vec{l}_p = -\vec{n} \cdot (\vec{l} \cdot \vec{n}) + \vec{l} - \vec{n} \cdot (\vec{l} \cdot \vec{n}) = \vec{l} - 2\vec{n} \cdot (\vec{l} \cdot \vec{n})$$

Lomljenu zraku možemo dobiti pomoću Snellovog zakon loma:

$$\frac{\sin(\alpha)}{\sin(\beta)} = \frac{n_2}{n_1}$$

gdje su n_1 i n_2 faktori loma dolaznog i odlaznog materijala (medija). Iz njega možemo izvući izraz za sinus kuta lomljene zrake:

$$\sin(\beta) = \sin(\alpha)\frac{n_1}{n_2}$$

Možemo primijetiti da ako je faktor loma dolaznog medija veći od onoga od odlaznog medija, sinus kuta lomljene zrake veći od 1. U tom slučaju se događa totalna refleksija i nema lomljene zrake, samo reflektirane. U slučaju da ova zraka postoji, lomljenu zraku ćemo označiti s $\vec{t}$ koji je normirani vektor, i možemo ga zapisati kao:

$$\vec{t} = \vec{t}_p + \vec{t}_o$$

Kako je $\vec{t}$ normiran, vrijedi $|\vec{t_p}| = \sin(\beta)$. Isto tako i vrijedi $|\vec{l_p}| = \sin(\alpha)$. Kako su $\sin(\alpha)$ i $\sin(\beta)$ povezani Snellovim zakonom, slijedi:

$$||\vec{t}_p|| = \frac{n_1}{n_2}||\vec{l}_p||$$

Kako su $\vec{l_p}$ i $\vec{t_p}$ kolinearni i istog smjera, vrijedi:

$$\begin{aligned}\vec{t}_p &= \frac{n_1}{n_2}\vec{l}_p \\ &= \frac{n_1}{n_2}(\vec{l} - \vec{n} \cdot (\vec{l} \cdot \vec{n}))\end{aligned}$$

Komponenta $\vec{t_o}$ je kolinerna vektoru normale, ali je suprotnog smjera. Imajući na umu da vrijedi $|\vec{t_o}| = \sin(\beta)$, možemo izvući konačni izraz za $\vec{t_o}$:

$$\begin{aligned}\vec{t}_o &= -\vec{n} \cdot \cos(\beta) \\ &= -\vec{n} \cdot \sqrt{1 - \sin^2(\beta)} \\ &= -\vec{n} \cdot \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \sin^2(\alpha)} \\ &= -\vec{n} \cdot \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - \cos^2(\alpha))} \\ &= -\vec{n} \cdot \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - (\vec{l} \cdot \vec{n})^2)}\end{aligned}$$

..te na kraju i sam $\vec{t}$:

$$\begin{aligned}\vec{t} &= \vec{t}_p + \vec{t}_o \\ &= \frac{n_1}{n_2}(\vec{l} - \vec{n} \cdot (\vec{l} \cdot \vec{n})) - \vec{n} \cdot \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \left(1 - (\vec{l} \cdot \vec{n})^2\right)}\end{aligned}$$

Za detaljniji izvod, pogledati [3].

Sada znamo kako izračunati svaku zraku koja nam je potrebna za ostvarivanje algoritma. Čitav pseudokod algoritma je dan u nastavku:

```
funkcija prati_zraku(izvor, smjer, dubina):
  ako dubina <= 0:
    vrati boja(0, 0, 0)

  mjesto_pogotka = baci_zraku(izvor, smjer)
  ako !mjesto_pogotka:
    vrati boja(0, 0, 0)

  boja = (0, 0, 0)
  za svako svjetlo u sceni:
    sjena_pogodak = baci_zraku(mjesto_pogotka, svjetlo.pozicija - mjesto_pogotka)
    ako !pogodak:
      boja += phong(svjetlo, pogodak)

  boja += prati_zraku(mjesto_pogotka, reflektiraj(smjer, mjesto_pogotka.normala), d
  boja += prati_zraku(mjesto_pogotka, odbij(smjer, mjesto_pogotka.normala, 1), dubi

  vrati boja
```

2.3. Postupak praćenja puta

Kod postupka praćenja zrake, kada provjeravamo je li točka u sjeni obasjana svjetlom ili nije, ne provjeravamo u "kolikoj je količini" obasjana. Rezultat je toga da u sceni imamo

samo oštre sjene, nema mekih sjena. To je rezultat pretpostavke da se svjetlost pravocrtno širi te da se predvidljivo odbija ili lomi. Međutim, ako imamo objekt koji ima grublju površinu, svjetlost se može odbiti u nekom nasumičnom smjeru – što je grublja površina, to je nepredvidljivije gdje će se zraka odbiti. Zaključak – moramo nekako uračunati tu nasumičnu komponentu.

Kada malo bolje promislimo, osvjetljenje u nekoj točki je suma svih doprinosa osvjetljenja iz svih smjerova koje dolaze u tu točku. Već smo utvrdili da je nemoguće pribrojiti sve doprinose iz svih smjerova jer smo ograničeni računalnom moći, ali što ako bismo uzeli nasumične zrake koje se nasumične odbijaju po sceni? Bi li to u konačnici, ako uzmemo dovoljan broj zraka, dalo rezultat kao da smo uračunali sve zrake u sceni?

Odgovor je da je to istina, a to što smo upravo opisali se naziva **Monte Carlo metoda praćenja puta**.

Ako usporedimo ovaj postupak s postupkom praćenja zrake, prednost ovog postupka su manje oštri rubovi i sjene, a njihova oštrina će ovisiti o svjetlima koja obasjavaju i materijalima objekata. Mana ovog postupka je što, zbog nedeterminističke prirode algoritma, moramo baciti jako puno zraka da dobijemo zadovoljavajuće rezultate. Pri jako malo zraka, renderirana slika ima mnogo šuma. Kako bi ga smanjili, trebamo bacati zrake isponova te novi rezultat spajati s prijašnjima. Ovaj postupak se zove **Monte Carlova integracija** koja će detaljnije biti opisana u sljedećem potpoglavlju.

Možemo primijetiti da je ovaj postupak iterativan. Pozitivna stvar kod toga je što možemo ponavljati postupak koliko god hoćemo, a sa svakom iteracijom ćemo završiti s malo boljim rezultatom. Mana je što svaka nova iteracija sve manje i manje doprinosi konačnom rezultatu pa nam se nakon određenog broja iteracije više ne isplati raditi nove iteracije. Koliko iteracija je dovoljno ovisi od slučaja do slučaja.

2.3.1. Matematički opis algoritma

Osvjetljenje u točki L_o možemo opisati **jednadžbom renderiranja** [4]:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot n)\, d\omega_i$$

gdje je:

- $L_o(x, \omega_o)$: Izračena svjetlost od točke x u smjeru ω_o
- $L_e(x, \omega_o)$: Emitirana svjetlost iz točke x u smjeru ω_o - komponenta isijavane svjelosti
- $f_r(x, \omega_i, \omega_o)$: Dvosmjerna funkcija refleksije (BRDF) na točki x - opisuje kako se svjetlo odbija od tijelo
- $L_i(x, \omega_i)$: Dolazna svjetlost u točku x iz smjera ω_i - komponenta refleksije drugih objekata
- ω_i: Smjer dolazne svjetlosti
- ω_o: Smjer izlazne svjetlosti
- n: Normala površine na točki x
- Ω: Hemisfera oko točke x

Intuitivno tumačenje: svjetlost u nekoj točki je zbroj isijavane svjetlosti tog objekta i kombinaciji doprinosa svih zraka po hemisferi ovisno o BRDF ($f_r(x, \omega_i, \omega_o)$) i kutu upada ($\omega_i \cdot n$).

Ovaj integral po hemisferi je problematičan – prema njemu, trebali bi uračunati doprinos svake zrake po hemisferi. Već smo objasnili zašto to neće ići. Jednadžbu renderiranja ćemo aproksimirati Monte Carlovom aproksimacijom integrala:

$$L_o(x, \omega_o) \approx L_e(x, \omega_o) + \frac{1}{N} \sum_{i=1}^{N} \frac{f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot n)}{p(\omega_i)}$$

gdje je:

- N: Broj uzoraka
- $p(\omega_i)$: Funkcija gustoće vektora smijera ω_i

Integral smo zamijenili sa sumom doprinosa N uzoraka. Za svaki uzorak ćemo baciti zraku u nasumičan smjer ovisno o funkciji gustoće $p(\omega_i)$. Ovakav rezultat će dati pri-

bližno dobar rezultat ovisno o broju uzoraka. Kako je ova aproksimacija stohastički proces, prema zakonu velikih brojeva će za vrlo veliki N ova aproksimacija odgovarati izvornoj jednadžbi renderiranja:

$$\lim_{N\to\infty}\left(L_e(x,\omega_o)+\frac{1}{N}\sum_{i=1}^{N}\frac{f_r(x,\omega_i,\omega_o)L_i(x,\omega_i)(\omega_i\cdot n)}{p(\omega_i)}\right)=L_o(x,\omega_o)$$

U praksi mi ne znamo koliki N će nam trebati za zadovoljavajuće dobar rezultat. Iz tog razloga renderiranje obavljamo iterativno – renderiramo sliku po sliku te uzimamo njihovu srednju vrijednost.

Pseudokod algoritma praćenja puta prema opisanom postupku:

```
funkcija prati_put(izvor, smjer, dubina):
  ako dubina <= 0:
    vrati boja(0, 0, 0)

  mjesto_pogotka = baci_zraku(izvor, smjer)
  ako !mjesto_pogotka:
    vrati boja(0, 0, 0)

  boja = (0, 0, 0)
  za svako svjetlo u sceni:
    sjena_pogodak = baci_zraku(mjesto_pogotka, svjetlo.pozicija - mjesto_pogotka)
    ako !pogodak:
      boja += phong(svjetlo, pogodak)

  nasumicni_smjer = točka_na_hemisferi(mjesto_pogotka.normala, 1)
  boja += prati_put(mjesto_pogotka, nasumicni_smjer, dubina - 1)

  vrati boja
```

3. Implementacija kroz primjere

Primjeri će biti ostvareni u programskom jeziku C++ i OpenGL-u. Zbog velike količine koda, u daljnjem tekstu će se pretpostaviti da čitatelj ima postavljen osnovni program u OpenGL-u ili da je barem svjestan njegove osnovne strukture. Algoritme ćemo, zbog jednostavnosti, implementirati programski, ali na način da se mogu lako ostvariti i sklopovski putem "compute" sjenčara, OpenCL-a ili nekog drugog API-ja za ubrzani paralelni izračun.

Za početak ćemo definirati spremnik (engl. *buffer*) u koji ćemo renderirati sliku. Alocirat ćemo polje vrijednost s pomičnim zarezom (engl. *float*) veličine $width \times height \times 3$:

```
raster = static_cast<float*>(calloc(width * height * 3, sizeof(float)));
```

Ovdje 3 označava broj kanala po pikselu – crveni, zeleni i plavi. Sada je potrebno napraviti objekt kojemu je samo zadaća pomoću OpenGL-a renderirati našu teksturu preko cijelog zaslona. To možemo napraviti postavljanjem njegovih vrhova u kutove kanonskog OpenGL sustava u ravnini $Z = 0$.

Definirat ćemo pomoćnu funkciju `osvijetliFragment` pomoću koje ćemo postaviti boju fragmenta na nekoj koordinati:

```
aoid Renderer::osvijetliFragment(int x, int y, glm::vec3 boja) {
  int h = height - 1 - y;
  raster[h * width * 3 + x * 3] = boja.x;
  raster[h * width * 3 + x * 3 + 1] = boja.y;
  raster[h * width * 3 + x * 3 + 2] = boja.z;
}
```

Sada trebamo definirati petlju izrade prikaza (engl. *render loop*). Želimo ostvariti "per-

spektivnu projekciju" kao na slici 2.2. bez obzira na algoritam. Treba imati na umu da ovim algoritmima mi ustvari ne radimo eksplicitno perspektivnu projekciju jer ne množimo vrhove objekata perspektivnom matricom, već je ćemo ostvariti algoritme na način da će biti prividno kao da jesmo.

Znamo da u svakom algoritmu trebamo pustiti zraku iz očišta za svaki piksel zaslona. Kako bi izračunali smjer svake zrake, trebamo podijeliti "ravninu projekcije" na *width* stupaca i *height* redaka i za svaki redak i stupac izračunati odgovarajuću točku. Započnimo s gornjim lijevim kutem. Ako pogledamo sliku 2.2., možemo vidjeti da ga možemo izračunati na sljedeći način:

```
glm::vec3 topLeft = camera.getPos() + camera.near * camera.forward()
             + camera.top * camera.up() + camera.left * camera.right();
```

Sada možemo definirati još duljinu retka, stupica, podjele retka i podjele stupca koji će biti potrebni za računanje pozicije svake podjele:

```
glm::vec3 row = (camera.right - camera.left) * camera.right();
glm::vec3 dx = row * (1.0f / width);
glm::vec3 column = -(camera.top - camera.bottom) * camera.up();
glm::vec3 dy = column * (1.0f / height);
```

Pomoću ovih varijabli, moguće je konstruirati petlju izrade prikaza na sljedeći način:

```
while (!glfwWindowShouldClose(window)) {
  glm::vec3 current = camera.getPos() + camera.near * camera.forward()
            + camera.top * camera.up() + camera.left * camera.right();
  glm::vec3 row = (camera.right - camera.left) * camera.right();
  glm::vec3 dx = row * (1.0f / width);
  glm::vec3 column = -(camera.top - camera.bottom) * camera.up();
  glm::vec3 dy = column * (1.0f / height);

  for (int i = 0; i < height; i++) {
    current = start + column * ((float)i / (height - 1));
    for (int j = 0; j < width; j++) {
```

```
      glm::vec3 boja = // algoritam ide ovdje
      osvijetliFragment(j, i, boja);
      current += dx;
    }
  }
  iscrtajRaster();
  glfwSwapBuffers();
}
```

Konkretni postupak računanja vrijednosti fragmenta je izostavljen za sada. Pomoćna funkcija `iscrtajRaster` prebacuje raster OpenGL-u da ga iscrta te se poziva `glfwSwapBuffers` da se renderirana slika pokaže na prikazu.

Napravili smo svu pripremu koju smo trebali, sada možemo implementirati algoritme.

3.1. Implementacija tehnike bacanja zrake

Najprije ćemo napisati algoritam bacanja zrake u užem smislu. Ovaj algoritam traži najbližu točku presjeka sa svim objektima u sceni:

```
IntersectPoint baci_zraku(glm::vec3 origin, glm::vec3 direction) {
  IntersectPoint intersect;
  intersect.intersected = false;

  for (Object *o : objects) {
      IntersectPoint p = o->intersectPoint(origin, direction);
      if (!p.intersected) {
          continue;
      }
      if (!intersect.intersected || p.t < intersect.t) {
          intersect = p;
      }
  }
```

```
  return intersect;
}
```

Metoda `intersectPoint` je vritualna metoda razreda Object. Svaka klasa koji nasljeđuje klasu Object će sama definirati kako ispitati koliziju zrake s jednom njenom instancom - bila to sfera, ravnina, trokut ili mesh trokuta. Njihove konkretne implementacije nećemo nanijeti jer su već matematički opisane u potpoglavlju 2.1.1.

Sada ćemo imlementirati konkretani algoritam renderiranja:

```
glm::vec3 raycast(glm::vec3 origin, glm::vec3 direction) {
    Object *object = nullptr;
    IntersectPoint p = baci_zraku(origin, direction);
    if (!p.intersected)
        return SKYBOX_COLOR;

    glm::vec3 light = glm::vec3(0);
    glm::vec3 normal = glm::normalize(glm::cross(p.vertices[1] - p.vertices[0], p.v
    glm::vec3 color = p.colors[0] * light + phong(p, glm::vec3(0), normal);

    return color;
}
```

Kao što možemo vidjeti, ova funkcija baci zraku i ako postoji probodište, izračuna se osvjetljenje prema Phongovom modelu osvjetljenja, u suporotnom znači da nismo udarili ni u jedan objekt pa se vraća vrijednost s kupole scene.

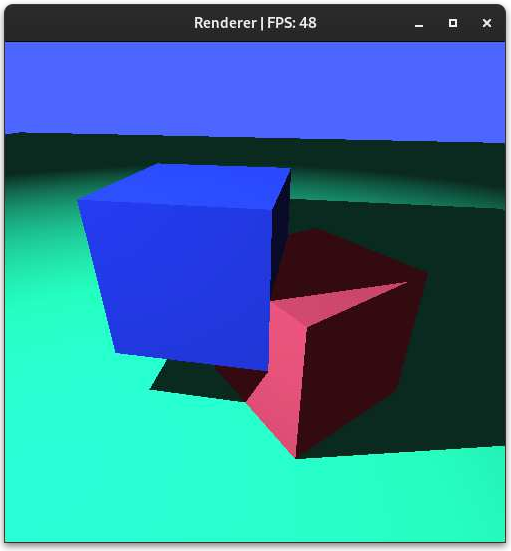
Valja napomenuti da pomoćna funkcija `phong` ne računa ambijentnu komponentu, posebno je pribrajamo. Razlog tome je što kasniji algoritmi (koji su rekurzivni) također koriste ovu funkciju za računanje osvjetljenja, pa bi je zbog toga pribrojili više puta. Mi je želimo pribrojiti samo jednom.

Još nam je preostalo samo uključiti ovu funkciju u petlju izrade prikaza:

```
// prethodni kod...
glm::vec3 boja = raycast(camPos, current - camPos);
```

```
        osvijetliFragment(j, i, boja);
        // daljnji kod...
```

Na slici 3.1. možemo vidjeti primjer scene renderirene pomoću postupka bacanja zrake.



Slika 3.1. Osvjetljenje pomoću bacanja zrake

3.2. Implementacija tehnike praćenja zrake

Kako bi implementirali ovu tehniku, iskoristit ćemo dobar dio prethodnog koda. Naime, samo trebamo napisati funkciju `raycast` koja izravno ostvaruje tehniku.

Implementacija prema pseudokodu iz 2.2.1.:

```
glm::vec3 raytrace(glm::vec3 origin, glm::vec3 direction, int depth) {
    if (depth == 0)
        return glm::vec3(0);

    IntersectPoint p = baci_zraku(origin, direction);
    if (!p.intersected)
        return SKYBOX_COLOR;

    glm::vec3 light = glm::vec3(0);
```

```
    glm::vec3 normal = glm::normalize(glm::cross(p.vertices[1] - p.vertices[0], p.v
    glm::vec3 color = p.colors[0] * light + phong(p, glm::vec3(0), normal);

    if (k_specular > 0) {
        color += k_specular * raytrace(p.point, glm::reflect(direction, normal), de
    }
    if (k_transmit > 0) {
        color += k_transmit * raytrace(p.point, glm::refract(-direction, normal, 1.
    }

    return color;
}
```

Kao što smo već ustanovili, algoritam je nadogradnja na bacanje zrake. Sada također uračunavamo i doprinos komponente odsjaja kao i doprinos komponente prozirnosti.

Možemo vidjeti da algoritam također prima i dubinu o kojoj ovisi konačni izgled rezultata kao i vrijeme provođenja postupka. Primijetimo da ako provedemo algoritam za dubinu 1, dobit ćemo raycastanu sliku. Prema tome, raycasting je poseban slučaj raytracinga te funkciju `raycast` možemo zapisati jednostavno kao:

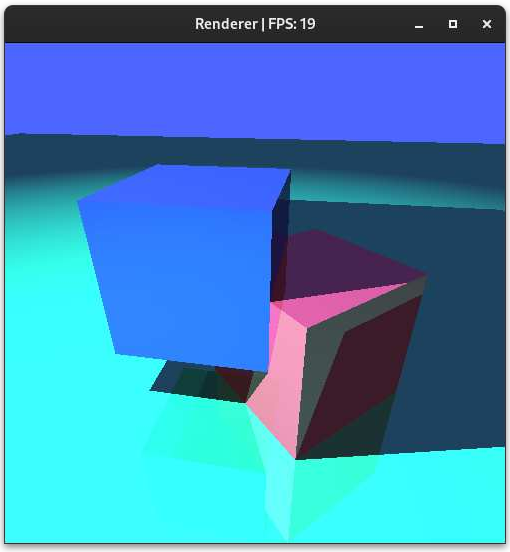
```
glm::vec3 raycast(glm::vec3 origin, glm::vec3 direction) {
  return raytrace(origin, direction, 1);
}
```

Još nam je preostalo u petlju izrade prikaza zamijeniti prijašnji algoritam s ovim:

```
    // prethodni kod...
    glm::vec3 boja = raytrace(camPos, current - camPos, RAYTRACE_DEPTH);
    osvijetliFragment(j, i, boja);
    // daljnji kod...
```

Varijablu *RAYTRACE_DEPTH* ćemo negdje definirati tako da ju možemo lako promijentiti ako želimo ako želimo algoritam pustiti u veću dubinu.

Na slici 3.2. možemo vidjeti sliku renderiranu tehnikom bacanja zrake.



Slika 3.2. Primjer scene renderirane uz pomoć tehnike praćenja zrake

3.3. Implementacija tehnike praćenja puta

Ovu metodu ćemo također ostvariti nadograđivanjem na prijašnju.

Prvo ćemo ostvariti funkciju `pathtrace` prema pseudokodu iz 2.3.1.:

```
glm::vec3 raytrace(glm::vec3 origin, glm::vec3 direction, int depth) {
    if (depth == 0)
        return glm::vec3(0);

    IntersectPoint p = baci_zraku(origin, direction);
    if (!p.intersected)
        return SKYBOX_COLOR;

    glm::vec3 light = glm::vec3(0);
    glm::vec3 normal = glm::normalize(glm::cross(p.vertices[1] - p.vertices[0], p.v
    glm::vec3 color = p.colors[0] * light + phong(p, glm::vec3(0), normal);

    if (k_diffuse > 0) {
        glm::vec3 d = glm::sphericalRand(1.0f);
        if (glm::dot(direction, normal) < 0.0f)
            d = -d;
```

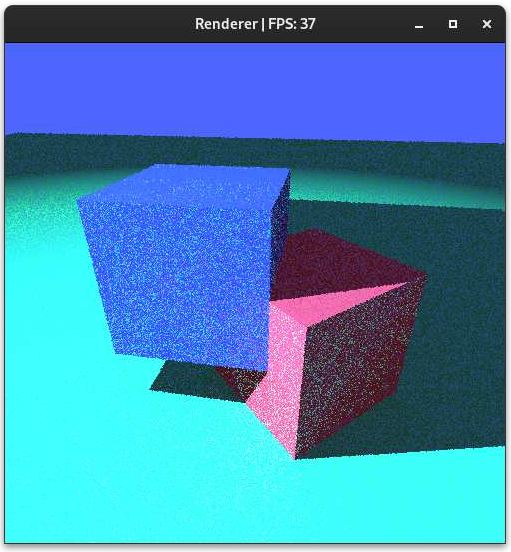
```
        color += k_diffuse * raytrace(p.point, d, depth - 1);
    }

    return color;
}
```

Vidimo da sada algoritam stvara samo jednu novu zraku koju baca u nasumičnom smijeru, odnosno efektivnu prati jednu putanju kroz scenu.

Sada još trebamo ažurirati kod u petlji izrade prikaza. Međutim, sada zraku ne šaljemo uvijek u kut jedne podjele nego negdje nasumično u kvadratu kojem je `current` gornji lijevi kut, a `current + dx + dy` donji lijevi kut. Prema tome, trebamo prilagaditi kod na sljedeći način:

```
// prethodni kod...
float offsetx = ((double) rand() / (RAND_MAX));
float offsety = ((double) rand() / (RAND_MAX));
glm::vec3 target = current + dx * offsetx + dy * offsety;
glm::vec3 boja = pathtrace(camPos, target - camPos, RAYTRACE_DEPTH);
osvijetliFragment(j, i, boja);
// daljnji kod...
```

Na slici 3.3. možemo vidjeti rezultat renderiran tehnikom bacanja zrake.



Slika 3.3. Primjer scene renderirane uz pomoć tehnike praćenja puta

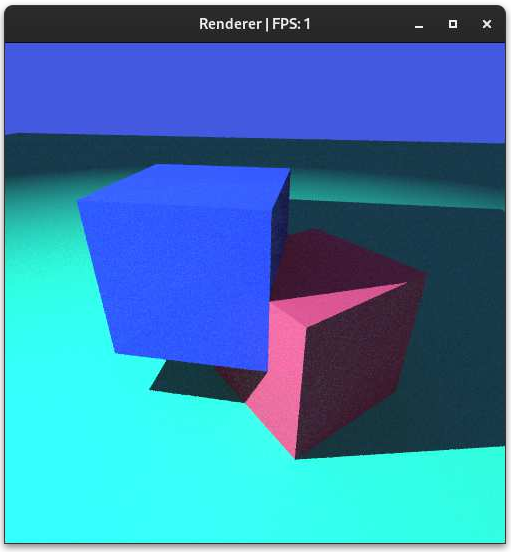
Možemo jasno razaznati zrnatost koje je posljedica što je ova tehnika ostvarena Monte Carlo algoritmom. Kako bi smanjili zrnatost, trebamo sliku izrenderirati više puta te uzeti srednju vrijednost svih slika. Za to ćemo trebati još jedan spremnik u kojeg ćemo spremati konačnu sliku koju treba prikazati – `displayRaster`. U varijabli `raster` će se tako nalaziti spremnik u kojeg renderiramo, a kad završimo s renderiranjem, u pomoćnoj funkciji `iscrtajRaster` ćemo ustvari slati `displayRaster` OpenGL teksture kao izvor.

Možemo na sljedeći način izračunati sliku koju trebamo renderirati:

```
// prijašnji kod u petlji izrade prikaza
float *other = rasteri + width * height * 3 * !currentRasterIndex;
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        glm::vec3 boja1 = getFragmentColor(displayRaster, j, i);
        glm::vec3 boja2 = getFragmentColor(raster, j, i);
        glm::vec3 konacnaBoja = (boja1 * (broj_rendera - 1)) + boja2)
        osvijetliFragment(displayRaster, j, i, color);
    }
}
iscrtajRaster();
glfwSwapBuffers();
```

Ovaj kod se nalazi netom uoči crtanja rastera u spremnik.

Primjer renderiranja nakon 10. iteracije algoritma se nalazi na slici 3.4.



Slika 3.4. Primjer scene renderirane uz pomoć postupka praćenja puta nakon 10 iteracija

4. Rezultati i rasprava

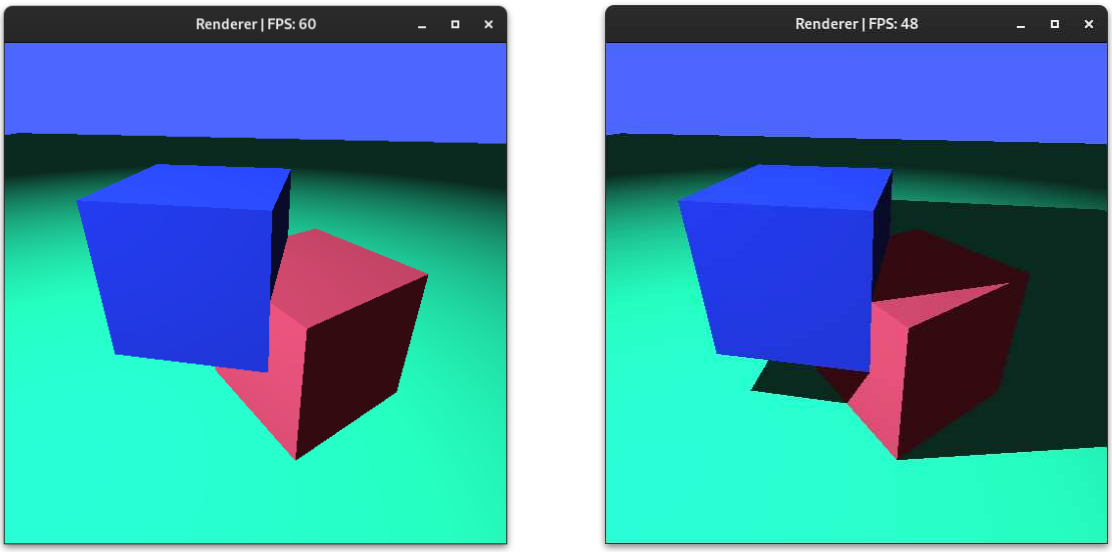
Usporedit ćemo algoritme prema u dvije kategorije:

- vizualni rezultat
- performance

4.1. Vizualni usporedba

4.1.1. Bacanje zrake

Prvo ćemo usporediti postupak bacanja zrake s Phongovim osvjetljenjem. Na slici 4.1. možemo vidjeti usporedbu između. Oba su lokalni modeli osvjetljenja pa možemo izravno usporediti jedan s drugim.



(a) Phongovo osvjetljenje **(b)** Osvjetljenje pomoću bacanja zrake

Slika 4.1. Usporedba osvjetljenja Phongovim postupkom i bacanjem zrake

Kako postupak bacanja zrake radi lokalno osvjetljavanje pomoću Phongovog modela

osvjetljenja, lokalno osvjetljenje kod jednog i drugog slučaja je vrlo slično. Možemo primijetiti da sjena nema kod Phongovog osvjetljenja, on ih sam po sebi ne uključuje, zbog toga moramo na neki drugi način dodati sjene u scenu (npr. mape sjene).

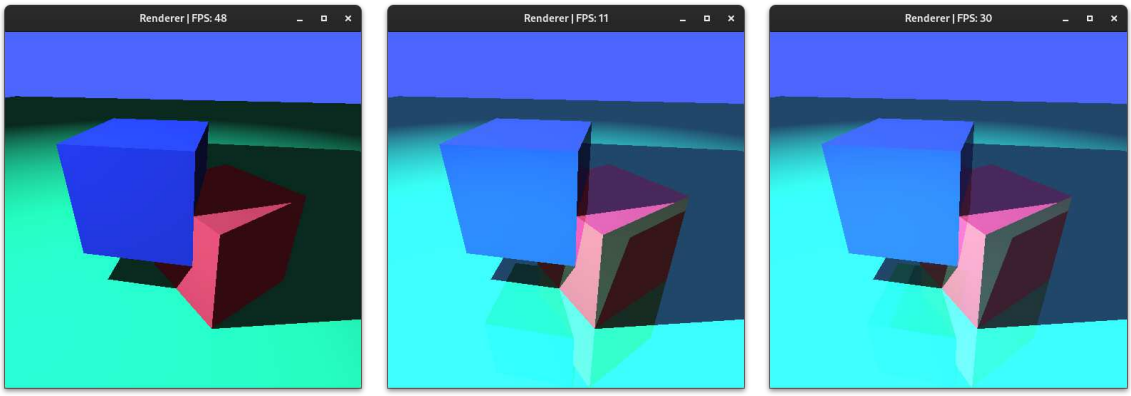
Kod bacanja zrake, sjene su ugrađene u sam model osvjetljenja pa se ne moramo oslanjati na dodatne tehnike za njihov prikaz. Sjene su vrlo oštre što je posljedica toga da izvor točkast, ovakve sjene su karakteristične za ovaj algoritam, kao i za algoritam praćenja zrake (što je očekivano jer je ovaj algoritam poseban slučaj tog algoritma).

4.1.2. Praćenje zrake

Ovaj algoritam je zanimljiviji za promatranje jer možemo, osim parametara Phongovog osvjetljenja, mijenjati i sljedeće parametre:

- dubina provođenja algoritma
- zrcalna konstanta (za svaki objekt ili globalno)

Na slici 4.2. možemo vidjeti renderiranu scenu za dubine 1, 2 i 10.



(a) Dubina 1 **(b)** Dubina 2 **(c)** Dubina 10

Slika 4.2. Usporedba renderiranih scena s različitim dubinama

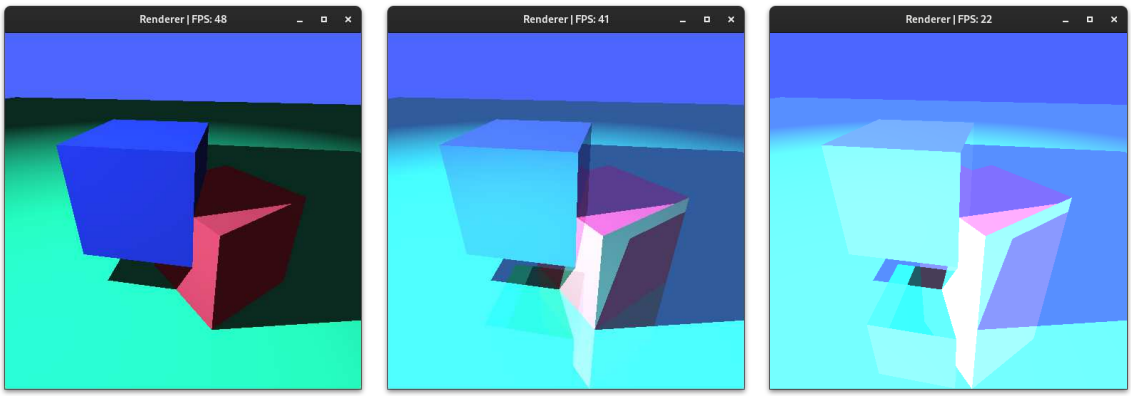
Render dubine 1 je specijalni slučaj. Ne puštaju se dodane zrake nakon prvog sudara pa osvjetljenje zapravo ne ovisi o drugim objektima u sceni – lokalni model osvjetljenja. Ovo je ustvari tehnika bacanja zrake.

Vidimo da render dubine 2 uključuje i odsjaje kod visokoreflektivnih objekata. Možemo primijetiti da se kupola scene također reflektira pa objekti dobijaju plavkastu ni-

jansu.

Također možemo vidjeti da render dubine 10 nema neku veliku razliku od rendera dubine 2, jedino možemo vidjeti detalje ispod kocke gdje je blagi zaklon, u svakom drugom slučaju zraka se odmah reflektira i udara u kupolu scene.

Sljedeće ćemo usporediti različite faktore odsjaja za fiksiranu dubinu. Na slici4.3. možemo vidjeti primjere rendera za konstante odsjaja 0, 0.5 i 1.



(a) Koeficijent 0 **(b)** Koeficijent 0.5 **(c)** Koeficijent 1

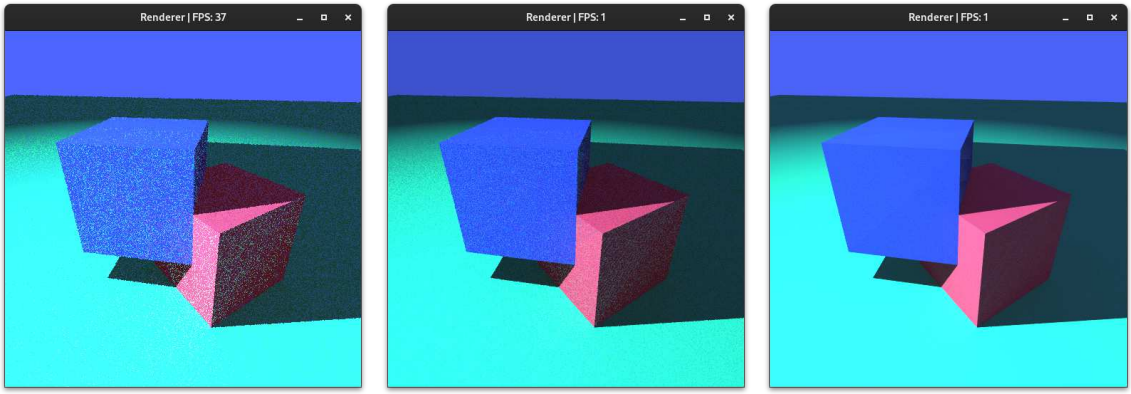
Slika 4.3. Usporedba renderiranih scena s različitim koeficijentima odbijanja

Možemo vidjeti da ako je koeficijent 0 da, predvidljivo, nema refleksija. Za koeficijent 0.5, objekti se vide u refleksijama, ali ne u potpunom osvjetljenju. Za koeficijent 1, objekti u refleksiji su potpuno vidljivi (nisu prigušeni) – savršeno zrcalo.

4.1.3. Praćenje puta

Kod ovog algoritma ćemo pogledati kako se algoritam ponaša za različiti broj iteracija.

Na slici 4.4. možemo vidjeti kako izgledaju renderi od 1, 5 i 30 iteracija.

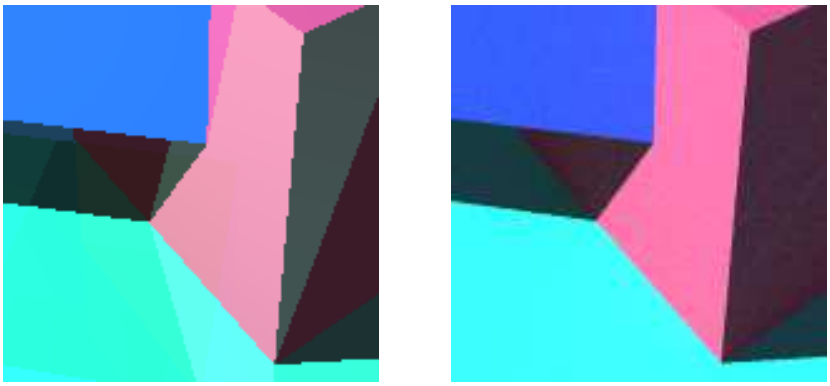


(a) 1 iteracija **(b)** 5 iteracija **(c)** 30 iteracija

Slika 4.4. Usporedba renderiranih scena s različitim brojem iteracija

Zrnatost rezultata se jasno vidi na prvoj slici. Nakon 5 iteracija, zrnatost se vidljivo smanji kao što se vidi na drugoj slici. Na trećoj slici vidi rezultat nakon 30 iteracija, vidimo da se zrnatost opet smanjila, ali ne toliko koliko bi očekivali. Renderirana slika vrlo sporo konvergira prema savršeno nezrnatoj slici pa je zbog toga potrebno uzeti neku proizvoljnu granicu broja iteracija ovisno o promatranim rezultatima.

Sljedeće ćemo promotriti uvećani segment slike.



(a) Praćenje zrake **(b)** Praćenje puta, 30 iteracija

Slika 4.5. Usporedba rubova kod slike renderiranom bacanjem i praćenjem zrake

Promotrimo sliku 4.5. Na prvoj slici možemo jasno raspoznati da se radi o algoritmu praćenja zrake po oštrim rubovima. Druga slika je očito renderana praćenjem puta što možemo prepoznati po donekle raspoznatljivom zrnatošću, ali što još zapinje za oko su glatki rubovima objekata i sjena zbog učinka antialiasinga. Ovaj učinak je ugrađen u sam algoritam – ne moramo ga dodatno dodavati kao kod postupka bacanja zrake, a razlog

zašto on postoji je to što kad biramo prvu zraku iz očišta, biramo nasumično kroz točku segmenta ravnine projekcije kroz koju će proći. Na rubovima nekad pogodi u objekt, a nekad ne, rezultat će bit srednja vrijednost ovisno o tome koliki udijel objekta se nalazi u tom segmentu.

4.2. Performance

Kako su u ovom slučaju algoritmi implementirani programski, nema ih smisla uspoređivati s renderiranjem rasterizacijom jer se ono izvršava u potpunosti na grafičkoj kartici. Također, ove algoritme možemo također ubrzati pomoću "compute" shadera, OpenCL-a, ili pomoću tehnologije CUDA. Zbog toga ćemo više skrenuti pažnju na relativne performance umjesto apsolutnih radi lakše usporedbe i kada se scena renderira u nekoj drugoj konfiguraciji. Kao osnovu ćemo koristiti algoritam bacanja zrake kao osnovu za uspoređivanje.

Prvo ćemo usporediti renderiranje postupkom praćenja zrake s postupkom bacanja zrake i međusobno ovisno o dubini renderiranja. U tablici 4.1. možemo vidjeti srednje rezultate uzastopno ponavljanja za svaku kategoriju.

Algoritam	**Prosječno vrijeme renderiranja (ms)**	**Relativno usporenje**
Bacanje zrake	2547	1.0x
Praćenje zrake, dubina 1	2779	1.09x
Praćenje zrake, dubina 2	4114	1.62x
Praćenje zrake, dubina 5, otvorena scena	4557	1.79x
Praćenje zrake, dubina 5, zatvorena scena	8443	3.31x

Tablica 4.1. Uspredba vremena renderiranja bacanja zrake i praćenja zrake

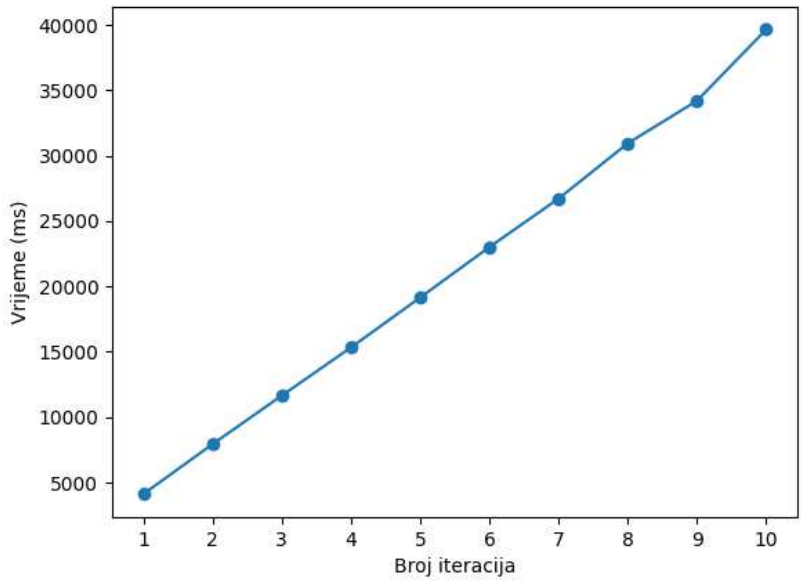
Vidimo da je algoritmu praćenja zrake na dubini 1 treba otprilike jednako koliko i algoritmu bacanja zrake, što je očekivano jer je su algoritam praćenja jednak algoritmu bacanja zrake na dubini 1. Vidimo da na dubini 2 imamo znatno usporenje jer u konačnici imamo dvostruko više zraka za obradu, pa ovoliko usporenje nije neočekivano. Razlog zašto ne vidimo dvostruko usporenje je taj što neke zrake odmah pogode u kupolu scene što prekida rekurziju. To znači da imamo maskimalno dvostruku više zraka nego u prvom slučaju, ali ćemo vjerojatno imati manje, a to vrlo ovisi o sceni koju renderiramo.

Pratimo dva slučaja dubine 5: u jednom slučaju renderiramo scenu gdje se nalazimo

na otvorenom području, a drugom smo unutar prostorije. Možemo primijetiti da su performance puno bolje kada smo na otvorenom. To je zato jer već nakon prvog odbijanja većina zraka završi u kupoli scene pa nema daljnjih refleksija. Kod zatvorene prostorije to nije slučaj jer se zrake ne mogu ranije zaustaviti udarom u kupolu scene jer ne mogu doći do njega unutar sobe.

Možemo primijetiti da vrijeme renderiranja raster poprilično linearno ako zrake ne udaraju u kupolu scene ili ne udaraju u nerefleksivan objekt.

Još ćemo promotriti performance algiritma praćenja puta. Algoritam se za različite dubine ponaša isto kao i algoritam bacanja zrake pa to nećemo ponovno razmatrati. Razmotrit ćemo kako se algoritam ponaša za različiti broj iteracija. Na slici 4.6. možemo vidjeti ovisnost vremena renderiranja o broju iteracija. Predvidljivo, ova je ovisnost linearna.



Slika 4.6. Prikaz vremena potrebnog za renderiranje u ovisnosti o broju iteracija

5. Zaključak

Usporedimo li lokalne modele osvjetljenja s metodama bacanja zrake i praćenja puta, vidimo da su one mnogo bolje rješenje za ostvarivanje fotorealistične slike pomoću računalne grafike. Kako su one globalni modeli osvjetljenja, svaki izvor svjetlosti i objekt u sceni utječe na osvjetljenje u svakoj točki koju renderiramo. Njima možemo postići realistične osvjetljenje i sjene bez korištenja aproksimacija na već postojeći model čime se izbjegavaju artefakti što doprinosi uvjerljivosti konačnog rezultata.

Međutim, ove tehnike imaju i svoje nedostatke i oni se u većini slučajeva tiču performanci. Tehnike praćenja zrake i praćenja puta su osjetno računalno skuplje od procesa rasterzacije što ih čini poprilično nepogodnima za renderiranje u stvarnom vremenu. Moguće je mijenjati dubinu provođenja ovih algoritama ovisno o tome želimo li veću kvalitetu slike ili brže renderiranje.

Imajući to na umu, uz razumnu dubinu, pomoću današnjih grafičkih akceleratora je moguće vršiti renderiranje tehnikom praćenja zrake u stvarnom vremenu. Za to su razvijena posebna sučelja pomoću kojih možemo učinkovito renderirati sliku bez potrebe za implementiranjem algoritama ručno – DXR za DirectX 12 i Ray Tracing familija ekstenzija za Vulkan.

Tehnika praćenja puta daje još bolji rezultat od tehnike bacanja zrake, ali je, zbog svoje iterativne prirode, još uvijek jako skupa za izračun. Zbog toga se danas koristi u aplikacijama gdje je potrebna najviša moguća kvaliteta slike, na primjer kod specijalnih efekata u filmskoj industriji.

Literatura

[1] F. ZEMRIS, “Empirijski i prijelazni modeli”, https://www.zemris.fer.hr/predmeti/irg/predavanja/8_sjenca1.pdf, 2024., pristupljeno: 2024-06-08.

[2] NVIDIA, “Ray tracing”, https://developer.nvidia.com/discover/ray-tracing, 2024., pristupljeno: 2024-06-10.

[3] M. Čupić i Željka Mihajlović, *Interaktivna računalna grafika kroz primjere u OpenGL-u*, 2021., 67–69, 334–337, 339–341. [Mrežno]. Adresa: https://www.zemris.fer.hr/predmeti/irg/knjiga.pdf

[4] S. University, “Monte carlo path tracing”, https://graphics.stanford.edu/courses/cs348b-10/lectures/path/path.pdf, 2004., pristupljeno: 2024-06-02.

Sažetak

Postupak praćenja zrake i praćenja puta

Davor Najev

U ovom radu istražene su metode bacanja zrake, praćenja zrake i praćenja puta, kojima je moguće renderirati fotorealističe slike u računalnoj grafici. Obrađeni su njihovi matematički modeli, s naglaskom na teorijske osnove koje omogućuju precizno simuliranje svjetlosnih interakcija. Dane su konkretne implementacije ovih metoda u programskom jeziku C++, što uključuje opis algoritama i kodne primjere. Također, uspoređene su njihove performance u različitim scenarijima renderiranja, kao i kvaliteta dobivenih slika te su, na temelju tih rezultata, pokazane prednosti i nedostatci svake metode.

Ključne riječi: bacanje zrake; praćenje zrake; praćenje puta; računalna grafika; fotorealistično renderiranje

Abstract

Ray tracing and path tracing

Davor Najev

In this paper, the methods of ray casting, ray tracing, and path tracing, which can render photorealistic images in computer graphics, are explored. Their mathematical models are discussed, with an emphasis on the theoretical foundations that enable precise simulation of light interactions. Specific implementations of these methods in the C++ programming language are provided, including descriptions of the algorithms and code examples. Additionally, their performance in various rendering scenarios and the quality of the resulting images are compared. Based on these results, the advantages and disadvantages of each method are demonstrated.

Keywords: ray casting; ray tracing; path tracing; computer graphics; photorealistic rendering

Privitak A: Repozitorij programskog dijela

Sav kod koji se nalazi u ovom radu se nalazi u Git repozitoriju na sljedećoj poveznici: `https://github.com/spinzed/rt-renderer`