Globalni iluminacijski modeli bazirani na praćenju zrake

Martinjak, Mateo

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mateo Martinjak

GLOBALNI ILUMINACIJSKI MODELI BAZIRANI NA PRAĆENJU ZRAKE

Diplomski rad

Voditelj rada:
doc. dr. sc. Tina Bosner

Zagreb, veljača, 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Kako snaga računala u današnje vrijeme sve više raste, tako raste i snaga grafičkih kartica. Prije par desetljeća nije se ni razmišljalo o ovakvim algoritmima jer su jednostavno preskupi, ali zadnjih par godina ide se prema tome da se omogoućuje na sve više kartica brže izvođenje algoritama za osvijetljenje baziranim na praćenju zraka. Ti algoritmi su i dalje jako skupi za izvršavanje pa ih se najviše isplati pokretati na grafičkim karticama umjesto na procesoru.

U konvencionalnom računalu, koristi se tzv. rasterizacija kako bi se scena crtala na ekran. Rasterizacijom se scena (koja je opisana nekim matematičkim modelima i koordinatama), konvertira u boju koja se prikazuje za svaki piksel. Dakle može se razmišljati o rasterizaciji kao crnoj kutiji kojoj se proslijedi scena i koja vrati boju svakog piksela. (Rasterizacija je detaljnije opisana u poglavlju 3.1)

Algoritmi bazirani na praćenju zraka rade isti zadatak ali na kompletno suprotan način. Za svaki se piksel generira zraka koja se isplali prema sceni te se ona prati dok ne udari u model u sceni. Tada ta zraka vraća boju koju će taj piksel poprimiti.

U ovom radu proći ću nekoliko algoritama za praćenje zrake, kako su se oni kronološki kroz povijest otkrili, te ćemo nekolicinu njih pokrenuti na grafičkoj procesnoj jedinici da se uoče razlike u brzini izvođenja.

Kako su se otkrivali novi algoritmi za praćenje zraka, nije se samo radilo na tome da se smanji njihova kompleksnost, nego se išlo prema što realističnijem prikazu. Prethodnici tih algoritama, kao na primjer ray casting, bili su rudementarni te jednostavni za shvaćanje, pa time i nisu imali realistične prikaze scena. S vremenom kako je zainteresiranost rasla, algoritmi su se unaprijedili. Prvotno se zapitalo zašto zrake u ray castingu stanu, zašto se ne bi mogle odbiti i nastaviti put (tako je nastao ray tracing), ili možda dodati mogućnost da se dodaju neka tijela u scenu u obliku čestica (kao prašina, magla, pa je tako nastao ray marching).

Napomena 0.0.1. *U ovom radu koriste se engleski nazivi za algoritme, jer je točnije i jednostavnije za razumijevanje. Stoga će se od sada koristiti naziv* ***ray tracing****, a ne* ***praćenje zrake uz pomoć odbijanja****. Također će se koristiti naziv* ***path tracing****, a ne* ***praćenje puta uz pomoć odbijanja****.*

Napomena 0.0.2. *U ovom radu naizmjence ću koristiti pojmove okolina i biblioteka (na engleskom aplikacijsko programsko sučelje ili **API**). U ovom radu manje semantičke razlike između tih pojmova su nebitne, općenito se misli na istu stvar. Pa je na primjer OpenGL **okolina** isti pojam kao i OpenGL **API**.*

Opis razvoja projekta

Ukratko ću opisati u ovom poglavlju strukturu projekta. Kao što je rečeno u uvodu, cilj ovog rada je pokazati razne algoritme za praćenje zrake, te ih implementirati i neke od njih pokrenuti na grafičkoj kartici kako bi se pokazala razlika u brzini izvođenja. Ali to "pokrenuti na grafičkoj kartici" nije tako jednostavno. Postoji puno alata koji se koriste za pokretanje algoritama na grafičkoj kartici, te su kompliciranu za korištenje pa su napisana uvodna poglavlja o tim okruženjima u poglavlju 3.

Algoritam praćenja zrake

Algoritam se općenito opisuje tako da se zraka izbacuje iz kamere u neku scenu koja sadrži objekte. Scena sadržava izvore svjetlosti. Algoritam ima podršku za refrakciju i refleksiju koje ćemo obratiti u ovom radu. Sjenčari rade veliku ulogu u ovom algoritmu, i njih ćemo isto detaljno preći u kasnijim poglavljnima. Naime, algoritam radi na taj način da prilikom svakog sudara s objektom pozove sjenčar (možemo o njima zasad razmišljati kao funkcijama koje za određenu točku računaju boju i količinu svjetlosti u točki sudara). Osim sjenčara, potrebno je za algoritam definirati i algoritme koji računaju presjek između zrake i geometrijskih tijela u sceni, kao što su i algoritmi koji generiraju smjerove novih zraka pri udaru.

Algoritam praćenja zrake jest u stvari **skup** puno malih algoritama od kojih svaki ima neku preddefiniranu funkciju koju obavlja.

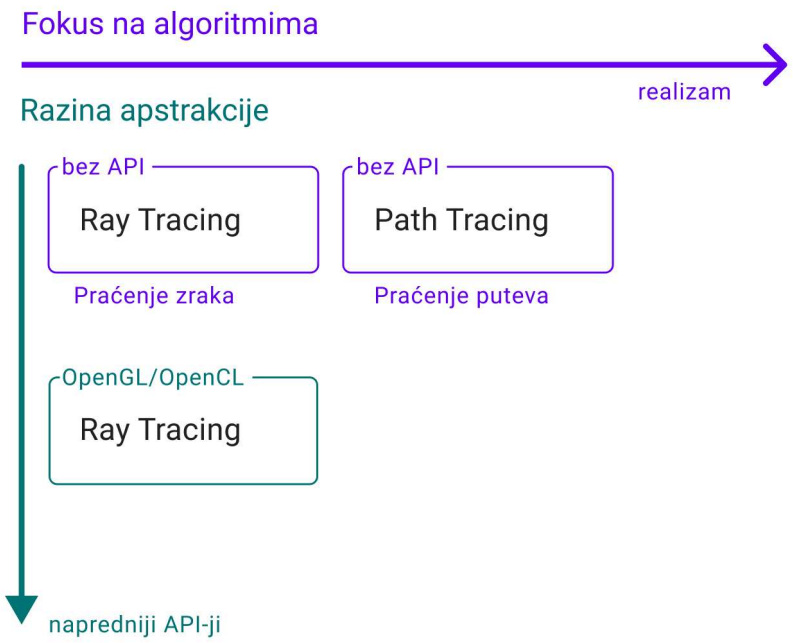
Algoritam praćenja puta

Definicija samog algoritma nije posve jednoznačna jer negdje ćete pročitati da je Algoritam praćenja puta (*eng. Path Tracing*) samo specijalni slučaj praćenja zrake a negdje drugdje da je to potpuno novi koncept algoritma.

Problem kod algoritma praćenja zrake jest da algoritam, umjesto da simulira svjetlost samo prilikom udarca u objekt izračuna količinu svjetlosti ovisno o tome koje izvore vidi u tom trenutku. To nije potpuno realistično, i mada je bolje od klasične rasterizacije, moguće je definirati algoritam koji u potpusnosti simulira zrake svjetlosti sve do izvora.

To algoritam praćenja puta radi. Umjesto da računa količinu svjetlosti u svakom sudaru, algoritam će generirati novu zraku i ”čekati” dok se to stablo zraka ne sudari s izvorom svjetlosti.

Može se zapitati, zar se svjetlost ne kreće iz izvora prema promatraču?
To bi bio puno skuplji proces, te se obrnuti smjer kretanja može shvatiti kao komprimis jer ćemo dobiti sličnu kvalitetu slike ali u puno manjoj vremenskoj složenosti. Neke verzije tog algoritma uistinu rade na taj način, da prate zraku iz smjera izvora svjetlosti do promatrača, ali su ti algoritmi puno više kompleksni i izvan su domene ovog rada.



Slika 0.1: Skica koraka razvoja i okolina korištenih u projektu

Ukratko ćemo objasniti razlike između tih algoritama koje ćemo kasnije proći u detalje u poglavlju 2.

Ray Tracing

Whitted je objavio članak [32] u kojem opisuje kako zrake nastavljaju put nakon prvobitne kolizije, a time možemo opisati ponašanja poput refleksija.

Path Tracing

Kajiya je proširio i tu metodu ray tracinga [19] koristeći svoju integracijsku formulu za svjetlo 1986. godine, te time definirao egzaktniji matematički model za protok svjetlosti u 3D sceni.

Opis okruženja koje ćemo koristiti

1. Implementacija bez okruženja

Implementacija bez okruženja je implementacija koja se izvodi na računalu bez ikakvog grafičkog okruženja. Ova implementacija je najjednostavnija, te nema mogućnost prikaza rezultata u realnom vremenu. Izlaz je tekstualna datoteka koja sadrži sliku u obliku matrice piksela.

2. OpenCL / OpenGL

OpenCL je višenamjenska platforma za paralelno računanje. OpenCL se može koristi za izračunavanje globalnog osvjetljenja na GPU-u, te se također može koristiti za izračunavanje globalnog osvjetljenja u realnom vremenu. OpenGL je specijaliziranija platforma koja se koristi za računalnu grafiku. Mi ćemo OpenGL koristiti kao omot oko OpenCL programa kako bi OpenCL programu dali mogućnost izvršavanja u realnom vremenu.

Poglavlje 1

Definicije pojmova

Za potrebe ovog rada dovoljno je gledati vektorski prostor $\mathbb{R}^n$ za $n \in \{2, 3, 4\}$. Prije nego definiramo pojmove u vezi praćenja zraka, definirat ćemo par uvodnih matematičkih pojmova koji su nam potrebni u kasnijim definicijama. Definicije ćemo definirati za neki generalni $n \in \mathbb{N}$.

Definicija 1.0.1 (Linearna transformacija)**.** *Linearna transformacija (oznaka LT) $\mathbb{R}^n$ je funkcija $T : \mathbb{R}^n \to \mathbb{R}^n$ takva da za svaka dva vektora $\mathbf{u}$ i $\mathbf{v}$ u $\mathbb{R}^n$ i svaki skalar $\alpha \in \mathbb{R}$ vrijedi:*

$$T(\alpha\mathbf{u} + \mathbf{v}) = \alpha T(\mathbf{u}) + T(\mathbf{v}) \text{ (svojstvo linearnosti za T)}$$

Definicija 1.0.2. *Translacija (oznaka T) je transformacija za koju vrijedi*

$$A(x) = x + u, x \in \mathbb{R}^n$$

Definicija 1.0.3. *Afina transformacija (oznaka AF) je transformacija za koju vrijedi*

$$A(x) = B(x) + u, x \in \mathbb{R}^n$$

gdje je B neka LT.

Definicija 1.0.4 (Skalarni produkt)**.** *Skalarni produkt dva vektora $\mathbf{a}$ i $\mathbf{b}$ je definiran kao:*

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

.

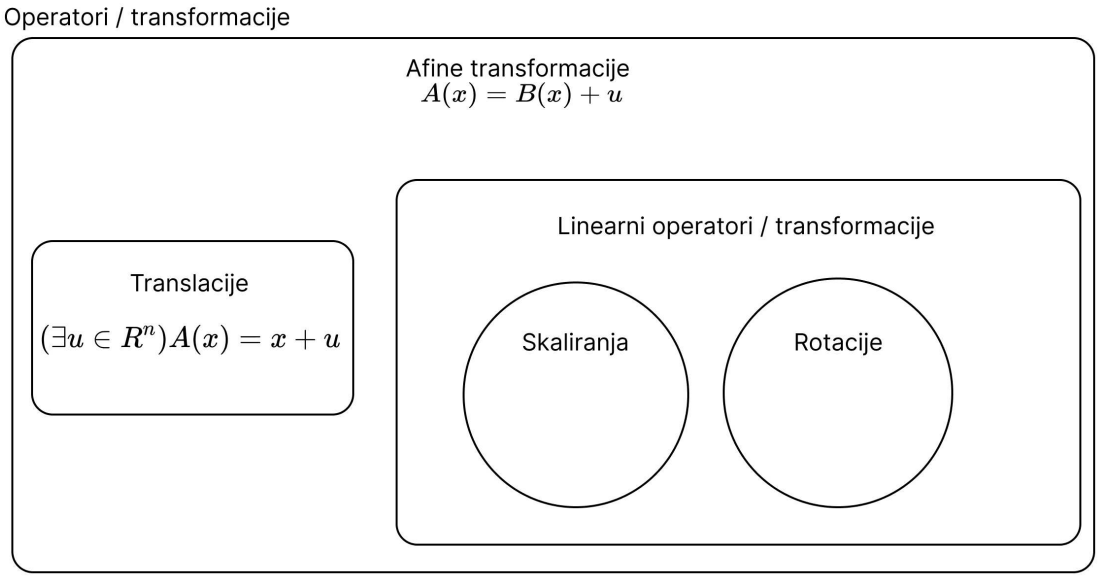
Definicija 1.0.5 (Euklidska norma u $\mathbb{R}^3$)**.** *Euklidska norma je funkcija $\|\cdot\|$ u $\mathbb{R}^3$. Za svaki vektor $\mathbf{v} \in \mathbb{R}^3$ vrijedi $\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + v_3^2}$. U $\mathbb{R}^3$, norma definira duljinu vektora.*

Definicija 1.0.6 (Vektorski produkt)**.** *Vektorski produkt dva vektora* **a** *i* **b** *iz* $\mathbb{R}^3$ *je vektor:*

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2b_3 - a_3b_2)\, i + (a_3b_1 - a_1b_3)\, j + (a_1b_2 - a_2b_1)\, k$$

.

Napomena 1.0.7. *Jedno važno svojstvo koje ćemo koristiti kasnije je: za vektore* **a** *i* **b***, norma* $\|\mathbf{a} \times \mathbf{b}\|$ *je jednaka površini paralelograma kojeg razapinju* **a** *i* **b***.*



Slika 1.1: Venn-ov diagram operatora u $\mathbb{R}^n$

1.1 Zraka

U kontekstu računalne grafike i algoritma praćenja zraka, zraka je matematički pojam koji opisuje polupravac u prostoru $\mathbb{R}^3$ koji se proteže u beskonačnost, iz neke ishodišne točke. Dakle, zraka se općenito definira svojim ishodištem i vektorom smjera, koji je uvijek jediničan. Zraku koristimo u sceni na način da pratimo objekte s kojima se presjeca, koristeći algoritme za presjek zrake i objekata niže u odjeljku, tu ćemo samo prvo dati bazičnu definiciju.

Definicija zrake

Definicija 1.1.1 (Pravac: parametarska definicija)**.** *Pravac je funkcija*

$$P : \mathbb{R} \to \mathbb{R}^3, R(t) = A + tb$$

gdje je $A \in \mathbb{R}^3$ *proizvoljna točka, te* $b \in \mathbb{R}^3$ *proizvoljni vektor za koji vrijedi* $|b| = 1$.

Definicija 1.1.2 (Zraka ili polu-pravac)**.** *Zraka je funkcija* $P : \mathbb{R}_{\geq 0} \to \mathbb{R}^3, R(t) = A + tb$*, gdje je* $A \in \mathbb{R}^3$ *proizvoljna točka, te* $b \in \mathbb{R}^3$ *proizvoljni jedinični vektor.*

Napomena 1.1.3. *U implementacijama i programskim kodôvima uvijek se uzima za pretpostavku da je* $|b| = 1$*. Pokenad će se normalizirati b za svaki slučaj prije računa iako je prije već normalizirana vrijednost.*

Implementacija zrake

Isječak kôda 1.1: Implementacija zrake [27]

```
class Ray
{
public:
    Ray(const vec3& origin, const vec3& direction)
        : origin(origin)
        , direction(direction)
    {
        // normalizira se vrijedost koji određuje smjer
        direction = normalize(direction);
    }

    vec3 origin() const { return origin; }
    vec3 direction() const { return direction; }

    // vraća 3D točku za neku vrijednost parametra t
    vec3 pointAt(float t) const { return origin + t *
    ↪  direction; }

private:
    vec3 origin;
    vec3 direction;
};
```

Ova klasa definira zraku koja se sastoji od smjera i točke ishodišta, te se također sastoji od funkcije koja vraća točku $\mathbf{x} \in \mathbb{R}^3$ za parametar t, koji određuje udaljenost od ishodišta. Funkcija će se koristiti kod testiranja presjeka zrake s drugim objektima u sceni, te ćemo pomoću nje vraćati točku presjeka. Može se vidjeti da se vektor smjera normalizira pri inicijalizaciji vektora, što je u skladu s definicijom, te je nužna predpostavka za algoritme koje ćemo raditi.

1.2 Sfera

Sferu kao elementarni geometrijski oblik definiramo prvu jer su operacije s zrakama nad njom dosta jednostavne. Sfera je definirana svojim centrom $\mathbf{c} \in \mathbb{R}^3$ i radiusom $r \in \mathbb{R}$. Proći ćemo kroz definiciju i implementaciju te ćemo kasnije dati konkretan algoritam kako pronaći točku presjeka zrake i sfere.

Definicija sfere

Definicija 1.2.1 (Sfera)**.** *Sfera je geometrijski objekt definiran centrom $\mathbf{c} \in \mathbb{R}^3$ i radiusom $r \in \mathbb{R}$. Eksplicina definicija sfere je sljedeća:*

$$S = \{\mathbf{x} \in \mathbb{R}^3 : \|\mathbf{x} - \mathbf{c}\| = r\}$$

gdje je $\mathbf{x}$ proizvoljna točka u $\mathbb{R}^3$. Jednadžba opisuje da točka prostora pripada sferi ako je udaljena od središta sfere za udaljenost r.

Implementacija sfere

Isječak kôda 1.2: Implementacija sfere [27]

```
class Sphere
{
public:
    Sphere(vec3 origin, double radius) : origin(origin),
    ↪  radius(radius) {}

    vec3 origin() const { return origin; }
    double radius() const { return radius; }

    bool CollisionWithRay(Ray ray) {
        // izračunaj da li zraka presjeca sferu i vrati
        ↪  boolean
        // konkretan algoritam se nalazi u idućoj sekciji
    };

private:
    vec3 origin;
    double radius;
};
```

1.3 Bézier-ove krivulje i krpice

Bézier-ove krivulje

Bézier-ova krivulja je vrsta parametarske krivulje koja se koristi u računalnoj grafici. Opisuje ju skup točaka koje se zovu **kontrolne točke** krivulje. Definirana je ovako:

$$P(t) = \sum_{i=0}^{n} b_i^n(t) P(i), t \in [0,1] \tag{1.1}$$

gdje su $P_i, i \in 1,2,3,4$ kontrolne točke krivulje, a $b_i^n(t)$ Bernstein-ovi polinomi koji definiraju doprinos svake točke:

$$b_i^n(u) := \binom{n}{i} u^i (1-u)^{(n-i)} \tag{1.2}$$

Napomena 1.3.1. *Samo razmatramo kubične Bézier-ove krivulje (gdje su b_i^n kubični polinomi) jer su nam bitni za algoritme:*

$$P(t) = \sum_{i=0}^{3} b_i^3(t)P(i), t \in [0,1] \tag{1.3}$$

pa su time težine:

$$\begin{aligned} b_0(t) &= (1-t)^3 \\ b_1(t) &= 3(1-t)^2 t \\ b_2(t) &= 3(1-t)t^2 \\ b_3(t) &= t^3 \end{aligned}$$

Bézier-ove krpice

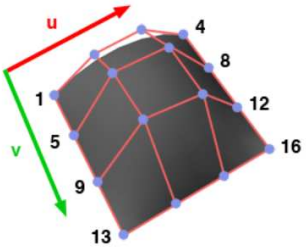
Bézier-ove krpica je vrsta parametarske plohe koja je definirana s 4x4 točke koje zovemo **kontrolnim točkama**. Parametri su $u, v \in [0,1]$. Slika tog preslikavanja je glatka ploha u R^3. Konkretno, Bézier-ova krpica je definirana:

$$P(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} b_i^n(u) b_j^m(v) P_{ij} \tag{1.4}$$

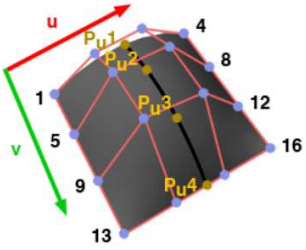
gdje su $b_i^n(u)$ Bernstein-ovi polinomi (1.2). Presjek zrake i Bézier-ove krpice se može direktno evaluirati u algoritmima praćenja zraka, ali metode za to su jako spore. Puno brže rješenje je da se krpice transformiraju u Mesh.

Evaluacija Bézier-ove krpice

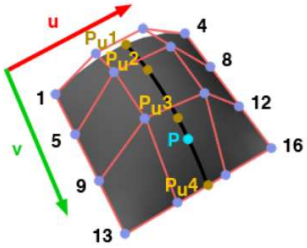
Evaluacija točke na krpici za neki par (u, v) je dosta jednostavno. Svaki red kontrolnih točaka se tretira kao jedna Bézier-ova krivulja. Svaku od tih krivulja evaluiramo posebno u točki u. Nakon toga se evaluira Bézier-ova krivulja s 4 rezultantne točke u točki v, time smo dobili konačnu točku $P = P(u, v)$ (pogledati sliku 1.2).



(a) Evaluira se svaka od v Bézier-ova krivulja u točki u, te se rezultantna točka sprema u $Pu[i], 0 \leq i \leq v$



(b) Evaluira se Bézier-ova krivulja definirana s vektorom točaka Pu u točki v

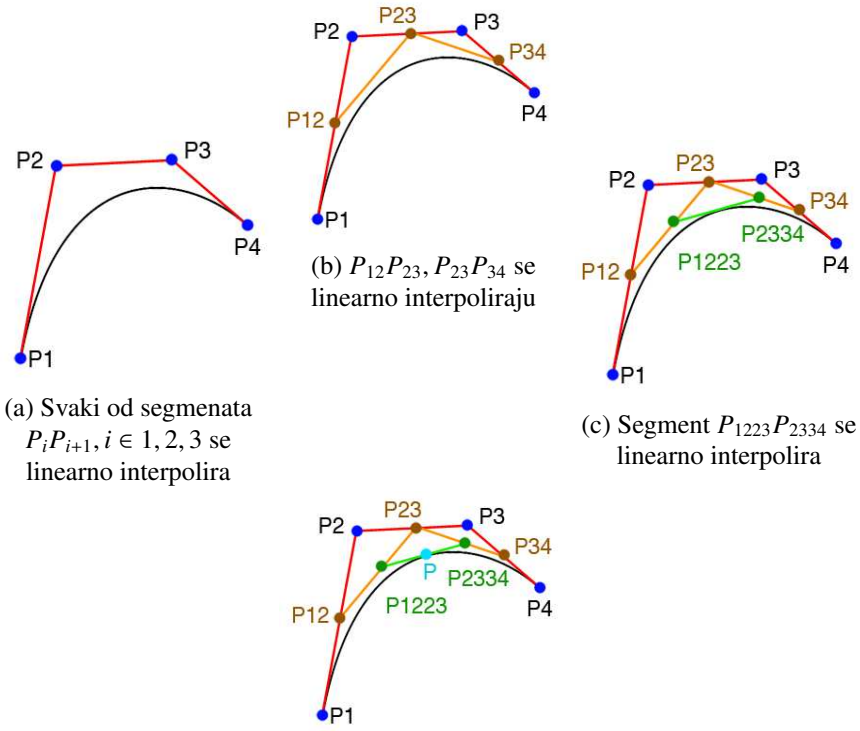


(c) Rezultat je točka $P = P(u, v)$

Slika 1.2: Evaluacija Bézier-ove krpice: prikazane su osi u i v te 16 kontrolnih točaka [2]

Isječak kôda 1.3: Evaluacija Bézier-ove krpice [2]

```
vec3 evalBezierCurve(const vec3 *P, const float t)
{
    float b0 = (1 - t) * (1 - t) * (1 - t);
    float b1 = 3 * t * (1 - t) * (1 - t);
    float b2 = 3 * t * t * (1 - t);
    float b3 = t * t * t;
    return P[0] * b0 + P[1] * b1 + P[2] * b2 + P[3] * b3;
}

vec3 evaluateBezierSurface(vec3 P[16], float u, float v)
{
    vec3 Pu[4];
    for (int i = 0; i < 4; ++i) {
       vec3 curveP[4];
       curveP[0] = P[i * 4];
       curveP[1] = P[i * 4 + 1];
       curveP[2] = P[i * 4 + 2];
       curveP[2] = P[i * 4 + 3];
       Pu[i] = evalBezierCurve(curveP, u);
    }
    return evalBezierCurve(Pu, v);
}
```



(a) Svaki od segmenata $P_iP_{i+1}, i \in 1, 2, 3$ se linearno interpolira

(b) $P_{12}P_{23}, P_{23}P_{34}$ se linearno interpoliraju

(c) Segment $P_{1223}P_{2334}$ se linearno interpolira

Slika 1.3: Evaluacija Bézier-ove krivulje pomoću de Casteljau-ovog algoritma[2]

Kako bi smanjili broj množenja kod evaluacije Bézier-ove krivulje, koristimo de Casteljau-ov algoritam (također poznat kao Bézier-ov konstruktivni algoritam) [3] [2]. Algoritam koristi rekurzivne linearne interpolacije segmenata kontrolnih točaka: u prvoj iteraciji dobimo 3 nove točke koje definiraju 2 nova segmenta, koji se ponovno interpoliraju kako bi se dobio novi segment, te se na kraju dobi sama evaluirana točka P (pogledati sliku 1.3).

Isječak kôda 1.4: Evaluacija Bézier-ove krivulje (de Casteljau)[2]

```
vec3 evalBezierCurveDeCasteljau(const vec3 *P, const float t)
↪  {
    vec3 P12 = (1 - t) * P[0] + t * P[1];
    vec3 P23 = (1 - t) * P[1] + t * P[2];
    vec3 P34 = (1 - t) * P[2] + t * P[3];
    vec3 P1223 = (1 - t) * P12 + t * P23;
    vec3 P2334 = (1 - t) * P23 + t * P34;
    return (1 - t) * P1223 + t * P2334; }
```

1.4 Normala i tangencijalna ravnina

Pošto se bavimo presjecima tijela i zraka, normala u točki presjeka nam je bitna kako bismo generirali nove zrake pomoću izračunatih smjerova refrakcije i refleksije, te kako bismo izračunali intezitet svjetlosti u točki, jer algoritmi osvijetljenja zahtjevaju da znamo smjer normale.

Definicija 1.4.1 (Ploha [15])**.** *Ploha je skup točaka iz* $(\mathbb{R} \cup \infty)^3$*, čije su točke neprekidno povezane.*

Definicija 1.4.2 (Regularna točka [15])**.** *Za točku* $\mathbf{x}$ *plohe* S *kažemo da je regularna ako sve tangente plohe* S *kojima je* $\mathbf{x}$ *diralište leže u jednoj ravnini.*

Definicija 1.4.3 (Tangencijalna ravnina [15])**.** *Tangencijalna ravnina plohe* S *u nekoj njezinoj regularnoj točki* $\mathbf{x}$ *je ravnina koja sadrži sve tangente plohe* S *s diralištem u* $\mathbf{x}$*, tj. tangente s diralištem u* $\mathbf{x}$ *svih onih krivulja plohe* S *koje prolaze točkom* $\mathbf{x}$*.*

Definicija 1.4.4 (Vektor normale [15])**.** *Normala plohe* S *u nekoj njezinoj regularnoj točki* $\mathbf{x}$ *je jedinični vektor koji je okomit na tangencijalnu ravninu plohe* S *u točki* $\mathbf{x}$*.*

Napomena 1.4.5. *Tangencijalna ravnina u točki* $\mathbf{x_0}$ *se također može opisati preko sljedeće jednadžbe koristeći skalarni produkt 1.0.4,ako je poznata normala u* $\mathbf{N_{x_0}}$ *u točki* $\mathbf{x_0}$*:*

$$(\forall \mathbf{x})(\mathbf{x} - \mathbf{x_0}) \cdot \mathbf{N_{x_0}} = 0$$

Vektor normale trokuta

Za potrebe prikazivanja **Mesh** objekta (skupa trokuta), navodimo račun za traženje normale nekog fiksnog trokuta: Za trokut ABC, (gdje su vrhovi definirani u smjeru obrnutom kazaljke na satu) s bridovima $e := B - A$ i $f := C - A$, normala je jednaka:

$$N := \frac{e \times f}{\|e \times f\|} \tag{1.5}$$

Vektor normale sfere

Za sferu s središtem u točki $\mathbf{S}$ i radiusom $r \in \mathbb{R}_+$, normala u točki sfere $\mathbf{x}$ jednaka je:

$$N := \frac{\mathbf{x} - \mathbf{S}}{r} \tag{1.6}$$

Vektor normale ravnine

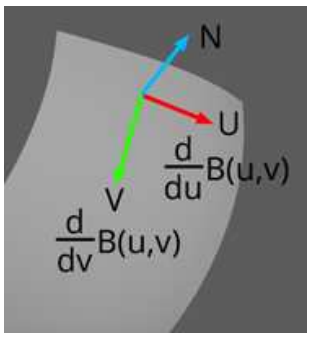
Za ravninu $ax + by + cz + d = 0$ zadanu implicitno, normala je jednaka:

$$N := \frac{(a, b, c)}{\|(a, b, c)\|}$$

Ali ravninu ćemo često u programima zadavati preko 3 točke koje opisuju proizvoljni trokut u toj ravnini, te se za to primjenjuje metoda 1.5.

Vektor normale Bézier-ove krpice

Pošto nam je za sjenčanje potrebno definirati normalu točke krpice, potrebno ih je naći.



Slika 1.4: Parcijalne derivacije Bézier-ove krpice daju u i v komponentu tangencijalne ravnine u traženoj točki, koje ćemo koristiti kako bi našli normalu [2]

Nalaženje parcijalnih derivacija za svaku os je jednostvano jer znamo definiciju Bézier-ove krpice (1.4):

$$\frac{\partial}{\partial u}B(u,v) = -3(1-u)^2 \sum_{j=0}^{3} b_j^3(v)P_{0,j} + (3(1-u)^2 - 6u(1-u)) \sum_{j=0}^{3} b_j^3(v)P_{1,j} +$$
$$(6u(1-u) - 3u^2) \sum_{j=0}^{3} b_j^3(v)P_{2,j} + 3u \sum_{j=0}^{3} b_j^3(v)P_{3,j}$$
$$\frac{\partial}{\partial v}B(u,v) = -3(1-v)^2 \sum_{i=0}^{3} b_i^3(u)P_{i,0} + (3(1-v)^2 - 6v(1-v)) \sum_{i=0}^{3} b_i^3(u)P_{i,1}$$
$$(6v(1-v) - 3v^2) \sum_{i=0}^{3} b_i^3(u)P_{i,2} + 3v^2 \sum_{i=0}^{3} b_i^3(u)P_{i,3}$$

Time zadajemo sljedeći dio koda, kojim računamo parcijane derivacije u traženoj točki te nakon toga koristimo postupak

$$N = \frac{\frac{\partial}{\partial u}B(u,v) \times \frac{\partial}{\partial v}B(u,v)}{\|\frac{\partial}{\partial u}B(u,v) \times \frac{\partial}{\partial v}B(u,v)\|} \tag{1.7}$$

kako bi konačno našli traženu normalu. Samo pratimo gornje formule:

Isječak kôda 1.5: Računanje normale točke Bézier-ove krpice [2]

```
vec3 dUBezier(const vec3 *controlPoints, const float &u, const
↪  float &v)
{
   vec3 P[4];
   vec3 vCurve[4];
   for (int i = 0; i < 4; ++i) {
       P[0] = controlPoints[i];
       P[1] = controlPoints[4 + i];
       P[2] = controlPoints[8 + i];
       P[3] = controlPoints[12 + i];
       vCurve[i] = evalBezierCurveDeCasteljau(P, v);
   }

   return -3 * (1 - u) * (1 - u) * vCurve[0] +
       (3 * (1 - u) * (1 - u) - 6 * u * (1 - u)) * vCurve[1] +
       (6 * u * (1 - u) - 3 * u * u) * vCurve[2] +
       3 * u * u * vCurve[3];
}
```

Isječak kôda 1.6: Računanje normale točke Bézier-ove krpice [2]

```
vec3 dVBezier(const vec3 *controlPoints, const float &u, const
↪  float &v)
{
   vec3 uCurve[4];
   for (int i = 0; i < 4; ++i) {
       uCurve[i] = evalBezierCurveDeCasteljau(controlPoints +
       ↪  4 * i, u);
   }

   return -3 * (1 - v) * (1 - v) * uCurve[0] +
       (3 * (1 - v) * (1 - v) - 6 * v * (1 - v)) * uCurve[1] +
       (6 * v * (1 - v) - 3 * v * v) * uCurve[2] +
       3 * v * v * uCurve[3];
}
```

Isječak kôda 1.7: Računanje normale točke Bézier-ove krpice [2]

```
// napokon možemo naći normalu za traženu točku
vec3 dU = dUBezier(controlPoints, u, v);
vec3 dV = dVBezier(controlPoints, u, v);
N = normalize(cross(dU,dV));
```

Zbog identičnog problema s brojem množenja koji smo imali kod krivulja, i tu ćemo navesti alternativni algoritam. Najprije navodimo pomoćnu lemu iz predavanja [3]:

Lema 1.4.6. *Za Bézier-ovu krivulju definiranu s (1.1) vrijedi:*

$$P'(t) = 3(P_3^{[2]}(t) - P_2^{[2]}(t)) \tag{1.8}$$

gdje je $P_k^{[s]}(t)$ *linearna intrerpolacija u t točaka* $P_k^{[s-1]}(t)$ *i* $P_{k-1}^{[s-1]}(t)$*, te vrijedi* $P_k^{[0]} := P_k$

Time možemo definirati sljedeću funkciju za računanje derivacije:

Isječak kôda 1.8: Računanje derivacije Bezier-ove krivulje

```
vec3 deriveBezierCurve(const vec3 *P, const float &t)
{
    vec3 P12 = (1 - t) * P[0] + t * P[1];
    vec3 P23 = (1 - t) * P[1] + t * P[2];
    vec3 P34 = (1 - t) * P[2] + t * P[3];
    vec3 P1223 = (1 - t) * P12 + t * P23;
    vec3 P2334 = (1 - t) * P23 + t * P34;
    return 3(P2334 - P1223);
}
```

Bézier-ovu krpicu (1.4) možemo drukčije napisati, tako da definiramo da je unutarnja suma $Q_i(v) := \sum_{j=0}^{3}(b_j^3(v)P_{ij})$:

$$P(u,v) = \sum_{i=0}^{3} b_i^3(u)Q_i(v) \tag{1.9}$$

ako iskoristimo lemu (1.8) parcijalna derivacija u smjeru u je:

$$\frac{\partial P}{\partial u}(u,v) = 3(Q_3^{[2]}(u) - Q_2^{[2]}(u)) \tag{1.10}$$

dok je parcijalna derivacija u smjeru v:

$$\frac{\partial P}{\partial v}(u,v) = \sum_{i=0}^{3} b_i^3(u)Q_i'(v) \tag{1.11}$$

Dakle, za prvu parcijalnu derivaciju bi trebali prvo evaluirati Bézier-ove krivulje u smjeru u, te nakon toga u tim evaluacijama kao novim kontrolnim točkama izračunati derivaciju Bézier-ove krivulje u točki (u, v).

S druge strane, drugu parcijalnu derivaciju nalazimo tako da prvo deriviramo Bézier-ove krivulje u smjeru v, te nakon toga s tim derivacijama kao novim kontrolnim točkama evaluiramo Bézier-ovu krivulju u točki (u, v).

Isječak kôda 1.9: Računanje parcijalne derivacije Bezier-ove krpice

```
vec3 dUBezier(const vec3 *controlPoints, const float &u, const
↪  float &v)
{
   vec4 Q[4],P[4];
   for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j)
            Q[j] = controlPoints[i+j*(4+1)];
        P[i] = evalBezierCurveDeCasteljau(Q,v);
   }
   return deriveBezierCurve(P,u); // pogledati (1.10)
}
```

Isječak kôda 1.10: Računanje parcijalne derivacije Bezier-ove krpice

```
vec3 dVBezier(const vec3 *controlPoints, const float &u, const
↪  float &v)
{
   vec4 Q[4],P[4];
   for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j)
            Q[j] = controlPoints[i+j*(4+1)];
        P[i] = deriveBezierCurve(Q,v);
   }
   return evalBezierCurveDeCasteljau(P,u); // pogledati (1.11)
}
```

1.5 Presjeci zrake i objekata u sceni

Zraka se na svojoj putanji može sudariti s objektima u sceni. Pod objektom se misli na geometrijske likove i tijela koji su najčešće jedno od sljedećeg:

- Ravnina
- Sfera
- Trokut
- Valjak

- Neko drugo tijelo - za koje se može definirati pojam *kolizija s zrakom*

Presjek zrake i ravnine

Za potrebe ovog rada promatramo samo vektorski prostor $\mathbb{R}^3$.

Definicija 1.5.1 (Ravnina: implicitna jednadžba)**.** *Ravnina u prostoru* $\mathbb{R}^3$ *je jednadžba*

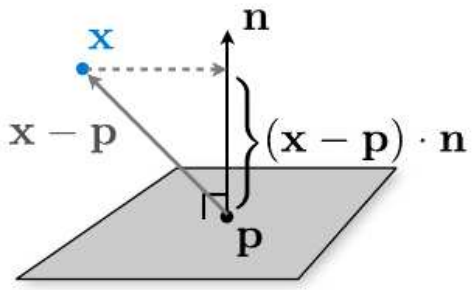
$$ax + by + cz + d = 0$$

gdje su $a, b, c, d \in \mathbb{R}$ *proizvoljni parametri, s time da barem jedan od* a, b, c *treba biti različit od nule.*

Definicija 1.5.2 (Ravnina: vektorska jednadžba)**.** *Ravnina se definira kao jednadžba*

$$(\boldsymbol{x} - \boldsymbol{p}) \cdot \boldsymbol{N} = 0$$

gdje je $\boldsymbol{N} \in \mathbb{R}^3$ *normala ravnine,* $\boldsymbol{p} \in \mathbb{R}^3$ *točka na ravnini, a* $\boldsymbol{x} \in \mathbb{R}^3$ *proizvoljna točka u prostoru.*



Slika 1.5: Izvor: [16][17]

Ako ubacimo parametarsku definiciju zrake u jednadžbu ravnine dobivamo:

$$\begin{aligned} (\boldsymbol{O} + t\boldsymbol{D} - \boldsymbol{p}) \cdot \boldsymbol{N} &= 0 \\ t(\boldsymbol{D} \cdot \boldsymbol{N}) + (\boldsymbol{O} - \boldsymbol{p}) \cdot \boldsymbol{N} &= 0 \\ t &= -\frac{(\boldsymbol{O} - \boldsymbol{p}) \cdot \boldsymbol{N}}{\boldsymbol{D} \cdot \boldsymbol{N}} \end{aligned} \tag{1.12}$$

Dobili smo parametar t za zraku $\boldsymbol{O} + t\boldsymbol{D}$, koji nam govori na kojem udaljenom mjestu od početka zrake se nalazi točka presjeka zrake i ravnine. Ako je $t \geq 0$, to znači da je točka presjeka na zraci, a ako je $t < 0$, to znači da je točka presjeka iza početka zrake.

Time smo definirali sljedeću funkciju:

Isječak kôda 1.11: Implementacija kolizije zrake i ravnine [2]

```
bool CollisionWithPlane(Ray ray, Plane p){
    double denominator = dot(ray.direction, p.normal);

    vec3 point = p.getPointOnAPlane();

    float t = - dot((ray.origin - point), p.normal) /
    ↪  denominator;

    return t >= 0;
}
```

Presjek zrake i trokuta

Postoji 3 uvjeta na točku presjeka $P_0 = P(t_0)$:

1. uvjet: točka presjeka nalazi se na zraci $P(t) = \boldsymbol{O} + t\boldsymbol{D}$
2. uvjet: točka presjeka nalazi se na ravnini
3. uvjet: točka presjeka nalazi se između 3 stranice trokuta

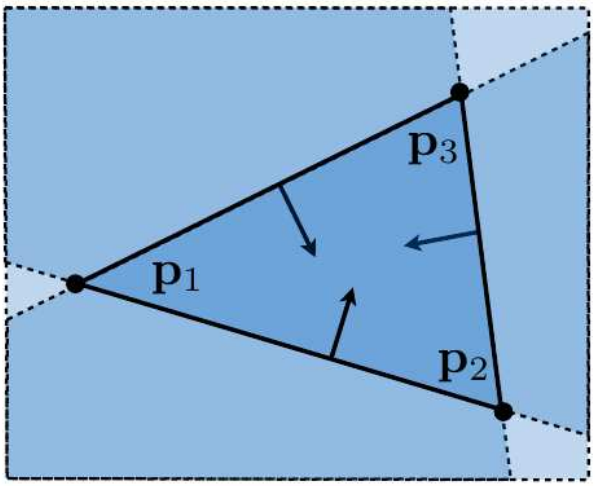
Baricentrične koordinate

Baricentrične koordinate su sistem koordinata koji se koirsti u grafici koji nam dozvoljava da definiramo poziciju točke unutar trokuta, u ovisnosti o težinama koje ta točka daje svakom od vrhova trokuta. Pretpostavimo da imamo trokut ABC.

Proizvoljnu točku P unutar trokuta možemo zapisati pomoću baricentričnih koordinata:

$$(\exists u, v \in [0,1], u + v \leq 1)P = (1 - u - v) \cdot P_1 + u \cdot P_2 + v \cdot P_3$$

gdje težine u, v reprezenziraju omjere površine trokuta naspram trokuta ABC (Slika 1.7).



Slika 1.6: Izvor: [16][17]

Kako bi dobili težine u, v i $w := (1 - u - w)$, za početak nam treba površina trokuta ABC, što ćemo dobiti preko površine paralelograma kojeg generiraju stranice trokuta $u' := (C - A)$ i $v' = (B - A)$. S H ćemo označiti visinu paralelograma iz točke C na stranicu v':

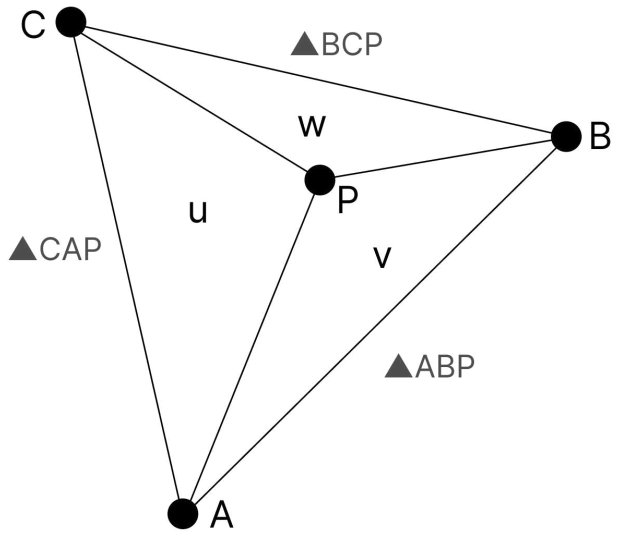
$$P_{ABC} = \frac{H\,\|v'\|}{2} = \frac{H\,\|v'\|\,\|u'\|\,\sin\theta}{2} \tag{1.13}$$

gdje θ označava kut između stranica u' i v'. To možemo pojednostaviti koristeći svojstvo vektorskog produkta 1.0.7:

$$P_{ABC} = \frac{\|u' \times v'\|}{2} \tag{1.14}$$

Kako bi izračunali površine malih trokuta CAP, BCP i ABP koristimo istu metodu, te su onda težine jednake:

$$w = \frac{P_{BCP}}{P_{ABC}} \quad u = \frac{P_{CAP}}{P_{ABC}} \quad v = \frac{P_{ABP}}{P_{ABC}} \tag{1.15}$$



Slika 1.7: Proizvoljna točka P trokuta ABC prikazana u baricentričnim koordinatama

Računanje presjeka

Računanje se radi u koracima [2]:

Primjer 1.5.3 (Računanje presjeka trokuta i zrake)**.** *Pretpostavimo da je poznata zraka $P = \boldsymbol{O} + t\boldsymbol{D}$, te da je poznat trokut ABC. Prvo se izračuna normala ravnine N na kojoj se nalazi trokut. To se može direktno izračunati iz vrhova trokuta. Iz $N \cdot D$ testiramo da li je zraka paralelna s ravninom. Ako nije, tj. $N \cdot D \neq 0$, pomoću (1.12) dobivamo t_0.*

$$t_0 = \frac{-(\boldsymbol{N} \cdot \boldsymbol{O}) - (-N \cdot A)}{\boldsymbol{N} \cdot \boldsymbol{D}} \tag{1.16}$$

Na kraju računamo samu točku presjeka P:

$$P = O + t_0 \cdot D$$

Još moramo provjetiti da li točka leži unutar trokuta, tu se koriste baricentrične koordinate. Za svaku stranicu $e = B - A$ trokuta $\triangle ABC$ računa se: $VP = P - A$ te pomoću njega računamo:

$$C := e \times VP \tag{1.17}$$

gdje koristimo vektorski produkt 1.0.6. Da bi testirali s koje strane trokuta se P nalazi, provjerimo

$$N \cdot C$$

Ako je taj skalarni produkt manji od nule, to znači da je točka P s druge strane trokuta i ne presjeca trokut. Identične provjere radimo i za preostala dva slučaja

$$e = C - B, VP = P - B$$
$$e = A - C, VP = P - C$$

U sljedećem kodu koristi se identičan koncept da bi se našla točka presjeka [2]:

Isječak kôda 1.12: Implementacija kolizije zrake i trokuta [2]

```
bool rayTriangleIntersect(
    const vec3 &orig, const vec3 &dir,
    const vec3 &A, const vec3 &B, const vec3 &C,
    float &t)
{
    vec3 N = getNormal();
    // PRESJEK RAVNINE I ZRAKE
    // računamo točku presjeka ravnine i zrake P
    //----------------------------------------
    float NdotRayDirection = N.dotProduct(dir);
    if (fabs(NdotRayDirection) < kEpsilon)
        return false;
    float d = -N.dotProduct(A);

    // računamo parametar t po jednadžbi (1.16)
    t = -(N.dotProduct(orig) + d) / NdotRayDirection;
    vec3 P = orig + t * dir;

    // PRESJEK TROKUTA I ZRAKE
    // računamo da li je P unutar trokuta
    //----------------------------------------
    // test prve stranice
    vec3 edge0 = B - A;
    vec3 vp0 = P - A;
    C = edge0.crossProduct(vp0); //po jednadžbi (1.17)
    if (N.dotProduct(C) < 0) return false;
    // test druge stranice
    vec3 edge1 = C - B;
    vec3 vp1 = P - B;
    C = edge1.crossProduct(vp1); //po jednadžbi (1.17)
    if (N.dotProduct(C) < 0)  return false;
    // test trece stranice
    vec3 edge2 = A - C;
    vec3 vp2 = P - C;
    C = edge2.crossProduct(vp2); //po jednadžbi (1.17)
    if (N.dotProduct(C) < 0) return false;
    return true;
}
```

Presjek zrake i sfere

Za sve točke P na sferi vrijedi $||P - C|| = r$, gdje je C centar sfere, a r radijus sfere. Zraka $R(t) = O + Dt$ prolazi kroz sferu ako vrijedi $||R(t) - C|| = r$. Smjer D je jediničan. Tada je $||O + Dt - C|| = r$. Tada se dobiva kvadratna jednadžba za t:

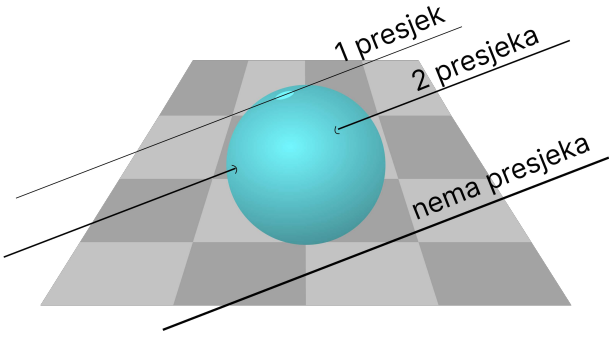
$$\begin{aligned} ||O + Dt - C|| &= r \\ (O + Dt - C) \cdot (O + Dt - C) &= r^2 \\ (O - C) \cdot (O - C) + 2t(O - C) \cdot D + t^2 D \cdot D &= r^2 \\ t^2 D \cdot D + 2t(O - C) \cdot D + (O - C) \cdot (O - C) - r^2 &= 0 \end{aligned}$$

Dobije se sljedeća kvadratna formula:

$$\begin{aligned} b &= (O - C) \cdot D \\ c &= (O - C) \cdot (O - C) - r^2 \\ t^2 D \cdot D + 2tb + c &= 0 \\ t^2 + 2tb + c &= 0 \\ t &= -b \pm \sqrt{b^2 - c} \end{aligned}$$

Imamo 3 slučaja:

1. $b^2 - c < 0$ - nema presjeka
2. $b^2 - c = 0$ - jedan presjek
3. $b^2 - c > 0$ - dva presjeka



Slika 1.8: Prjesek zrake i sfere

Time smo definirali sljedeću funkciju:

Isječak kôda 1.13: Implementacija kolizije zrake i sfere

```
bool CollisionWithSphere(Ray ray, Sphere s){
    float b = (ray.origin - s.origin) * ray.direction;
    float c = square(ray.origin - s.origin) -
    ↪  square(s.radius);

    return b*b - c >= 0 ;
}
```

1.6 Monte carlo integracija

Uvod

Pošto je integracija na računalu skup proces, možemo ju na neki način analitički aproksimirati.

Općenito, integral oblika

$$I = \int_a^b f(x)dx$$

možemo zapisati kao

$$I = \text{area}\,(f(x), a, b)$$

tj, u kontekstu ovog poglavlja možemo koristiti teorem srednje vrijednosti i zapisati integral na sljedeći način

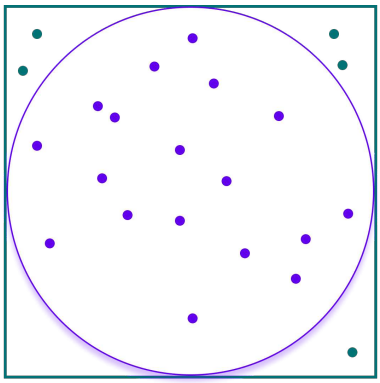
$$I = (b - a) \cdot \text{average}\,(f(x), a, b)$$

Definicija 1.6.1 (Monte Carlo integracija)**.** *Monte Carlo integracija je numerička metoda računanja vrijednosti integrala. U toj metodi koriste se nasumični slučajni brojevi kako bi se aprosksimirao integral.*

Monte Carlo integracija se sastoji od nekoliko koraka:

1. *definiraj generator slučajnih vrijednosti koji generira vrijednosti unutar intervala* $[a, b]$
2. *za neku gornju granicu N, generiraj N vrijednosti pomoću generatora*

3. *za svaku generiranu vrijednost* x_{rand}*, izračunaj* $f(x_{rand})$
4. *izračunaj prosječnu vrijednost od svih N izračunatih vrijdnosti od* f*:* $f_N := (f(x_1) + f(x_1) + ... + f(x_N)) \cdot \frac{1}{N}$
5. *aproksimacija integrala je onda jednaka* $(b - a) \cdot f_N$



Slika 1.9: Aproksimacija PI pomoću generiranja nasumičnih točaka

Primjer 1.6.2 (Primjer Monte Carlo integracije - aproksimacija vrijednosti π)**.** *Pretpostavimo da imamo kružnicu s radijusom* $r = 1$ *omeđenu kvadratom s stranicama duljine* $2r$*. Sada uzimamo nasumično točke u kvadratu i računamo koliko je točaka u kružnici. Dolazimo do zaključka da omjer broja točaka unutar kružnice i broja točaka unutar kvadrata je približno jednak sljedećem:*

$$\lim_{N\to\infty} \frac{N_{in}}{N_{out}} = \frac{\pi}{4} = \frac{\pi r^2}{(2r)^2} = \frac{P_{krug}}{P_{pravokutnik}}$$

gdje je N_{in} *broj točaka unutar kružnice, a* N_{out} *broj točaka unutar kvadrata. Ako se jednadžba pomnoži s 4, zapravo smo dobili metodu aproksimacije* π *pomoću generiranja nasumičnih točaka.*

Pokušajmo sad to napraviti u programu. Koristit ćemo funkciju *random_double*$(-1, 1)$ koja vraća nasumično generirani broj između -1 i 1.

Potom ćemo provjeriti je li točka unutar kružnice ili ne:

$$x^2 + y^2 \leq 1$$

Ako jest unutar kružnice, povećavamo brojač *inside_circle* za 1.

Isječak kôda 1.14: Aproksimacija vrijednosti π [29]

```
int main() {
    int inside_circle = 0;
    int runs = 0;
    std::cout << std::fixed << std::setprecision(12);
    while (true) {
        runs++;
        auto x = random_double(-1, 1);
        auto y = random_double(-1, 1);
        if (x*x + y*y < 1)
            inside_circle++;

        if (runs % 100000 == 0)
            std::cout << 4*double(inside_circle) / runs;
    }
}
```

Program vraća aproksimaciju broja π, gdje broj generiranih točaka određuju točnost aproksimacije.

Primjer 1.6.3 (Integracija x² na (0, 2))**.**

$$\int_0^2 x^2 dx = \frac{1}{3}x^3\Big|_0^2 = \frac{8}{3}$$

Možemo ga zapisati kao:

$$I = 2 \cdot average\left(x^2, 0, 2\right)$$

što sugerira da možemo koristiti Monte Carlo integraciju.

Isječak kôda 1.15: Integracija x² na (0, 2)

```
int main() {
    int N = 1000000;
    auto sum = 0.0;
    for (int i = 0; i < N; i++) {
        auto x = random_double(0, 2);
        sum += x*x;
    }
    std::cout << "I= 2 *" << sum/N;
}
```

Dobili smo rezultat $I = 2 \cdot 1.\dot{3}$ što je vrlo blizu točnog rezultata $I = \frac{8}{3}$.

Taj postupak možemo koristiti i za funkcije koje ne možemo analitički integrirati kao npr $log(sin(x))$.

Funkcije gustoće vjerojatnosti (PDF)

Definicija 1.6.4 (Slučajna varijabla)**.** ***Slučajna varijabla*** *je varijabla čija je vrijednost definirana nekim slučajnim procesom, ili u računarskom kontekstu, generatorom slučajnih vrijednosti.*

Kontinuirana slučajna varijabla *je slučajna varijabla koja može poprimiti beskonačno vrijednosti iz nekog intervala. Razlikuje se od* ***diskretne slučajne varijable*** *u tome što diskretne varijable poprimaju vrijednosti samo iz nekog skupa diskretnih vrijednosti.*

Primjeri kontinuiranih slučanih varijabli uključuju: visine osoba, težina, temperatura prostorije, vrijeme dolaska autobusa itd.

Primjeri diskretnih slučajnih varijabli uključuju: broj karata u špilu, rezultat bacanja kocke ili novčića, broj posjeta web stranici, broj učenika u učionici itd.

Definicija 1.6.5 (Funkcija gustoće vjerojatnosti (PDF))**.** *Funkcija gustoće vjerojatnosti je (u većini slučajeva glatka) funkcija koja opisuje vjerojatnost neke kontinuirane slučajne varijable. Koristi se da se izračuna vjerojatnost da kontinuirana slučajna varijabla poprima neku određenu vrijednost ili vjerojatnost da varijabla spada u neki podinterval vrijednosti.*

Primjer 1.6.6. *Pretpostavimo da $\mathcal{X}$ slučajna varijabla koja opisuje visinu ljudi. U tom slučaju, PDF od $\mathcal{X}$ bi opisivala vjerojatnosti za neku konkretnu visinu. Na primjer iz evaluacija $PDF(190) = 0.0001$, $PDF(180) = 0.002$ da se pročitati da je mala vjerojatnost da su osobe visoke 190, ali puno veća da su visoke 180.*

Napomena 1.6.7. *Općenito se PDF ne koristi kako bi se dobila vjerojatnost da slučajna varijabla bude jednaka nekoj konkretnoj vrijednosti nego se umjesto toga koristi interval. Na primjer, ako slučajna varijabla $\mathcal{X}$ opisuje udaljenost do posla, i ako pokušamo ispitati koja je vjerojatnost da je $\mathcal{X} = y$ metara, $y > 0, y \in \mathbb{R}$, dobit ćemo da je vjerojatnost jednaka* 0 *jer postoji neprebrojivo mnogo vrijednosti koje bi $\mathcal{X}$ mogao poprimiti. Umjesto toga, može se ispitati sljedeće: $y < \mathcal{X} < z, 0 <= y < z, y, z \in \mathbb{R}$*

Motivacija

Pretpostavimo da želimo generirati nasumično vrijednost x u intervalu $[a, b]$ s vjerojatnošću koja je proporcionalna x, tj. u $\frac{x}{N}$ slučajeva želimo da generirana vrijednost bude x.

Definicija 1.6.8 (Kumulativna funkcija vjerojatnosti)**.** *Kumulativna funkcija vjerojatnosti (CDF) je fukcija koja vraća vjerojatnost da je neka slučajna varijabla $\mathcal{X}$ manja ili jednaka od neke fiksne vrijednosti. Matematički, CDF je integral od funkcije gustoće vjerojatnosti:*

$$CDF(x) = \int_{-\infty}^{x} PDF(t)dt$$

gdje je $x \in \mathbb{R}$ ta fiksna vrijednost.

Konstrukcija PDF-a za intezitet svjetlosti

Kako bi u našoj sceni simulirali realističan intezitet svjetlosti u nekoj točki (označiti ćemo taj intezitet slučajnom varijablom r u domeni $[0, 2]$) konstruirat ćemo PDF $p(r)$ koji će biti proporcionalan samom r, dakle $p(r)$ će linearno ovisiti o r. Tom konstrukcijom ćemo definirati realističnu distribuciju svjetlosti koja dolazi iz nekog izvora. $p(r)$ će reprezentirati vjerojatnost da izvor s intezitetom r uzimamo u obzir pri osvjetljavanju točke. Očito je da će izvori jačeg inteziteta biti češće odabrani.

Primjer 1.6.9 (Konstrukcija PDF-a)**.** *Pretpostavimo da imamo slučajnu varijablu r. Vjerojatnost da r poprima vrijednost iz domene $[x_0, x_1]$ jednaka je površini ispod grafa PDF-a $p(r)$ između x_0 i x_1. Također smo za uvjet uzeli da $p(r)$ mora biti linearan r, tj. $p(r) := Cr$, za neki $C \in \mathbb{R}$. To zapisujemo kao:*

$$Vjerojatnost\ x_0 < r < x_1 := area(p(r), x_0, x_1)$$

Pošto PDF ispod cijele domene mora biti jednak 1*, vrijedi:*

$$area(p(r), 0, 2) = 1$$

Kada to raspišemo:

$$area(Cr, 0, 2) = \int_0^2 Crdr = \frac{Cr^2}{2}\Big|_{r=0}^{r=2} = \frac{C \cdot 2^2}{2} - \frac{C \cdot 0^2}{2} = 2C$$

dobivamo konačno riješenje $p(r) = r/2$.

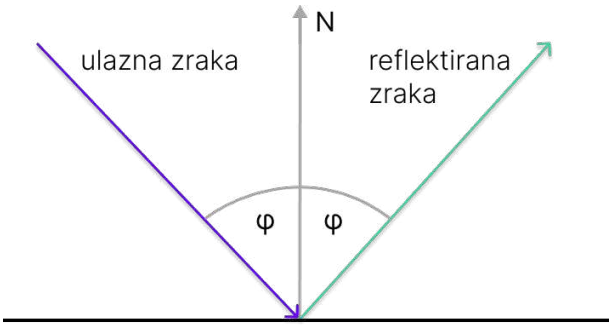
Tema se nastavlja u odjeljku 5.3, gdje se obrađuje korištenje Monte Carlo integracije u svrhu poboljšanja slike u našim algoritmima.

1.7 Vrste refleksije

U računalnoj grafici, pod refleksijom se misli na simuliranje svjetlosti koja se odbija od površina. Dvije su bitne klasifikacije refleksije: točkasta refleksija i difuzna refleksija. Bitna je činjenica da difuzna refleksija daje jednaku boju neovisno o kutu kamere.

U računalnoj grafici, te dvije vrste refleksije simulirat ćemo kroz par poznatih algoritama kao što su Lambertian, Phong i ostali, svaki od njih ima svoje pogodnosti i nedostatke. [22] [3]

Točkasta refleksija



Slika 1.10: Točkasta refleksija

Točkasta (eng. specular) refleksija je vrsta refleksije koja se događa kada se svjetlost reflektira od površine pod otprilike istim kutem kao i upadna zraka, generira tzv. zrcalnu zraku [3]. Kao rezultat dobivamo na tom dijelu slike fokusirani snop svjetlosti.

Simulirati ćemo difuznu refleksiju kroz jedan primjer. Generirati ćemo zraku svjetlosti i pratiti ju kroz scenu dok zraka presjeca neku površinu.

Primjer 1.7.1 (Simulacija zrake koja točkasto reflektira od površine)**.** *Pretpostavimo da imamo zraku* $R(t) = O + tD$ *s ishodištem* $O \in \mathbb{R}^3$ *i smjerom kretanja* $D \in \mathbb{R}^3$.

Pretpostavimo da zraka udari u neku površinu sfere,trokuta ili ravnine u točki $R_{in}(t_0) =: x_0 \in \mathbb{R}^3$. *Već smo pokazali kako naći normalu tom slučaju u 1.4. Neka je* $N \in \mathbb{R}^3$ *normala*

te površine u točki x_0.

Cilj je generirati novu zraku. To ćemo učiniti korištenjem malo linearne algebre:

$$R_{out} = R_{in} - 2 \cdot (R_{in} \cdot N) \cdot N$$

Ovdje skalarni produkt dva vektora, ulaznog vektora i normale, je skalar koji je jednak kosinusu kuta između ta dva vektora. On se množi s 2 da bi se kut udvostručio, koji se množi s vektorom normale da bi se dobio vektor koji ima duljinu jednaku gore navedenom dvostrukom kutu (smjer ima isti kao normala). Sada samo oduzmemo to od ulaznog vektora da dobijemo izlazni vektor.

Definicija 1.7.2 (Točkasta refleksija)**.** *Intezitet točkaste refleksije se računa na sljedeći način:*

$$I_{specular} = \rho_s \cdot I \cdot (R \cdot V)^p$$

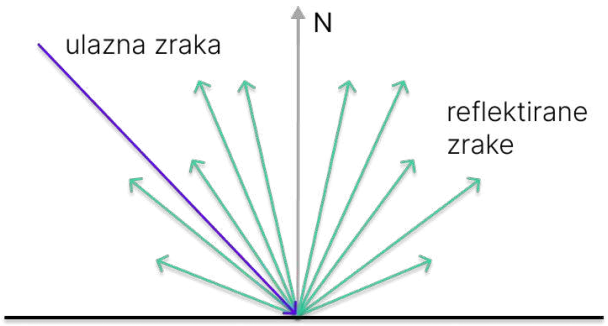
gdje je

- $I_{specular}$ *intezitet točkaste refleksije*
- ρ_s *je koeficijent točkaste refleksije (subjektivno, povećava sjaj materijala)*
- I *intezitet upadajuće zrake svijetla*
- R *zrcalni vektor dobiven u gornjem primjeru*
- V *vektor smjera zrake kamere koji gleda prema kameri*
- p *je točkasti eksponent koji određuje koliko je snop svjetlosti fokusiran u točki*

U algoritmu ćemo simulirati zrcaljenje zrake tako da definiramo funckiju koja će za smjer ulazne zrake v i normalu n izračunati zrcalnu zraku i vratiti ju kao rezultat[27]:

Isječak kôda 1.16: Refleksija zrake

```
vec3 reflect(const vec3& v, const vec3& n) {
    return v - 2*dot(v, n)*n;
}
```



Slika 1.11: Difuzna refleksija

Difuzna refleksija

Difuzna refleksija je svjetlost se reflektira podjednako u svim smjerovima od plohe. To je dominantan način osvjetljenja za plohe koje nisu sjajne. [3]. Svjetlost se odbije od površine u slučajnim smjerovima, te kao rezultat dobivamo teksturu površine sličnu *'mat'* teksturi. [22]

Definicija 1.7.3 (Lambertov (kosinusni) zakon)**.** *Zakon kaže da je refleksija na površini proporcionalna kosinusu kuta θ između normale N u točki na površini i smjera ulaska svjetlosti L koji je orijentiran prema izvoru svjetlosti:*

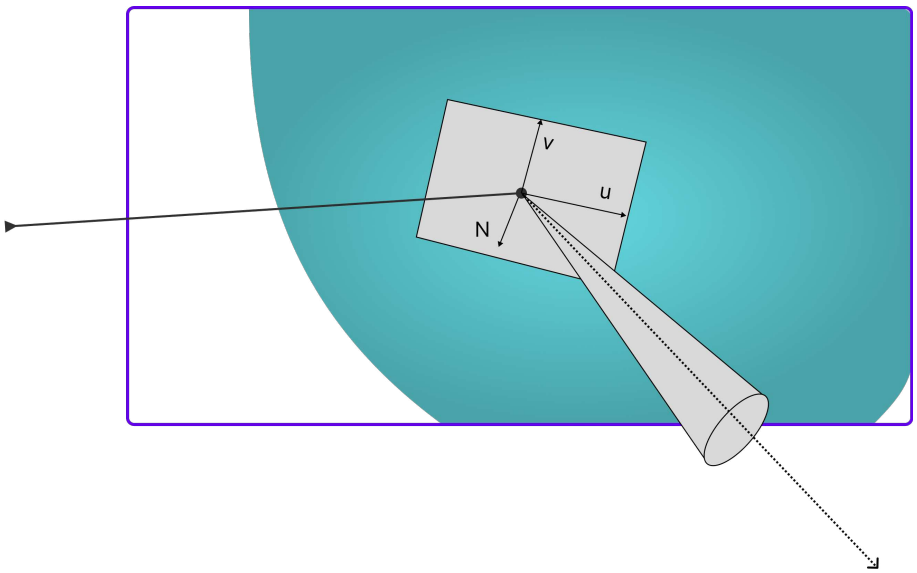
$$I_{diffuse} \propto k \cdot cos(\theta) \tag{1.18}$$
$$\propto k \cdot N \cdot L \tag{1.19}$$

Definicija 1.7.4. *Difuzna refleksija se može opisati na sljedeći način:*

$$I_{diffuse} = \rho_d \cdot I \cdot \cos(\theta)$$

gdje je

- $I_{diffuse}$ *intezitet difuzne refleksije u točki*
- ρ_d *je koeficijent difuzne refleksije (označava omjer svjetlosti koji se odbio difuzno)*
- I *je intenzitet ulaznog svijetla,*
- $cos(\theta)$ *je kosinus kuta između normale N u točki na površini i kuta pod kojim ulazi svjetlo L*



Slika 1.12: Difuzna refleksija: prilikom simulacije, nećemo uzimati u obzir sve smjerove za izlaznu zraku već samo neke koji su malo pomaknuti (”offset” u kodu) od savršeno reflektiranog smjera izlazne zrake (ako uzmemo veliki offset, očito je da se dobije rezultat s slike 1.11)

Kako bi simulirali difuznu refleksiju u algoritmu, koristi ćemo gornju funkciju ali ćemo rezultat malo ”pomaknuti”:

Isječak kôda 1.17: Difuzna refleksija zrake

```
vec3 reflect_diffuse(const vec3& v, const vec3& n, double
↪   offset) {
    vec3 reflected = reflect(v, n);
    return reflected + offset*random_in_unit_sphere());
}
```

Vidimo da kada generiramo smjer reflektirane zrake, zbrojimo mu neki nasumični vektor jedinične sfere koji je pomnožen s nekim malim pomakom. Time zapravo simuliramo difuzno odbijanje svjetlosne zrake.

1.8 Refrakcija

Refrakcija (ili lom svjetlosti, eng. *refraction*) je fenomen koji se javlja kada svjetlost prolazi kroz materijale različitih gustoća, tj. s različitim indeksima refrakcije, zbog činjenice

da se brzina svjetlosti mijenja u različitim materijalima. Primjer toga je kada svjetlost iz zraka prelazi u vodu, te se na granici s vodom smjer svjetlosti mijenja.

Intezitet refrakcije ovisi o indeksima refrakcije oba materijala, te o kutu upadne zrake. Kako bismo u računalnoj grafici mogli simulirati jednu takvu zraku, moramo izračunati njen novi smjer nakon refrakcije s drugim materijalom. U tome nam može pomoći Snellov zakon. [22]

Snellov zakon

Snellov zakon je formula koja opisuje odnos između kuta ulazne zrake i indeksa refrakcije. Zakon je nazvan po nizozemskom astronomu i matematičaru Willebrordu Snelliusu.

Teorem 1.8.1 (Snellov zakon)**.** *Zakon kaže da za dani par materijala, omjer sinusa kuta upada θ_2 (eng. angle of incidence, kut između smjera upadne zrake i normale) i sinusa refraktiranog kuta θ_1 (eng. angle of refraction, kut između smjera refraktirane zrake i normale) je jednak omjeru indeska refrakcije dva materijala, n_1 i n_2.*

$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{n_1}{n_2} \tag{1.20}$$

Primjer 1.8.2 (Simulacija refrakcije)**.** *Za potrebe algoritma praćenja zraka, simulirati ćemo generiranje refrakcijske zrake kroz neki par materijala. U izvodu se koristi Euklidska norma 1.0.5.*

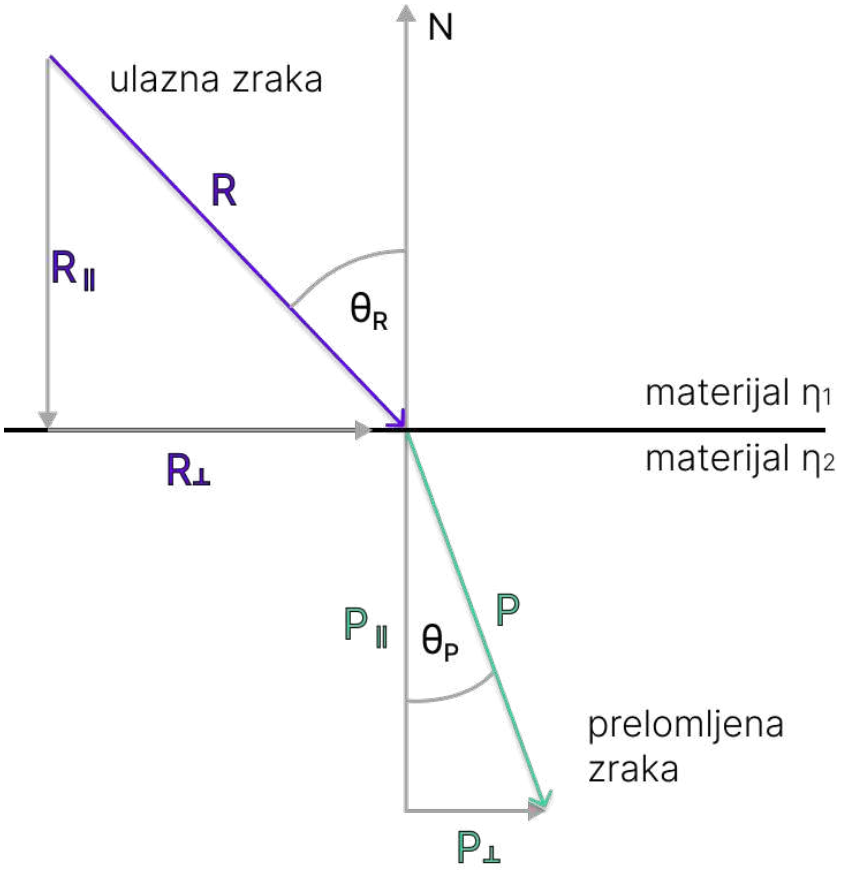
Pretpostavimo da imamo smjer zrake R koja udara u granicu između materijala u nekoj točki x_0. Također pretpostavimo da je R ***normaliziran****, tj. $\|R\| = 1$. Za normalu N u točki presjeka također vrijedi $\|N\| = 1$. Smjer refraktirane zrake P ćemo promatrati kroz dvije komponente, jedna koja je paralelna normali i druga koja joj je okomita:*

$$P = P_\perp + P_\parallel$$

Ustanovimo prvo da vrijede jednostavni trigonometrijski identiteti: $\sin\theta_R = \|R_\perp\|$ *i* $\sin\theta_P = \|P_\perp\|$

Tada vrijedi sljedeće:

$$\begin{aligned}
\|P_\perp\| &= \sin\theta_P \\
&= \frac{\sin\theta_P}{\sin\theta_R} \cdot \sin\theta_R \\
&= (1.20) \\
&= \frac{n_R}{n_P}\|R_\perp\|
\end{aligned}$$



Slika 1.13: Snellov zakon

Treba nam još jedan identitet:

$$R_\perp = R + ||R_\parallel|| \cdot N = R + \cos\theta_R \cdot ||R|| \cdot N = R + \cos\theta_R \cdot N \tag{1.21}$$

Pošto je $R_\perp$ paralelan s $P_\perp$ vrijedi i (1.21) možemo zapisati:

$$P_\perp = \frac{n_R}{n_P}(R + \cos\theta_R \cdot N)$$

Preostalo je izračunati paralelnu komponentu $P_\parallel$, za to koristimo činjenicu da $||P|| = 1$:

$$P_\parallel = -||P_\parallel|| \cdot N = -\sqrt{1 - ||P_\perp||^2} \cdot N$$

Jednom kada konačno imamo P možemo znatno pojednostaviti formulu:

$$\begin{aligned}
P &= P_{\perp} + P_{\parallel} \\
&= \frac{n_R}{n_P}(R + \cos\theta_R \cdot N) - \sqrt{1 - \|P_{\perp}\|^2} \cdot N \\
&= \frac{n_R}{n_P}R + \left(\frac{n_R}{n_P}\cos\theta_R - \sqrt{1 - \|P_{\perp}\|^2}\right)N \\
&= \frac{n_R}{n_P}R + \left(\frac{n_R}{n_P}\cos\theta_R - \sqrt{1 - \sin^2\theta_P}\right)N \\
&= (1.20) \\
&= \frac{n_R}{n_P}R + \left(\frac{n_R}{n_P}\cos\theta_R - \sqrt{1 - \left(\frac{n_R}{n_P}\right)^2(1 - \cos^2\theta_R)}\right)N
\end{aligned}$$

U zadnjem koraku ćemo primijeniti sljedeće, jer je računanje kosinusa skupa operacija:

$$\cos\theta_R = -\frac{R \cdot N}{\|R\| \cdot \|N\|}$$

Detalji oko trigonometrijskog dijela dokaza se mogu naći u [27], te u [5] gdje je dana detaljna razrada.

Za kraj, kako bi koristili gornji koncept u algoritmu, definiramo funkciju[27] koja će za ulaznu zraku izračunati novi smjer i generirati refraktiranu zraku:

Isječak kôda 1.18: Generiranje nove refraktirane zrake

```
vec3 refract(
        const glm::vec3& R, //jediničan v. smjera ulazne zrake
        const glm::vec3& N,  //jedinična normala
        double n2_over_n1) {
    auto cos_theta = dot(-R, N);
    //Okomita komponenta
    glm::vec3 r_out_perp =  n2_over_n1 * (R + cos_theta*N);
    //Paralelna komponenta
    glm::vec3 r_out_parallel = -sqrt(fabs(1.0 -
    ↪  r_out_perp.length_squared())) * N;

    return r_out_perp + r_out_parallel;
}
```

1.9 BxRF funckije

Uvod u generalizaciju: BxRF funckije

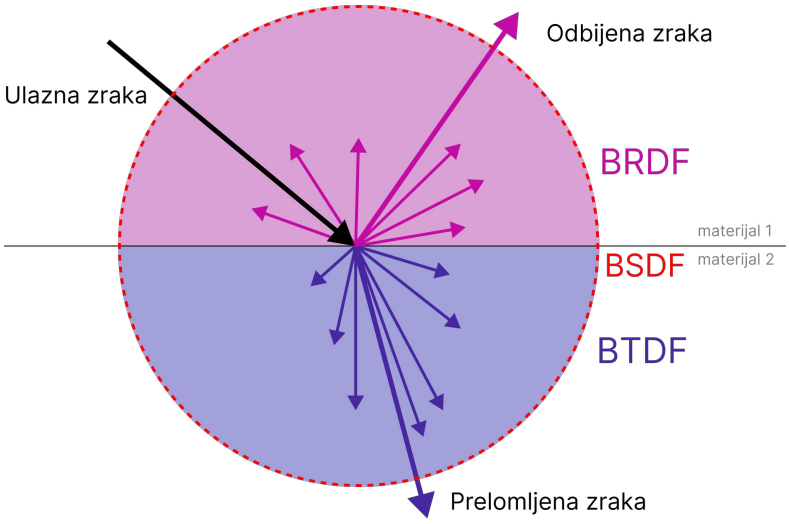
BSDF, BRDF i BTDF su akronimi koji se odnose na matematičke funkcije koje se koriste u polju računalne grafike. [22]

Definicija 1.9.1. *BSDF je kratica za "bidirectional scattering distribution function". To je funkcija koja opisuje kako se svjetlost raspršuje od površina u sceni, i također je nadskup od BRDF i BTDF.*

Definicija 1.9.2. *BRDF je kratica za "bidirectional reflectance distribution function". To je funkcija koja opisuje kako se svjetlost reflektira od površina u sceni.*

Definicija 1.9.3. *BTDF je kratica za "bidirectional transmission distribution function". To je funkcija koja opisuje kako svjetlost propagira kroz polu-prozirni materijal.*

BSDF, BRDF i BTDF se u kontekstu računalne grafike koriste da bi se egzaktno simulirala svjetlost i njena reakcija s različitim površinama u sceni.



Slika 1.14: Ilustracija svjetlosti koja reagira s materijalom, i BxDF funkcije BRDF i BTDF koje opisuju pojedinu komponentu te interakcije

Primjeri BRDFa: Phong model

Phong model računa lokalnu iluminaciju u nekoj točki materijala, autor modela je Bui Tuong Phong. U literaturi se ponekad zove Phong *sjenčar* što je nezgodno jer sjenčari imaju drukčiju definiciju u kontekstu računalne grafike [32].

Phong model u točki, tj. intezitet svjetlosti u točki koju Phong računa može se dobiti po formuli:

$$I_{\text{Phong}} = I_a\rho_a + I\rho_d(N \cdot L) + I\rho_s(V \cdot R)^p$$

gdje je:

- I_{Phong} intezitet reflektiranog svjetla u točki presjeka
- I_a je količina ambijentnog svjetla
- ρ_a ambijentni koeficijent (količina amb. svjetla koji se reflektira)
- N vektor normale u točki presjeka
- L vektor smjera izvora svjetlosti
- I intezitet izvora svjetlosti
- ρ_d difuzni koeficijent (količina difuznog svjetla koji se reflektira)
- V vektor smjera kamere/piksela
- R reflektirani vektor izvora svjetlosti
- ρ_s točkasti koeficijent (količina točkastog svjetla koje se reflektira)
- p točkasti eksponent (kontrolira sjaj materijala)

Primjer BRDFa: ostali modeli

Neki dodatni modeli slični Phongu su Blinn-Phong, Cook-Torrance i Whitted model. Blinn-Phong je najsličniji Phong modelu što se tiče rezultata i vremenske kompleksnosti, Cook-Torrance je je više realističan od predhodnika ali pod cijelu kompleksnosti dok Whitted model, koji je najkompleksniji od spomenutih, koristi u računu globalne informacije o sceni, što ga čini najtočnijim, ali je zato jako vremenski skup. Popis ostalih modela s detaljnijom međusobnom usporedbom se može naći u članku [10].

Poglavlje 2

Pregled kroz algoritme za globalno osvjetljenje baziranih na praćenju zrake

Definicija 2.0.1 (Globalno osvijetljenje)**.** *Globalno osvijetljenje (Global illumination ili skraćeno GI) je indirektno osvijetljenje u 3D sceni. To je skup algoritama koji se koriste u računalnoj grafici koji se koriste kako bi se u sliku dodao bolji realizam i osvijetljenje. Jedinstveni su po tome što uzimaju u obzir ne samo izravnu svijetlost iz izvora svijetlosti nego i one zrake koje su se reflektirale od neke druge površine.*

Slike koje se dobiju pomoću tih algoritama su puno više fotorealistične nego one koje se dobiju konvencionalnim metodama, kao rasterizacijom. Mana je u tome što su ti algoritmi puno veće složenosti te je generacija slika puno sporija. [22] [16]

Neki od algoritama za globalno osvijetljenje:

- Radiosity
- Ray tracing
- Path tracing
- Ambient occlusion
- Photon mapping

Neke od metoda se u praksi koriste i zajedno da bi se dobije fotorealistične slike visoke točnosti. Ovi algoritmi modeliraju difuznu inter-refleksiju (naknadna odbijanja u sceni nakon odbijanja primarne zrake) koja je jako bitna za globalno osvijetljenje, te oni koje ćemo mi prolaziti (bazirane na praćenju zrake) također modeliraju i točkastu refleksiju, koja je potrebna da bi algoritam bio visoke točnosti kod rješavanja svjetlosne jednadžbe, te time da u završnici daje što realističniji prikaz scene (u članku [12] su dane malo detaljnije usporedbe među modelima).

2.1 Uvod: sjenčari

Napomena 2.1.1. *Sjenčar je dvosmislen pojam, u jednom kontekstu može označavati svjetlosni algoritam koji "sjenča" neku točku, dok se u kontekstu OpenGL-a koji ćemo kasnije obraditi, označavaju specifične programe koji se izvršavaju na grafičkoj kartici. U ovom uvodnom poglavlju opisujemo koncept sjenčara kojim možemo pokriti oba konteksta.*

Definicija 2.1.2. *Sjenčar (eng.shader) je mali program koji definira kako će se realistična slika prikazati tokom izvođenja računalnog grafičkog programa. Oni definiraju kako se boja, teksture, svjetlost, sjene i ostali parametri primjenjuju na objekte u 3D sceni tokom izvođenja programa. Sjenčari se zbog svoje prirode izvršavaju na grafičkim karticama, jer je tamo moguće izvršiti paralelno velik broj tih programa.*

Mi koristimo sjenčare u kontekstu praćenja zraka da definiramo kako će se svjetlost kretati u sceni jednom kad presjeca neki 3D objekt. Koristimo ih da simuliramo refleksiju, refrakciju i sjene u sceni. Sjenčari se općenito definiraju za specifični piksel, te u kontekstu tog piksela rade nužne kalkulacije. Možemo o njima razmišljati kao o skupu instrukcija, ali te instrukcije se izvršavaju sve odjednom za svaki piksel na ekranu. To znači da će **svako izvršavane istog sjenčara biti drukčije ovisno o pikselu nad kojim se kôd sjenčara izvršava**.

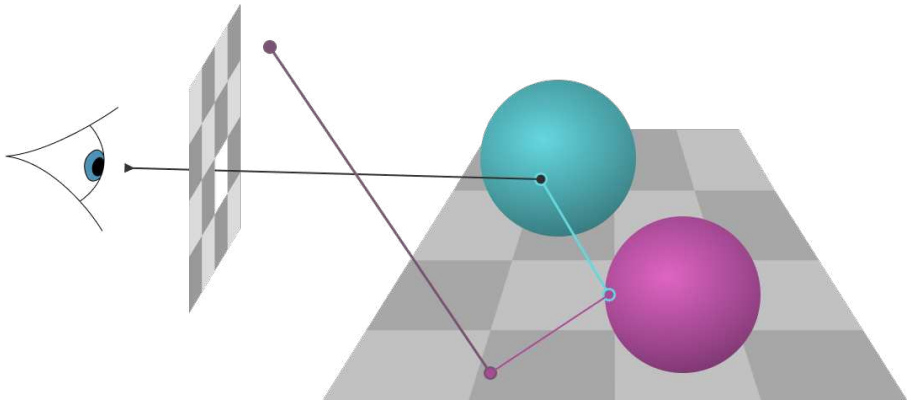
Sjenčari u praksi kroz paralelizam

Pošto su sjenčari mali programi koji su obavljaju istu funkciju za svaki piksel, dosta ih je jednostavno paralelizirati. Taj posao radi grafička jedinica računala (GPU). GPU je uređaj dizajniran da obavlja puno istih manjih poslova odjednom. Možemo zamisliti GPU kao *nxn* matricu s cijevima kroz koje teku podaci. Pretpostavnimo da imamo kao ulaz 3D scenu koja se sastoji od obične sfere. Naš sjenčar s ulazom *i*, *j* (koordinate piksela) i kopijom te scene ima jednostavan kôd: *ako je zraka koja prolazi kroz piksel i, j u presjeku s sferom, oboji i, j piksel u crno, inače vrati bijelu.* Na izlazu dobivamo *nxn* dvodimenzionalnu sliku scene. Konkretniji primjer se može naći u poglavlju 3.1 gdje će se ići malo dublje u priču kako sjenčari točno rade.

Isječak kôda 2.1: Pseudokôd sjenčara

```
color obojaj(int i, int j, Scene& scena) {
    for(int k; k < scena.objects.length(); k++)
            if(inCollisionWith(i, j, scena.objects[i])
                return (0, 0, 0);
    return (1, 1, 1);
}
```

2.2 Ray tracing (praćenje zrake)



Slika 2.1: Ray tracing algoritam: prikaz zrake koja kroz piksel na ekranu prolazi scenom i odbija se od objekata, time generirajući nove zrake

Algoritam praćenja zrake (*eng. Ray Tracing*) je algoritam za globalno osvijetljenje koji je baziran na praćenju zrake. Algoritam prati zraku kroz scenu te na svaki presjek zrake sa objektom računa boju piksela. Potom se računaju nove zrake s tim presjekom kao ishodištem i ponovno se računa boja idućeg presjeka s drugim objektom.

Uvod i motivacija

Od početaka, grafika je težila prema sve boljem realizmu slike. I najraniji alati za sjenčanje sadržavali su algoritme koji bi računali točkastu refleksiju, sjene i prozirnost materijala. Zadatak svakog iluminacijskog modela jest da odredi koliko svjetlosti je reflektirano u kameru (motritelja) s neke vidljive točke objekta, tako da se definira funkcija koja kao argumente prima smjerove svjetlosnih izvora, jačinu izvora, orijentaciju objekta, poziciju kamere te ostala svojstva tog objekta i njegove teksture i materijala, itd.

Prema Whittedu, (koji je prvi otkrio ovu metodu u [32]), kako bi egzaktno prikazali 3D scenu na ekran, u koraku kolizije zrake s nekim objektom (kada računamo intezitet boje piksela), podaci o globalnom osvijetljenju moraju biti dostupni. To u principu znači da se informacije pamte u nekoj vrsti stabla zraka (eng. *tree of rays*), korijen stabla je zraka koja kreće iz kamere, kroz piksel$_{ij}$ pa prema sceni. Jednom kada algoritam stvori prvu zraku koja kreće iz kamere za svaki piksel, tada to stablo zraka predaje *sjenčaru*, programu koji uzima kao ulaz jednu zraku te ju prati kroz scenu. Sjenčar tada, prolazeći stablo, tj. sve presjeke s objektima koje su zrake napravile, računa i vraća konačnu boju za taj piksel. Pomoću svih tih čimbenika, algoritam može simulirati globalno osvijetljenje, što

uključuje korektno prikazivanje sjena, refleksija, i lomova svjetlosti kroz objekte. Modus operandi svakog sjenčara, tj. razine apstrakcije u kojem može raditi su: **mikroskopski**, **lokalno** ili **globalno**. Mikroskopski, refleksije je dosta teško modelirati. Da bi se to napravilo, mora postojati jako dobar i točan model teksture materijala te kako da se svjetlost odbija od svih nepravilnosti na njemu. Većina sjenčara radi u lokalnom načinu rada, te za većinu slučajeva to iziskuje dovoljno točnim prikazom slike. Na primjer OpenGL radi na tom načinu uz pomoć rasterizacije, sjenčar dobiva samo lokalne informacije iz kojih mora izračunati korektnu boju za piksel, kao što su npr. lokacije izvora svjetlosti i orijentacija objekta . Naravno, nažalost to nije dovoljno informacija ako se želi proizvesti realistična slika 3D scene.

Dakle preostaje globalan rad sjenčara. Kod globalnog rada, sjenčar mora znati sve globalne varijable scene u svakom trenutku, što uključuje na primjer informacije o svim objektima u sceni, pozicije svih izvora svjetlosti u sceni.

Preduvjeti za implementaciju

U ovom koraku ćemo spojiti termine koje smo definirali u Poglavlju 1 kako bi izveli konkretan algoritam za praćenje zrake. Kako bi se implementirao algoritam praćenja zraka, potrebno je implementirati sljedeće algoritme:

1. **Kolizija objekata u sceni**: algoritam za koliziju je najvažniji preduvjet jer je potrebno za svaku zraku otkriti s kojim se objektima u sceni ona sudarila. Taj algoritam se zove algoritam presjeka (eng. intersection algorithm). Općenito se implementira samo za jednostavne objekte u sceni (kao što su sfera, trokut, ravnina) te za **Mesh** objekte (skup trokuta koji definiraju kompleksni objekt). Izlaz tog algoritma jest **točka presjeka zrake**. Više o implementaciji tog algoritma se može naći u odjeljku 1.5.

2. **Refleksije i refrakcije**: Ovisno o materijalu, ovaj mali algoritam bi trebao moći izračunati smjer kretanja nove zrake (ili više njih) nakon kolizije s objektom, s točkom kolizije kao novim ishodištem. Na primjer, pri koliziji s zrcalom, algoritam bi trebao dati savršeno odbijen smjer nove zrake (pod savršeno se misli na točkastu refleksiju o kojoj se može više naći u 1.7), dok na primjer za materijal koji simulira metal ili drugu hrapavu površinu, smjer kretanja bi trebao odstupati za neku slučajnu vrijednost od savršenog odbijanja (tu se misli na difuznu refleksiju, o kojoj se isto može naći više u 1.7). Algoritam bi također, korištenjem Snellovog zakona (1.7), trebao izračunati zraku refrakcije ukoliko se radi o prozirnom materijalu.

3. **Sjenčanje**: algoritam Sjenčar (eng. Shader) sjenčanja stupa na snagu jednom kada se otkrije točka presjeka. Zadatak algoritma jest izračunati boju koja će se prikazati na

ekranu za neki konkretni piksel. Taj algoritam može biti jednostavan, poput Phong ili Lambertian algoritma, ili to može biti kompleksna BRDF funkcija koja opisuje neki složeni materijal koji je teško realno prikazati koristeći se jednostavnim algoritmima sjenčanja. Dakle ti algoritmi općenito kao ulaz primaju kut između izvora svjetlosti i tangencijalne ravnine i normalu u točki presjeka, a kao izlaz vraćaju jačinu odbijene svjetlosti u zadanom smjeru. Više o tim algoritmima se može naći u sekciji o BRDF funkcijama, 1.9, te u sekciji gdje opisujemo kako sjenčari funkcioniraju 2.1, te u sekciji gdje se opisuju sjenčari u konkretnim primjerima 3.1.

4. **Pomoćni algoritmi**: Potrebno je također prilikom nalaska točke presjeka izračunati normalu te tangencijalnu ravninu, koje nam služe kao preduvijet za korištenje algoritama za sjenčanje i algoritama za računanje smjera refleksije i refrakcije. U Odjeljku 1 detaljno je opisano kako se to može izračunati.

Koraci i opis algoritma

Algoritam je kao što se može zaključiti rekurzivan, te se ponavlja sve dok se ne dođe do nekog od uvjeta zaustavljanja. Uvjeti su: **maksimalna dubina**, tj. maksimalni broj skokova i ako se zraka **ne sijeće s nijednim objektom u sceni** tada se boja postavlja na boju pozadine.[16]

Isječak kôda 2.2: Pseudokôd sjenčara

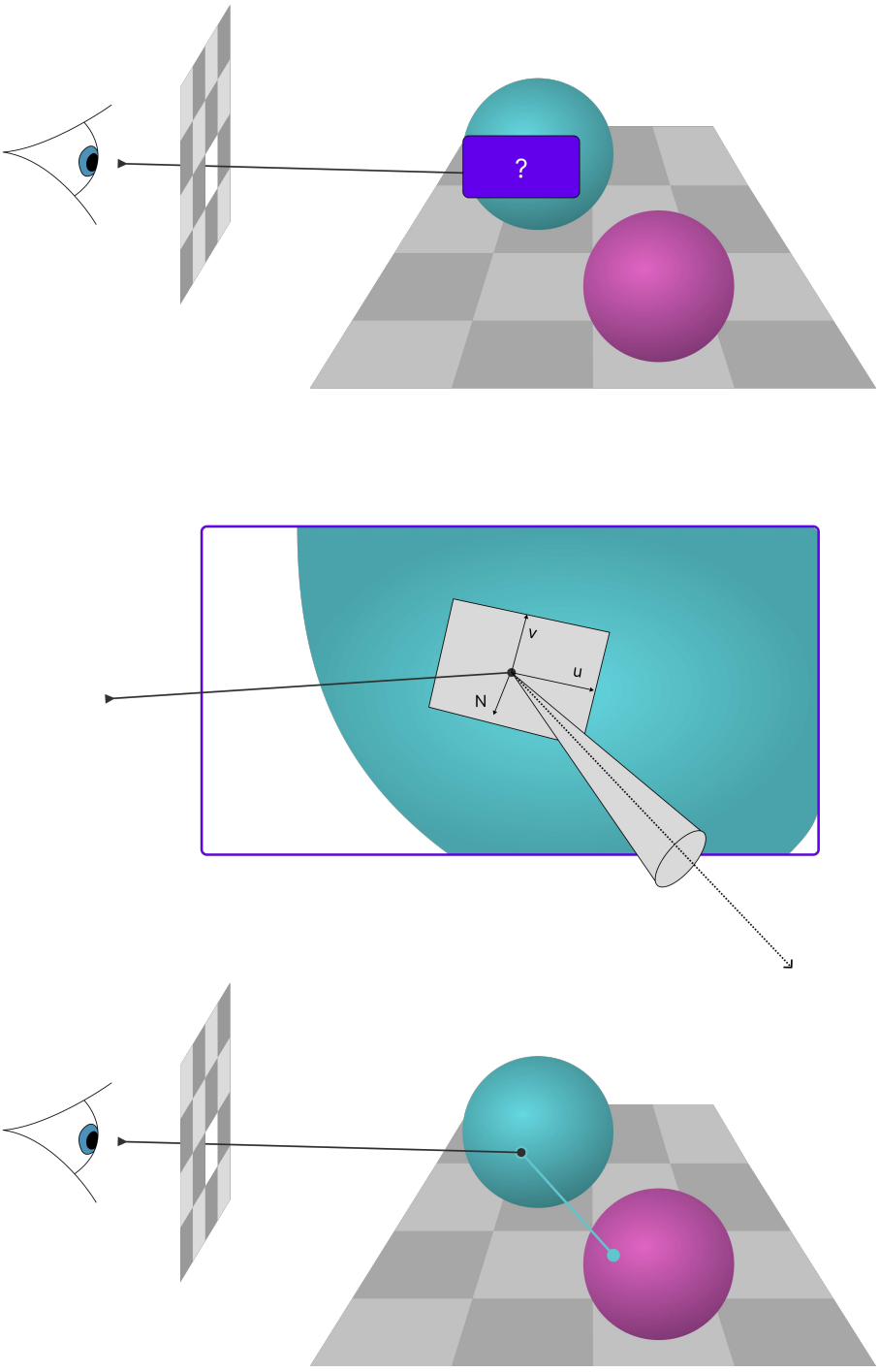
```
scena = učitajScenu();
shadingAlgo = odaberiAlgo("Phong");

pratiZraku(zraka, dubina):
  ako dubina == 0:
    vrati boju pozadine;
  za svaki objekt u scena:
    ako točka := kolizija(objekt, zraka):
      N, materijal = izračunajSvojstvaSudara(točka, objekt);
      // uz boju, shadingAlgo vraća težine f_i koje opisuju u
      ↪ kojoj mjeri
      // će Phong boja, reflektirana i refr. boja utjecati na
      ↪ konačnu boju
      boja c, f1, f2, f3 = shadingAlgo(N, materijal,
      ↪ scena.izvoriSvjetla);
      ako materijal.proziran:
        smjer s = noviSmjerRefrakcije(materijal, N, zraka);
        boja p = pratiZraku(novaZraka(p, točka), dubina - 1);
      ako materijal.refl:
        smjer s = noviSmjerRefleksije(materijal, N, zraka);
        boja r = pratiZraku(novaZraka(s, točka), dubina - 1)
      vrati c * f1 + p * f2 + r * f3
    inače:
      vrati bojaPozadine;

za svaki piksel i, j:
    zraka = novaZraka(i, j);
    slika[i][j] = pratiZraku(zraka, 10);
```

Ovaj pojednostavljeni pseudokod algoritma praćenja zrake ilustrira koji su sve algoritmi potrebni za njegov rad.

1. kolizija() odgovara klasi algoritama za kolizije u točki #1
2. noviSmjerRefrakcije() i noviSmjerRefleksije() koji računaju smjerove za nove zrake ovisno o materijalu, odgovaraju klasi algoritama za refleksije i refrakcije u točki #2
3. funkcija shadingAlgo() odgovara klasi algoritama za sjenčanje u točki #3

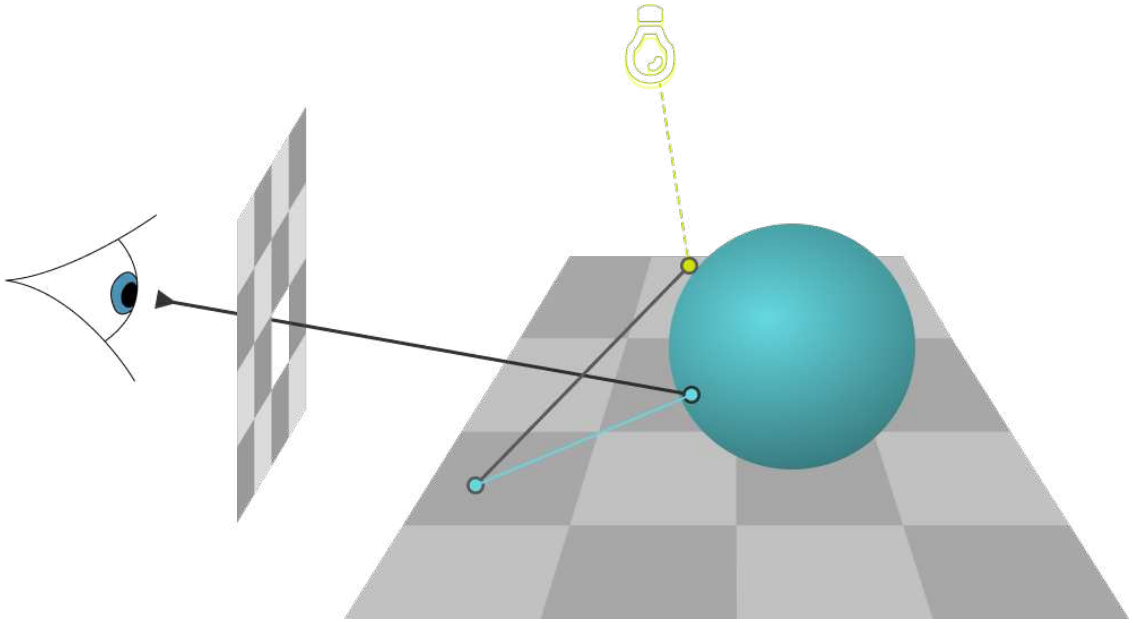
4. Klasi #4 odgovara funkcija izračunajSvojstvaSudara() koja izračunava tangencijalnu ravninu, normalu i sve ostale informacije koje su potrebne kao ulazi za ostale algoritme

Na ilustraciji sa slike 2.2 se može vidjeti kako primarna zraka udara u objekt.
Na mjestu kolizije se tada računa tangencijalna ravnina i normala, kako bi se pomoću njih izračunao novi smjer kretanja reflektirane zrake. (stožac reprezentira sve moguće smjerove reflektirane zrake). Tada ”kontrolu” preuzima opet algoritam za kolizije kako bi za našao novu koliziju za generiranu zraku.



Slika 2.2: Opis algoritma praćenja zrake

2.3 Path tracing (praćenje puta)



Slika 2.3: Path tracing

Algoritam praćenja zrake daje dobre rezultate u prikazu realističnih slika na ekranu, ali tokom godina otkriven je jedan koji rezultira puno realističnijim slikama. Naime praćenje zraka samo "simulira" svjetlost i sjene tako da gleda koji su izvori svjetlosti vidljivi u toj točki, i to je moguće jer u točki kolizije algoritam sadrži te informacije. U nastavku definiramo napredniju verziju tog algoritma.

Definicija 2.3.1 (Algoritam praćenja puta (*eng. Path Tracing*))**.** *Klasa algoritama "praćenja puta" su algoritmi koji sadrže fizičku definiciju simulacije svjetlosti. Ti algoritmi simuliraju svjetlost na "slijepi" način. U trenutku kolizije, informacije o izvorima svjelosti nisu vidljive.*

Da bi se piksel iz kojeg je primarna zraka ispaljena bio vidljiv na ekranu (tj. imao neku netrivijalnu boju), taj lanac zraka se mora u nekom trenutku sudariti s svjetlosti. Inače, piksel sadrži boju pozadine, koja je po definiciji crna.

1986. godine, James T. Kajiya objavio je članak "The rendering equation", u kojem je iznio matematičku podlogu za renderiranje 3D slike na ekran [19]. Predstavljeni model je rigorozno matematički definiran te nije dana uputa za njegovu implementaciju, stoga ćemo mi predstaviti pojednostavljeni algoritam koji se konceptualno ne razlikuje puno s onim u članku.

Praćenje puta - pregled algoritma

Isječak kôda 2.3: Pseudokôd sjenčara

```
scena = učitajScenu();

pratiZraku(zraka, dubina):
    HitInfo hitInfo;

    if (dubina <= 0)
        return background;

    if (!world.hit(zraka, hitInfo))
        return background;

    ray rasprsenaZraka;

    // materijal svjetlosnih izvora nemaju boju nego parametar
    ↪  emitted, tj. intezitet svjetlosti
    color emitted = hitInfo.mat_ptr->emitted(hitInfo);

    // ako materijal nema boju, znači da je izvor i tada vraća
    ↪  samo intezitet tog izvora
    if (!hitInfo.mat_ptr->scatter(r, hitInfo,
    ↪  hitInfo.prigusenje, rasprsenaZraka))
        return emitted;

    // inače vraća trenutnu boju i prigušene boje svih
    ↪  naknadnih odbijanja (svako naknadno odbijanje se sve
    ↪  više prigušuje)
    return emitted + hitInfo.prigusenje *
    ↪  pratiZraku(rasprsenaZraka, background, world,
    ↪  dubina-1);

za svaki piksel i, j:
    zraka = novaZraka(i, j);
    slika[i][j] = pratiZraku(zraka, 5);
```

Ovaj kôd prati zraku kroz scenu, s ishodištem u kameri. Nove zrake se generiraju prilikom refleksije i refrakcije, te se rekurzija zaustavlja jednom kada se postigne limit

rekurzije (koji također zovemo eng. maxDepth) ili ako zraka izađe iz scene. U svakom koraku, računa se točka presjeka s objektom u sceni te se koristi svojstva materijala na kojem se dogodio presjek da se izračuna potrebne informacije za generaciju boje i smjer nove zrake.

Algoritam zbraja sve boje materijala s kojim je to stablo zraka bilo u kontaktu te na kraju primarna zraka vraća konačnu boju tog piksela.

Razlika u odnosu na praćenje zrake

Glavna razlika je u tome što algoritam praćenja zrake računa količinu svjetlosti u točki presjeka tako da provjerava koji izvori su vidljivi iz dane točke sudara. ”shadingAlgo” u algoritmu praćenja zrake radi taj posao.

Algoritam praćenja puta u drugu ruku ”simulira” sjetlosne zrake, te ih nastavlja pratiti u sceni, bez konteksta gdje se svjetlosni izvori nalaze. To uvelike povećava kvalitetu slike, ali je vremenski skuplje.

Praćenje puta - nedostaci algoritma

Jedan od glavnim nedostataka algoritma jest što je jako spor. Da bi završna boja piksela bila netrivijalna, lanac zraka mora doći do svjetlosti. Ako ne dođe, taj se piksel boja bojom pozadine koja je najčešće crna. Zato su slike generirane tim algoritmima često zrnate, svi crni pikseli u slici prestavljaju lance zraka koji nisu ”stigli” do izvora svjetlosti.

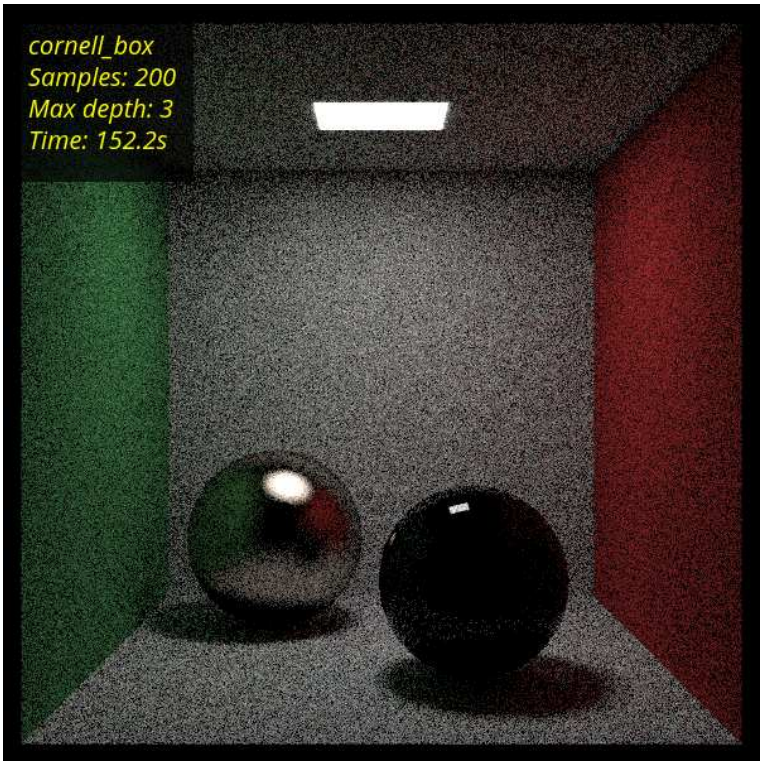
Problem se ne može riješiti u potpunosti ali se može minimizirati na sljedeće načine:

- **Zaglađivanje**: algoritam koji se koristi u grafici da bi se smanjila nazubljenost rubova u slici, odlično pridonosi rješavanju problema zrnatosti: umjesto da ispaljujemo kroz svaki piksel jednu zraku, ispaljujemo ih n s malim pomakom vektora smjera, te na kraju uzimamo prosječnu vrijednost vraćenih boja. Na taj način povećavamo šanse da će neka zraka tog piksela pogoditi izvor svjetla. Više o tome se može naći u odjeljku 5.1.

- **Monte Carlo metoda** metoda uzima uzorke zraka svjetlosti na način da u trenutku kolizije nasumice odabire točke na površinama objekata u sceni i ispaljuje nove zrake u njih te nađe prosjek svih tih zraka. Ovo je pojednostavljeno objašnjenje koncepta o kojem se može više pročitati u 5.3. On uvelike popravlja problem zrnatosti, ali uz veću cijenu vremenske složenosti.

Praćenje puta - naknadne iteracije i poboljšanja algoritma

Razvile su se mnoge iteracije algoritma nakon njegovog predstavljanja 1986. godine. Navest ćemo neke od njih i kako točno oni unaprijeđuju dani algoritam praćenja puta.



Slika 2.4: Praćenje puta: zrnatost slike je rezultat malog izvora svjetlosti i prirode algoritma
Rezultat izvršavanja: [24]

- **Monte Carlo praćenje puta**: otkrio Peter Kelemen i Steve Marschner u radu "Monte Carlo path tracing" (2001), više o tome u 5.3

- **Dvosmjerno praćenje puta (eng. Bidirectional path tracing)**: predstavljen od strane Henrika Wann Jensena, Stephena R. Marschnera, Marca Levoya, i Pata Hanrahana u radu "Bidirectional path tracing" (2001), ovaj algoritam opisuje praćenje puta, ali se istovremeno prate zrake iz izvora i iz kamere, time znatno povećavaju realost interakcija svjetlosti s materijalima u sceni, čime je povećan ralizam mekih sjena i indireknog osvijetljenja

- **Photon mapping**: predstavljen od strane Henrik Wann Jensena u radu "Global illumination using photon maps" (1996). U ovom algoritmu umjesto da se kao kod praćenja puta zraka prati iz kamere prema izvorima svjetlosti, ovdje se zrake prate u suprotnom smjeru, što uvelike popravlja realizam u slici.

- **Metropolis light transport**: predstavljen od strane Roberta L. Cooka, Kena Mus-

gravea, i Petera Shirleya u radu ”Distributed ray tracing” (1984). Algoritam pruža znatno realističnije slike jer za razliku od predhodnih algoritama, može istraživati i naći zrake koje dolaze do izvora koje je teško naći u sceni.

Poglavlje 3

Demistifikacija pojmova u radu s grafičkim i GPU bibliotekama

3.1 Uvod u rad s grafičkim bibliotekama

Grafičke procesne jednice se koriste u grafici jer mogu rukovati s ogromnim količinama podataka, kao što su podaci (točke, linije) koji opisuju komplicirani model, teksture visoke rezolucije, transformacijske matrice itd.

Jezici i okoline koji se koriste za grafičke procesne jedinice su *low-level*, tj. niske razine apstrakcije. To ustvari znači da za korištenje tih jezika i okolina je potrebno znati i nekolicinu stvari u vezi sklopovlja računala, konkretno grafičke kartice, jer niska razina jezika pruža da se direktno komunicira s tim sklopovljem, što nije slučaj s jezicima više apstrakcije kao što su na primjer Java i C#, gdje poznavanje sklopovlja na kojem program radi nije potrebno.

Što se konkretno grafičkih kartica tiče, za njihovo programiranje koriste se biblioteke i okoline koji služe kao ***posrednici*** između sklopovlja i programera. Mi zapravo ne upravljamo registrima grafičke kartice direktno, nego pomoću tih biblioteka. Postoji par biblioteka koje se koriste u te svrhe [13]:

- OpenGL: *open source* grafička biblioteka, radi na velikoj većini grafičkih kartica, razvijena je kako bi algoritmi koji su implementirani u tome mogli raditi na bilo kojoj kartici neovisno o proizvođaču
- OpenCL[1]: *open source* biblioteka opće namjene, od istog je proizvođača kao i OpenGL, nastoji pružiti iste funkcionalnosti neovisno o kartici na kojoj je implementiran program

[1] Ove biblioteke nisu *grafičke* biblioteke nego višenamjenske, takozvane GPGPU biblioteke, koriste se za generalno programiranje GPU-a, npr. za strojno učenje, ali se mogu koristiti i za grafiku

- DirectX: konkurent OpenGLa, Nvidijina grafička biblioteka, algoritmi pisani u njoj su puno efikasniji nego u OpenGLu jer je DirectX puno više specijalizirana biblioteka Nvidia grafičkih kartica
- CUDA: konkurent OpenCLa, Nvidijina biblioteka za opće programiranje grafičkih kartica, koristi se u razne svrhe, ponajviše za strojno učenje.
- Vulkan: biblioteka nastala zadnjih godina, naslijednik OpenGL-a, puno je više low level pisana i zahtjeva puno veće poznavanje sklopovlja grafičke kartice
- WebGPU
- Metal

U prošlosti GPU nisu bili programibilni, imali su skup fiksiranih low-level funkcija. OpenGL-ov dizajn je nastao u tom dobu. On je osmišljen kao stroj stanja, pa se njegovo sučelje puno razlikuje od modernijih biblioteka. S vremenom, pojavila se potreba za programibilnošću GPUa kako je potražnja od proizvođača aplikacija rasla. DirectX je puno bliži modernijim GPU arhitekturama od OpenGLa, te pokušava biti puno jednostavniji za programere tako da sam obavi dio razvojnog procesa, dok npr. Vulkan sve to stavalja u ruke programeru. U zadnjih par godina proizvođači kartica uveli su hardversku podršku za ubrzanje ray tracing algoritama te razna ubrzanja za AI računanja velikih tenzora. Nvidijine grafičke kartice klase RTX na primjer dolaze s time, te se ta spomenuta hardverska unaprijeđenja mogu koristiti kroz njihovu biblioteku DirectX, jer kako se hardware mijenjao kroz godine, tako su se i biblioteke ažurirale da budu u toku s promijenama.

Životni tok grafičke aplikacije

Aplikacija implementirana pomoću jednog od gore spomenitih biblioteka uglavnom slijedi sljedeće korake (neovisno o kojoj biblioteci je riječ):

1. Inicijalizacija biblioteke / APIja: stvaranje struktura podataka i potrebnih varijabli koji nam uopće dozvoljavaju početak korištenja APija
2. Učitavanje podataka: stvaraje struktura podataka potrebnih da se mogu učitati stvari kao što su sjenčari, učitavanje modela, tekstura i svega potrebnog što bi se nalazilo u sceni, podešavanje varijabli / buffera koji govore GPU-u koje instrukcije treba izvršiti te i samo fizičko slanje podataka na GPU
3. Ažuriranje podataka: generalna aplikacijska logika
4. Prezentacija: šalje se lista komandi u GPU te se čeka izvršavanje

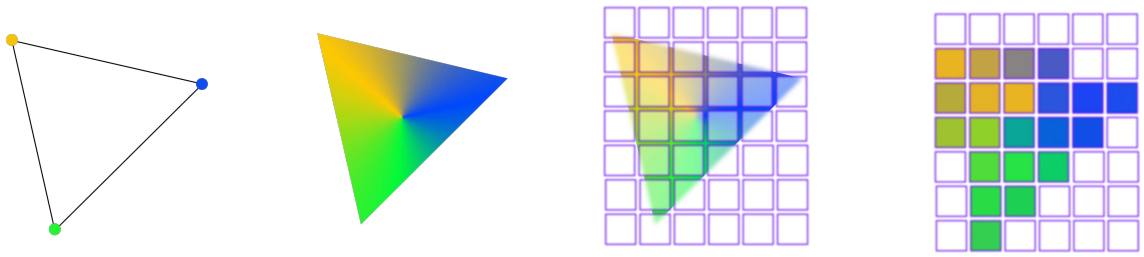
5. **Ponavljaj korake 2, 3, 4 dok aplikacija ne završi**

6. Pričekaj dok GPU ne bude gotov s radom, potom uništi sve korištene strukture

Rasterizacija

Rasterizacija je proces u računalnoj grafici u kojem se 3D geometrijski likovi pretvaraju u rasteriziranu sliku, što je u stvari 2D matrica piksela.

Tijekom rasterizacije, svaki se geometrijski lik dijeli na manje likove, ili kako se u grafici zovu, primitive. Tim se primitivima tada prolazi da se vidi koji od njih presjeca koji piksel, da bi se taj piksel obojao korektnom bojom. [3] [11].



Slika 3.1: Rasterizacija trokuta: proces prima na ulazu trokut, te se računa za svaki piksel koju boju treba poprimiti

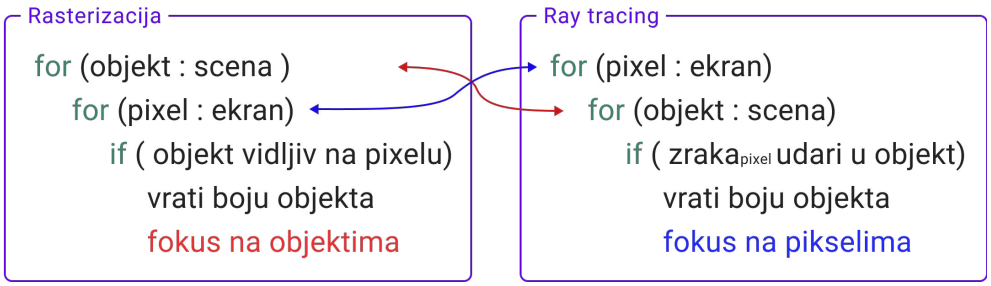
Rasterizacija se općenito pokreće na grafičkim jedinicama računala, tamo su i osmišljene da rade. Svaka se grafička jedinica može brinuti samo za piksel koji joj je dodijeljen, što čini proces jako brzim i jednostavnim.

Postoji nekoliko koraka koji čine rasterizaciju, od početnih transformacija (koje transformiraju 3D scenu na 2D ekran), pa do obrezivanja (koje uklanja sve primitive koje nisu vidljive na ekranu) i na kraju sjenčanja (procesa koji odlučuje boju piksela u ovisnosti o postavkama svjetlosti u sceni i vrsti materijala).

Nećemo ulaziti u detalje rasterizacije, ali ćemo objasniti distinktnu razliku između koncepta rasterizacije i algoritma praćenja zraka.

Proces rasterizacije za svaki objekt u sceni evaluira da li je on vidljiv u nekom fiksnom pikselu, te ovisno o tome ga uzima u obzir kod bojanja tog piksela.

Kod praćenja zraka je to drukčije, za svaki piksel se generira zraka (koju možemo zamisliti kao crnu kutiju koja samo vraća boju piksela) koja tada prolazi kroz objekte u sceni.



Slika 3.2: Razlike u algoritmima između konvencionalne rasterizacije i algoritma praćenja zraka
Izvor: [16]

Poredak kojim se petlje izvršavaju je bitan, jer više ne možemo jednostavno koristiti grafičke jedinice da naprave svoj posao za svaki piksel, pošto zraka mora cijelo vrijeme znati koji se objekti nalaze u sceni, što bi u praksi značilo da svaki od grafičkih jedinica mora to isto znati.

3.2 OpenGL grafička biblioteka

OpenGL [11] je jednostavan low-level grafički API kojeg koriste razni developeri igara, web stranica (preko WebGL-a), inženjeri koji vizualiziraju podatke i drugi. Trenutno je podržan na skoro svakoj platformi, uključujući Windows, Linux, Android i web. Na MacOSu više nije dostupan, ali zato jest njihov novi API Metal. OpenGL se može programirati u nekolicini jezika, koji uključuju C (službeni jezik), C++, Rust (preko Glium wrappera), Python (preko PyOpenGL wrappera), JavaScript (korištenjem WebGL-a).

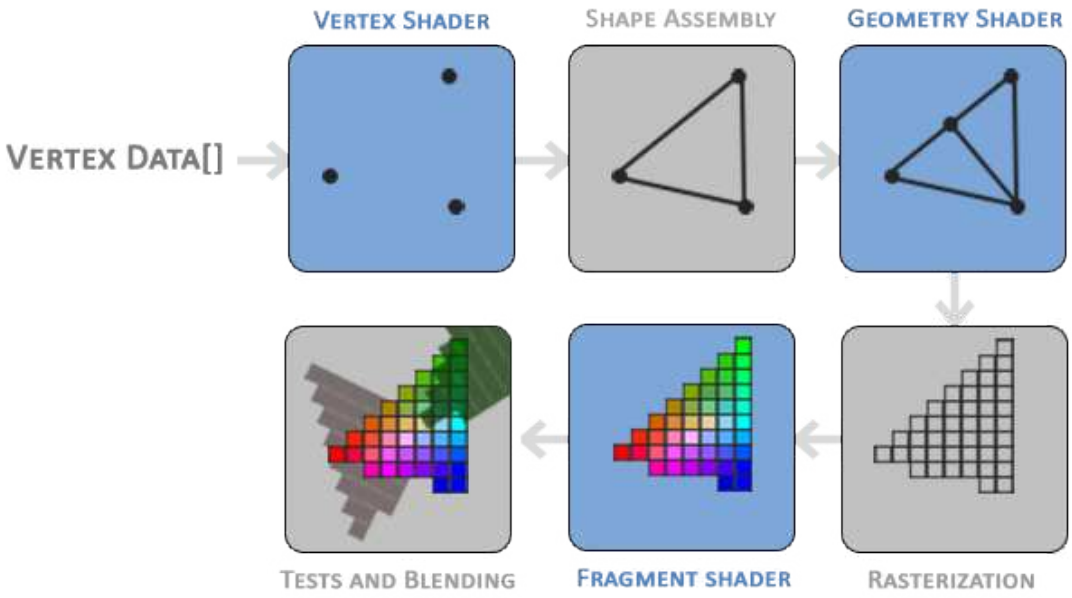
Kao što je već rečeno u prethodnom poglavlju, OpenGL je stroj stanja. Programer je dužan postaviti stanja GPUa prije pokretanja aplikacije. Preko tih stanja se komunicira s hardwareom GPU-a.

OpenGL je stari API te je kroz godine, kako se hardware mijenjao i omogućavao sve veću programibilnost, dobivao nove funkcionalnosti.

OpenGL sjenčar (eng. Shader)

Sjenčari su programi koji se izvršavaju na jezgrenim procesorima grafičke kartice. U izvršnom kodu definiramo kodove sjenčara koje predamo OpenGLu, koji se tada izvrše

na grafičkoj kartici nad podacima koje damo kao ulaz.



Slika 3.3: Protok informacija kroz sjenčare u OpenGLu: plavi sjenčari su programibilni te se u izvršnom programu mogu definirati, dok se za sive brine OpenGL. Izlaz prethodnog sjenčara je ulaz idućeg.
Izvor: [6]

OpenGL se sastoji od njih nekoliko, najbitniji su **vertex** i **fragment** sjenčari.

Na slici se vidi pojednostavljeni prikaz sjenčara. Ulazni podaci iz izvršnog programa se prosljeđuju vertex sjenčaru.

Dajemo u nastavku primjer jednog vertex sjenčara:

Isječak kôda 3.1: Primjer vertex shadera [6]

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 ourColor;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;
}
```

Posao tog sjenčara je da nad svakim vrhom koji dobije kao ulaz napravi određene transformacije te da taj vrh preda sljedećem sjenčaru. Sjenčar dobiva dvije informacije iz ulaza, poziciju i boju vrha. Sa pozicijom ništa ne radi nego ju sprema u gl_Position, gdje se po standardu sprema završna pozicija vrha nakon svih transformacija koje je sjenčar primijenio nad vrhom. Boju vrha sjenčar samo prosljeđuje u fragment sjenčar, koju će fragment sjenčar dobiti na ulazu.

Nakon toga definiramo i fragment sjenčar,

Isječak kôda 3.2: Primjer fragment shadera [6]

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;

void main()
{
    FragColor = vec4(ourColor, 1.0);
}
```

Sjenčar kao ulaz dobiva informacije o boji, sličnno kao i u vertex sjenčaru, u FragColor se po standardu upisuje konačna boja tog fragmenta.

Definicija 3.2.1. *OpenGL kontekst (eng. context) je struktura podataka koja sadrži sve informacije o stanju OpenGLa. Rekli smo da je OpenGL stroj stanja, a ta stanja se spremaju u kontekst. Sve naredbe koje naknadno šaljemo OpenGLu, šaljemo kroz taj kontekst.*

Naravno, moguće ih je imati i više i prilikom izvršavanja programa, mijenjati iz jednog u drugi ako je potrebno.

Ostali sjenčari: geometry sjenčar

Između vertex sjenčara i fragment sjenčara nalazi se opcionalni geometry sjenčar. Taj sjenčar kao ulaz uzima **skup** vrhova koji opisuju neku geometrijsku primitivu (npr. točku, trokut, niz linija koji su definirani ovdje [8]). Sjenčar nad tih vrhovima primjenjuje transformacije, na sličan način kao što vertex sjenčar primjenjuje za pojedinu točku, te pri završetku šalje transformirane vrhove na ulaz idućeg sjenčara.

Napomena 3.2.2. *Geometry sjenčar također može i stvarati nove vrhove, uz transformaciju postojećih, te može na izlazu definirati neku različitu primitivu od one koje je dobio kao ulaz.*

Isječak kôda 3.3: Primjer geometry shadera [6]

```
#version 330 core
layout (points) in; //ulazni podaci
layout (line_strip, max_vertices = 2) out; //primitiva na izlazu i ukupan broj točaka koje shader vraća

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

U primjeru, geometry sjenčar dobiva na ulazu jedan vrh. Svakim pozivom **EmitVertex()**, sjenčar sprema točku definiranu s **gl_Position** u novu primitivu. Kada se pozove **EndPrimitive**, sjenčar šalje na izlaz novo nastalu primitivu koja se sastoji od 2 vrha, te će krajnji rezultat toga biti da će se svaki vrh prikazati na slici kao mala linija, jer geometry sjenčar u ovom primjeru tranformira svaki vrh scene u liniju.

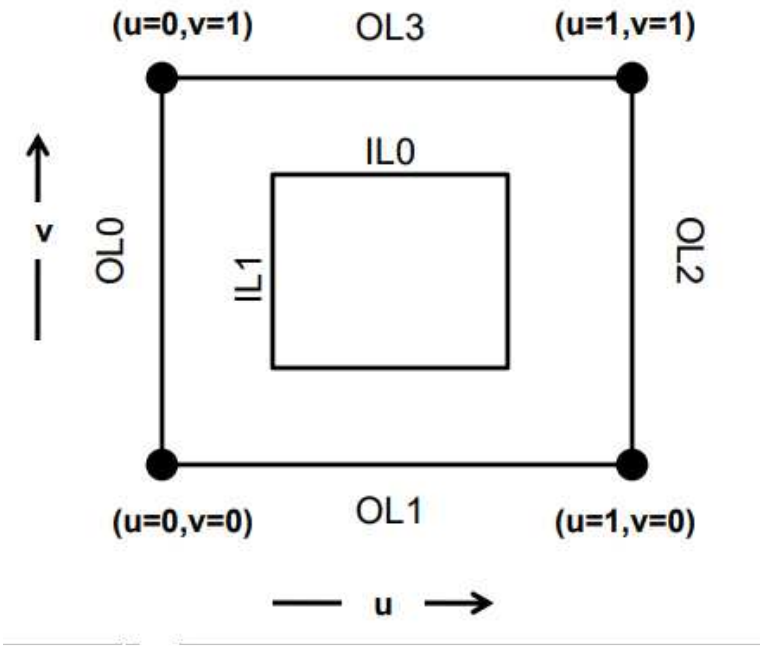
Ostali sjenčari: tesselation sjenčari

Tesselation (popločenje) je proces kojim nad nekom primitivom [8] radi subdivizija na manje poligone, te se nakon toga manji poligoni transformiraju po potrebi. U praksi, taj proces se koristi za crtanje parametarski zadanih krivulja i ploha, tako da sjenčaru samo damo željenu domenu i pravilo (funkciju) po kojem da transformira novo stvorene točke. Taj sjenčar je zato vrlo koristan jer se skup proces crtanja parametarski zadanih krivulja i ploha delegira na GPU, umjesto da na CPU računamo vrijednosti tih točaka. Za razliku od geometrijskog sjenčara, tesselation sjenčara ima dva (tri, ako se broji i sjenčar koji povezuje ta dva, njega nije moguće programirati), te su također opcionalni. Tesselation sjenčari se sastoje od:

- **Tessellation Control Shader (TCS)** - opcionalni sjenčar (u slučaju da se ne definira, koristiti će se zadane postavke) koji određuje razinu tesselacije, tj. gustoću subdivizije koju želimo. Ukoliko želimo sami definirati subdiviziju, imamo izbor za svaku stranicu primitive definirati razinu subdivizije. Primitive među kojima možemo birati su: **trokut**, **pravokutnik** ili **izolinije** [9]. Kao primjer ćemo uzeti **pravokutnik**. Primitiva je zadana kao jedinični pravokutnik. Svaki od stranica može imati različitu razinu subdivizije (*gl_TessLevelOuter*). Uz to, također definiramo i *unutarnju* (*gl_TessLevelInner*) subdiviziju za svaku os. Pogledati sliku 3.5.

Isječak kôda 3.4: Primjer TCS shadera [6]

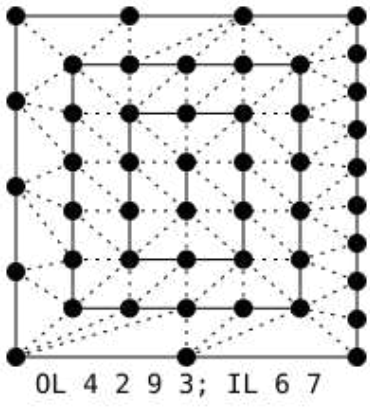
```
#version 410 core
// razine subdivizije direktno čitamo iz uniformnih
↪  varijabli
uniform float uOuter02, uOuter13, uInner0, uInner1;
#define ID gl_InvocationID
layout( vertices = 16 ) out;
void main( )
{
    gl_out[ID].gl_Position = gl_in[ID].gl_Position;
    gl_TessLevelOuter[0] = uOuter02;
    gl_TessLevelOuter[2] = uOuter02;
    gl_TessLevelOuter[1] = uOuter13;
    gl_TesslevelOuter[3] = uOuter13;

    gl_TessLevelInner[0] = uInner0;
    gl_TessLevelInner[1] = uInner1;
}
```



Slika 3.4: Subdiviziju pravokutnika definiramo preko 6 vrijednosti: 4 razine subdivizije za svaku vanjsku stranicu, te 2 razine za unutarnju subdiviziju.
Izvor: [6]

- **Generator primitiva tesselacije** - drugi sjenčar koji nije programibilan. Generira subdiviziju koja je bila zadana s TCS sjenčarom (slika 3.5)

- **Tessellation Evaluation Shader (TES)** - treći sjenčar koji dobiva kao ulaz novo generiranu točku, kao rezultat prethodne subdivizije. Ovisno o poziciji unutar primitive (varijabla *gl_TessCoord*) možemo nad točkom napraviti neke napredne tranformatcije. Npr. ako znamo parametarsku definiciju neke plohe, koristimo ju u ovom koraku da transformiramo primitivni četverokut u tu plohu



Slika 3.5: Generirana subdivizija iz razina koje smo u TCS definirali: u primjeru su vanjske razine postavljene na 4, 2, 9, 3, dok su unutarnje postavljene na 6, 7
Izvor: [6]

Isječak kôda 3.5: Primjer TES shadera [6]

```
#version 410 core
layout (quads) in;

void main()
{
    // koordinate generirane točke unutar primitive
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    //ovdje koristimo parametarsku definiciju željene
    ↪  plohe
    vec3 P = izracunajTransformaciju(u,v);
    gl_Position =  P;
}
```

Slanje podataka na GPU

Slanje podataka iz memorije u grafičku karticu mora biti brzo i efikasno. Strukture podataka koje ćemo koristiti za slanje moraju nuditi spremanje podataka za točke u sceni na neki generičan način, tako da ih grafička kartica na brz način može učitati u svaku jezgrenu jedinicu.

Definicija 3.2.3 (VBO)**.** *VBO, ili vertex buffer object je vrsta objekta u OpenGLu koji se koristi za spremanje i slanje podataka. U njega se spremaju informacije o točkama kao što su pozicija, boja i teksturne koordinate.*

Definicija 3.2.4 (VAO)**.** *VAO, ili vertex array object je vrsta objekta koja sprema sva stanja potrebna da se pošalju podaci o točkama spremljenim u VBO. Ako u neki VBO spremimo neke informacije o točkama poput koordinata, a u drugi VBO neke druge informacije o istim točkama kao npr. njihove boje, VAO sadrži informacije o tome što je gdje spremljeno i u kojem formatu. VAO je moćan alat jer ga je moguće mijenjati usred izvršavanja programa, i time upravljati koje točke i njihove atribute šaljemo na grafičku karticu.*

Definicija 3.2.5 (Binding)**.** *Bind, binding (eng. povezivanje) je proces u OpenGLu gdje OpenGL stroj stvori neku referentnu varijablu koja se povezuje s varijablama u našem programu, u svrhu promjene nekog stanja u OpenGLu. Na primjer, ukoliko imamo više VBO u programu, bind daje OpenGLu do znanja koji VBO je "trenutni" i sve daljnje akcije se odnose na taj VBO, pa kao rezultat te naredbe same po sebi ne traže kao argument koji VBO je u pitanju, jer se to inducira iz posljednjeg bind-a koji je pozvan.*

Primjer 3.2.6 (Vertex attribute pointer)**.** *To je funkcija kroz koju, kada popunimo podacima jedan VBO, zapisujemo točnu strukturu tih podataka. Na primjer, neka postoje sljedeći podaci:*

$$x = [1, 0, 2, 2, 0, 3, 0, 3, 0, 0, 5, 0]$$

koji sadrže koorinate od prve točke (x, y, z), boju prve točke (r, g, b), te isto i za drugu. OpenGL zahtjeva da svi podaci šalju u VBO u 1D polju. Tada pozivamo sljedeću funkciju dvaput:

$$glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \cdot sizeof(float), 0);$$

$$glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 3 \cdot sizeof(float), 3 \cdot sizeof(float));$$

Kroz prvi poziv definiramo kako će se spremati koordinate točaka, a kroz drugi boje. Prvi argument je jako bitan: kroz njega program sjenčara može dohvatiti te podatke.

$$layout(location = 0)\ in\ \ vec3\ aPos;$$

kroz ovu liniju sjenčar dobiva konkretne podatke o točkama iz polja x.
Drugi argument funkcije označava koliko neposrednih vrijednosti u polju označava koordinate, tj. boju, u oba slučaja je to 3.
3. argument je tip podataka, 4. ne koristimo pa ga stavimo na ***GL_FALSE****.*
5. argument se zove ***stride****, ili pomak, tu definiramo koliko idućih podataka u polju moramo preskočiti da bi učitali atribut jednake vrste iduće točke, to je u oba slučaja isto 3.*
6. argument je ***početni pomak****, u našem slučaju to je 0 za koordinate, a 3 za boje, jer prve informacije za boju se u x nalaze tek iza 3. mjesta.*

Opis tijeka rada OpenGL-a

Navest ćemo elementarne korake svakog OpenGL programa za lakše shvaćanje programskog koda:

1. **Podešavanje kontexta i OpenGL prozora**: OpenGL-u su potrebne neke od pomoćnih biblioteka kako bi se upravljalo prozorom pošto sam OpenGL nema podršku za to. Također je potrebno i podesiti OpenGL kontekst, kako bi ga mogli koristiti za izvršavanje naredbi za OpenGL

2. **Stvaraju se Vertex buffer i vertex array objekti** (VAO, VBO): Oni se koriste kako bi se slali podaci između izvršnog programa i grafičke kartice. Mora ih se podesiti.

3. **Stvara se program sjenčara**: OpenGL nam daje referencu preko koje možemo početi stvarati program sjenčara.

4. **Učitavanje sjenčara**: iz datoteka se učitava kodovi vertex i fragment sjenčara (i ostalih opcionalnih sjenčara), te se kodovi povezuju s OpenGL referencama.

5. **Kompajliranje koda sjenčara**: program tada kompajlira kod od svakog sjenčara, koji se sprema u internu memoriju OpenGL biblioteke

6. **Linkanje sjenčara**: Linkanje je proces u kojem se uzimaju kompajlirane verzije više sjenčara kako bi se stvorio konačan shader program. Linkaju se vertex i fragment sjenčari, i ostali ako postoje.

7. **VAO binding**: Bind je proces u kojem OpenGLu damo do znanja da je VAO koji se binda postao trenutni/aktualni, jer u OpenGLu je moguće sa samo jednim VAO i jednim VBO upravljati u nekom trenutku.

8. **VBO binding**: Isto kao i za VAO. Jednom kada bindamo VBO, može se početi na njemu raditi.

9. **Kopiranje podataka**: podaci točaka se kopiraju u VBO strukturu

10. **Poziva se VertexAttribPointer funkcija**: To su naredbe kroz koje damo informacije u VAO koja je struktura podataka u trenutnom VBO objektu. Na primjer, ako podaci koje smo spremili u VBO sadrže koordinate i boju točaka, radimo 2 poziva te funkcije.

11. **Petlja u kojoj se crta program**

12. **Bind programa sjenčara**: Sjenčar program koji smo gore napravili bindamo te time postaje aktualan.

13. **Podešavanje uniformnih varijabli**: Sve varijable koje želimo koristiti za vrijeme izvršavanja programa koje želimo slati retrokativno u grafičku karticu moramo ih označiti kao uniformne. Te vrste varijabli se ne mijenjaju tijekom izvršavanja sjenčar programa. Na primjer ako želimo mijenjati položaj kamere kada pomaknemo miš, te informacije se šalju na GPU kroz uniformne varijable.

14. **Crtanje**: inicira se crtanje na ekran, što u potpunosti OpenGL radi u pozadini, mi samo iniciramo naredbu

15. **Zamjena (eng. swap) buffera**: Zato što očekujemo od programa da crta nekoliko desetaka puta u sekundi, ne crta se na ekran direktno nego kroz dva buffera. Kada program crta sliku u jedan buffer, rezultat drugog je vidljiv na ekranu, te se onda zamijene. Jedan buffer dakle može biti u stanju crtanja ili u stanju prikaza tokom izvršavanja. To svojstvo se koristi kod animiranih scena te ga nema svrhe koristiti kod statičnog prikaza.

16. **Petlja gotova**

17. **Program uništava VBO i VAO**

18. **Program uništava program sjenčara**

19. **Program uništava kontekst i prozor**

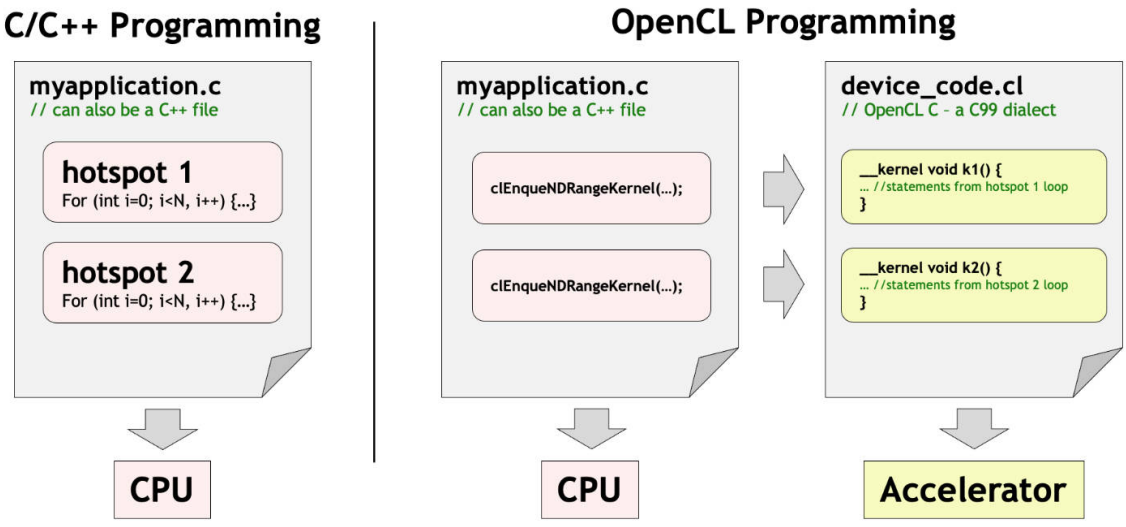
20. **Program se gasi.**

3.3 OpenCL biblioteka

OpenCL je biblioteka koja se koristi za izvršavanje programa u sustavima koji se sastoje od više procesora i grafičkih procesnih jedinica. OpenCL time nudi sučelje za paralelno računanje. Koristi se u raznim primjenama, primarno u procesiranju slike, zvuka i video materijala, fizici i drugim raznim znanstvenim simulacijama i računima.
Mi primarno koristimo OpenGL kako bismo pokrenuli program sjenčara paralelno na više jezgri, i time drastično uštedjeli na vremenu.

Definicija 3.3.1 (Host)**.** ***Host*** *je u ovom kontekstu naziv za server, računalo ili program čija je primarna uloga inicirati i upravljati, te komunicirati s ostalim hardware-om u sustavu. Host program je primarna i završna točka OpenCL programa. U većini slučajeva, host kôd se piše u C/C++ jeziku.*

Definicija 3.3.2 (Device)**.** ***Kernel*** *(eng. jezgra, jezgreni kôd) je kôd koji se izvršava na računalima, tj. procesorima u sustavu koji nisu u ulozi host-a, tj* ***device*** *(eng. uređaj), koji je još poznat pod nazivom accelerator processors/devices (eng. ubrzivajuć procesor/uređaj) Konkretno u OpenCL-u, jezik koji se koristi za pisanje kernel kôda jest OpenCL Kernel Language, koji je sintaksom nalik C-u. U našem slučaju, ulogu kernela imaju* ***sjenčari****.*



Slika 3.6: Razlika između konvencionalnog izvođenja programa na CPU-u i izvođenja preko OpenCL sučelja. Ulogu *Accelerator*-a ima GPU.
(Izvor: Khronos)

Definicija 3.3.3 (Context)**.** *OpenGL context ili kontekst jest objekt u koji se sprema konekcija do određenog uređaja. Context se generira na host uređaju, te se koristi kako bi host upravljao memorijom i ostalim resursima tog uređaja. Jednom kada se generira context, taj context generira* ***komandni red*** *specifično za taj uređaj. Context također generira buffere, to su objekti koji se koriste za slanje i primanje informacija između hosta i uređaja.*

Definicija 3.3.4 (Komandni red)**.** *Komandni red (eng. command queue) se koristi kako bi host slao naredbe uređajima, naredbe poput pokretanja kernela ili slanja podataka sa ili u uređaj.*

Opis tijeka rada OpenCL programa

Opis rada OpenGLa se može opisati u par ključnih koraka:

1. Host program (napisan u C/C++) **inicira zahtjev** za korištenje CPU i GPU uređaja.

2. Host aplikacija tada kompajlira kernel kôd, koji u potpunosti definira što će odabrani uređaji na sebi izvršavati.

3. Host aplikacija tada generira jedan ili više **komandnih redova** (eng. command queue). Ti redovi se sastoje od akcija koje će uređaju obavljati, npr. vrste akcija uključuju: izvršenje kernel kôda, kopiranje podataka sa ili u uređaj.

4. Uređaji izvršavaju akcije dane u spomenutim redovima te svaki od njih vrati rezultat host programu.

5. Host program, jednom kada ima sve rezultate računanja, može inicirati naknadne zahtjeve uređajima ili generirati željeni rezultat.

Opis jednog OpenCL programa

Isječak kôda 3.6: OpenCL pseudokôd jednostavnog programa [1]

```
// Ulazni i izlazni nizovi
int n = 4;
float array1[] = {1, 2, 3, 4};
float array2[] = {1, 1, 1, 1};
float result[n];

// Stvaranje OpenCL konteksta i biranje uređaja
cl_context context = clCreateContext(&device_id);
// Stvaranje komandnog reda
cl_command_queue queue = clCreateCommandQueue(context);


// Učitavanje kernel kôda iz datoteke
string source = loadProgram("add.cl");
cl_program program = clCreateProgramWithSource(context,
↪  &source);
// Kompajliranje kernel programa
clBuildProgram(program);
// Stvaranje kernel programa iz danog kôda
cl_kernel kernel = clCreateKernel(program, "add_arrays",
↪  &err);

// Stvaranje buffer objekata za ulaznu i izlaznu memoriju
cl_mem array1_buf = clCreateBuffer(context, CL_MEM_READ_ONLY,
↪  sizeof(float) * n, array1);
cl_mem array2_buf = clCreateBuffer(context, CL_MEM_READ_ONLY,
↪  sizeof(float) * n, array2);
cl_mem result_buf = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
↪  sizeof(float) * n, result);

// Ulaz kernel programa je oblika
// add_arrays(arg1, arg2, ...), ovdje se definira što su
// ti argumenti
clSetKernelArg(kernel, 0, sizeof(array1), &array1_buf);
clSetKernelArg(kernel, 1, sizeof(array2), &array2_buf);
clSetKernelArg(kernel, 2, sizeof(result), &result_buf);
clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
// Izvršavanje kernel programa
clEnqueueNDRangeKernel(queue, kernel, n);
// Kopiranje izlaza iz izlaznog buffera u result array
clEnqueueReadBuffer(queue, result_buf, sizeof(float) * n,
↪  result);
// u results se nalaze rezultati izvođenja
std::cout << results;
```

Gornji pseudokôd je primjer programa koji se sastoji od dvije liste brojeva veličine 4, te je cilj zbrojiti ih na uređaju, tj. GPU-u.

Slijedi pripadajući kernel kôd:

Isječak kôda 3.7: Kernel kôd koji pripada gornjem programu [1]

```
__kernel void add_arrays(__global float *a, __global float *b,
↪  __global float *c, int n) {
    int i = get_global_id(0);
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
```

Ovaj kernel kôd definira funkciju ”add_arrays” koja prima 4 parametra:
dvije liste brojeva (ulazi), listu brojeva u koju ispisujemo izlaze, *n* koji označava veličinu listi. ”get_global_id” je funkcija koju kernel koristi kako bi odredio **kojem indeksu u listi** je taj kernel pridodijeljen.

Pokretanje OpenCL programa

OpenCL kôd se u Linux okruženju kompajlira naredbom:

gcc main.c -o vectorAddition -l OpenCL

Detalji oko pokretanja programa mogu se naći u dodatku A.

Poglavlje 4

Implementacija

Implementaciju algoritma praćenja zrake prati opsežni skup članaka na [2]. Kao svjetlosni algoritam će se koristiti Phong te će također podržavati refrakciju i refleksiju. Kao što je u poglavlju 2.2 objašnjeno, prolazimo kroz svaki piksel i "gradimo" sliku piksel po piksel.

Implementacija algoritma praćenja puta prati *Ray Tracing in One Weekend*[27] seriju knjiga, pomoću kojih slažemo cijeli algoritam iz temelja u C++ bez korištenja aplikacijskih sučelja poput OpenGLa ili OpenCLa. Implementirat ćemo i niz dodatnih klasa i funkcija koje nisu opisane u knjigama.

I na kraju kao ilustracijski primjer i dokaz da je moguće implementirati praćenje puta u **realnom vremenu**, prikazujemo verziju implementiranu u OpenCL biblioteci [20].

4.1 Uvod

Implementacije algoritma praćenja zrake i praćenja puta dosta su slične te ćemo u idućoj sekciji objasniti konkretne razlike. Oba programa se izvršavaju na procesoru (ne na grafičkoj kartici), stoga su oba algoritma dosta spora. Pošto vremenska složenost algoritama uvelike ovisi o broju objekata u sceni, vidjet ćemo da su implementacije na CPU jako nepraktične za velike scene, jer vremenska složenost strmo raste s rastom broja objekata. Svaki od ponuđenih implementacija imat će nekoliko hiperparametara:

Definicija 4.1.1 (Hiperparametar)**.** *Hiperparametar je parametar implementacije koji se namješta ručno prije izvršavanja programa, dok se obični parametri mijenjaju programski tokom izvršavanja. Neki konkretni hiperparametri u danim implementacijama su:*

- ***Broja uzoraka** (samples) koliko će se primarnih zraka generirati za svaki piksel, koje će se na kraju uprosječiti da se dobije konačna boja piksela*

- ***Dubina rekurzije*** *(depth) koliko će se puta zraka odbiti u sceni prije nego li rekurzija stane*
- ***Širina i visina slike*** *(width, height) veličina slike se definira kroz dva broja, broj piksela u konačnoj slici je tada širina x visina.*
- ***Pozicija kamere*** *Kameru ćemo definirati kroz 3D koordinatu u sceni*

Svi ovi hiperparametri će se definirati u YAML formatu datoteke, koju program čita prije izvršavanja.

Uz navedene hiperparametre, koje je nužno navesti, također je nužno navesti i konkretne objekte u sceni. Kategorizirat ćemo ih po tipu, jer ovisno kojeg su tipa, drukčije se računaju kolizije, sjenčanja nad njima i slično.

Algoritmi bez API-ja - uvodne napomene

Unos podataka

Učitavanje objekata u scenu je dosta trivijalno, ali bitno je da se drži nekog standarda unosa. Za ove projekte ćemo koristiti YAML datoteku jer njihov zapis nije kompliciran i lako se čita, te ga program lako parsira.

Ulaz u program će biti popis objekata. U datoteci će uz svaki objekt biti definirani njegovi podatci, koji uključuju: **Ime** objekta, **Tip**, **Poziciju u sceni** i **Materijal**.

Definicija 4.1.2 (Tipovi objekata u sceni)**.** *Tip objekta je jedno od sljedećeg: tip koji smo definirali u 1 koji se sastoje od trokuta, ravnine i sfere, ili* ***Mesh*** *tj. skup trokuta, koji definira neki kompleksniji objekt.*

Što se materijala tiče, vidjet ćemo u idućem odjeljku da se implemetacija malo razlikuje između praćenja zrake i praćenja puta. Naime, praćenje zrake koristi ”gotove” svjetlosne algoritme poput Phonga koji računaju boju u točki, pa se za neki objekt umjesto materijala samo definira neki svjetlosni algoritam. S druge strane, u praćenju puta imamo veću slobodu pri definiciji materijala, te žemo prikazati par primjera tih materijala.

Također je bitno napomenuti na koji način ćemo učitati veće objekte u scenu. Koristiti ćemo format ”.obj”, u kojem će se nalaziti opis jednog kompleksnog modela.

Definicija 4.1.3 (Učitavanje **Mesh** objekta)**.** ***Mesh*** *objekt se učitava preko .obj datoteke. Punog naziva Object Wavefront File, ”.obj” datoteka se koristi za spremanje geometrijskih informacija o objetku ili cijeloj sceni. Jedana takva datoteka sadržava:* ***Pozicije točaka***

u sceni (lista, po tri broja u novom redu, ***Listu teksturnih koordinata*** *(po dva broja u svakom redu),* ***listu normala svakog trokuta*** *(po tri broja u svakom redu), te najbitnije,* ***listu definiranih trokuta*** *koji čine taj objekt. Na primjer, jedan element te liste je:*
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
gdje su vi indeksi točaka koje čine taj trokut, vti indeksi teksturnih koordinata, a vni indeksi normala. Više o tome se može naći u ekstenzivnom članku koji opisuje detalje te datoteke. [4].

Unos podataka zadanih preko Bézier-ovih krpica

Kao što smo rekli u uvodnom poglavlju o Bézier-ovim krpicama, kolizije s zrakama nećemo evaluirati direktno nego ćemo krpice transformirati u Mesh. Za početak definirajmo ulazne podatke:

Isječak kôda 4.1: Ulazni podatci za param. definiran objekt [2]

```
const int numOfPatches = 32;
const int numOfVerts = 306;
int objectPatches[numOfPatches][16] = {
    {  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,
    ↪  12,  13,  14,  15,  16},
    ...
    {270, 270, 270, 270, 300, 305, 306, 279, 297, 303, 304,
    ↪  275, 294, 301, 302, 271}
};
vec3 objectVertices[numOfVerts] = {
    { 1.4000,  0.0000,  2.4000},
    ...
    { 1.4250, -0.7980,  0.0000}
};
```

Objekt se sastoji od *numOfPatches* broja krpica, te od *numOfVerts* broja kontrolnih točaka (od kojih se neke ponavljaju kroz više krpica).

Isječak kôda 4.2: Implementacija kolizije zrake i ravnine [2]

```
int divs = 16; //definiramo subdiviziju mesha
auto P = std::vector<vec3>((divs + 1) * (divs + 1));
auto nvertices = std::vector<int>(divs * divs);
auto vertices = std::vector<int>(divs * divs * 4);
vec3 controlPoints[16];
for (int np = 0; np < numOfPatches; ++np) {
    // obrađujemo np-tu krpicu
    // definiramo kontrolne točke za np-tu krpicu
    for (int i = 0; i < 16; ++i){
        controlPoints[i][0] =
        ↪ objectVertices[objectPatches[np][i] - 1][0],
        controlPoints[i][1] =
        ↪ objectVertices[objectPatches[np][i] - 1][1],
        controlPoints[i][2] =
        ↪ objectVertices[objectPatches[np][i] - 1][2];
    }
    // evaluiramo, te spremamo konretne točke novog mesha
    for (int j = 0, k = 0; j <= divs; ++j) {
        for (int i = 0; i <= divs; ++i, ++k) {
            P[k] = evalBezierPatch(controlPoints, i /
            ↪ (float)divs, j / (float)divs);
        }
    }
    // sada generiramo indekse točaka za svaku malu
    ↪ subdiviziranu stranicu
    for (int j = 0, k = 0; j < divs; ++j) {
        for (int i = 0; i < divs; ++i, ++k) {
            nvertices[k] = 4;
            vertices[k*4]= (divs + 1) * j + i;
            vertices[k*4+1]= (divs + 1) * (j + 1) + i;
            vertices[k*4+2]= (divs + 1) * (j + 1) + i + 1;
            vertices[k*4+3]= (divs + 1) * j + i + 1;
        }
    }

    objects.push_back(new PolygonMesh(o2w, divs * divs,
    ↪ nvertices, vertices, P));
}
```

Pokazivači

Zbog ogromne količine podataka kojima program upravlja, koriste se pamtene vrste pokazivača u C++. Ti pokazivači su ”pametni” jer sami upravljaju alociranjem memorije. Opasnost ne-korištenja te vrste pokazivača je da ako slučano ne de-alociramo memoriju, jako brzo ćemo proizvesti curenje memorije, što inače ne stvara toliki problem, ali u ovom slučaju stvara zbog velike količine memorije koju koristimo, jer kada dovoljno memorije procuri, ta memorija ne daje prostor ostalim procesima i jedini način da se oslobodi natrag jest da se radna memorija (RAM) ugasi, tj. da se računalo ugasi i upali.

Isječak kôda 4.3: Primjer unique pointera

```
#include <memory>
int main() {
    std::unique_ptr<int[]> velikoPolje(new int[1000000]);
    velikoPolje[0] = 5;
    velikoPolje[1] = 7;
    // ...
}
```

Napomena 4.1.4. *Od C++14 nadalje, koristimo posebnu funkciju* ***std::unique_ptr*** *za alociranje. Konkretno, u programu to znači da nećemo koristiti ključnu riječ new za alociranje nego će taj posao raditi nova funkcija.*

Isječak kôda 4.4: Primjer unique pointera

```
#include <memory>
int main() {
    auto velikoPolje = std::make_unique<int[]>(1000000);
    // ...
}
```

Još jedan bitan koncept je njihovo premještanje, što često koristimo kada želimo premjestiti cijeli taj objekt s jednog pametnog pokazivača na drugi:

Isječak kôda 4.5: Primjer premještanja unique pointera

```
#include <memory>

int main() {
    std::unique_ptr<int[]> uniqueArray(new int[1000000]);
    velikoPolje[0] = 5;
    velikoPolje[1] = 7;
    // ...

    // premještamo velikoPolje u pokazivač velikoPolje2
    std::unique_ptr<int[]> velikoPolje2 =
    ↪  std::move(velikoPolje);

    // BITNO: početni pokazivač ne sadrži više sadržaj polja
    std::cout << uniqueArray.get() << std::endl; // vraća
    ↪  nullptr
    std::cout << uniqueArray2.get() << std::endl; // vraća
    ↪  adresu polja u memoriji
}
```

Algoritam u OpenCL/OpenGL - uvodne napomene

Algoritam koristi interoperabilnost između OpenCL i OpenGL biblioteka. (pogledati [26]) OpenGL se u našem slučaju koristi kao ”wrapper” biblioteka oko OpenCL programa kako bi omogućili OpenCL izvršanje u realnom vremenu. Naime, OpenCL sam po sebi vraća rezultate koje kernel na GPU izračuna te se sami brinemo kako ćemo te rezultate prikazati na ekran. Mogli smo samo te podatke spremiti u sliku, ali ovako, pomoću OpenGL biblioteke, rezultat izvršavanja OpenCL programa šaljemo u OpenGL koji prikazuje rezultate na ekran u realnom vremenu, OpenGL petlja je u stvari ta koja poziva OpenCL izvršavanje u svakom novom prikazu (frame-u).

Napomena 4.1.5. *Mogli smo umjesto OpenGL koristiti neku drugu biblioteku koja bi služila kao ”wrapper” oko OpenCL-a. Primjer jedne takve je SFML, koja se koristi u proizvodnji C++ video igrica. SFML bi samo služio da ”pozove” OpenGL program u svakom ”frame-u”, te bi rezultat prikazao na ekran. Razlog korištenja OpenGL-a je da je SFML dosta visoke apstrakcije, te je dosta trivijalan za korištenje, a pošto pišemo program u OpenCL-u, odabrali smo biblioteku slične razine apstrakcije.*

Rekurzije nisu podržane

Kao što naslov kaže, niti u OpenCL niti OpenGL-u ne možemo pisati rekurzivan kod, stoga nam ne preostaje ništa drugo nego da petlju iz 2 pretvorimo u iterativnu verziju. Problem se javlja jer nemaju sve GPU podršku za rekurziju, stoga je to zabranjeno u OpenCl-u i OpenGL-u [7]. (Članak [21] u detalje objašnjava kako je moguće "iteratizirati" rekurziju praćenja zraka koju je onda moguće koristiti na GPU)

Dati ćemo primjer pseudokoda kako je to riješeno u OpenCL implementaciji.

Isječak kôda 4.6: Primjer rješavanja problema rekurzije u OpenCL-u

```
function trace(spheres, camray, sphere_count, seed0, seed1)
    ray = camray
    accum_color = (0, 0, 0)
    mask = (1, 1, 1)
    randSeed0 = seed0
    randSeed1 = seed1

    // sljedeća pretlja glumi rekurziju
    for bounces = 0 to 7
        t = 0
        hitsphere_id = 0
        if not intersect_scene(spheres, ray, t, hitsphere_id,
        ↪  sphere_count)
            return accum_color + mask * (0.15, 0.15, 0.25)
        hitsphere = spheres[hitsphere_id]

        // račun ...

        accum_color += mask * hitsphere.emission
        mask *= hitsphere.color
        mask *= dot(newdir, normal_facing)
    return accum_color
```

Ovdje je bitno opisati što rade dvije varijable, **mask** i **accum_color**. **accum_color** akumulira boju svjetlosti koja se prati kroz scenu. Inicijalizira se na crnu, te prilikom svakom hitca s objektom, dodaje joj se količina svjetlosti **koja se isijava** iz materijala objekta. Varijabla **mask** čuva boju površine. Inicijalizira se na bijelu te prilikom svakog hitca, boja novog materijala se množi s tom varijablom. Maska time modulira kroz svaku iteraciju boju svjetlosti koja se isijava od materijala. Dakle zadnje tri linije petlje rade

sljedeće:

1. **accum_color += mask * hitsphere.emission**: Boja koju trenutni objekt **emitira** (hitsphere.emission) se dodaje u **accum_color**. Količinu koja se dodaje u iteraciji, se modulira (ograničava količinu svjetlosti koja se dodaje) pomoću trenutne vrijednosti varijable **mask**, koja čuva u sebi boju hitaca koji su napravljeni kroz iteraciju.

2. **mask *= hitsphere.color**: U ovom se koraku maska ažurira s bojom trenutnog materijala (hitsphere.color). Boja se množi s trenutnom maskom. To omogućava maski da čuva sve boje prethodnih materijala gdje su se hitci dogodili.

3. **mask *= dot(newdir, normal_facing)**: Ova linija modificira masku tako da ju množi s skalarnim produktom **newdir** (novi smjer zrake) i **normal** (normale u točki hitca). Rezultat skalarnog produkta je skalar koji reprezentira kosinus kuta između ta dva vektora. Taj proces ustvari glumi svjetlost u stvarnom svijetu koja se odbija od difuznog materijala. Taj skalar prestavlja doprinos svjetlosti koja upada iz smjera **newdir**. Množimo masku s tim skalarom, pa time algoritam uzima veću količinu svjetlosti iz smjerova koji su bliže normali.

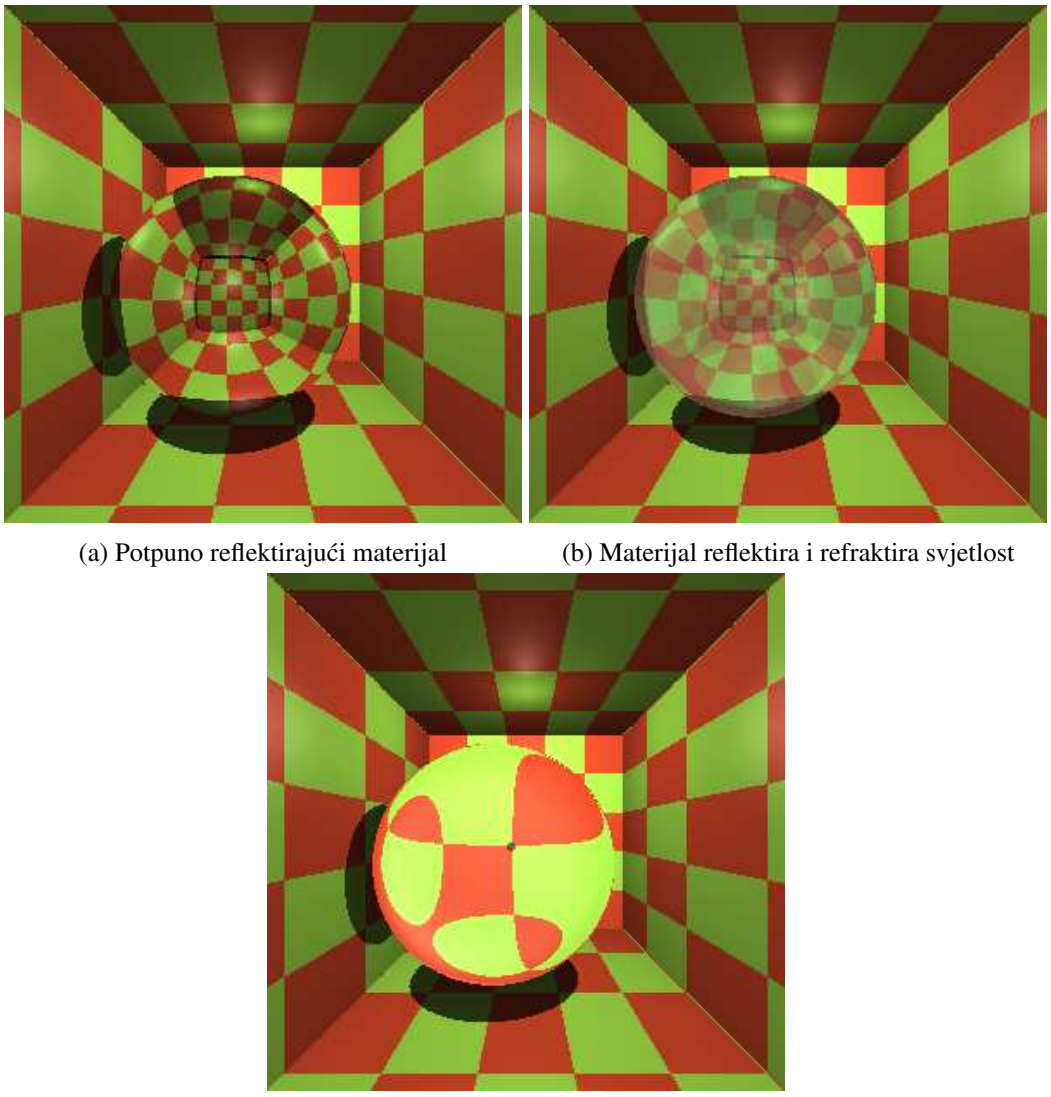
4.2 Rezultati

Praćenje zraka bez korištenja API-ja

Algoritmom praćenja zraka simulirali smo refrakciju i refleksiju materijala. Primjeri uključuju scenu u obliku kutije koja sadrži geometrijske likove nad kojima smo definirali neki materijal. Važno je uočiti da i zidovi koji okružuju te objekte također imaju definiran materijal, u primjerima je taj materijal žarke boje i reflektira svjetlost.

Napomena 4.2.1. *U primjerima praćenja zraka, za osvjetljavanje pozadine (zidova scene) koristimo Phong algoritam, može se uočiti da algoritam ispaljuje dodatne zrake nakon primarne, prema izvorima kako bi generirao sjene objekata. Nad ostalim materijalima Phong nije prisutan. Kako bi se dobili kompleksni rezultati, potrebno je kombinirati Phong ili neki drugi svjetlosni algoritam, zajedno s refleksijom i refrakcijom, te time definirati konkretnu BxRF funkciju za svaki od materijala.*

Slika 4.1: Praćenje zrake: simulacija visoko reflektirajućih površina, poput metala



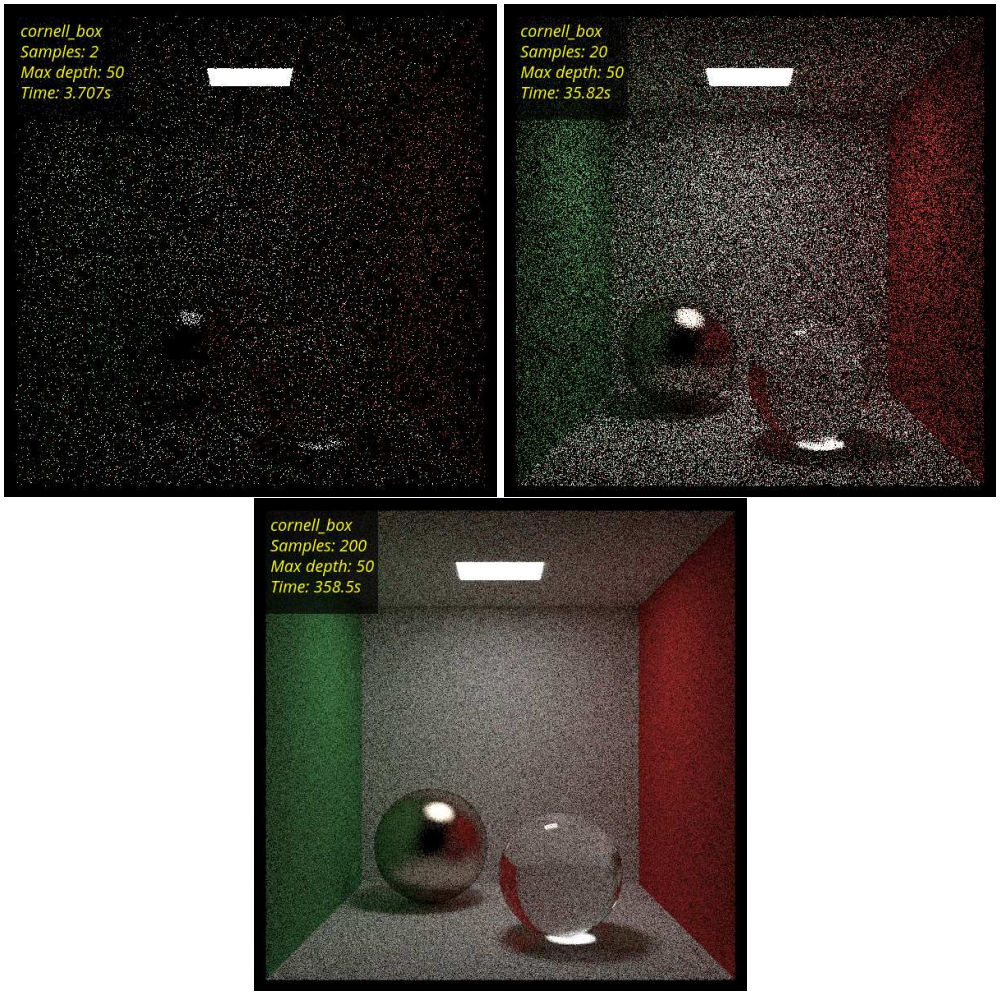
(a) Potpuno reflektirajući materijal (b) Materijal reflektira i refraktira svjetlost

(c) Potpuno refraktirajući materijal

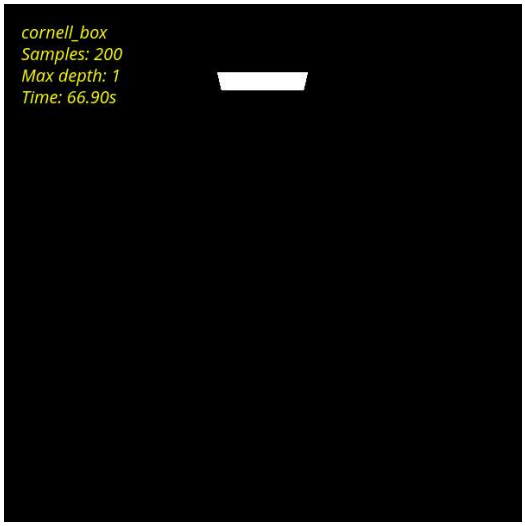
Slika 4.2: Praćenje zrake: simulacija prozirnog materijala

Praćenje puta bez korištenja API-ja

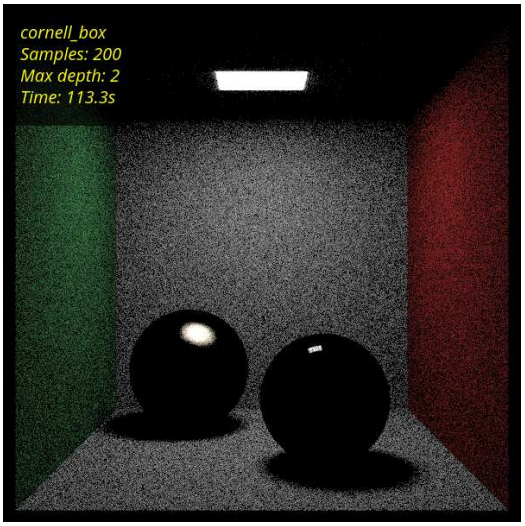
Kod praćenja puta, ne postoji poseban svjetlosni algoritam koji osvjetljava pojedini materijal, nego se oslanjamo na to da će zrake ispaljene iz kamere možda udariti u svjetlost na svome putu, i to zapravo pridodaje velikom realizmu. Neka ”svojstva” svjetlosti kao na primjer meke sjene i bolja točkasta refleksija jako su teške za implementaciju kod praćenja zrake jer ih neki svjetlostni algoritam mora napraviti, ali kod praćenja puta je to zadano ponašanje svjetosti. To naravno košta, može se vidjeti na primjerima 5.1 da je potrebno i par minuta za najmanje zrnatu sliku.



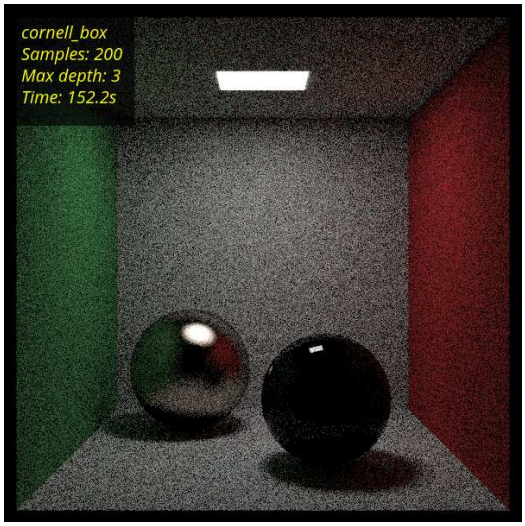
Slika 4.3: Praćenje puta: dubina lanca zraka je konstantna, varira broj uzoraka po pikselu (samples)



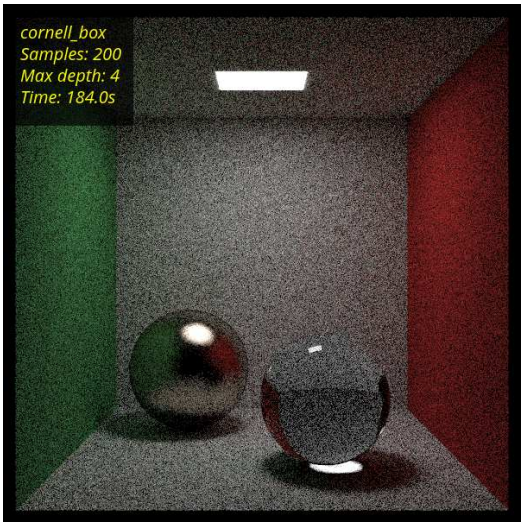
(a) Dubina: $n = 1$, nema refleksija, samo se vide izvori svjetlosti



(b) Dubina: $n = 2$, točkaste i difuzne refleksije počinju biti vidljive



(c) Dubina: $n = 3$, odrazi u refleksivnim materijalima počinju biti vidljivi



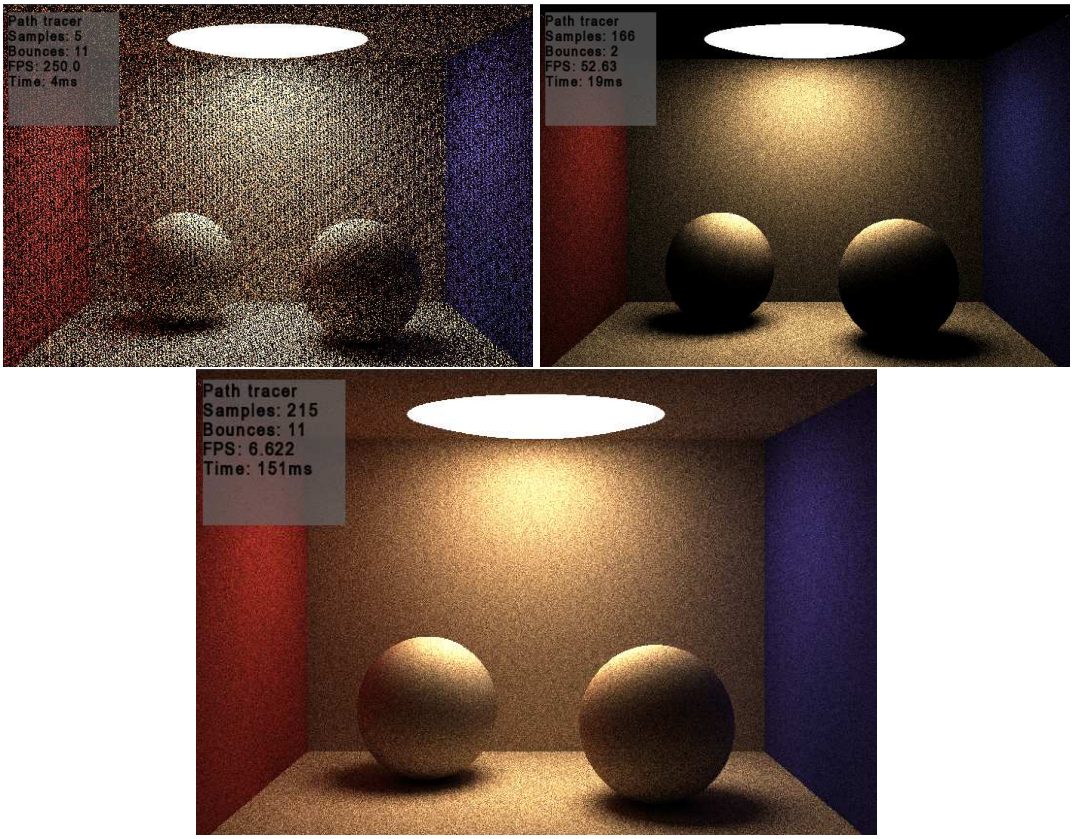
(d) Dubina: $n = 4$, zrake uspješno refraktiraju kroz prozirne materijale

Slika 4.4: Praćenje puta: varira dubina rekurzije

Praćenje puta u OpenCL-u

Kod slika u OpenCL-u, slike koje dobivamo su u stvarnom vremenu. Vidimo na ovoj jednostavnoj sceni da kod 215 uzoraka, više nije moguće u stvarnom vremenu izvršavati program, jer je 151ms puno vremena za jednu sliku.

Napomena 4.2.2. *Na slici 4.5 može se vidjeti neobična zrnatost kada smanjimo broz uzoraka. Zrnatost sama po sebi je predviđena, ali u rezultatu se javlja zrnatost u pravilnim uzorcima, a ne kao nepravilan šum, što nije za očekivati. Taj problem se javlja zbog činjenice da OpenCL nema mogućnosti generiranja slučajnih vrijednosti, pa je jedini način da se slučajne vrjednosti generiraju jest preko* (i, j) *koordinata piksela u OpenCL kernelu. Slučajni brojevi su nužni za algoritam jer preko njih zadajemo smjerove zrakama koje generiramo.*



Slika 4.5: Praćenje puta u OpenCL-u: u prvoj slici smanjen je broj uzoraka, u drugoj je manja dubina [20]

Poglavlje 5

Moguća poboljšanja realizma i brzine izvođenja

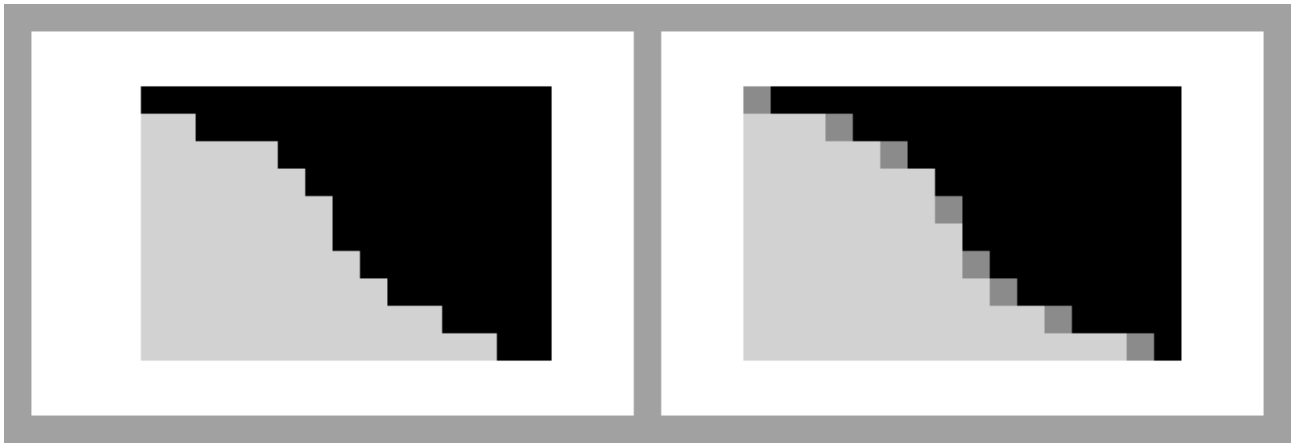
U nastavku slijedi par algoritama i tehnika koji bi znantno trebali ubrzati brzinu izvođenja programa te poboljšati realizam.

5.1 Anti-aliasing

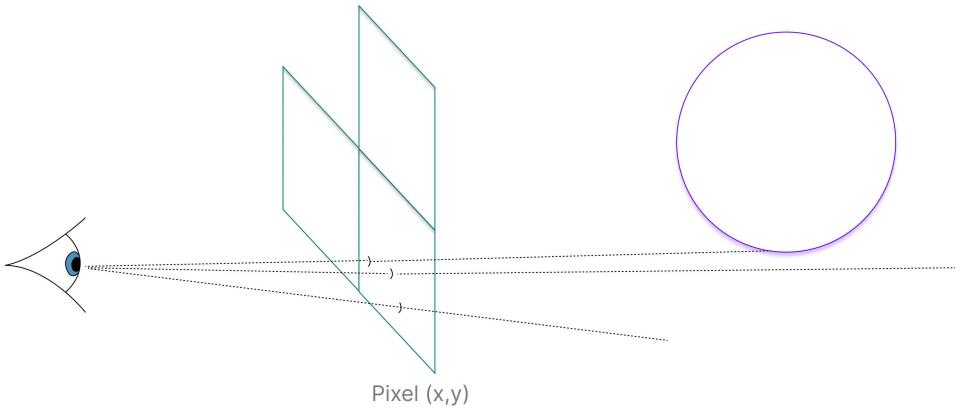
Anti-aliasing (hrv. zaglađivanje nazubljenosti, stohastičko zaglađivanje) je tehnika zaglađivanja nazubljenih rubova kako bi se poboljšala kvaliteta renderirane slike na ekranu. Općenito su kose linije problematične (nazubljene). Nazubljenost nastaje zbog konačnog broja piksela na ekranu koji su posloženi u matricu, i ne mogu realistično osjenčati linije koje nisu u ravnini s tom matricom piksela.

Zaglađivanje u praksi

U praksi, najlakše je problem riješiti tako da kroz svaki piksel ne ispalimo samo jednu zraku nego N njih, taj broj ćemo zvati broz uzoraka, ili **samples**, što se može vidjeti na slici 5.2. Svaka od zraka ima vektor smjera s vrlo malim pomakom. Rezultat toga je da svaka od tih zraka vrati drugu nijansu boje (ovisno u materijalima objekata koje presjeca). Ova metoda najviše pridonosi onim pikselima koji prikazuju **rubove** objekata, koji će time biti uglađeni (jer neke zrake će "taman" promašiti objekt dok će ga druge pogoditi, te će taj piksel poprimiti prosjek tih boja).



Slika 5.1: Rezultat uglađivanja
(Izvor: Wikipedia Anti_Aliasing)



Slika 5.2: Kroz piksel (x, y) je ispaljeno više zraka. Smjer svake zrake je malo pomaknut.

Opis dijela kôda

U kôdu ćemo riješitit taj problem tako da pri generiranju primarne zrake za $pixel_{xy}$, generiramo njih N, te ih sve zbrojimo u varijablu $finalcolor$. Također uvodimo varijablu $sampleShare$ koja određuje doprinos boje od svake zrake.

Isječak kôda 5.1: Primjer zaglađivanja

```
int numOfSamples = 50;
float finalcolor = 0;

float sampleShare = 1.0f / numOfSamples;

for (int i = 0; i < numOfSamples; i++)
    finalcolor += trace(...) * sampleShare;

return finalcolor;
```

5.2 Strukture ubrzanja: BVH

Hijerarhija omeđujućih volumena naziv je za abstraktnu stablastu struktru koja sadržava skup geometrijskih modela. Listovi tog stabla su sami objekti koji se nalaze u nekom omeđujućem volumenu, a ti volumeni se mogu nalazit u drugim većim volumenima, te time čine stablo. Volumen koji karakterizira korijenski čvor stabla gledamo kao jedan unificirani objekt u sceni, a zapravo je hijerarhija manjih objekata [16].

Zašto koristiti BVH-ove?

Pretpostavimo da se scena sastoji od puno malih objekata koji nisu uniformno raspoređeni u sceni, na primjer drvo s listovima. Ukoliko bi tretirali svaki taj list kao poseban objekt, kod računanja putanje svake zrake, petlja bi morala prolaziti kroz svaki od 1000+ listova. Bilo bi korisno da za one zrake koje ne udare u drvo, ne moramo računati da li zraka udari u svaki pojedini list. Tu ulaze BVH u igru. Ukoliko definiramo BVH koji omeđuje stablo, tada će svaka zraka **samo jednom** proći kroz stablo.

Sudar zrake s BVH

Ukoliko zraka udari u BVH, tada se neposredna djeca od korijenskog čvora tretiraju kao posebni objekti u sceni, te se primjenjuje isti postupak rekurzivno. Ako se dogodi da zraka ipak ne pogađa niti jedno neposredno dijete čvora, ništa se ne događa. U suprotnom, elementi tog djeteta se tretiraju kao posebni objekti itd.

Performanse

Gore opisani postupak znatno povećava performanse algoritma. Spremajući ih u takvo stablo, može se dostići relativno mala vremenska složenost u broju čvorova.

Dokaz. Pretpostavimo da imamo n objekata u sceni, te da svaki volumen može imati $m >= 2$ neposredne djece koji su drugi volumeni, ili objekti u sceni. Označimo s $T(n)$ vremensku složenost algoritma traženja presjeka u sceni. Očito je za vidjeti da algoritam, ukoliko zraka udari u volumen roditelja, će udariti u jedno neposredno dijete, ali u ostala neće, pa smijemo zapisati: $T(n) = T(n/m) + g$, gdje je g kompleksnost računanja presjeka za neki objekt, što je vremenski konstantno. Koristeći Master teorem [31], dobije se da $T(n) \in O(logn)$.

□

5.3 Monte carlo metoda u Ray tracingu

> *I have the most difficult kind of bug to find in a Monte Carlo program — a bug that produces a reasonable-looking image.*
>
> —Peter Shirley, *Ray Tracing: The Rest of Your Life*

Monte Carlo metoda se koristi kako bi se simulirao put zraka svjetlosti, umjesto da se uzdamo samo na čistu vjerojatnost da će zraka pogoditi neki izvor. Glavni cilj metode jest da nekako probamo naći dobre smjerove za zrake koje generiramo da što prije nađu izvor, i time ćemo postići manju zrnatost u slici [22]. Prisjetimo se Monte Carlo integracije:

1. imamo $f(x)$ koji moramo integrirati po nekoj domeni $[a, b]$
2. odaberemo neki *PDF* p na toj domeni
3. generiramo i uzmemo prosjek gomile točaka $\frac{f(x)}{p(x)}$, gdje je x vrijednost dobivena nekim nasumičnim generatorom vrijednosti

Cilj nam je dakle, našu distribuciju koja opisuje smjerove (koja je do sada bila uniformna, jer smo uzimali nasumične smjerove) nekako "usmjeriti" da nam generira upravo smjerove koji su nam bitni (oni koji su bliže izvoru). Od tud dolazi naziv Uzorkovanje po važnosti (eng. importance sampling). U nastavku teksta ćemo označiti s $\boldsymbol{A}$ vjerojatnost da se svjetlost rasprši od neke površine, očito je da je onda vjerojatnost da se ne rasprši (tj. da materijal upije svjetlost) jednaka $1 - \boldsymbol{A}$. U slučaju da se svjetlost rasprši, opisivat ćemo to pomoću raspšujućeg PDF-a s.

Definicija 5.3.1. *Raspšujući PDF* $\boldsymbol{s}$ *je PDF koji prima smjer* $dir \in \mathbb{R}^3$ *i vraća vjerojatnost raspršivanja svjetlosti* $s(dir)$ *u tom smjeru. Važno je napomenuti da daj PDF nije uniforman, te da ovisi o kutu ulazne zrake.*

Konačnu boju točke možemo definirati onda na sljedeći način:

$$\mathrm{I}_{\text{final}} = \int \boldsymbol{A} \cdot s(dir) \cdot \mathrm{I}(dir)$$

$\mathrm{I}(dir)$ određuje proizvoljni svjetlosni algoritam (koji također ovisi o kutu ulazne zrake). Navedeni integral za konačnu boju koja zraka vraća definira količinu ukupne svjetlosti na jediničnoj sferi (koja se u ovom slučaju raspršuje).

Primjer Monte Carlo integracije: Lambertijan materijal

Za rješavanje integrala generiramo sljedeće vrijednosti, te uzimamo njihov prosjek:

$$\mathrm{I}_{\mathrm{final}} = \frac{\boldsymbol{A} \cdot s(dir) \cdot \mathrm{I}(dir)}{p(dir)}$$

gdje je p(dir) proizvoljna PDF (prosjek u kontekstu algoritma praćenja puta znači isto što i broj uzoraka (samples)). U literaturi [22] se može naći dane BRDF funkcije za Lambertijan materijal:

$$BRDF = \frac{A \cdot s(dir)}{\cos\theta}$$

pa ćemo ju koristiti. Za raspršujuću PDF treba uočiti da je proporcionalan $\cos\theta$, gdje je θ kut između normale i kuta izlazne zrake. Radimo sličan račun kao u uvodnom poglavlju 1.6.9 da bi dobili valjani koeficijent (jer integral nad domenom neke PDF uvijek mora biti jednak 1, samo što je ovdje domena polusfera).

$$s(dir) := \frac{\cos\theta}{\pi}$$

Time dobivamo konačni BRDF koji ćemo koristiti:

$$\mathrm{I}_{\mathrm{final}} = \frac{A}{\pi}$$

Uspjeli smo naći formulu za konačnu boju. Kada bi ju ubacili u kraj petlje u algoritmu za praćenje puta (tamo gdje vraćamo konačnu boju zrake) ne bi još dobili poboljšanje kakvo želimo jer još mora postojati način da generiramo nove zrake koje iskorištavaju distribuciju koju smo za njih definirali. Generiranje takvih zraka ne ulazi u domenu ovog odjeljka, ali se može naći vrlo detaljan opis izgradnje istog u [29], a rigorozna matematička pozadina iza tog modela se može naći u [2], dok se u predavanjima [16] i u članku [14] mogu naći isti koncepti, ali puno jednostavnije objašnjeni.

Dodatak A

Upute za pokretanje primjera u Linux okruženju

Potrebno instalirati odgovarajuće biblioteke za OpenGL i OpenCL, te CMake.

Postavljanje projekta obavlja CMake program, tako da je jedino potrebno instalirati make datoteku pomoću CMake-a i pokrenuti ju da bi se kompilacija započela. Projekti se pokreću na sljedeći način:

1. ***git clone *link**** (repozitoriji [23], [24] i [25])
2. U korijenskom direktoriju otvoriti konzolu
3. ***mkdir build***
4. ***cd build***
5. ***cmake ..*** kreiranje make datoteke koja sadrži instrukcije za kompilaciju i linkanje projekta
6. ***make*** kompajlira se projekt
7. ***./PROJEKT*** naredba za pokretanje projekta

Pojmovnik

Algoritam praćenja puta (*eng. Path Tracing*) Algoritam praćenja puteva, "realističniji i skuplji" algoritam od praćenja zraka. 2, 50

Algoritam praćenja zrake (*eng. Ray Tracing*) Dvosmislen pojam za algoritam praćenja zrake. Također može značiti i Whittedov svjetlosni algoritam. 44

Baricentrične koordinate Sustav koordinata trokuta koji koristi udaljenosti od vrhova kako bi se odredila pozicija neke točke unutar trokuta. 20

Lambertijan materijal Ta vrsta materijala ima svojstvo da neovisno o smjeru promatrača, na cijeloj površini ima uniformnu distribuciju difuzne refleksije (pogledati sekciju 1.7). 92

Mesh naziv za objekt u sceni koji se sastoji od skupa trokuta (ponekad i četverokuta ili kompleksnijih poligona, ali se u tim slučajevima ti poligoni razbiju u trokute). Kada pošaljemo upit da li zraka presjeca mesh, mesh općenito samo prolazi kroz petlju i provjerava koji trokut se točno presjeca. 10, 13, 45, 74, 75

OpenCL Aplikacijsko sučelje koje služi kao posrednik između izvršnog programa i grafičke kartice. Fokus mu je na generalnom računarstvu. 68

OpenGL Aplikacijsko sučelje koje služi kao posrednik između izvršnog programa i grafičke kartice. Fokus na računalnoj grafici. 58

OpenGL sjenčar (eng. Shader) Mali program koji se izvršava paralelno u svakoj pojedinoj jezgri grafičke kartice. Postoji više vrsta sjenčara koji obavljaju različite funkcije, te se općenito izvršavaju paralelno u nizu: izlazi svih sjenčara prve vrste služe kao ulazi sjenčarima neke druge vrste. 58

Sjenčar (eng. Shader) U kontekstu algoritma praćenja zraka, sjenčar je algoritam koji osjenčava točku presjeka u koju je zraka udarila. Ulogu sjenčara može imati Phongov algoritam ili neki drugi. 45

Tangencijalna ravnina Tangencijalna ravnina plohe S u nekoj njezinoj regularnoj točki $\mathbf{x}$ je ravnina koja sadrži sve tangente plohe S s diralištem u $\mathbf{x}$. 13

Uzorkovanje po važnosti (eng. importance sampling) je uzorkovanje kod kojeg, pomoću neke pažljivo odabrane ne-uniformne distribucije generiramo uzorke koji su nam "važni", a to su u slučaju ovih algoritama, smjerovi novih zraka koji teže smjerovima izvora svjetlosti u sceni. 91

Popis isječaka kôdova

Popis slika

Bibliografija

[1] *Oak Ridge National Laboratory*, 2021, https://www.olcf.ornl.gov/tutorials/opencl-vector-addition/, Primjeri programa u OpenCL okruženju.

[2] *Scratchapixel*, 2021, https://www.scratchapixel.com/, Imena autora (ako nisu anonimni) se mogu naći na ovom linku: https://docs.google.com/spreadsheets/d/19ljS9fyrRFchaIn_TF8L2YaZw5p89kFqo6QfTad9Av0/edit?usp=sharing.

[3] Tina Bosner, *Računalna grafika - predavanja*, 2021, https://web.math.pmf.unizg.hr/nastava/CG.

[4] Paul Bourke, *Object Files*, http://paulbourke.net/dataformats/obj/.

[5] Bram de Greve, *Reflections and Refractions in Ray Tracing*, https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf.

[6] Joey de Vries, *Learn OpenGL*, https://learnopengl.com/.

[7] Khronos dokumentacija, *GLSL - rekurzije*, https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)#Recursion.

[8] ———, *Primitive u OpenGL-u*, https://www.khronos.org/opengl/wiki/Primitive.

[9] ———, *Teselacijske primitive u OpenGL-u*, https://www.khronos.org/opengl/wiki/Tessellation#Tessellating_primitives.

[10] Alain Galvan, *Advances in Material Models*, 2019, https://alain.xyz/blog/advances-in-material-models, blog Alaina Galvana, AMD software inženjera.

[11] ———, *Raw OpenGL*, 2019, https://alain.xyz/blog/raw-opengl, blog Alaina Galvana, AMD software inženjera.

[12] _____, *Real Time Ray Tracing*, 2019, https://alain.xyz/blog/real-time-ray-tracing, blog Alaina Galvana, AMD software inženjera.

[13] _____, *A Comparison of Modern Graphics APIs*, 2020, https://alain.xyz/blog/comparison-of-modern-graphics-apis, blog Alaina Galvana, AMD software inženjera.

[14] _____, *Ray Tracing Denoising*, 2020, https://alain.xyz/blog/ray-tracing-denoising, blog Alaina Galvana, AMD software inženjera.

[15] Sonja Gorjanc, Ema Jurkin, Iva Kodrnja i Helena Koncul, *DESKRIPTIVNA GEOMETRIJA*, 2023, https://www.grad.hr/geometrija/udzbenik/ (hrvatski).

[16] Wojciech Jarosz, *Lecture notes for Physics-based Rendering ar Dartmouth*, 2019, https://canvas.dartmouth.edu/courses/35073.

[17] _____, *Lecture notes for Physics-based Rendering at Carnegie Mellon University*, 2022, http://graphics.cs.cmu.edu/courses/15-468/.

[18] A. Yoshida J.H. Reif, J.D. Tygar, *Computability and Complexity of Ray Tracing*, Descrete and computational geometry **11** (1994), 265–287, https://users.cs.duke.edu/~reif/paper/tygar/raytracing.pdf (engleski).

[19] James T. Kajiya, *The Rendering Equation*, SIGGRAPH Comput. Graph. **20** (1986), br. 4, 143–150, ISSN 0097-8930, https://doi.org/10.1145/15886.15902 (engleski).

[20] Sam Lapere, *OpenCL path tracing tutorial*, 2016, https://raytracey.blogspot.com/2016/11/opencl-path-tracing-tutorial-2-path.html, blog Sama Laperea, Nvidia RTX inženjera.

[21] Dr. Orion Lawlor, *Recursive Raytracing, on GPU Hardware without Recursion*, https://www.cs.uaf.edu/2012/spring/cs481/section/0/lecture/02_07_recursion_reflection.html.

[22] Greg Humphreys Matt Pharr, Wenzel Jakob, *Physically Based Rendering: From Theory To Implementation*, Morgan Kaufmann, 2016, ISBN 978-0128006450, https://www.pbr-book.org/ (engleski).

[23] Github repozitorij, *Algoritam praćenja puta u OpenCL i OpenGL*, https://github.com/aeoden96-cg/path-tracing-opencl-interop, Pretežno bazirano na člancima Sama Laperea: OpenCL path tracing tutorials.

[24] _____, *Algoritam praćenja puta*, https://github.com/aeoden96-cg/path-tracing, Pretežno bazirano na knjigama Petera Sherleyja Ray Tracing: The Rest of Your Life.

[25] _____, *Algoritam praćenja zrake*, https://github.com/aeoden96-cg/ray-tracing-phong, Pretežno bazirano na skupu Scratchpixel članaka (autori i atribucija u Scratchpixel bibliografiji).

[26] Matthew Scarpino, *OpenCL in Action: How to accelerate graphics and computations*, 2011, https://livebook.manning.com/book/opencl-in-action/chapter-15/.

[27] Peter Shirley, *Ray Tracing: In One Weekend*, 2020, https://raytracing.github.io/books/RayTracingInOneWeekend.html, editori: Steve Hollasch i Trevor David Black.

[28] _____, *Ray Tracing: The Next Week*, 2020, https://raytracing.github.io/books/RayTracingTheNextWeek.html, editori: Steve Hollasch i Trevor David Black.

[29] _____, *Ray Tracing: The Rest of Your Life*, 2020, https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html, editori: Steve Hollasch i Trevor David Black.

[30] Tom Deakin Simon McIntosh-Smith, *Hands On OpenCL dvodnevno predavanje*, 2022, https://handsonopencl.github.io/.

[31] Saša Singer, *Složenost algoritama*, 2005, https://web.math.pmf.unizg.hr/~singer/oaa/materijali/skripta/00.pdf (hrvatski).

[32] Turner Whitted, *An Improved Illumination Model for Shaded Display*, Commun. ACM **23** (1980), br. 6, 343–349, ISSN 0001-0782, https://doi.org/10.1145/358876.358882 (engleski).

Sažetak

Zadnjih par godina, kako je rasla snaga grafičkih kartica, dolazi i pitanje može li se koristiti neki bolji načini za prikaz korektnog osvijetljenja kod prikaza 3D objekata u računalnoj sceni. Tu stupaju na snagu algoritmi globalnog osvijetljenja, konkretno koji rade na principu praćenja zrake. Zrake se ”ispucavaju” u scenu iz svakog piksela i vrati se boja koju taj piksel poprimi. Ti algoritmi su poznati još od 80tih, ali mana im je što zahtjevaju jako puno procesorskog vremena, pa su godinama ostali u sjeni. U zadnjih par godina, na karticama je uz malo truda moguća njihova implementacija, pa time ne moramo više koristiti procesor jer jednostvno možemo zaposliti male, brze, efikasne jezgre u grafičkoj jedinici da paralelno računaju boje tih zraka.

U ovom radu smo istražili koji su to točno algoritmi, analizirali ih te dali implemtacije, na grafičkoj kartici i na CPU, kroz par različitih okolina, te smo istražiti njihove mane i kompleksnosti, i prikazali rezulate renderiranja.

Summary

In the last couple of years, as the power of graphic cards continued to grow, a primary question we should ask is, is there a better way to render correct lightning in 3D computer models? Global illumination algorithms, specifically ray tracing ones, show promise. Rays are shot toward the scene from each pixel, and they return a color that that pixel should be. These algorithms are known since the 80s, but their biggest flaw is their huge computing time, so they stayed in shadow for years. In the last couple of years, it became possible to use graphics cards for that purpose, where we can implement them, with some effort. So now we no longer need to use CPU, but instead use these small fast efficient GPU cores to compute our ray colors, all in parallel. In this thesis, we explored some of these algorithms, analyzed and gave implementations of them on GPU and on CPU through a couple of different environments, and then explored flaws of these algorithms and complexities. Finally, we showed final renders for each implementation.

Životopis

Mateo Martinjak, rođen 2.5.1996 godine u Zagrebu. Pohađao sam srednju školu Ruđera Boškovića (2011-15) gdje sam stekao titulu Tehičara za računarstvo. Slijedno tome, završio sam preddiplomski smjer matematike (2015-20) na PMF - Matematičkom odsjeku. Moje radno iskustvo je započelo 2018. zapošljavanjem u *Mercedes Benz Leasing - Hrvatska*. Tamo sam radio godinu dana kao IT tehičar i developer softvera, te sam u sklopu toga administrirao SQL baze podataka, razvijao Python i Bash skripte za automatiziranje plaćanja, te razvio PHP full-stack aplikaciju za aukcije vozila. U 2021. godini sam se zaposlio u Končar KET-u kao softver developer, gdje sam razvijao SCADA sustave u Pythonu (back-end) i Javascript/snapdom (front-end). 2022. godine (do danas) sam zaposlen u firmi Thespian kao web front-end React i Javascript developer gdje radim na raznim web rješenjima, te u sklopu toga radim u WebGL-u i srodnim okruženjima za rad u 3D grafici.