Prototipni sustav za trasiranje zraka svjetlosti u računalnoj simulaciji

Vuković, Patrik

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Bjelovar University of Applied Sciences / Veleučilište u Bjelovaru**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:144:208566

VELEUČILIŠTE U BJELOVARU
PREDDIPLOMSKI STRUČNI STUDIJ RAČUNARSTVO

PROTOTIPNI SUSTAV ZA TRASIRANJE ZRAKA SVJETLOSTI U RAČUNALNOJ SIMULACIJI

Završni rad br. 13/RAČ/2021.

Patrik Vuković

Bjelovar, travanj 2022.

obrazac ZR - 001

Veleučilište u Bjelovaru

Trg E. Kvaternika 4, Bjelovar

1. DEFINIRANJE TEME ZAVRŠNOG RADA I POVJERENSTVA

Kandidat: **Vuković Patrik** Datum: 13.09.2021. Matični broj: 001997

JMBAG: 0314020007

Kolegij: **RAZVOJ RAČUNALNIH IGARA**

Naslov rada (tema): **Prototipni sustav za trasiranje zraka svjetlosti u računalnoj simulaciji**

Područje: **Tehničke znanosti** Polje: **Računarstvo**

Grana: **Programsko inženjerstvo**

Mentor: **Ante Javor, struč. spec. ing. comp.** zvanje: **predavač**

Članovi Povjerenstva za ocjenjivanje i obranu završnog rada:

1. **dr.sc. Zoran Vrhovski, predsjednik**
2. **Ante Javor, struč.spec.ing.comp., mentor**
3. **Krunoslav Husak, dipl. ing. rač., član**

2. ZADATAK ZAVRŠNOG RADA BROJ: 13/RAČ/2021

Izraditi prototip sustava za trasiranje zraka svjetlosti koji će se koristiti u računalnoj simulaciji. Aplikacija će omogućiti stvaranje scene, pozadine, kamere, kreiranje objekata raznih oblika i boja na sceni. Svi objekti na sceni biti će dio sustava za trasiranja zraka svjetlosti i sudjelovati u fizičkoj interakciji svjetla. Objekti sustava implementirati će osnovne materijale o kojima će ovisiti refrakcija svjetla. Prototipna aplikacija će omogućiti promjenu osnovnih postavki i konfiguraciji scene. Aplikacija će biti napisana u programskom jeziku C++.

Zadatak uručen: 13.09.2021. Mentor: **Ante Javor, struč. spec. ing. comp.**

REPUBLIKA HRVATSKA BJELOVAR VELEUČILIŠTE U BJELOVARU

Sadržaj

1. UVOD

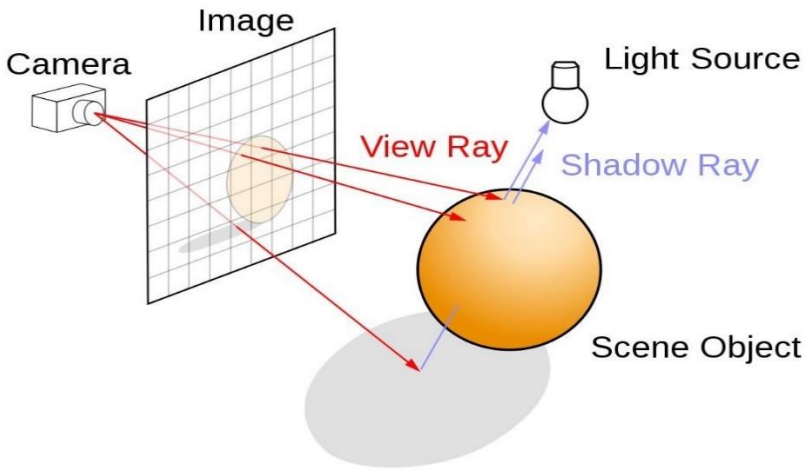
Još otkako je NVIDIA 2018. godine najavila trasiranje zraka svjetlosti na Gamescom konferenciji, trasiranje zraka je postala tema o kojoj se često govorilo. Tehnika trasiranja zraka svjetlosti se niz godina smatrala nedostižnom tehnikom za primjenu u računalnoj grafici. Primarno zbog nedostatnog razvoja hardvera. NVIDIA je omogućila primjenu tehnike trasiranja zraka kako bi postala iskoristiva u praksi. Trasiranje zraka svjetlosti jedno je od važnijih postignuća u području računalne grafike u posljednjih nekoliko godina. No, znajući da trasiranje zraka troši jako puno procesne i grafičke snage računala ne primjenjuje se u potpunosti ni u današnjim video igrama. Istina je da se trasiranje zraka koristi u igrama, ali uz ograničenja. Kroz igre se najčešće vidi djelomično renderiranje, refleksiju i sjene [1].

Ova tehnologija nagovještava svoj utjecaj u budućnosti u području računalne grafike. Zbog toga je cilj ovog rada upoznavanje s pojedinim dijelovima trasiranja zraka svjetlosti. Ovaj rad obuhvaća implementaciju sustava koji će prikazivati trasirane zrake svjetlosti i željene objekte. Prototipni sustav implementiran je pomoću C++ programskog jezika, a izlaz programa prikazan je pomoću prijenosne mape piksela (engl. *Portable Pixel Map, PPM*) za slike u boji.

Prvo poglavlje govori o ispaljivanju i trasiranju zraka svjetlosti. U drugom poglavlju prikazano je i opisano funkcioniranje zraka za trasiranje te njihova izvedba kroz program. Treće poglavlje opisuje detalje o prikazu izlaza programa pomoću PPM formata, opisan je programski kod za kameru i pozadinu programa. Kroz četvrto poglavlje objašnjen je način prepoznavanja objekata pomoću zraka svjetlosti te je opisan postupak kreiranja sjena i više objekata. U petom poglavlju govori se o zaglađivanju rubova kugli. Šesto poglavlje opisuje materijale objekata koji će se koristiti. Sedmo poglavlje bavi se podesivom kamerom i njenim fokusom te prikazom programa s navedenim značajkama. U osmom poglavlju obrađen je umjetni izvor svjetlosti, kvadratni oblici, proizvoljna slika na kugli i njihove primjene kroz programski kod.

2. ISPALJIVANJE, TRASIRANJE I PRIKAZ ZRAKE

Ispaljivanje zraka (engl. *ray-casting*) je proces pronalaženja najbližeg ili ponekad bilo kojeg objekta koji se nalazi na putu zrake. Zraka napušta kameru kroz piksel i putuje dok ne dođe u koliziju s objektom. Ovisno o smjeru odbijanja ispaljene zrake, prema izvoru svijetla ili prema sjeni, dolazi do iscrtavanja objekta i njegovih karakteristika.

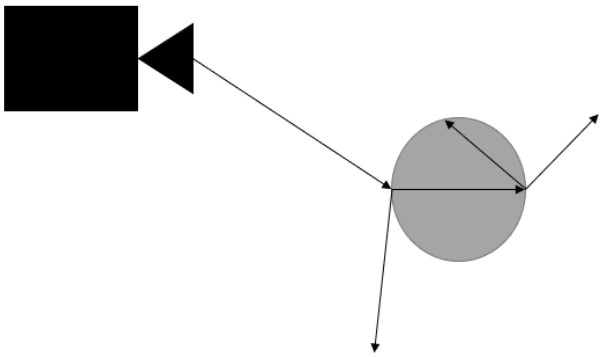


Slika 2.1: Ispaljivanje zrake [2]

Prema slici 2.1 vidi se da zraka putuje od kamere kroz mrežu piksela na scenu. U svakoj točki i na svakom odredištu zraka je ispaljena prema izvoru svjetlosti kako bi se odredilo je li prvobitna dodirna točka zrake i površine objekta osvjetljenja ili u sjeni. Na ovaj način ispaljivanje zraka dovodi do trasiranja zraka.

Trasiranje zraka (engl. *ray-tracing*) koristi mehanizam ispaljivanja zraka za rekurzivno prikupljanje svjetlosti od reflektirajućih objekata i objekata na kojima dolazi do loma svjetlosti. Primjerice, kada se na scenu doda ogledalo, zraka je odbijena od dodirne točke s ogledala u smjeru refleksije. Što god reflektirana zraka presijeca utječe na konačno zasjenjivanje zrcala. Također prozirni i stakleni objekti imaju sličan učinak na zrake. Ove vrste objekta mogu stvoriti reflektirajuće i lomne zrake. Ovaj proces se pojavljuje rekurzivno gdje postoji šansa da sa svakom novom nastalom zrakom dolazi do stvaranja nove reflektirajuće i lomne zrake. Stoga za rekurziju mora biti postavljena granica kao što je maksimalan broj odbijanja zraka. U suprotnom dolazi do beskonačnih iteracija što zagušuje i onemogućuje završavanje zadatka programa [3]. Primjer je vidljiv na slici 2.2. Prilikom dodira ispaljene zrake i površine staklene kugle stvorit će se dvije nove zrake.

Reflektirajuća zraka će se odbiti od staklene kugle u reflektirajućem smjeru dok će lomna proći kroz kuglu i unutar nje stvoriti dvije nove zrake koje će ponavljati postupak.



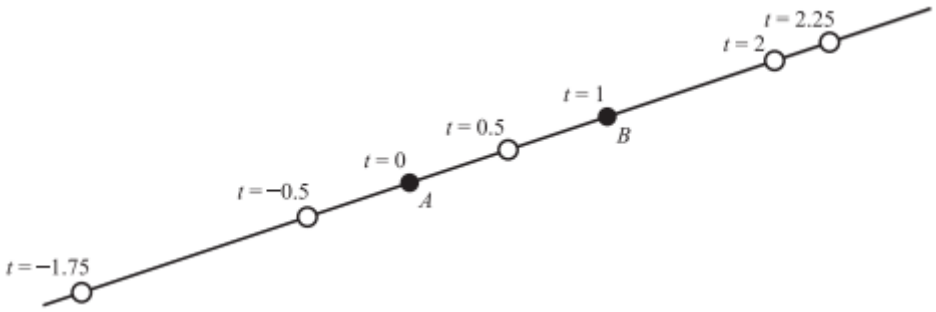
Slika 2.2: Prikaz odbijanja zraka od staklenog objekta

2.1 Zraka

Za trasiranje važna proračunska komponenta je zraka. Zraka se obično navodi kao interval na liniji. Ne postoji implicitno jednadžba za trodimenzionalnu liniju analognu dvodimenzionalnoj koja je zapisana jednadžbom $y = mx + b$, pa se obično koristi parametarski oblik. Parametarska linija može se predstaviti kao ponderirani prosjek točaka A i B što je vidljivo jednadžbom 2.1 [4].

$$p(t) = A + tB. \tag{2.1}$$

U programiranju, ovakav prikaz zamišljen je kao funkciju $p(t)$ koja za ulaz prima argument u obliku realnog broja t i vraća točku p. Za cijelu liniju, parametar može poprimiti bilo koju stvarnu vrijednost. Argument t može biti bilo koja vrijednost u rasponu od $[-\infty, +\infty]$, a točka p se kontinuirano kreće uzduž linije kako se t mijenja. Kako bi se implementiralo ovu funkciju potreban je način za predstavljanje točaka A i B. Može biti upotrijebljen bilo koji koordinatni sustav, ali se gotovo uvijek koristi Kartezijev koordinatni sustav. U sučeljima za programiranje aplikacija (engl. *Application Programming Interface, API*) i programskim jezicima, ovakve se funkcije često nazivaju *vec3* ili *float3* koje prikazuju točku i sadrže tri realna broja x, y i z. Ista linija prikazana je s bilo koje dvije različite točke kroz koje ta linija prolazi. Međutim, odabirom tih različitih točaka dolazi do promijene u rasponu prikaza t vrijednosti.



Slika 2.3 Promjena vrijednosti ti duž zrake [3]

Na slici 2.3 prikazano je kako različite vrijednosti *t* stvaraju različite točke na zraci. *A* predstavlja ishodište zrake, dok je *B* koeficijent smjera kojim se definira smjer u kojem zraka ide. Parametar *t* zrake realan je broj pa je predstavljen kao *float* u programskom kodu. Ako se *t* postavi kao negativan moći će se ići bilo gdje po zraci, a za pozitivan *t* prikazani su samo dijelove ispred ishodišta *A* koje predstavlja kameru. Zbog toga pozitivan dio predstavlja zraku koja će biti vidljiva na sceni.

Programski kod 2.1: Prikaz zrake pomoću p(t) funkcije [4]

```
class ray
{
public:
	ray() {}
	ray(const vec3& a, const vec3& b) { A = a; B = b; }
	vec3 origin() const { return A; }
	vec3 direction() const { return B; }
	vec3 point_at_parameter(float t) const { return A + t * B; }

	vec3 A;
	vec3 B;
};
```

U programskom kodu 2.1 prikazana je funkcija *p*(*t*) koja je implementirana kao vektor, a naziv funkcije je *point_at_parameter*. Srž trasiranja zraka je ispucati zrake kroz piksele i pogoditi objekt te dohvatiti boju objekta kojeg je zraka dotaknula. Programski kod 2.1 računa upravo to. Kroz kameru je poslana zraka od izvora, kada se dogodi dodirna točka zraka mora očitati boju za tu dodirnu točku.

3. PRIKAZ IZLAZA PROGRAMA, KAMERA I POZADINA

3.1 Izlaz programa

Kako bi se mogla prikazati cijela željena scena, u programskom kodu potrebno je izraditi dio programa koji će biti zadužen za to. Kod svake izmjene u programskom kodu potrebno je ispisati izlazni prikaz u nekom formatu. Najjednostavniji način je zapisati prikaz u za to predodređenu datoteku. Problem kod takvog zapisivanja je što postoji mnogo formata kao što su JPEG, TIFF, GIF, BMP, PNG, BAT i drugi. Mnogi od njih su kompleksni, stoga je u ovom slučaju korišten PPM format pomoću kojega je lako interpretirati sliku i ispisati njen izlazni prikaz.

```
P3
800 400
255
217 233 255
217 233 255
217 233 255
217 233 255
217 233 255
217 233 255
217 233 255
217 233 255
```

Slika 3.1: Izgled PPM formata

Na slici 3.1 prikazan je način pohranjivanja obojenih piksela u PPM format. *P3* predstavlja ASCII oznaku da je slika prikazana pomoću RGB boja bez formatiranja u format za datoteke čistog teksta. U drugom retku prvi broj predstavlja širinu, a drugi broj visinu slike prikazanu u broju piksela. Treći redak prikazuje maksimalnu vrijednost koju je moguće zadati za pojedinu boju koja se prikazuje. Preostali zapis unutar PPM formata prikazuje svaki piksel koji je pojedinačno obojen pomoću osnovne tri boje zadane u ovom formatu crvene, zelene i plave, tim redom zapisane u svakom retku.

3.2 Kamera i pozadina

Za početnu kameru postavljena je jednostavna kamera kako bi se uz pokretanje programa moglo vidjeti što on vraća kao izlaz. Postavljena je i jednostavna pozadina koja će vraćati boju. Veličina izlazne slike je postavljena na širinu od 1500 stupaca i visinu od 750 redaka piksela. Izvor zraka odnosno kameru, potrebno je postaviti u početnu točku u ovom slučaju s koordinatama 0, 0, 0. Kamera će biti postavljena prema pravilu desne ruke, dakle y os će gledati prema gore, x os prema desno, a prema ekranu negativna z os. Očitavanje vrijednosti piksela koji čine sliku će se kretati iz donjeg lijevog kuta i uz pomoć dva vektora na krajevima ekrana koji će određivati krajnje točke na ekranu.

Programski kod 3.1: Prikaz jednostavne kamere [4]

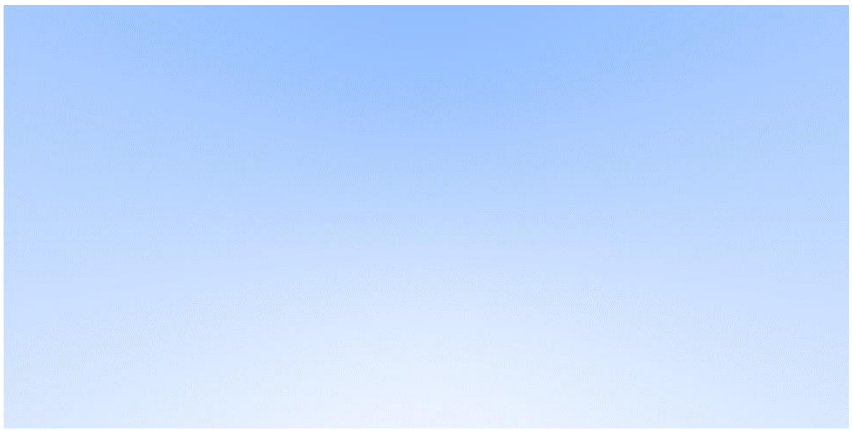
```
int main() {
	int nx = 1500;
	int ny = 750;
	int ns = 100;
	std::cout << "P3\n" << nx << " " << ny << "\n255\n";
	vec3 lower_left_corner(-2.0, -1.0, -1.0);
	vec3 horizontal(4.0, 0.0, 0.0);
	vec3 vertical(0.0, 2.0, 0.0);
	vec3 origin(0.0, 0.0, 0.0);
	for (int j = ny - 1; j >= 0; j--) {
		for (int i = 0; i < nx; i++) {
			float u = float(i) / float(nx);
			float v = float(j) / float(ny);
			ray r(origin, lower_left_corner + u * horizontal +
				v * vertical);
			vec3 col = color(r);
			int ir = int(255.99 * col[0]);
			int ig = int(255.99 * col[1]);
			int ib = int(255.99 * col[2]);

			std::cout << ir << " " << ig << " " << ib << "\n";
		}
	}
}
```

U programskom kodu 3.1 kamera je postavljena u početni položaj, a dva vektora koja će određivati krajnje točke na ekranu su predstavljeni kao *float u* i *float v*. Zraka svjetlosti *r* postavljena je da cilja na približno centar kako bi zrake prikupile boju objekta ako pogode. Mala odstupanja od centra ne predstavlja veliki problem jer će se oštri rubovi slike ublažiti tehnikom zaglađivanja.

Programski kod 3.2: Kreiranje jednostavne pozadine [4]

```
vec3 color(const ray& r, hitable* world, int depth) {
    vec3 unit_direction = unit_vector(r.direction());
    float t = 0.5 * (unit_direction.y() + 1.0);
    return (1.0 - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
}
```

Programski kod 3.2 prikazuje kako funkcija linearno stapa i miješa plavu i bijelu boju ovisno o visini *y* koordinate. Skalirana je vrijednost y da odgovara rasponu $0.0 < t < 1.0$. S ovime je postignuto da $t = 1.0$ daje plavu boju, a $t = 0.0$ daje bijelu. Sve što se nalazi unutar tog raspona stopit će se i pomiješati. To će kreirati linearno stapanje što je kroz programski kod dobiveno uz pomoć formule gdje je vrijednost stapanja jednaka *(1 - t) * početna vrijednost + t * završna vrijednost, t* prati *y* odnosno poprima vrijednosti od nula do jedan što znači da putuje od dna do vrha ekrana i daje izlaz prikazan na slici 3.2.



Slika 4.2: Prikaz jednostavne pozadine

4. STVARANJE I PRIKAZ KUGLE

4.1. Kreiranje kugle

Kao objekt koji će biti kreiran odabrana je kugla, no prvo je potrebno zadati kako zraka pogađa kuglu. Računanje je li zraka pogodila kuglu nije složeno, iz tog razloga se često baš kugla i rabi kao objekt kod trasiranja zraka. S obzirom da se sve formule moraju prikazati pomoću vektora u programskom kodu jednadžba kugle pomoću vektora glasi [6]:

$$dot((p - c),(p - c)) = RR \tag{4.1}$$

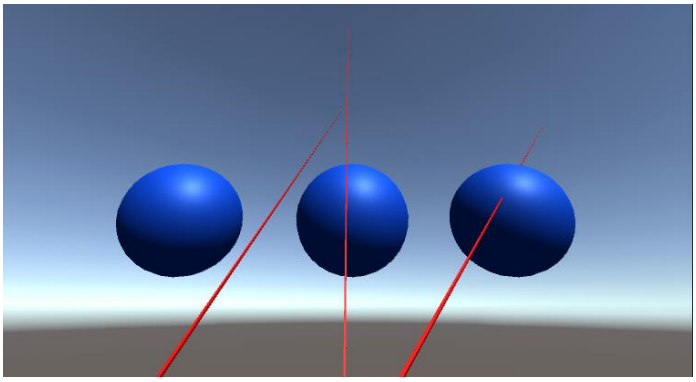
gdje je *dot* skalarni umnožak, a vektor iz centra *c* do točke pogotka *p* jednak je (*p* - *c*). Ovaj izraz protumačen je kao bilo koja točka *p* koja zadovoljava jednadžbu nalazi se na kugli. Potrebno je znati hoće li zraka $p(t) = A + tB$ ikada pogoditi kuglu. Ako dolazi do pogotka kugle to znači da postoji određeni *t* za koji $p(t)$ zadovoljava jednadžbu kugle. Dakle potreban je bilo koji *t* za koji je to istinito:

$$dot((p(t) - c),(p(t) - c)) = RR \tag{4.2}$$

ako se potpuno proširi oblik zrake $p(t)$ i uredi jednadžba dobiva se:

$$ttdot(B,B) + 2tdot(A - C, A - C) + dot(C, C) – RR = 0 \tag{4.3}$$

vektori i *R* u jednadžbi su konstante i poznanice. Ono što je nepoznato je *t* do kojeg je moguće doći pomoću kvadratne jednadžbe. Ukoliko postoje dva realna rješenja ona su točke u kojima zraka sječe kuglu, ukoliko je rješenje negativan broj to znači da ne postoje realna rješenja odnosno nema dodirnih točaka između zrake i kugle te ako je rješenje nula znači da postoji jedno realno rješenje i zraka je tangenta zadanoj kugli.

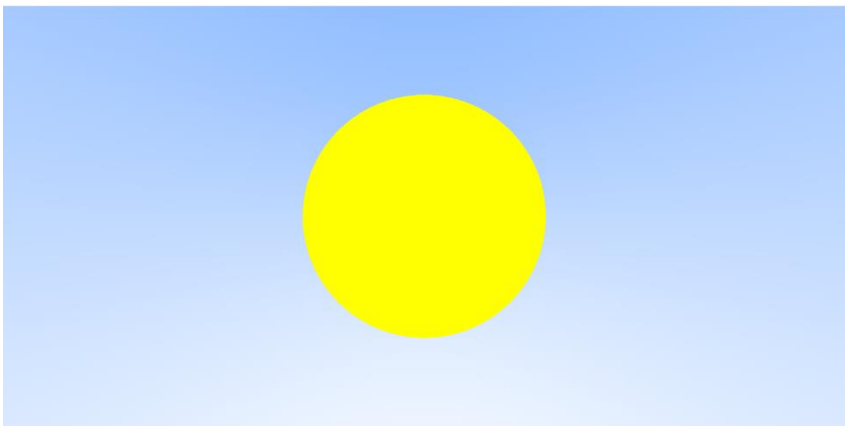


Slika 4.1: Moguća sjecišta zrake i kugle [6]

Programski kod 4.1: Pronalaženje i bojanje dodirnih točaka zrake i kugle

```
bool hit_sphere(const vec3& center, float radius, const ray& r) {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = 2.0 * dot(oc, r.direction());
    float c = dot(oc, oc) - radius * radius;
    float discriminant = b * b - 4 * a * c;
    return (discriminant > 0);
}

vec3 color(const ray& r) {
    if (hit_sphere(vec3(0, 0, -1), 0.5, r))
        return vec3(1, 1, 0);
    vec3 unit_direction = unit_vector(r.direction());
    float t = 0.5 * (unit_direction.y() + 1.0);
    return (1.0 - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
}
```

U programskom kodu 4.1 zapisana je prethodna matematička jednadžba 4.3. Jednadžba je testirana tako što za svaki piksel koji pogodi zraka postavljen je u žutu boju, uz to pikseli koji se nalaze na kugli po z osi su postavljen na 1. Ovakav oblik prikaza je nepotpun jer nedostaju sjene, odsjaji i više objekata na sceni pa kugla izgleda kao krug kao što je vidljivo sa slike 4.2.



Slika 4.2: Prikaz kugle

4.2. Površinska normala i sjene

Prioritet je napraviti površinsku normalu kako bi se mogla prikazati sjena na kugli kroz programski kod. Površinska normala je vektor koji je okomit na površinu i usmjeren je prema van. Za kuglu vrijedi da je površinska normala u smjeru točke sudara umanjena za centar zadane kugle, odnosno smjer normale je od točke P u smjeru $P - C$ gdje je P točka na površini, a C centar kugle. Ako je za primjer korištena zemlja kao kugla i zamišljena točka na njoj kao točka na kugli to bi značilo da vektor iz zemljinog središta do zamišljene točke pokazuje ravno prema gore te prolazi kroz nju i nastavlja se u istom smjeru. Uz pomoć normale može se osjenčati kuglu, a s obzirom na to da u trenutnom programskom kodu nema izvora svjetlosti, normala je vizualizirana pomoću boja. Treba zapisati svaku komponentu na intervalu od nula do jedan i zapisati koordinate *x, y, z* u boje *r, g, b.* Za normalu je sada bitno gdje je zraka pogodila kuglu, a ne samo je li kugla pogođena ili ne. Za najbliži udarac postavljen je na najmanji *t* kao što je prikazano u programskom kodu 4.2.

Programski kod 4.2: Kreiranje površinske normale i sjene [4]

```
double hit_sphere(const vec3& center, double radius, const ray& r) {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = 2.0 * dot(oc, r.direction());
    float c = dot(oc, oc) - radius * radius;
    float discriminant = b * b - 4 * a * c;
    if (discriminant < 0) {
        return -1.0;
    }
    else {
        return (-b - sqrt(discriminant)) / (2.0 * a);
    }
}

vec3 color(const ray& r) {
    float t = hit_sphere(vec3(0, 0, -1), 0.5, r);
    if (t > 0.0) {
        vec3 N = unit_vector(r.point_at_parameter(t) - vec3(0, 0, -1));
        return 0.5 * vec3(N.x() + 1, N.y() + 1, N.z() + 1);
    }
    vec3 unit_direction = unit_vector(r.direction());
    t = 0.5 * (unit_direction.y() + 1.0);
    return (1.0 - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
}
```

Kako bi se izradilo više objekata koji će sudjelovati u trasiranju zraka potrebna je klasa koja će sadržavati funkciju za dohvaćanje zrake. Za to će biti zaslužan interval za pogotke od *tmin* do *tmax*. Kako bi se pogodak računao mora vrijediti da je *tmin* $< t <$ *tmax*. Za ispaljene zrake to je pozitivan *t* u određenom rasponu koji ako je zadovoljen prikazuje objekt. Kako bi se moglo koristiti više objekata u simulaciji trasiranja zraka svjetlosti potrebno je implementirati nekoliko komponenti. Apstraktnu klasu koja će prihvaćati zrake i prepoznati sve što bi te zrake mogle pogoditi, klasu za kugle te listu kugli koje će moći biti pogođene zrakom.

Programski kod 4.3: Klasa za prihvaćanje zraka [4]

```
struct hit_record
{
    float t;
    vec3 p;
    vec3 normal;
};

class hitable {
public:
    virtual bool hit(const ray& r, float t_min,
                     float t_max, hit_record& rec) const = 0;
};
```

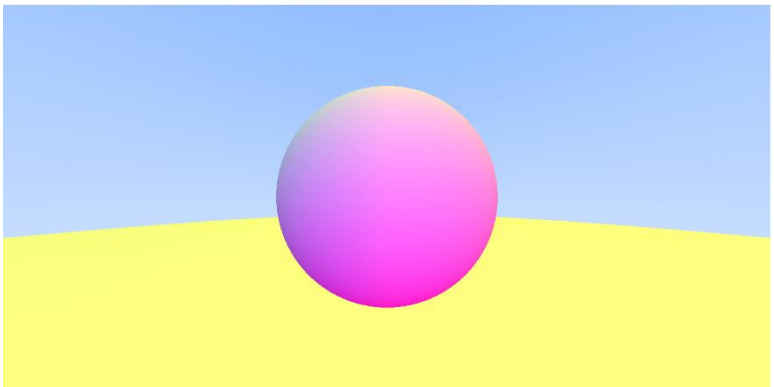
Programski kod 4.4: Klasa za listu objekata koji mogu biti pogođeni [4]

```
class hitable_list : public hitable {
public:
    hitable_list() {}
    hitable_list(hitable** l, int n) { list = l; list_size = n; }
    virtual bool hit(const ray& r, float tmin, float tmax,
                     hit_record& rec) const;
    hitable** list;
    int list_size;
};

bool hitable_list::hit(const ray& r, float t_min,
                       float t_max, hit_record& rec) const {
    hit_record temp_rec;
    bool hit_anything = false;
    double closest_so_far = t_max;
    for (int i = 0; i < list_size; i++) {
        if (list[i]->hit(r, t_min, closest_so_far, temp_rec)) {
            hit_anything = true;
            closest_so_far = temp_rec.t;
            rec = temp_rec;
        }
    }
    return hit_anything;
}
```

Programski kod 4.5: Klasa za kugle

```
class sphere : public hitable {
public:
    sphere() {}
    sphere(vec3 cen, float r) : center(cen), radius(r){};
    virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
    vec3 center;
    float radius;
};
bool sphere::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = dot(oc, r.direction());
    float c = dot(oc, oc) - radius * radius;
    float discriminant = b * b - a * c;
    if (discriminant > 0) {
        float temp = (-b - sqrt(discriminant)) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            return true;
        }
        temp = (-b + sqrt(discriminant)) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.point_at_parameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            return true;
        }
    }
    return false;
}
```

Nakon implementacije prethodno navedenog programskog koda, moguće je unutar programa dodavati liste s novim kuglama koje se mogu prilagođavati i pomicati prema želji. Naspram slike 4.2 na slici 4.3 vidljiva je dubina same slike čemu su doprinijele površinska normala i sjena.



Slika 4.3:Prikaz kugle s dodanom površinskom normalom i sjenom

5. ZAGLAĐIVANJE NAZUBLJENOSTI

Zaglađivanje nazubljenosti (engl. *anti-aliasing*) je metoda koja se koristi u obradi digitalne grafike za zaglađivanje linija i uklanjanje vizualnih izobličenja koja se javljaju kada se slika visoke razlučivosti prikazuje u nižoj razlučivosti. Nazubljenosti (engl. *aliasing*) se očituje kao oštre linije poput stepenica na rubovima i predmetima koji bi inače trebali biti glatki. *Anti-Aliasing* čini ove zakrivljene i kose linije ponovno glatkima tako što dodaje blagu promjenu boje rubova linija ili objekta, uzrokujući da se nazubljeni rubovi zamute i bolje stope s ostatkom objekta i pozadinom kojoj pripadaju. Na posljetku kada se ljudskim okom gleda udaljenu sliku u prvobitnom izdanju više se ne može primijetiti blagu promjenu boje i zamućivanje koje stvara *anti-aliasing* [7].

Jedna od komponenti koja je potrebna je generator nasumičnih brojeva koji vraća realne nasumične brojeve u rasponu od nula uključujući nulu do jedan koji nije uključen. Unutar C++ moguće je doći do *drand48()* funkcije koja je prvobitno bila idealna za trasiranje zraka no zahtijevala je previše vremena kod svakog pokretanja programa. Iz tog razloga napravljena je nova funkcija koja je značajno poboljšala brzinu izvođenja i svakog pokretanja, a naziva se *frand()* [8].

Za svaki proslijeđeni piksel kreira se nekoliko uzoraka unutar pojedinog piksela i šalju se zrake kroz svaki uzorak kako bi površina svake poslane zrake kroz uzorak poprimila nijansu susjednog piksela. Boje i nijanse tih uzoraka su zatim pomiješane kako bi se dobilo novu nijansu svakog takvog piksela. Kako bi se to sve lakše i preglednije napravilo kreirana je klasa za kameru u koju je prebačen programski kod iz glavnog dijela programa vezan uz nju. Unutar klase riješeno je poravnanje osi kamere od prije, što se može i vidjeti u programskom kodu 5.1.

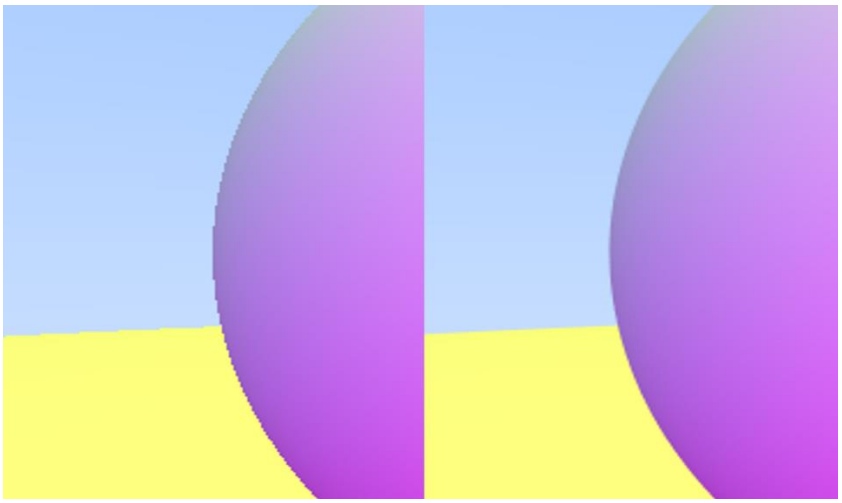
Programski kod 5.1: Klasa kamere [4]

```
class camera {
public:
    camera() {
        lower_left_corner = vec3(-2.0, -1.0, -1.0);
        horizontal = vec3(4.0, 0.0, 0.0);
        vertical = vec3(0.0, 2.0, 0.0);
        origin = vec3(0.0, 0.0, 0.0);
    }
    ray get_ray(float u, float v) {
        return ray(origin, lower_left_corner +
                   u * horizontal + v * vertical - origin);
    }

    vec3 origin;
    vec3 lower_left_corner;
    vec3 horizontal;
    vec3 vertical;
};
```

Programski kod 5.2: Dodavanje frand() funkcije i nove kamere

```
int main()
{
    int nx = 1500;
    int ny = 750;
    int ns = 100;
    cout << "P3\n" << nx << " " << ny << "\n255\n";
    hitable* list[2];
    list[0] = new sphere(vec3(0, 0, -1), 0.5);
    list[0] = new sphere(vec3(0, -100.5, -1), 100);
    hitable* world = new hitable_list(list, 2);
    camera cam;
    for (int j = ny - 1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            vec3 col(0, 0, 0);
            for (int s = 0; s < ns; s++) {
                float u = float(i + frand()) / float(nx);
                float v = float(j + frand()) / float(ny);
                ray r = cam.get_ray(u, v);
                col += color(r, world, 0);
            }
            col /= float(ns);
            int ir = int(255.99 * col[0]);
            int ig = int(255.99 * col[1]);
            int ib = int(255.99 * col[2]);

            cout << ir << " " << ig << " " << ib << "\n";
        }
    }
}
```

Nakon primjene programskog koda prikazanog u programskom kodu 5.1 i 5.2 vidljiva je razlika u rubovima između prijašnjeg izlaza i novonastalog izlaza kao što je prikazano na slici 5.1.



Slika 5.1: Lijeva kugla prikazana je bez tehnike za zaglađivanje nazubljenosti, a desna s njom

6. KUGLE RAZLIČITIH MATERIJALA

6.1 Difuzna kugla

Objekti načinjeni od difuznog materijala ne emitiraju svjetlost kao ostali objekti, već preuzimaju boju iz okoline poput pozadine ili nebeskog svjetla i moduliraju tu boju s njihovom osnovnom bojom. To rade tako da mijenjaju amplitudu ili frekvenciju elektromagnetskog vala ili druge oscilacije u skladu s varijacijama tog drugog signala, obično niže frekvencije, od kojeg preuzimaju boju. Difuzni odraz svjetlosti s površine funkcionira tako da se zrake koje padaju na površinu u istu točku raspršuju pod različitim kutovima, a ne samo pod jednim kutom kao što je to slučaju kod zrcalne refleksije [9]. Idealnom difuznom reflektirajućom površinom se smatra ona koja pokazuje Lambertov odraz, što znači da postoji jednak udio svjetlosti kada se objekt gleda iz svih smjerova koji leže uz površinu. Također više svjetlosti se apsorbira nego reflektira, to se još naziva i slabljenjem svjetlosti što je uzrokovano raspršivanjem svjetlosti prilikom odbijanja od difuznog materijala. Tamnija površina upija više svjetlosti kao što je to slučaj kod Vantablack tvari koja je do sada najtamnija umjetna tvar i upija do 99.965 posto zračenja u vidljivom spektru [9].

Kako bi se prikazao difuzni materijala za zraku koja dođe u doticaj s takvim materijalom mora se generirati slučajni smjer za prikaz reflektirajuće zrake. To se postiže tako da se vektor slučajno odabrane vrijednosti doda na postojeći normalan vektor za reflektirajuću zraku [9].

$$R = P + \vec{N} + \vec{E}. \tag{6.1}$$

P predstavlja točku u kojoj zraka pogađa kuglu. Vektor *N* je normala na točku pogotka kugle *P*, a vektor *E* je vektor slučajno odabrane vrijednosti. Na kraju je dobivena točka *R* koja je smjer odbijene zrake. Kako bi se odabralo slučajnu vrijednost treba odabrati nasumičnu točku u jediničnoj kocki gdje su koordinate x, y, z u rasponu od -1 do 1. Točka je odbačena i ponovno postavljena sve dok ne dobije jedna koja je unutar kugle. Ovo je postignuto uz pomoć programskog koda 6.1.

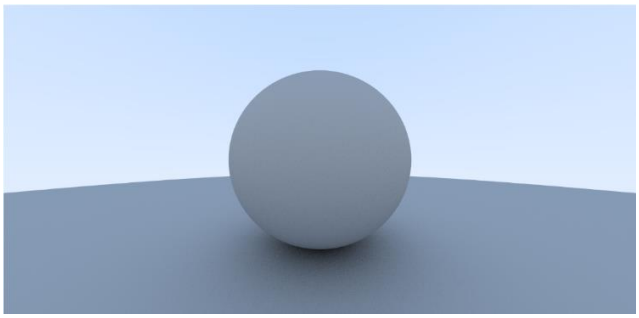
Programski kod 6.1: Dodavanje difuznog materijala

```
vec3 random_in_unit_sphere() {
    vec3 p;
    do {
        p = 2.0 * vec3(randf(), randf(), randf()) - vec3(1, 1, 1);
    } while (p.squared_length() >= 1.0);
    return p;
}
vec3 color(const ray& r, const hittable* world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.0, FLT_MAX, rec)) {
        vec3 target = rec.p + rec.normal + random_in_unit_sphere();
        return 0.5 * ray_color(ray(rec.p, target - rec.p), world);
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5 * (unit_direction.y() + 1.0);
        return (1.0 - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
    }
}
```

Nakon programskog koda 6.1 izlazna slika je bila jako tamna te se nije mogla jasno razaznati sjena. To se dešavalo jer je kugla apsorbirala polovinu energije svakog odbijanja, dakle 50% refleksija. Kugla bi trebala biti svijetlo-sive boje, no problem nastaje kod obrade slike. Pretpostavljeno je da je slika prilagođena bojama monitora odnosno gama ispravljena, što znači da su 0 i 1 vrijednosti transformirane prije nego su pohranjene u bajt. Kako bi se to ispravilo za boju je korjenovana svaka vrijednost vektora što je dalo željeni izlaz.

Programski kod 6.2: Podešavanje svjetline za difuzne objekte [4]

```
col /= float(ns);
col = vec3(sqrt(col[0]), sqrt(col[1]), sqrt(col[2]));
int ir = int(255.99 * col[0]);
int ig = int(255.99 * col[1]);
int ib = int(255.99 * col[2]);
cout << ir << " " << ig << " " << ib << "\n";
```



Slika 6.1: Prikaz difuzne kugle

6.2 Metalna kugla

Pošto je sada riječ ne samo o jednom materijalu nego o više njih potrebna je reorganizacija i kreiranje dodatnih klasa kako bi se postavke pojedine kugle mogle lakše mijenjati. Dakle, mora se kreirati način da program prepozna je li kreirana raspršena zraka ili je sama ispaljena zraka apsorbirana. Ako je riječ o raspršenoj mora se zadati koliko je raspršena zraka oslabljena kako bi se u konačnici dobili razni izgledi metalnih objekata kao što su glatki s odsjajem i hrapaviji bez odsjaja ili s jako slabim odsjajem. Zbog toga je prvo kreirana klasa za materijale.

Programski kod 6.3: Prikaz klase material [4]

```
class material {
public:
    virtual bool scatter(
         const ray& r_in, const hit_record& rec, vec3&
attenuation,
         ray& scattered) const = 0;
};
```

Programski kod 6.4: Prikaz klase hitable [4]

```
class material;

struct hit_record
{
    float t;
    vec3 p;
    vec3 normal;
    material* mat_ptr;
};

class hitable {
public:
    virtual bool hit(
        const ray& r, float t_min, float t_max,
        hit_record& rec) const = 0;
};
```

Prema programskom kodu 6.4 se vidi da je dodan *hit_record* unutra klase *hitable*, to je napravljeno kako bi se mogli potrebni argumenti i informacije pohraniti unutar strukture. Pošto je potrebno da klasa *material* i klasa *hitable* znaju jedna za drugu, klasa *material* je dodana ovdje.

Programski kod 6.5: Prikaz klase lambertian [4]

```
class lambertian : public material {
public:
    lambertian(const vec3& a) : albedo(a) {}
    virtual bool scatter(const ray& r_in, const hit_record& rec,
                         vec3& attenuation, ray& scattered) const {
        vec3 target = rec.p + rec.normal + random_in_unit_sphere();
        scattered = ray(rec.p, target - rec.p);
        attenuation = albedo;
        return true;
    }

    vec3 albedo;
};
```

Lambertian klasa koja je tako nazvana zbog istoimenog zakona za refleksiju svijetlosti iz programskog koda 6.5 predstavlja klasu za difuzne objekte. Unutar ove klase mogu se dogoditi tri stvari. Može uvijek doći do raspršivanja zraka izvora svjetlosti koje će se umanjiti svojom refleksijom *R*, može doći do raspršivanja, ali ne umanjivanja svojom refleksijom nego apsorbiranjem dijela 1-*R* zrake ili može biti mješavina ta dva uvjeta.

Programski kod 6.6: Prikaz koda za odbijanje zraka od metalnih materijala [4]

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

Za predmete s glatkom površinom zrake neće biti nasumično odbijene, nego se mora vidjeti kako će se odbijati od metalnih uglačanih predmeta koji funkcioniraju kao ogledalo. Taj problem riješen je s programskim kodom 6.6. Putanja početne zrake odnosno vektora *v* u ovom prikazu zadržava isti smjer i nakon udarca u površinu predmeta, *N* je i dalje jedinični vektor, *B* je zamišljena udaljenost od izvora zrake do točke udarca, odnosno *B* je skalarni umnožak vektora *v* i *N*. Dakle, odbijena zraka je (*v* + *2B*), ali pošto *v* pokazuje prema unutra to jest ide unutar kugle predznak je minus te to daje konačni izraz koji je prikazan u programskom kodu 6.6.

S obzirom na to da je ovim programskim kodom riješeno odbijanje zraka od metalnih materijala može se kreirati klasa koja računa i prikazuje takve objekte na sceni kao što je prikazano u programskom kodu 6.7.

Programski kod 6.7: Prikaz klase za kreiranje metalnih objekata [4]

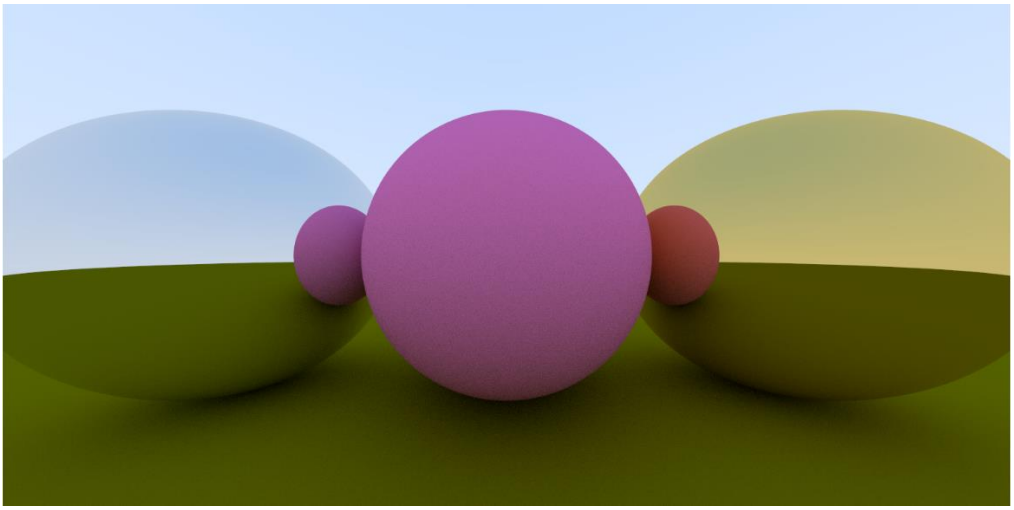
```
class metal : public material {
public:
    metal(const color& a) : albedo(a) {}
    virtual bool scatter(
        const ray& r_in, const hit_record& rec,
        vec3& attenuation, ray& scattered) const {
        vec3 reflected = reflect(unit_vector(r_in.direction()), rec.normal);
        scattered = ray(rec.p, reflected);
        attenuation = albedo;
        return (dot(scattered.direction(), rec.normal) > 0);
    }
    vec3 albedo;
};
```

Potrebno je također prilagoditi funkciju za bojanje s obzirom na to da se ovdje više ne traži nasumično odbijanje, bojanje i difuzni materijal. Dodana je zraka za raspršivanje i vektor za slabljenje pomoću kojih se dobiva realističan izgled metalnog objekta.

Programski kod 6.8: Prilagođena funkcija za bojanje

```
vec3 color(const ray& r, const hittable* world, int depth) {
    hit_record rec;
    if (world->hit(r, 0.001, FLT_MAX, rec)) {
        ray scattered;
        vec3 attenuation;
        if (depth < 50 && rec.mat_ptr->scatter(r, rec, attenuation, scattered))
{
            return attenuation * color(scattered, world, depth + 1);
        }
        else {
            return vec3(0, 0, 0);
        }
    }
    else {
        vec3 unit_direction = unit_vector(r.direction());
        float t = 0.5 * (unit_direction.y() + 1.0);
        return (1.0 - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
    }
}
```

Sada su kreirane dvije vrste materijala, a to su difuzni i metalni uglačani. Posljednje što je preostalo je promijeniti glavni dio programa gdje se dodaju liste s novim objektima i kreirati novi izlaz kao što je prikazano na slici 6.2.



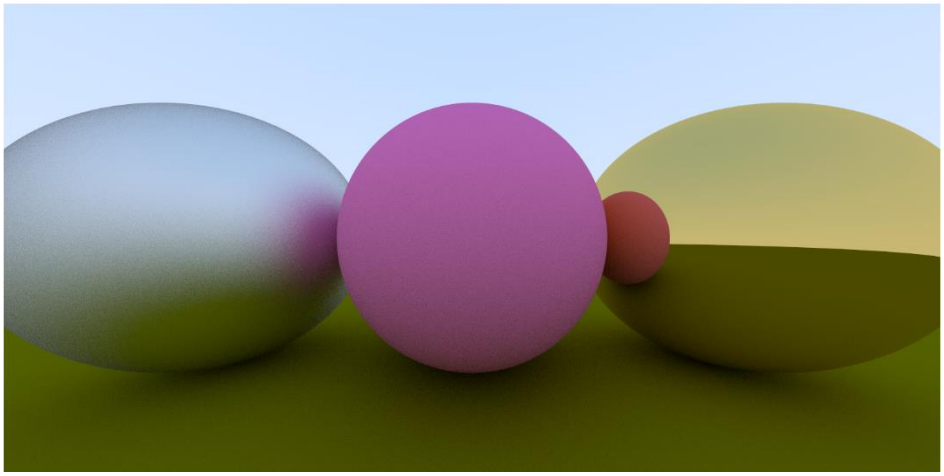
Slika 6.2: Metalne uglačane i difuzna kugla

Kako bi se napravio realističniji prikaz metalne kugle koja nije uglačana, postavlja se nasumičan smjer odbijene zrake unutar određenog radijusa koji će stvoriti novu krajnju točku u kojoj će ta odbijena zraka završiti. Što je veća kugla, veće će biti zamućenje odraza odbijene zrake. Problem nastaje kod velikih kugli gdje se zrake mogu raspršiti ispod površine tla radi blizine kugle i površine što dovodi do zrake koja odlazi u beskonačnost jer se nema od čega više odbiti. Taj problem će se riješiti tako što će površina biti postavljena da apsorbira takve zrake. Maksimalno zamućenje je postavljeno na 1.0 što znači da će kugle bez zamućenja odnosno sa zamućenjem 0.0 biti metalne uglačane kugle.

Programski kod 6.9: Postavljanje zamućenja na metalne objekte

```
class metal : public material {
public:
    metal(const vec3& a, float f) : albedo(a) {
        if (f < 1) fuzz = f; else fuzz = 1;
    }
    virtual bool scatter(const ray& r_in, const hit_record& rec,
        vec3& attenuation, ray& scattered) const
    {
        vec3 reflected = reflect(unit_vector(r_in.direction()),
rec.normal);
        scattered = ray(rec.p, reflected + fuzz * random_in_unit_sphere());
        attenuation = albedo;
        return (dot(scattered.direction(), rec.normal) > 0);
    }
    vec3 albedo;
    float fuzz;
};
```

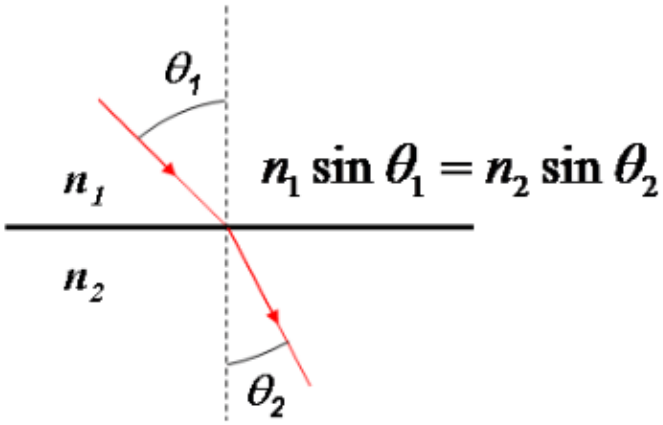
Nakon što je dodano zamućenje od 0.3 i 1.0 na metalne objekte dobiven je prikaz sa slike 6.3.



Slika 6.3: Prikaz zamućenih metalnih kugli

6.3 Staklena kugla

Prozirni materijali poput vode, stakla ili dijamanta funkcioniraju tako da kada ih svjetlosna zraka pogodi razdijeli se na odbijenu i prelomljenu ili prijenosnu zraku. Ovo će se riješiti nasumičnim odabirom između odbijene i prelomljene zrake te pritom stvarajući samo jednu raspršenu zraku po interakciji. Snellov zakon za prelomljene zrake kaže da je lomljenje savijanje putanje svjetlosnog vala pri prolasku preko granice koja razdvaja dva medija te da je lomljenje uzrokovano promjenom amplitude koju doživljava val pri promjeni medija [11].



Slika 6.4: Snellov zakon [10]

Sa slike 6.4 može se iščitati formula za lom zraka svjetlosti. n_1 i n_2 su indeksi loma, a indeks loma zraka je 1, stakla od 1.3 do 1.7, dok je za dijamant 2.4.

Jedan od problema do kojeg dolazi kada je zraka unutar materijala s većim indeksom loma, ne postoji stvarno rješenje za Snellov zakon pa iz tog razloga ni lom nije moguć. To je razlog zašto voda i zrak katkad izgledaju kao savršeno ogledalo ako se gledaju pod određenim kutom i ta se pojava još naziva potpunom refleksijom. Zbog toga je programski kod za lom nešto kompleksniji nego za odbijanje zraka svjetlosti.

Programski kod 6.10: funkcija za lom svjetlosti [4]

```
bool refract(const vec3& v, const vec3& n, float ni_over_nt, vec3&
refracted) {
    vec3 uv = unit_vector(v);
    float dt = dot(uv, n);
    float discriminant = 1.0 - ni_over_nt * ni_over_nt * (1 - dt * dt);
    if (discriminant > 0) {
        refracted = ni_over_nt * (uv - n * dt) - n * sqrt(discriminant);
        return true;
    }
    else
        return false;
}
```

U programskom kodu 6.10 *ni_over_nt* je n_1/n_2, *v* prikazuje smjer upada zrake, *n* je površinska normala, a *refracted* je smjer zrake. Kada je *discriminant* veća od 0 postoji lomna zraka r. Kada je *discriminant* manja od 0 dolazi do potpune refleksije i nema lomne zrake, odnosno zraka će biti ponovno odražena u objekt. Kada je *discriminant* jednaka 0 to je granica totalne refleksije, to znači da je lomna zraka okomita na površinsku normalu i tada ne dolazni do loma niti odbijanja zrake svjetlosti.

Programski kod 6.11: Primjena loma na staklene objekte [4]

```
class dielectric : public material {
public:
    dielectric(float ri) : ref_idx(ri) {}
    virtual bool scatter(const ray& r_in, const hit_record& rec,
        vec3& attenuation, ray& scattered) const
{
        vec3 outward_normal;
        vec3 reflected = reflect(r_in.direction(), rec.normal);
        float ni_over_nt;
        attenuation = vec3(1.0, 1.0, 1.0);
        vec3 refracted;
        if (dot(r_in.direction(), rec.normal) > 0) {
            outward_normal = -rec.normal;
            ni_over_nt = ref_idx;
        }
        else {
            outward_normal = rec.normal;
            ni_over_nt = 1.0 / ref_idx;
        }
        if (refract(r_in.direction(), outward_normal, ni_over_nt, refracted))
            scattered = ray(rec.p, refracted);
        else
            scattered = ray(rec.p, reflected);
        }
        return true;
    }

    float ref_idx;
};
```

Programski kod 6.11 prikazuje lom svjetla na staklenim objektima. S obzirom na to da stakleni objekti uvijek imaju lomne zrake dok je to moguće slabljenje jačine zrake svjetlosti je postavljeno na 1, odnosno staklene površine ne upijaju ništa. Nadalje, pravo staklo ima drugačiji lom kada se gleda kroz njega pod različitim kutovima. Za izračun toga postoji jednadžba, ali postoji i pojednostavljena polinomska aproksimacija koju je osmislio Christophe Schlick[11].

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5 \tag{6.2}$$

Gdje je:

$$R_0 = ((n_1 - n_2) / (n_1 + n_2))^2 \tag{6.3}$$

U formuli 6.3 θ predstavlja upadni kut, odnosno kut između smjera iz kojeg dolazi upadna svjetlost i normale površine između dva medija. n_1 i n_2 su indikatori loma dva medija i R_0 je koeficijent loma za svjetlost koja dolazi paralelno s normalom, odnosno vrijednost kada je θ jednaka 0 ili dolazi do minimalnog odraza.

Programski kod 6.12: Schlickova funkcija [4]

```
float schlick(float cosine, float ref_idx) {
    float r0 = (1 - ref_idx) / (1 + ref_idx);
    r0 = r0 * r0;
    return r0 + (1 - r0) * pow((1 - cosine), 5);
}
```

Sada je moguće upotpuniti klasu za staklene objekte.

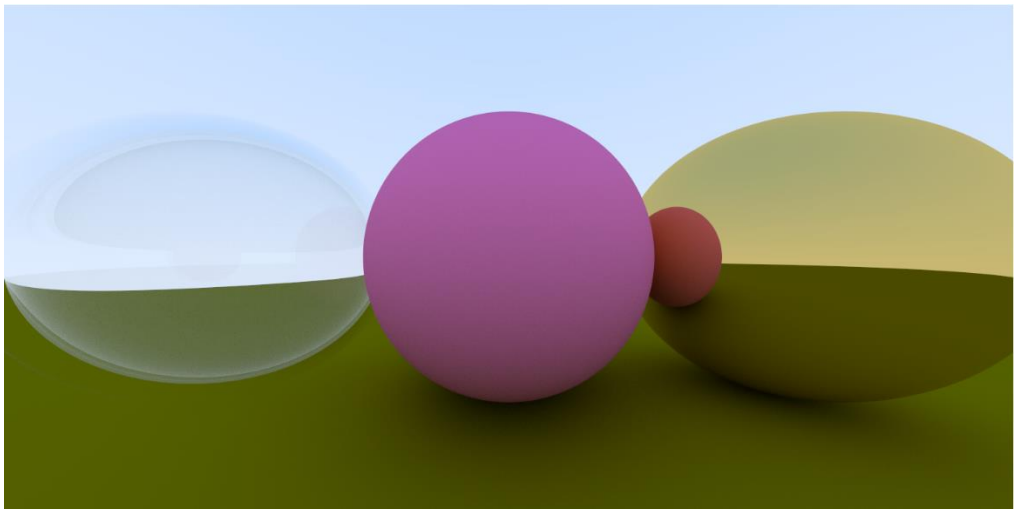
Programski kod 6.13: Upotpunjena klasa za lom svjetlosti nad staklenim objektima [4]

```
vec3 outward_normal;
vec3 reflected = reflect(r_in.direction(), rec.normal);
float ni_over_nt;
attenuation = vec3(1.0, 1.0, 1.0);
vec3 refracted;
float reflect_prob;
float cosine;
if (dot(r_in.direction(), rec.normal) > 0) {
    outward_normal = -rec.normal;
    ni_over_nt = ref_idx;
    cosine = ref_idx * dot(r_in.direction(), rec.normal) /
r_in.direction().length();
}
else {
    outward_normal = rec.normal;
    ni_over_nt = 1.0 / ref_idx;
    cosine = -dot(r_in.direction(), rec.normal) / r_in.direction().length();
}
if (refract(r_in.direction(), outward_normal, ni_over_nt, refracted)) {
    reflect_prob = schlick(cosine, ref_idx);
}
else {
    reflect_prob = 1.0;
}
if (randf() < reflect_prob) {
    scattered = ray(rec.p, reflected);
}
else {
    scattered = ray(rec.p, refracted);
}
```

Može se primijeniti trik sa staklenim kuglama gdje ako je korišten negativan polumjer, geometrija objekta ostaje nepromijenjena, ali onda površinska normala gleda prema unutra. Ovako se može koristiti kao mjehurić za izradu šuplje staklene kugle kao što je prikazano programskim kodom 6.14 i slikom 6.5.

Programski kod 6.14: Prikaz izrade staklene kugle

```
list[0] = new sphere(vec3(0, 0, -1), 0.5, new lambertian(vec3(0.8, 0.2, 0.5)));
list[1] = new sphere(vec3(0, -100.5, -1), 100, new lambertian(vec3(0.2, 0.2, 0.0)));
list[2] = new sphere(vec3(1, 0, -1), 0.5, new metal(vec3(0.8, 0.6, 0.2), 0.0));
list[3] = new sphere(vec3(-1, 0, -1), 0.5, new dielectric(1.5));
list[4] = new sphere(vec3(-1, 0, -1), -0.45, new dielectric(1.5));
```



Slika 6.5: Izlaz programa s prikazanom staklenom kuglom

7. PODESIVI POLOŽAJ I FOKUS KAMERE

7.1 Podesivi položaj kamere

Radi prikaza veće slike iz raznih kutova s više objekata moramo kreirati kameru s podesivim položajem. Kut trenutnog vidnog polja je sve što je vidljivo na dosadašnjoj slici. S obzirom na to da slika nije kvadrat vidno polje je drugačije s obzirom na vertikalnu i horizontalnu os. U ovom slučaju korištena je vertikalna os za vidno polje. Specifikacija će biti kroz stupnjeve, a kroz konstruktor pretvarat će se u radijane radi lakšeg zapisa i brzine programa. Nova kamera prikazana je programskim kodom 7.1.

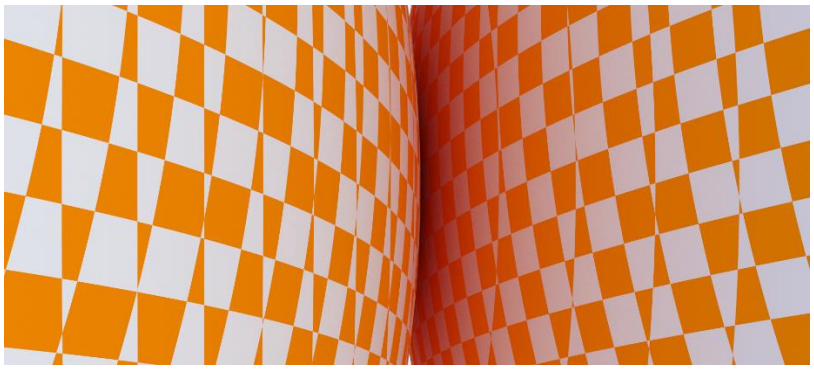
Programski kod 7.1: Novi kut kamere

```
class camera {
public:
    camera(float vfov, float aspect) {
        float theta = vfov * M_PI / 180;
        float half_height = tan(theta / 2);
        float half_width = aspect * half_height;
        lower_left_corner = vec3(-half_width, -half_height, -1.0);
        horizontal = vec3(2 * half_width, 0.0, 0.0);
        vertical = vec3(0.0, 2 * half_height, 0.0);
        origin = vec3(0.0, 0.0, 0.0);
    }
    ray get_ray(float u, float v) {
        return ray(origin, lower_left_corner + u * horizontal + v * vertical - origin);
    }

    vec3 origin;
    vec3 lower_left_corner;
    vec3 horizontal;
    vec3 vertical;
};
```

Programski kod 7.2: Klasa za prikaz rešetkaste teksture kugli

```
class texture {
public:
	virtual vec3 value(float u, float v, const vec3& p) const = 0;
};

class checker_texture : public texture {
public:
	checker_texture() {}
	checker_texture(texture* t0, texture* t1) : even(t0), odd(t1) {}
	virtual vec3 value(float u, float v, const vec3& p) const {
		float sines = sin(10 * p.x()) * sin(10 * p.y()) * sin(10 * p.z());
		if (sines < 0)
			return odd->value(u, v, p);
		else
			return even->value(u, v, p);
	}

	texture* odd;
	texture* even;
};
```

S pozivom kamere iz programskog koda 7.1 i dodavanjem programskog koda 7.2 za promjenu teksture kugli kroz glavni dio programa dodavanjem dvije kugle za prikaz se dobiva prikaz kao na slici 7.1.



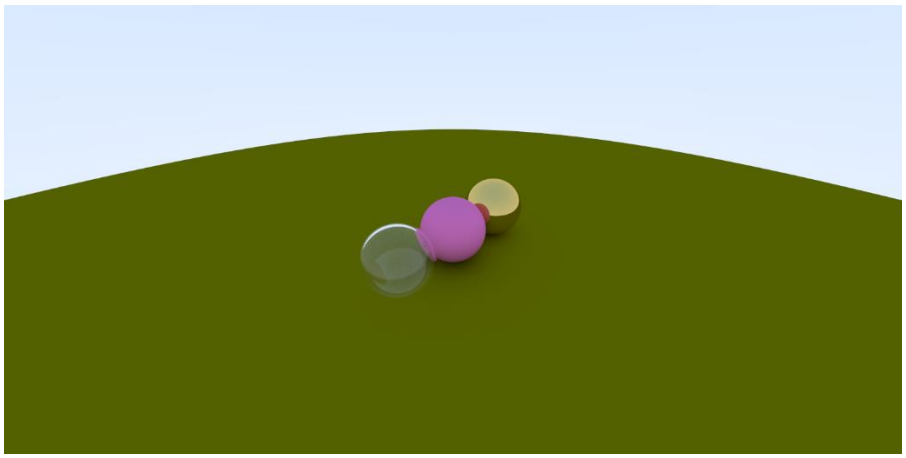
Slika 7.1:Prikaz kugli iz novog kuta kamere

Kako bi se dobilo proizvoljno gledalište prvo se moraju napraviti i imenovati točke koje su krucijalne za to. Točku u kojoj stoji kamera, odnosno s koje se gleda naziva se *lookfrom*, a točku na koju se gleda naziva se *lookat*. Također mora biti specificirana rotacija kamere, nagib lijevo i desno te rotaciju oko *lookat* i *lookfrom* osi. Nadalje, potreban je način za specificiranje vektor za kameru. S obzirom na to da već postoji ploha koja je ortogonalna na smjer pogleda kamere može se iskoristiti bilo koji vektor i može ga se projicirati na tu plohu da se dobije vektor potreban za kameru. Što će se u ovom slučaju zvati *vup* vektor.

Dodavanjem vektora s koordinatama *u, v, w* kreirana je potpuna ortonormalna osnova pomoću koje se opisuje orijentaciju kamere. *vup, v i w* se nalaze na istoj plohi. Isto kao prije kada je *-z* bio postavljen zbog pravila lijeve ruke, sada kamera proizvoljnog prikaza gleda prema *-w*.

Programski kod 7.3: Kamera podesivog vidnog polja i poziv u glavnom djelu programa

```
class camera {
public:
    camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect) {
        vec3 u, v, w;
        float theta = vfov * M_PI / 180;
        float half_height = tan(theta / 2);
        float half_width = aspect * half_height;
        origin = lookfrom;
        w = unit_vector(lookfrom - lookat);
        u = unit_vector(cross(vup, w));
        v = cross(w, u);
        lower_left_corner = origin - half_width * u –
                            half_height * v - focus_dist * w;
        horizontal = 2 * half_width * u;
        vertical = 2 * half_height * v;
    }
    ray get_ray(float s, float t) {
        return ray(origin, lower_left_corner + s * horizontal +
                   t * vertical - origin);
    }

    vec3 origin;
    vec3 lower_left_corner;
    vec3 horizontal;
    vec3 vertical;
};

camera cam(vec3(-2, 2, 1), vec3(0, 0, -1),
           vec3(0, 1, 0), 90, float(nx) / float(ny));
```



Slika 7.2: Prikaz scene iz podesivog vidnog polja kamere

Pomoću programskog koda 7.3 uspješno je kreiran prikaz sa slike 7.2 što je omogućilo daljnje realističnije prikazivanje objekata na sceni. U programskom kodu 7.3 posljednja linija predstavlja poziv u glavnom dijelu programa gdje se mogu odrediti točne koordinate u prostoru za pozicioniranje kamere.

7.2 Fokus

Finalna značajka je kreirati fokus na određenom dijelu slike. Razlog zašto dolazi do zamućenja na pravim kamerama je taj što one trebaju veliki otvor za prikupljanje svjetlosti. Na ovaj način sve bi bilo zamućeno, ali ako se stavi leću u otvor kamere postojat će razdaljina na kojoj će sve biti u fokusu. Udaljenost od te ravnine u kojoj su stvari u fokusu kontrolira udaljenost između leće i senzora odnosno filma. To je razlog pomicanja leće u odnosu na kameru kada se mijenja ono što je u fokusu. Otvor prednjeg djela kamere kontrolira učinkovitost leće. Ako je kameri potrebno više svjetlosti otvor se povećava ali se time povećava i zamućenje slike. Za ovu primjenu virtualna kamera je idealna jer ima savršeni senzor i nikada ne treba više svjetlosti, ali da bi izlazna slika izgledala realistično mora se dodati zamućenje od krajeva prema centru slike, drugim riječima želi se postići da je u fokusu samo ono što se nalazi u centru izlaza programa.

Prava kamera ima složenu leću koja je kompleksna. Za prikaz prave kamere trebalo bi se simulirati redoslijed obavljanja funkcija unutar prave kamere: senzor, leća, širina otvora te odrediti kada poslati zrake i okrenuti izlaz jednom kada je izračunat jer je slika na pravom filmu okrenuta naopako. No, ne mora se raditi sve navedeno u računalnoj grafici obično se koriste aproksimacije tankih leća. Ne mora se simulirati slika unutar kamere jer je to za prikazivanje slike izvan kamere potpuno nepotrebno. Umjesto toga može se postaviti početak zraka na površini leće i poslati ih na virtualnu plohu koja će projicirati sliku na sebi samoj i prikazati što je u fokusu. Za to je potrebno da zrake ne idu iz jedne točke već su ispaljene s kruga oko točke *lookfrom* kamere.

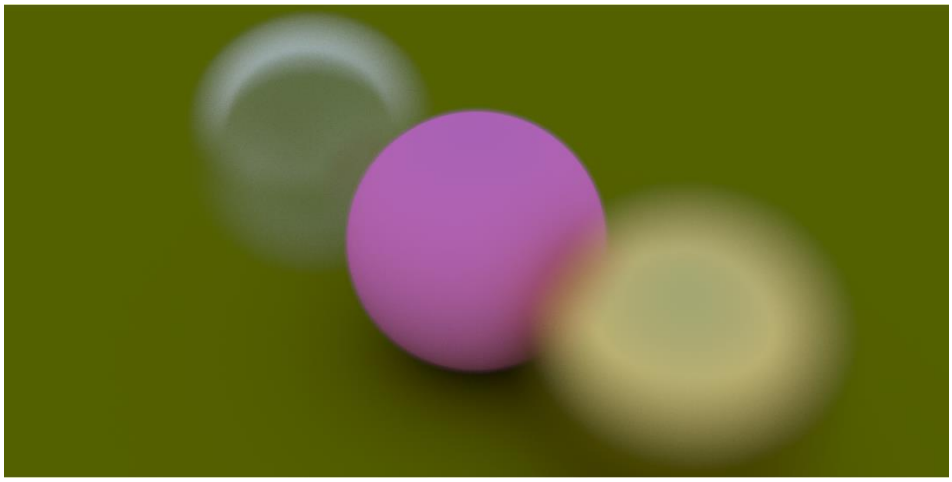
Programski kod 7.4: Prilagodba kamere u glavnom dijelu programa

```
vec3 lookfrom(3, 3, 2);
vec3 lookat(0, 0, -1);
float dist_to_focus = (lookfrom - lookat).length();
float aperture = 2.0;

camera cam(lookfrom, lookat, vec3(0, 1, 0), 20,
       float(nx) / float(ny), aperture, dist_to_focus);
```

Pomoću programskog koda 7.4 i 7.5 kreirana je kamera koja ima mogućnost fokusiranja odnosno zamućenja dijela slike. Ovisno o razini zamućenja koje je zadano, dobivaju se različiti izlazi. Za sliku 7.3 korišteno je veliko zamućenje od 2.0.

Programski kod 7.5: Dodavanje mogućnosti fokusiranja kamere unutar njene klase [4]

```
vec3 random_in_unit_disk() {
       vec3 p;
       do {
              p = 2.0 * vec3(randf(), randf(), 0) - vec3(1, 1, 0);
       } while (dot(p, p) >= 1.0);
       return p;
}

class camera {
public:
       camera(vec3 lookfrom, vec3 lookat, vec3 vup, float vfov, float aspect,
              float aperture, float focus_dist) {
              lens_radius = aperture / 2;
              float theta = vfov * M_PI / 180;
              float half_height = tan(theta / 2);
              float half_width = aspect * half_height;
              origin = lookfrom;
              w = unit_vector(lookfrom - lookat);
              u = unit_vector(cross(vup, w));
              v = cross(w, u);
              lower_left_corner = origin - half_width * focus_dist * u -
half_height *
                     focus_dist * v - focus_dist * w;
              horizontal = 2 * half_width * focus_dist * u;
              vertical = 2 * half_height * focus_dist * v;
       }
       ray get_ray(float s, float t) {
              vec3 rd = lens_radius * random_in_unit_disk();
              vec3 offset = u * rd.x() + v * rd.y();
              return ray(origin + offset, lower_left_corner + s * horizontal + t *
                     vertical - origin - offset);
       }
       vec3 origin;
       vec3 lower_left_corner;
       vec3 horizontal;
       vec3 vertical;
       vec3 u, v, w;
       float lens_radius;
};
```



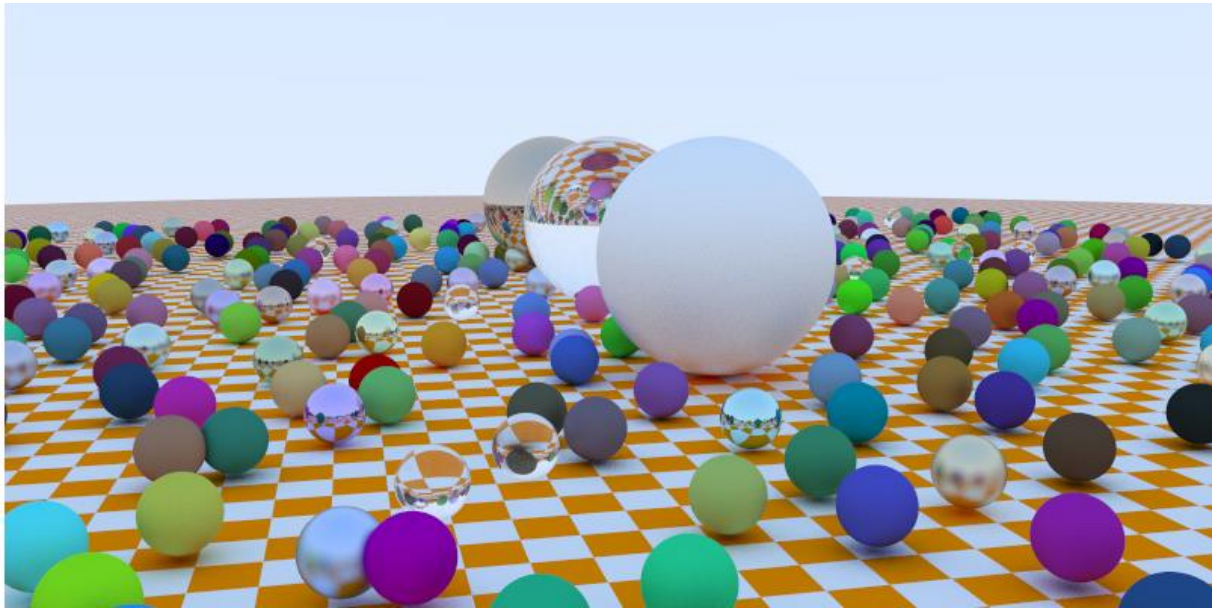
Slika 7.3: Prikaz izlaza programa s dodanim fokusom slike

Sada kada su dodane željene značajke moguće je kreirati sliku sa svim dodanim obilježjima koja su napravljena.

Programski kod 7.6: Implementacija značajki

```
hitable* random_scene() {
    int n = 500;
    hitable** list = new hitable * [n + 1];
    texture* checker = new checker_texture(new constant_texture(vec3(1.0, 0.31, 0.0)),
                       new constant_texture(vec3(0.9, 0.9, 0.9)));
    list[0] = new sphere(vec3(0, -1000, 0), 1000, new lambertian(checker));
    int i = 1;
    for (int a = -11; a < 11; a++) {
        for (int b = -11; b < 11; b++) {
            float choose_mat = randf();
            vec3 center(a + 0.9 * randf(), 0.2, b + 0.9 * randf());
            if ((center - vec3(4, 0.2, 0)).length() > 0.9) {
                if (choose_mat < 0.8) {
                    list[i++] = new sphere(center, 0.2,
                                new lambertian(new constant_texture(vec3(randf() * randf(),
                                               randf() * randf(), randf() * randf()))));
                }
                else if (choose_mat < 0.95) {
                    list[i++] = new sphere(center, 0.2, new metal(vec3(0.5 * (1 + randf()),
                                0.5 * (1 + randf()), 0.5 * (1 + randf())), 0.5 * randf()));
                }
                else {
                    list[i++] = new sphere(center, 0.2, new dielectric(1.5));
                }
            }
        }
    }
    list[i++] = new sphere(vec3(0.0, 1.0, 0.0), 1.0, new dielectric(1.5));
    list[i++] = new sphere(vec3(-4.0, 1.0, 0.0), 1.0, new metal(vec3(0.7, 0.6, 0.5), 0.0));
    list[i++] = new sphere(vec3(-4.0, 1.0, 0.0), 1.0, new lambertian(
                                                  new constant_texture(vec3(0.5,0.5,0.5))));
    return new hitable_list(list, i);
}
```

U programskom kodu 7.6 kreirana je nova scena na kojoj su nasumično dodane, ovisno o nasumično generiranom broju, vrste kugla na scenu. Ako je odabrani broj bio manji od 0.8 na scenu su dodane difuzne kugle, ako je broj bio manji od 0.95, a veći od 0.8 dodane su metalne kugle te za sve ostale slučajeve dodane su staklene kugle. U sredini slike ručno su dodane 3 veće kugle od svake vrste po jedna koje su u fokusu. Slika 7.4 izgleda ovako:



Slika 7.4: Slika sa kreiranim značajkama

Slika 7.4 nešto je manje rezolucije od ostatka slika. Zbog svoje zahtjevnosti nije na klasičnoj rezoluciji od 1500x750 kao što su sve ostale slike nego u rezoluciji od 800x400. Razlog tome je nemoguće završavanje izvođenja programa zbog prekomjernog vremena nakon kojeg je program prestao s izvođenjem ili je memorija bila zapunjena zbog količine obrade podataka. Slika 8.4 u rezoluciji 1500x750 mogla bi se prikazati kada bi se cijeli postupak preselilo na grafičku procesorsku jedinicu.

8. IZVOR SVJETLOSTI I KVADRATNI OBLICI

Svjetlost je glavna komponenta trasiranja zraka zbog toga će se u ovom poglavlju prikazati kako pretvoriti klasičan objekt u nešto što emitira svjetlost na scenu.

Prvo treba kreirati materijal koji emitira svjetlost. Potrebno je dodati funkciju za emitiranje, ona će biti poput pozadine, govoriti će zrakama svoju boju i boju koju proizvodi ali neće imati odraz.

Programski kod 8.1: Dodavanje klase svjetlosti i funkcije za emitiranje svjetlosti

```
class diffuse_light : public material {
public:
       diffuse_light(texture* a) : emit(a) {}
       virtual bool scatter(const ray& r_in, const hit_record& rec,
                            vec3& attenuation, ray& scattered)
                            const { return false; }
       virtual vec3 emitted(float u, float v, const vec3& p) const {
              return emit->value(u, v, p);
       }
       texture* emit;
};

class material {
public:
    virtual bool scatter(const ray& r_in, const hit_record& rec,
                         vec3& attenuation, ray& scattered)
                         const = 0;
    virtual vec3 emitted(float u, float v, const vec3& p) const {
        return vec3(0, 0, 0);
    }
};
```

Kroz programski kod 8.1 prikazana je klasa svjetlosti naziva *diffuse_light* i funkcija *emitted()* u klasi *material* koja služi kako se ne bi moralo na sve materijale koji ne emitiraju svjetlost primjenjivati funkcija *emitted()*.

Programski kod 8.2: Klasa za prikaz kvadratnih oblika

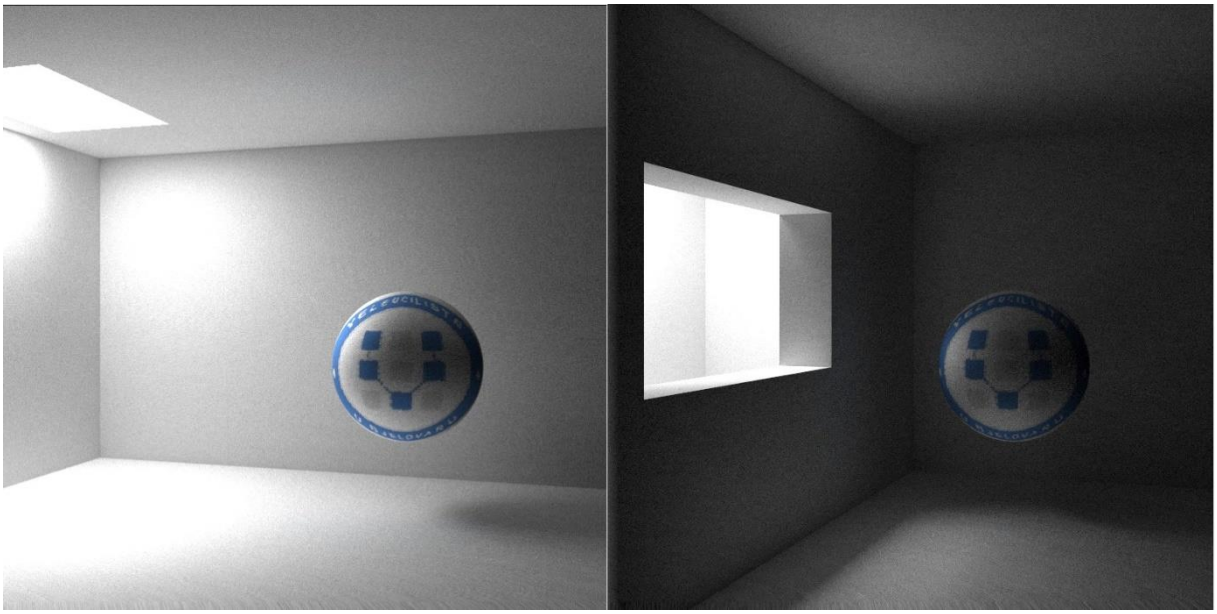
```
class box : public hitable {
public:
	box() {}
	box(const vec3& p0, const vec3& p1, material* ptr);
	virtual bool hit(const ray& r, float t0, float t1, hit_record& rec)
const;
	virtual bool bounding_box(float t0, float t1, aabb& box) const {
		box = aabb(pmin, pmax);
		return true;
	}
	vec3 pmin, pmax;
	hitable* list_ptr;
};

box::box(const vec3& p0, const vec3& p1, material* ptr) {
	pmin = p0;
	pmax = p1;
	hitable** list = new hitable * [6];
	list[0] = new xy_rect(p0.x(), p1.x(), p0.y(), p1.y(), p1.z(), ptr);
	list[1] = new flip_normals(
		new xy_rect(p0.x(), p1.x(), p0.y(), p1.y(), p0.z(), ptr));
	list[2] = new xz_rect(p0.x(), p1.x(), p0.z(), p1.z(), p1.y(), ptr);
	list[3] = new flip_normals(
		new xz_rect(p0.x(), p1.x(), p0.z(), p1.z(), p0.y(), ptr));
	list[4] = new yz_rect(p0.y(), p1.y(), p0.z(), p1.z(), p1.x(), ptr);
	list[5] = new flip_normals(
		new yz_rect(p0.y(), p1.y(), p0.z(), p1.z(), p0.x(), ptr));
	list_ptr = new hitable_list(list, 6);
}

bool box::hit(const ray& r, float t0, float t1, hit_record& rec) const {
	return list_ptr->hit(r, t0, t1, rec);
}
```

Programskim kodom 8.2 dodana je klasa *box* koja je načinjena od pravokutnika koji čine kvadratnu strukturu. Pomoću ove klase kreirati će se stranice sobe finalne scene što će pomoći pri odbijanju svjetlosti i isticanju dubine scene.

Finalna značajka koja je korištena je prikaz proizvoljne fotografije na kugli. Kako bi se to postiglo korištena je biblioteka *stb_image* i kreirana je klasa *image_texture* što je prikazano programskim kodom 8.3.

Programski kod 8.3: Klasa za prikaz proizvoljne slike kugle

```
class image_texture : public texture {
public:
	image_texture() {}
	image_texture(unsigned char* pixels, int A, int B) :
					data(pixels), nx(A), ny(B) {}
	virtual vec3 value(float u, float v, const vec3& p) const {
		int i = u * nx;
		int j = (1 - v) * ny - 0.001;
		if (i < 0) i = 0;
		if (j < 0) j = 0;
		if (i > nx - 1) i = nx - 1;
		if (j > ny - 1) j = ny - 1;
		float r = int(data[3 * i + 3 * nx * j]) / 255.0;
		float g = int(data[3 * i + 3 * nx * j + 1]) / 255.0;
		float b = int(data[3 * i + 3 * nx * j + 2]) / 255.0;
		return vec3(r, g, b);
	}
	unsigned char* data;
	int nx;
	int ny;
};
```



Slika 8.1: Finalna scena s novim značajkama

Izvor svjetlosti slike 8.1 predstavlja pravokutnik u gornjem lijevom kutu te sva svjetlost dolazi od njega i odbija se te stvara sjene kako je prikazano na slici. Kada je dodana pregrada kreirana od kvadratnih oblika koja zaklanja izvor svjetlosti može se primijetiti znatno zamračenje slike što nam kazuje da svjetlost uistinu dolazi iz izvora u gornjem lijevom kutu. Kao proizvoljna slika kugle odabran je logotip Veleučilišta u Bjelovaru.

9. ZAKLJUČAK

Trasiranje zraka svjetlosti neizbježno je u sadašnjosti i budućnosti u području računalne grafike zbog svoje atraktivnosti da sve digitalno prikaže na što zorniji i realističniji način. Atraktivnosti trasiranja zraka pridonosi činjenica da u području video igara puno više vremena oduzima izgradnja igre za računala bez mogućnosti trasiranja zraka nego za računala koje imaju tu mogućnost. Razlog tome je činjenica da grafičke procesorske jedinice s mogućnosti trasiranja zraka mogu to odraditi u stvarnom vremenu i prikazati one dijelove koji su vidljivi igraču na realističniji način. Za grafičke procesorske jedinice koje nemaju mogućnost trasiranja kreatori moraju prilagoditi igru kako bi i oni vidjeli sjene odsjaje i slično, ali ručna izrada navedenog zahtjeva više resursa. Zbog napretka hardvera, primarno grafičkih procesorskih jedinica s mogućnosti trasiranja, standardizirati će se postupak izrade i omogućiti kreatorima izradu kvalitetnijeg produkta u manje utrošenog vremena.

Izrađeni prototip stoji kao zasebna cjelina i bez daljnje nadogradnje nije primjenjiv u svrhe kompleksnijeg grafičkog prikaza ili video igre. Također kameru bi trebalo pokrenuti kako bi mogla prikazivati predmete u pokretu. Trasiranje zraka svjetlosti je kompleksan i opširan pojam pomoću kojeg se može ići u raznim smjerovima. Cilj ovog sustava je bilo zorno prikazati sliku pomoću zraka svjetlosti što je i postignuto u zadovoljavajućem omjeru.

Sustav je razvijen korištenjem C++ programski jezik pomoću čijeg okruženja su postignuti svi ciljevi ovog sustava. Zorni prikaz izlaza programa prikazan je pomoću PPM formata za obojene slike. Za poboljšanje ovog sustava moguće je napraviti objekte raznih oblika kao i grafičko korisničko sučelje preko kojeg korisnici mogu na lakši način dodavati objekte. Također bi trebalo poraditi na brzini izvođenja sustava što bi se moglo realizirati prebacivanjem tereta obrade na grafičku procesorsku jedinicu.

10. LITERATURA

[1] PC CHIP [Online]. Dostupno na:

https://pcchip.hr/hardver/komponente/sto-je-to-ray-tracing/ (1.11.2021.)

[2] NVIDIA DEVELOPER Ray Tracing [Online]. Dostupno na:

https://developer.nvidia.com/discover/ray-tracing. (1.11.2021.)

[3] Eric Haines, Tomas Akenine-Möller - Ray Tracing Gems_ High-Quality and Real-Time Rendering with DXR and Other APIs-Apress; 2019. str. 7-13.

[4] Peter Shirley – Ray Tracing in One Weekend; 2018.

[5] Scratchpixel 2.0 [Online]. Dostupno na:

https://www.scratchapixel.com/index.php?redirect (3.11.2021.)

[6] CMICHEL [Online]. Dostupno na:

https://cmichel.io/howto-raytracer-ray-sphere-intersection-theory (3.11.2021.)

[7] Techopedia [Online]. Dostupno na:

https://www.techopedia.com/definition/1950/antialiasing (4.11.2021.)

[8] Inigo Quilez [Online]. Dostupno na:

https://iquilezles.org/www/articles/sfrand/sfrand.htm (5.11.2021.)

[9] 1000 Forms Of Bunnies [Online]. Dostupno na:

http://viclw17.github.io/2018/07/20/raytracing-diffuse-materials/ (6.11.2021.)

[10] ResearchGate [Online]. Dostupno na:

https://www.researchgate.net/figure/Illustration-of-Snells-law-where-n2-n1-The-angle-th1-is-measured-between-the-incident_fig9_235132960 (7.11.2021.)

[11] The Schlick Fresnel Approximation [Online]. Dostupno na:

https://link.springer.com/content/pdf/10.1007%2F978-1-4842-7185-8_9.pdf (7.11.2021.)

11. SAŽETAK

Prototipni sustav za trasiranje zraka svjetlosti u računalnoj simulaciji

Cilj ovog rada izrada je sustava za trasiranje zraka svjetlosti koristeći C++ programski jezik. Sustav omogućava stvaranje scene na kojoj je moguće stvoriti pozadinu, podesivu kameru i kreirati objekte raznih materijala i boja o kojima ovisi lom svijetla. Svi objekti na sceni dio su sustava za trasiranje zraka i sudjeluju u fizičkoj interakciji svjetla. Za prikaz scene koristi se PPM format za prikaz slika u boji. Realizacija sustava prikazana je programskim kodom te izlaz programa priloženim slikama.

Ključne riječi: Ispaljivanje zraka svjetlosti, trasiranje zraka svjetlosti, anti-aliasing, difuzni, metalni i stakleni objekti, podesiva kamera, fokus, umjetni izvor svjetlosti, kvadratni oblici i proizvoljna slika kugle.

12. ABSTRACT

Prototype system for ray tracing in computer simulation

The aim of this paper is to develop a system for tracing light rays using C++ programming language. The system allows you to create scenes where it is possible to create a background, an adjustable camera, and create objects of various materials and colors on which the refraction of light depends. All objects in the scene are part of the ray tracing system and participate in the physical interaction of light. The PPM format for displaying color images is used to display the scene. The realization of the system is shown by the program code and the program outputs with the attached images.

Keywords: Ray casting, ray tracing, anti-aliasing, diffuse, metal and glass objects, adjustable camera, focus, artificial light source, square shapes and an arbitrary image of the sphere.

IZJAVA O AUTORSTVU ZAVRŠNOG RADA

Pod punom odgovornošću izjavljujem da sam ovaj rad izradio/la samostalno, poštujući načela akademske čestitosti, pravila struke te pravila i norme standardnog hrvatskog jezika. Rad je moje autorsko djelo i svi su preuzeti citati i parafraze u njemu primjereno označeni.

Mjesto i datum	Ime i prezime studenta/ice	Potpis studenta/ice
U Bjelovaru, 7 TRAVNJA 2022.	PATRIK VUKOVIĆ	Patrik Vuković

Prema Odluci Veleučilišta u Bjelovaru, a u skladu sa Zakonom o znanstvenoj djelatnosti i visokom obrazovanju, elektroničke inačice završnih radova studenata Veleučilišta u Bjelovaru bit će pohranjene i javno dostupne u internetskoj bazi Nacionalne i sveučilišne knjižnice u Zagrebu. Ukoliko ste suglasni da tekst Vašeg završnog rada u cijelosti bude javno objavljen, molimo Vas da to potvrdite potpisom.

Suglasnost za objavljivanje elektroničke inačice završnog rada u javno dostupnom nacionalnom repozitoriju

PATRIK VUKOVIĆ

ime i prezime studenta/ice

Dajem suglasnost da se radi promicanja otvorenog i slobodnog pristupa znanju i informacijama cjeloviti tekst mojeg završnog rada pohrani u repozitorij Nacionalne i sveučilišne knjižnice u Zagrebu i time učini javno dostupnim.

Svojim potpisom potvrđujem istovjetnost tiskane i elektroničke inačice završnog rada.

U Bjelovaru, 7. TRAVNJA 2022.

Patrik Vuković

potpis studenta/ice