Implementacija algoritma za praćenje zrake svjetlosti

Martinec, Dorotea

SVEUILIŠTE U RIJECI

TEHNIČKI FAKULTET

Preddiplomski sveučilišni studij strojarstva

Završni rad

IMPLEMENTACIJA ALGORITMA ZA PRAĆENJE ZRAKE SVJETLOSTI / IMPLEMENTATION OF RAY TRACING ALGORITHM

Mentor: Izv. prof. dr. sc. Jerko Škifić

Rijeka, rujan 2022.

Dorotea Martinec
069075977

SVEUILIŠTE U RIJECI

TEHNIČKI FAKULTET

Preddiplomski sveučilišni studij strojarstva

Završni rad

IMPLEMENTACIJA ALGORITMA ZA PRAĆENJE ZRAKE SVJETLOSTI / IMPLEMENTATION OF RAY TRACING ALGORITHM

Mentor: Izv. prof. dr. sc. Jerko Škifić

Rijeka, rujan 2022.

Dorotea Martinec
069075977

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA ZAVRŠNE ISPITE

Rijeka, 12. ožujka 2021.

Zavod: **Zavod za mehaniku fluida i računarsko inženjerstvo**
Predmet: **Računarske metode**
Grana: **2.15.04 mehanika fluida**

ZADATAK ZA ZAVRŠNI RAD

Pristupnik: **Dorotea Martinec (0069075977)**
Studij: Preddiplomski sveučilišni studij strojarstva

Zadatak: **Implementacija algoritma za praćenje zrake svjetlosti / Implementation of ray tracing algorithm**

Opis zadatka:

Implementirati algoritam praćenja zrake svjetlosti u svrhu realističnog prikaza stacionarne scene. Voditi računa o različitim modelima sjenčanja, materijalu objekata u sceni, alias učinku te broju i tipu izvora svjetlosti.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Dorotea Martinec

Zadatak uručen pristupniku: 15. ožujka 2021.

Mentor:

Izv. prof. dr. sc. Jerko Škifić

Predsjednik povjerenstva za završni ispit:

Prof. dr. sc. Kristian Lenić

IZJAVA

Izjavljujem da sam samostalno izradila ovaj rad.

Rijeka, rujan 2022.

Dorotea Martinec

Dorotea Martinec

ZAHVALA

Zahvaljujem svom mentoru izv.prof.dr.sc. Jerku Škifiću na susretljivosti i svoj pomoći tijekom izrade završnog rada.

SADRŽAJ

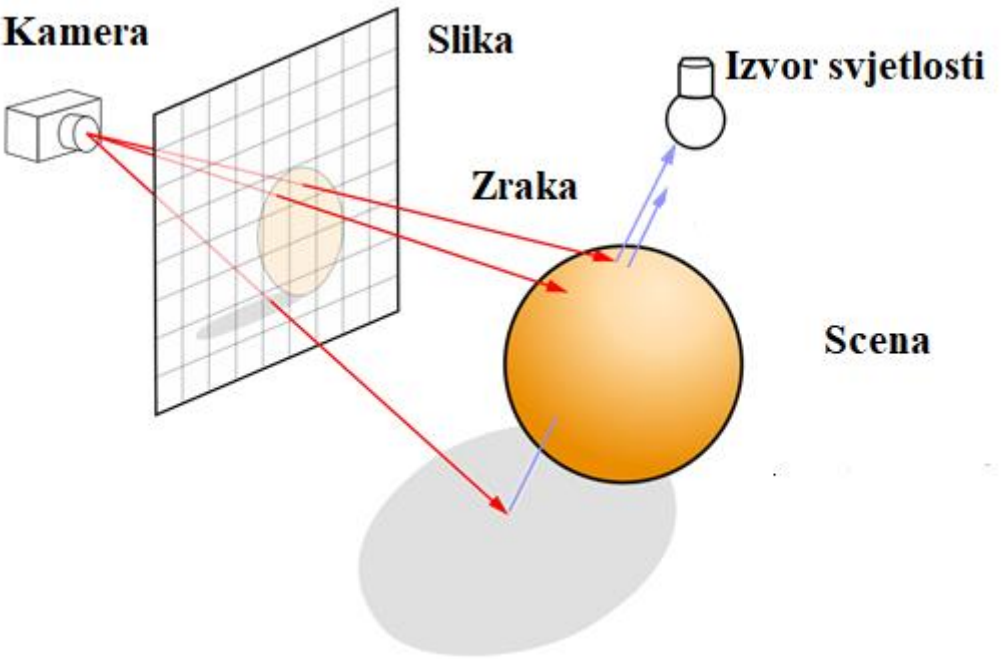
1. UVOD

Praćenje zraka svjetlosti (engl. ray tracing) je algoritam kojim se stvaraju realistične računalne slike. Radi na način tako da imitira prostiranje svjetlosti kroz prostor.

Za početak treba se postaviti scena koju promatramo. U sceni se nalazi objekt (sfera). Da bi se u sceni mogao vidjeti objekt potreban je izvor svjetlosti. Za zadanu scenu treba se odrediti projekcija objekta u dvodimenzionalnoj ravnini projekcije. Za prikaz objekta u ravnini projekcije potrebno je odrediti koordinate tog objekta u koordinatnom sustavu kamere. Položaj gdje se kamera nalazi naziva se očište. Ranije spomenuti izvor svjetlosti emitira zrake svjetlosti u svim smjerovima. Tako da će neke zrake udariti površinu promatranog objekt. Zatim se zrake svjetlosti odbiju od objekta i putuju prema očištu. Izvor svjetlosti emitira svjetlost u svim smjerovima, ali samo mali postotak zraka svjetlosti dođe do objekta, pa zatim do očišta. Algoritam koji izračuna putanju svih zraka bi bio vrlo spor i nepotreban. Algoritam za praćenje zraka svjetlosti koristi sve ove koncepte, ali njegova specifičnost je u tome što prati zrake svjetlosti unazad. Krećući od kamere, prateći zraku svjetlosti do objekta i na kraju do izvora svjetlosti. Tako da su samo vidljive zrake izračunate. Ravna projekcija sastoji se od piksela. Algoritam mora izračunati putanju zrake počevši od kamere prolazeći kroz piksel i dohvatit scenu. Može se dogodit nekoliko scenarija:

- Zraka udari najbliži objekt, površina objekta je reflektirajuća i ponaša se kao zrcalo. Izračuna se reflektirajuća zraka i slijedi se njezi put. Ona može doći do izvora svjetlosti. Ako zraka dođe do izvora svjetlosti tada znamo da se objekt može vidjeti kroz taj piksel.
- Zraka može indirektno stići do izvora svjetlosti, tako da se odbije više puta ili od više objekata dok ne naiđe na izvor svjetlosti. Tako se može vidjeti odraz nekih drugih objekata (kao na primjer tlo u kojem se vidi odraz gledanog objekta).
- Zraka ne može nikad dosegnuti izvor svjetlosti. Tada promatramo područje sjene objekta.

Pomoću tog načina polako se, piksel po piksel dobije dvodimenzionalni prikaz scene u ravnini projekcije tj. slika scene u kojoj se nalazi objekt. Zahvaljujući ovim postupkom slika scene na kraj završetka programa će izgledati vrlo realistično.



Slika 1.1 Vizualni prikaz djelovanja algoritma za praćenje zraka svjetlosti [1]

2. PRIMJENA PRAĆENJA ZRAKA SVJETLOSTI

Primarna primjena praćenja zraka svjetlosti se koristi u kompjuterskoj grafici, naviše u inženjerstvu, video igrama i animiranim filmovima. Također se može koristiti u dizajnu kazališne i televizijske rasvjete, kao što i u arhitekturi.

2.1. Primjena praćenja zraka svjetlosti u animaciji

Pitanja kako efikasnije animirati zanimaju informatičare već nekoliko desetljeća. Napredak računalne grafike, uključujući razvoj praćenja zraka, otvorila se mogućnost na tom području. Tradicionalno, pojedinačni okviri animiranih radova crtani su rukom. Kretanje je animirano kroz složen niz koraka podešavanja okvira. Taj sam proces trajao je vrlo dugo. Još uvijek postoji osjećaj nostalgije za tradicionalnim metodama animacije. Međutim, računalna grafika ima sve jaču i jaču ulogu u tom procesu. Računalna animacija izvodi se na modelima prije nego što se krene izrada prikaza pomoću alata za praćenje zraka svjetlosti. Ova se tehnika može visoko optimizirati. Praćenjem zraka mogu se dodavati efekata kao što su refleksija, sjenčanje i sjena. Koje je tradicionalnim umjetnicima često bilo teško i dugotrajno za proizvesti. Grafička tehnologija također je sposobna prikazati foto-realistične slike koje bi bilo gotovo nemoguće proizvesti bez računalnog praćenja zraka. Primjena računalne grafike i praćenja zraka u modernoj animaciji uključuju naprednu refleksiju, sjenčanje i zrcalnost. [2]

2.2. Primjena praćenja zraka svjetlosti u arhitekturi

Tijekom predstavljanja prijedloga dizajna klijentima, vrlo je važno za arhitekte da imaju realistične prikaze pomoću kojih se može vizualizirati njihov dizajn. Nekada su se arhitekti oslanjali na ručno izrađene crteže stvorene vodenim bojama i tušem. Nažalost, iznimno je teško proizvesti realne efekte osvjetljenja korištenjem tradicionalnih metoda iscrtavanja. Čak i mnogi programi za računalno potpomognuto projektiranje (CAD) koji pedantno modeliraju objekte ne mogu modelirati svjetlost.

„Neki programi za modeliranje također omogućuju arhitektima da uvedu svjetlo u sliku, određujući položaj svjetla, orijentaciju, boju i distribuciju. Ove značajke pomažu dizajnerima da stvore efekte kao što su sjene i reflektirajuća svjetla, ali još uvijek ne uspijevaju modelirati fizičku stvarnost da objekti stupaju u interakciju sa svjetlom. Kako bismo točno modelirali pravo ponašanje svjetlosti u određenom okruženju, moramo uzeti u obzir sve svjetlo u tom okruženju i prepoznati da se pravo svjetlo reflektira, lomi, stvara difuziju i apsorbira.

Arhitekte prvenstveno zanima stvaranje vizualno realistične slike. Praćenje zraka unatrag često je najkorisnija metoda za arhitektonske izrade prikaza. Budući da je modeliranje svjetla toliko bitno za ovo područje, razvoj programa koji uključuju praćenje zraka bio je san mnogih arhitekata. Globalna tehnologija osvjetljenja uspijeva oživjeti arhitektonski dizajn. „[3]

2.3. Primjena praćenja zraka svjetlosti u video igrama

„Prije nekoliko desetljeća, rasterizacija je zaslužila svoje mjesto u videoigrama jer je hardver potreban za to bio dovoljno pristupačan da mu pristupe glavni kupci, za razliku od onog koji je bio potreban za praćenje zraka. Ovo je još uvijek dobrim dijelom točno; grafičke kartice za igre optimizirane su za rasterizaciju i bit će tako još mnogo godina.

Uvođenje praćenja zraka svjetlosti u konvencionalnim video igrama započelo je 2018. lansiranjem Nvidia GeForce RTX linije kartica za stolna računala, u obliku GeForce RTX 2080. Nvidia je predstavila svoju drugu generaciju GeForce RTX 3000 kartica serije 2020. (na čelu s GeForce RTX 3080) , a konkurentski AMD brzo je slijedio primjer sa svojom Radeon RX 6000 serijom.

Ukratko, trebalo je toliko vremena da praćenje zraka svjetlosti uđe na sceni video igra jer su računalni resursi za njegovu provedbu bili nedostižni po cijenama koje bi omogućile konvencionalno usvajanje. Doduše, ulazna cijena je još uvijek relativno visoka - ni AMD ni Nvidia još ne nude jeftinu desktop grafičku karticu s hardverskim praćenjem zraka.

Trenutačno je početna razina video kartica sposobna za praćenje zraka u hardveru GeForce RTX 2060, koja je lansirana 2019. po ne konvekcionalnoj cijeni od 349 dolara, a danas se prodaje znatno više od te cijene iz većine izvora, zbog velikih potražnji i malih ponuda video kartica.“ [4]

3. IMPLEMENTACIJA VEKTORA

„Skoro svi grafički programi imaju neke strukture podataka za pohranjivanje geometrijskih vektora i boja. U mnogim sistemima ti vektori su 4D (3D plus homogene koordinate za geometriju, i RGB plus alfa transparentni kanal za boje. Tri koordinate su dovoljne. Koristit ću klasu vektora za boje, lokacije, smjerove, pomake.“ [5]

Vektor u matematici označava veličinu koja ima iznos, smjer i orijentaciju. Za predstavljanje podatka vektora u programskom jeziku Python postoje razne opcije. Neke od tih su:

- Lista s tri elemenata
- Tuple s tri elemenata
- Numpy array s tri elemenata
- Rječnik s x, y i z elementima
- Klasa s x, y i z elementima

Tuple nije privlačna opcija ako se usporedi s ostalima pošto je vrlo nefleksibilna. U prve dvije opcije se ne može zasebno pristupit svakom elementu. Numpy array ima optimizirane strukture podataka, ali ih se ne može dohvatiti njihovim imenom nego indeksom. Rječnikom mogu se dohvatiti individualni elementi kao što su x, y i z. Klasom se mogu definirati članovi i mogu se pohraniti na tim mjestima. Odabrana je opcija klase s x, y i z elementima.

Vektor se mora implementirati tako da podržava sljedeće operacije:

$$V = [1, -2, -2]$$

$$V = \sqrt{x^2 + y^2 + z^2} = \sqrt{9} = 3 \tag{3.1}$$

$$V_2 = [3, 6, 9]$$

Skalarni produkt dva vektora:

$$V_1 \cdot V_2 = \sqrt{a_1 b_1 + a_2 b_2 + a_3 b_3} = -27 \tag{3.2}$$

Zatim slijede obične osnovne operacije:

$$V_1 + V_2 = [4, 4, 7]$$

$$V_1 - V_2 = [-2, -8, -11] \tag{3.3}$$

$$2 \cdot V_1 = [2, 4, -4]$$

$$V_2/3 = [1, 2, 3]$$

3.1. Implementacija koda u programskom jeziku Python

```
import math
from math import sqrt

class Vector:

    def __init__(self, x=0.0, y=0.0, z=0.0):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)

    def dot_product(self, other):
        return self.x * other.x + self.y * other.y + self.z * other.z

    def magnitude(self):
        return math.sqrt(self.dot_product(self))

    def normalize(self):
        return self / self.magnitude()

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y, self.z + other.z)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y, self.z - other.z)

    def __mul__(self, other):
        assert not isinstance(other, Vector)
        return Vector(self.x * other, self.y * other, self.z * other)

    def __rmul__(self, other):
        return self.__mul__(other)

    def __truediv__(self, other):
        assert not isinstance(other, Vector)
        return Vector(self.x / other, self.y / other, self.z / other)
```

Slika 3.1 Implementacija vektora u programskom jeziku Python

4. ISPIS SLIKE

4.1. Izrada prikaza slike pomoću programskog jezika Python

Piksel je najmanji element koji se može definirati u slici. Svaki piksel je uzorak originalne slike, više uzoraka obično daje točnije prikaze izvornika. Intenzitet svakog piksela je promjenjiv. U sustavima slikanja u boji, pikseli su uglavnom sastavljeni od tri primarne boje: zelena, crvena i plava. Pomoću kojih se kreiraju ostale boje. Crvena, zelena i plava mogu se pohraniti u bajtu (byte) koji se sastoji od osam bitova (bits). Ako se multipliticira jedan bajt tj. osam bitova s tri (što predstavlja tri primarne boje) dobivamo 24 bitova koji mogu predstavljati bilo koju moguću boju. Virtualno bilo koju moguću boju koju vidimo na kompjuterskom zaslonu. Zadatak u ovome koraku završnog rada je kreirati sliku s 3 puta 2 piksela stvorenu od različitih boja. Koja će pohraniti kao PPM datoteku.

Pošto su zadana dva reda i ti stupca treba se definirati kako će svaki izgledati.

Prvi red:

- CRVENA = (1, 0, 0)
- ZELENA = (0, 1, 0)
- PLAVA = (0, 0, 1)

Drugi red:

- CRVENA + ZELENA = (1, 1, 0)
- CRVENA + ZELENA + PLAVA = (1, 1, 1)
- CRVENA * 0.001 = (0.001, 0, 0)

Crvena i zelena zajedno stvaraju žutu boju. Dok crvena, zelena i plava bijelu, a crvena multiplicirana s vrlo malim brojem stvara skoro potpuno crnu boju.

4.2. PPM Dokument

„PPM je skraćenica od Portable Pixmap Format. Pojavio se kasnih 1980-ih kako bi se olakšalo dijeljenje slika na različitim platformama. Svaka PPM datoteka koristi tekstualni format za pohranu informacija o određenoj slici. Unutar svake datoteke, svaki piksel ima određeni broj i informacije o visini i širini slike, plus sve podatke o razmaku.“ [6]

PPM example [edit]

This is an example of a color RGB image stored in PPM format. There is a newline character at the end of each line.

```
P3
# The P3 means colors are in ASCII, then 3 columns and 2 rows,
# then 255 for max color, then RGB triplets
3 2
255
255   0   0     0 255   0     0   0 255
255 255   0   255 255 255     0   0   0
```

Image (magnified)

Slika 4.1 Primjer PPM dokumenta [7]

U PPM dokumentu ako red počinje sa #, označava komentar. Slično kao i u programskom jeziku Python. Linije sa komentarima objašnjavaju što je PPM dokument. Prva linija je zaglavlje, u kojoj P3 označava da se radi o PPM datoteci, dok druga dva broja označavaju broj stupaca i redova. U četvrtom redu je prikazano da slika ima tri stupca s dva reda. Peti red predstavlja najvišu vrijednost boje. Što je uvijek 255. U sljedećem redu nalaze se tri trojke. Svaka trojka zasebno predstavlja jednu boju. Prva trojka (255 0 0) predstavlja crvenu boju. U prošlom paragrafu je spomenuto da crvena boja je (1, 0, 0), ali u PPM datoteci broj jedan je zamijenjen brojem 255. Tako da domena, umjesto da se kreće od 0 do 1, kreće se od 0 do 255. Sljedeći piksel predstavlja zelenu boju, treći plavu boju, itd. Kao što je objašnjeno u prijašnjem paragrafu.

4.3. Implementacija slike u programskom jeziku Python

Za izradu prikaza (engl. rendering) koristeći programski jezik Python treba se definirati u kojem formatu i kakva scena će se prikazati. To se definiralo u zasebnoj Python datoteci nazvanom main.py (hrv. Glavni dio). Nadalje main će isključivo služiti za izradu prikaz slika i u sljedećim koracima. Ostale komponente scene koje će se implementirati u sljedećim koracim neće se nalaziti u main već će se uvoziti po potrebi. Ostale komponente također će se nalaziti u zasebnim datotekama. Tako je posloženo radi lakšeg načina provedbe završnog rada, radi lakšeg snalaženja, da se znam gdje se točno koja komponenta nalazi. U Python datoteku koja je sadržavala samo klasu vektora definirano je također klase slike i boje. U slikama ispod paragrafa nalaze se isječci iz tih datoteka. Na dalje bit će postavljene samo slike isječaka komponenata koje su se primijenile u svakom novom koraku završnog rada.

```
class Color(Vector):
    pass
```

Slika 4.2 Implementacija boje u programskom jeziku Python

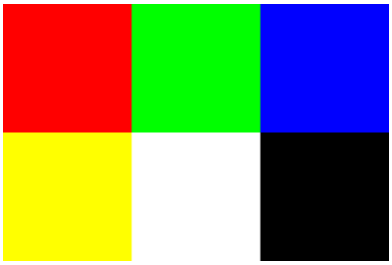
```
class Image:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.pixels = [[None for _ in range(width)] for _ in range(height)]

    def set_pixel(self, x, y, col):
        self.pixels[y][x] = col

    def write_ppm(self, img_file):
        def to_byte(c):
            return round(max(min(c * 255, 255), 0))

        img_file.write("P3 {} {}\n255\n".format(self.width, self.height))
        for row in self.pixels:
            for color in row:
                img_file.write(
                    "{} {} {} ".format(
                        to_byte(color.x), to_byte(color.y), to_byte(color.z)
                    )
                )
            img_file.write("\n")
```

Slika 4.3 Implementacija slike u programskom jeziku Python

```
from image import Image
from color import Color


def main():
    WIDTH = 3
    HEIGHT = 2
    im = Image(WIDTH, HEIGHT)
    red = Color(x=1, y=0, z=0)
    green = Color(x=0, y=1, z=0)
    blue = Color(x=0, y=0, z=1)

    im.set_pixel(0, 0, red)
    im.set_pixel(1, 0, green)
    im.set_pixel(2, 0, blue)

    im.set_pixel(0, 1, red + green)
    im.set_pixel(1, 1, red + blue + green)
    im.set_pixel(2, 1, red * 0.001)

    with open("test1.ppm", "w") as img_file:
        im.write_ppm(img_file)

if __name__ == "__main__":
    main()
```

Slika 4.4 Implementacija glavnog (main) programa u Pythonu

Na kraju ovog koraka izrađena je slika:



Slika 4.5 Slika u PPM dokument

5. SFERA (3D SFERA U 2D PROSTORU)

U ovom dijelu zadatka izradit će se prikaz 3D sfere u dvodimenzionalnoj ravnini projekcije. Slika će se također pohraniti u PPM formatu. Nakon pokretanja koda u programskom jeziku Python, slika sfere izgledat će kao 2D objekt (krug), pošto u ovom koraku nisu još primijenjeni modeli sjenčanja.

Zadatak algoritma praćenja zraka je vrlo kompleksan, ali će se ovdje pojednostaviti. Prvi korak je ispucavanje zrake prema svakom pikselu, tako će se pronaći najbliži objekt koji je pogođen zrakom u sceni. Ako je objekt udaren treba se pronaći boja na površini tog objekta. Također ako nikakav objekt nije udaren pretpostavit će se da nema objekt tj. da je prazan prosto i boja će biti crna.

5.1. Zraka

Zraka se definira samo s jednom točkom i smjerom. To čini zrakom savršenom da glumi ulogu svjetlosti u fizici. Svjetlo se počinje kretati od jedne točke i giba se samo u jednom smjer.

Formula za zraku:

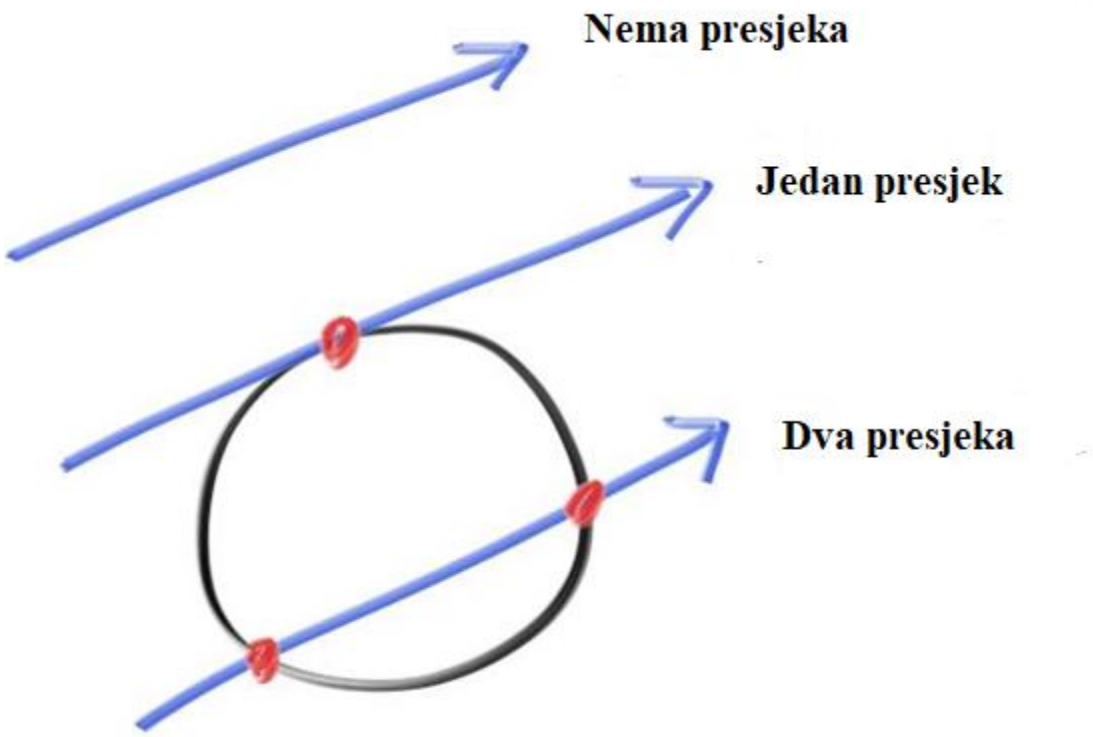
$$ray(t) = ray_{origin} + ray_{direction} \cdot t \tag{5.1}$$

- $ray(t)$ 3D pozicija duž crte u 3D
- ray_{origin} ishodište zrake
- $ray_{direction}$ vektor smjera zrake
- t parametar zrake

5.2. Presjek zrak i sfere

Geometrijski oblik sfera se često koristi u algoritmu praćenju zraka svjetlosti jer je vrlo svestrana. Pojednostavljeni algoritam praćenja zraka svjetlosti se odvija ovako: Prvo se zraka treba poslati prema svakom pikselu. Tako će se pronaći najbliži objekt koji je udaren zrakom u sceni. Ako je objekt udareni onda se treba pronaći boja na površini tog objekta. Kada zraka udari sferu postoje tri drugačija slučaja:

- Zraka udari sferu. Prođe kroz nju. Stoga znači da zraka udari sferu u dvije točke na njezinoj površini.
- Zraka dodirne samo jednu točku na površini sfere
- Zraka ne dodiruje sferu



Slika 5.1 Primjer sjecišta zraka i sfere [8]

diskriminanta:

$$a = 1$$

$$b = 2ray_{dir} \cdot sphere\_to\_ray$$

$$c = sphereto_{ray} \cdot sphere\_to\_ray - (sphere_{ceter})^2 \tag{5.2}$$

$$discriminant = b^2 - 4ac$$

- *sphere_to_ray* presjek sfere i zrake
- $sphere_{center}$ centar sfere
- *discriminant* diskriminanta
- ray_{dir} *smjer zrake*

Ako je diskriminanta veća od nule na kraju proračuna, presjek sfere i zrake je u dvije toče. To predstavlja slučaj kada zraka udari sferu i prođe kroz nju. Stoga znači da zraka udari sferu u dvije točke na njezinoj površini.

Ako je diskriminanta jednaka nuli, presjek je samo u jednoj točki. To je slučaj kada zraka presječe sferu u jednoj točki.

Ako je diskriminanta manja od nule presjeka nema. Zraka ne dodiruje sferu.

Jednadžba za udaljenost:

$$dist = \frac{-b \pm \sqrt{discrimiant}}{2a} \tag{5.3}$$

- *dist* udaljenost

jednadžba za proračun zrake svjetlosti:

$$ray(t) = ray_{origin} + ray_{direction} \cdot t \tag{5.4}$$

U sljedećem koraku zamijenit će se t (parametar zrake) s udaljenosti koja se treba prevaliti da se pronađe pozicija udarca zrake o površinu sfere. Rješenje će imati vektorsku vrijednost i točno će ukazivati gdje je točka u kojoj je zraka udarila u sferu:

$$hit_{pos} = ray_{origin} + ray_{direction} \cdot dist \tag{5.5}$$

- *sphere_to_ray* presjek sfere i zrake
- $sphere_{center}$ centar sfere
- *dist* udaljenost
- hit_{pos} pozicija udarca
- ray_{dir} smjer zrake

Pomoću pozicije udarca izračunat će se boja površine mjesta gdje je zraka udarila sferu.

Također se treba izračunati omjer slike. Uzmem li se da je:

$$x_{min} = -1$$

lijevi kut slike

$$x_{max} = +1$$

desni kut slike

gdje x predstavlja širinu slike.

Varijabla y koja predstavlja visinu slike ne može se pretpostaviti da je:

$$y_{min} = -1$$

gornji kut

$$y_{max} = +1$$

doljnji kut

jer bi slika izašla nepoželjnog formata.

Za izračun y_{min} i y_{max} potreban je omjer. Koji se ovako izračuna:

$$AspectRatio = \frac{width}{height} = \frac{320}{200} = 1.6 \tag{5.6}$$

Nakon što se izračunao omjer, kreće se s izračunom y koordinate:

$$y_{max} = \frac{1}{Aspectration} = \frac{1}{1.6} = +0.625$$
$$y_{min} = -0.625 \tag{5.7}$$

5.3. Implementacija u programskom jeziku Python

U ovom koraku završnog rada, u Python datoteci u kojoj je već uvrštena klasa za vektor, sliku i boju, implementirat će se zasebna klasa za sferu. Također će se primijenit zasebne klase za scenu i RenderEngin. Boja sfere bit će crvena. Klasa scene će sadržavati podatke o visini i širini izrađenog prikaza, objektu i kameri. Također će se implementirati klasa zrake koja će sadržavati smjer i početak, kao i klasa sfere sa centrom, polumjerom i materijalom. U sljedećim slika nalazi se implementacija koda u programskom jeziku Python.

Vrijednosti potrebne za kreiranje slike su:

- Slika 320 × 200
- Pozicija kamere: (0, 0, 1)
- Pozicija sfere: (0, 0, 0)

- Polumjer: 0.5
- Boja sfere: #FF0000 (crvena)

```
class Color(Vector):

    @classmethod
    def from_hex(cls, hexcolor="#000000"):
        x = int(hexcolor[1:3], 16) / 255.0
        y = int(hexcolor[3:5], 16) / 255.0
        z = int(hexcolor[5:7], 16) / 255.0
        return cls(x, y, z)
```

Slika 5.2 Implementacija boje u programskom jeziku Python

```
class Ray:

    def __init__(self, origin, direction):
        self.origin = origin
        self.direction = direction.normalize()
```

Slika 5.3 Implementacija zrake u programskom jeziku Python

```
class Point(Vector):

    pass
```

Slika 5.4 Implementacija točke u programskom jeziku Python

```
class Scene:

    def __init__(self, camera, objects, width, height):
        self.camera = camera
        self.objects = objects
        self.width = width
        self.height = height
```

Slika 5.5 Implementacija scene u programskom jeziku Python

```
from math import sqrt

class Sphere:

    def __init__(self, center, radius, material):
        self.center = center
        self.radius = radius
        self.material = material

    def intersects(self, ray):
        sphere_to_ray = ray.origin - self.center
        # a = 1
        b = 2 * ray.direction.dot_product(sphere_to_ray)
        c = sphere_to_ray.dot_product(sphere_to_ray
                                      ) - self.radius * self.radius
        discriminant = b * b - 4 * c

        if discriminant >= 0:
            dist = (-b - sqrt(discriminant)) / 2
            if dist > 0:
                return dist
        return None
```

Slika 5.6 Implementacija sfere u programskom jeziku Python

```
class RenderEngine:

    def render(self, scene):
        width = scene.width
        height = scene.height
        aspect_ratio = float(width) / height
        x0 = -1.0
        x1 = +1.0
        xstep = (x1 - x0) / (width - 1)
        y0 = -1.0 / aspect_ratio
        y1 = +1.0 / aspect_ratio
        ystep = (y1 - y0) / (height - 1)

        camera = scene.camera
        pixels = Image(width, height)

        for j in range(height):
            y = y0 + j * ystep
            for i in range(width):
                x = x0 + i * xstep
                ray = Ray(camera, Point(x, y) - camera)
                pixels.set_pixel(i, j, self.ray_trace(ray, scene))
        return pixels

    def ray_trace(self, ray, scene):
        color = Color(0, 0, 0)
        # Find the nearest object hit by the ray in the scene
        dist_hit, obj_hit = self.find_nearest(ray, scene)
        if obj_hit is None:
            return color
```

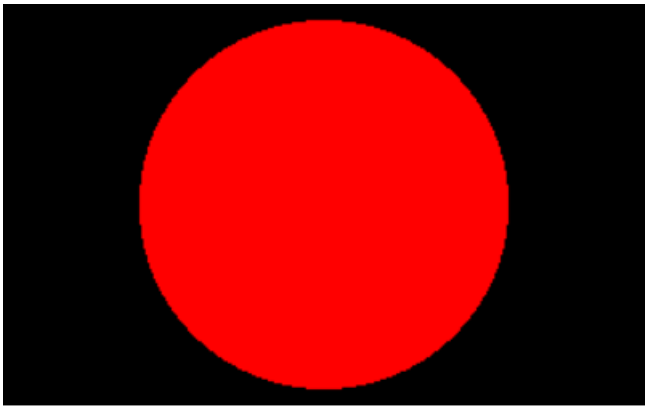
```
    def find_nearest(self, ray, scene):
        dist_min = None
        obj_hit = None
        for obj in scene.objects:
            dist = obj.intersects(ray)
            if dist is not None and (obj_hit is None or dist < dist_min):
                dist_min = dist
                obj_hit = obj
        return (dist_min, obj_hit)

    def color_at(self, obj_hit, hit_pos, scene)
        return obj_hit.material
```

Slika 5.7 Implementacija koda RenderEngine u programskom jeziku Python

```
def main():
    WIDTH = 320
    HEIGHT = 200
    camera = Vector(0, 0, -1)
    objects = [Sphere(Point(0, 0, 0), 0.5, Color.from_hex("#FF0000"))]
    scene = Scene(camera, objects, WIDTH, HEIGHT)
    engine = RenderEngine()
    image = engine.render(scene)

    with open("test11.ppm", "w") as img_file:
        image.write_ppm(img_file)


if __name__ == "__main__":
    main()
```

Slika 5.8 Implementacija glavnog dijela (main) u programskom jeziku Python

Na kraju ovog koraka izrađena je slika:



Slika 5.9 Prikaz 3D sfera u 2D prostoru

6. KOMPONENTE AMBIJENTA, DIFUZNOG I ZRCALNOG OSVJETLJENJA

Za sada u algoritmu za praćenje zraka svjetlosti nedostaju komponente svjetlosti, a osvjetljenje može predstavljati veliku razliku kod iscrtavanja slike scene. U ovom koraku će se uvesti klasa svjetlost. Svjetlost u stvarnosti se može odbiti od objekata, lomiti se, reflektirati itd. Za sada će bit fokus samo na tri komponente koje prikazuju kako se svjetlost odbija od objekata. Te tri komponente su ambijent, difuzno i zrcalno sjenčanje.

Ambijent (engl. ambient) je pojam kada postoji, u prostoru kojeg promatramo, barem malo svjetlosti. Dovoljno da se donekle prepoznaju oblici. Ustvari sjene u stvarnome životu nisu potpuno crne. Npr. kada se uđe u tamnu sobu i preleti dovoljno vremena da se oči priviknu na tamu mogu se vidjeti lagani obrisi objekata. Iako nema izvora svjetlosti. To se zove ambijent svjetlo.

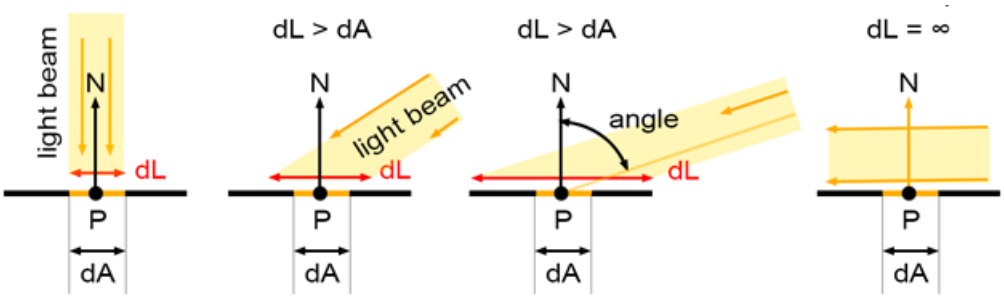
Difuzno sjenčanje (engl. diffuse reflection) se događa kada zraka svjetla udari o mat ili drvene površine. Tada se svijetlo raspršii ili rasijava u različitim smjerovima. Rasijavanje se događa na svim objektima osim potpuno reflektirajuće površine, kao što je zrcalo.

Zrcalno sjenčanje (engl. specular reflection) je sjajna točka svjetla koja se reflektira od glatke površine. Ta se pojava često može uočiti na materijalima od metala i mramora. Većina objekata pokazuje zrcalno sjenčanje do neke mjere.

6.1. Lambert-ov model sjenčanja

Umjesto da se promatra točku koja u fizičkom svijetu nema nikakvo značenje, promatrat će se vrlo malo područje oko točke dA. To sve je prikazano na slici ispod paragrafa. Ono čemu će se posvetit pažnja je količina svjetlosti koja pada na površinu malog područja oko točke P. Svjetlost koja pada na P nije prikazana kao samo jedna svjetlosna zraka, već snop svjetlosti. Pošto nas ne zanima samo jedna točka, već površina dA. Svjetlost koja pada na ovu površinu

sadržana je u malom volumenu okomitom na dA kao što je prikazano lijevo na slici. Što ukazuje da sva svjetlosna energija sadržana u ovom volumenu koja ima isti presjek kao dA pada na dA. Ako konstantan tok svjetlosne energije putuje kroz ovaj volumen kao što je prikazano lijevo na slici, tada se može pretpostaviti da postoji stalna količina svjetlosne energije koja udara o površinu dA u bilo kojem trenutku. Ali kada se svjetlosni snop malo nagne u desno kod nekim kutom tako da više nije paralelan s normalom N (kao što se može vidjeti na slikama desno od prve) , poprečni presjek snopa s površinom postaje veći od dA. Što je kut između normale N i svjetlosnog snopa sve veći to se sve više povećava poprečni presjek područja oko dA, koje je obasjano svjetlosnim snopom. Stoga će sva energija koja proizlazi iz svjetlosnog snopa biti rasprostranjena na veću površinu. Pošto će ista količina svjetlosne energije biti raspoređena na manje područja koje nam predstavlja dA i na veće područje koje je izvan određene površine dA, površina dA će primati manje svjetlosne energije. Na posljednjoj slici, potpuno na desno, kut se toliko povećavao dok svjetlosni snop nije postao okomit na normalu N. Tako da na posljednjoj slici svjetlosni snop uopće ne dodiruje površinu dA tj. svjetlosna energija ne pogađa dA.



Slika 6.1 Kutovi pada svjetlosnog snopa na površinu [9]

Prijevodi sa slike:

- Light beam (Svjetlosni snop)
- Angle (Kut)

Može se zaključiti da najviše svjetlosne energije površina poprima kada je svjetlosni snop okomit na površinu tj. on je u smjeru normale. A kada svjetlosni snop osvijetli površinu pod nekim kutom tada površina primi manje energije, i kako kut raste površina dA će primati sve manje i manje svjetlosne energije. Stoga može se zaključiti što je kut veći površina će primat manje svjetlosne energije tj. reflektirat će manje svjetlosti. Površina će postati tamnija kako se kut povećava. Taj princip je poznat pod nazivom Lambertov kosinusni zakon. Iluminacija

površine je direktno proporcionalna kosinusu kuta (Ø) između smjera svjetlosti (L) i normale površine (N). Što je izraženo u jednadžbi, gdje N i L predstavljaju vektore:

$$\cos\theta = N \cdot L \tag{6.1}$$

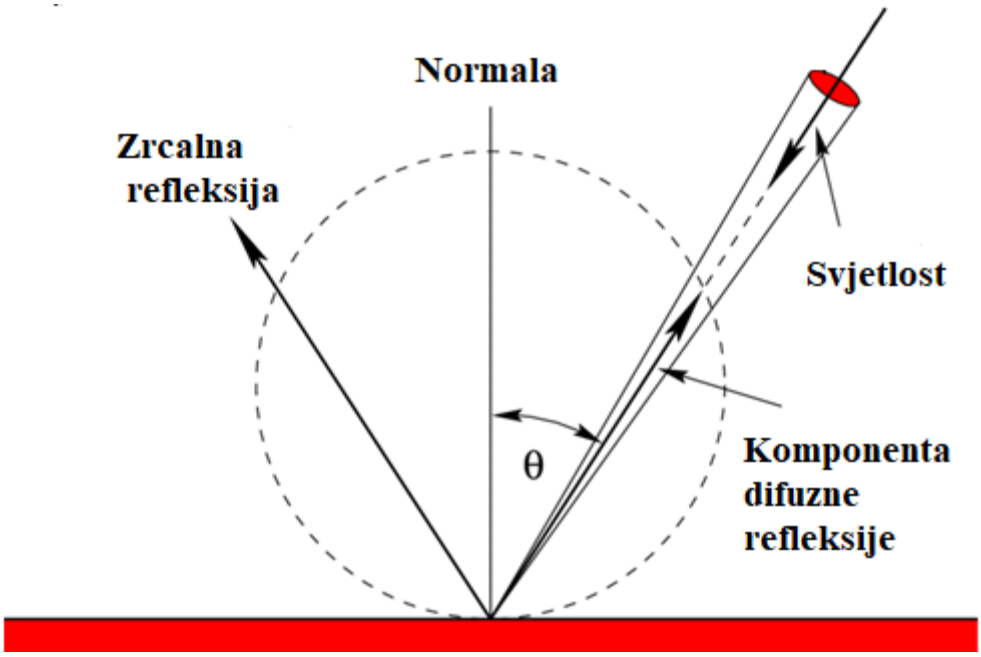
- N normala
- L smjer svjetla
- $cos\theta$ kut između normale i smjera svjetlosti

Ovako će izgledati modificirani Lambert-ov model sjenčanja:

$$C_{surface} = L \cdot N \cdot M_{diffuse} \cdot C_{object} \tag{6.2}$$

- $C_{surface}$ boja objekta u području difuznog sjenčanja
- $M_{diffuse}$ koeficijent materijala kod difuzije
- C_{object} boja objekta
- N normala
- L smjer svjetla

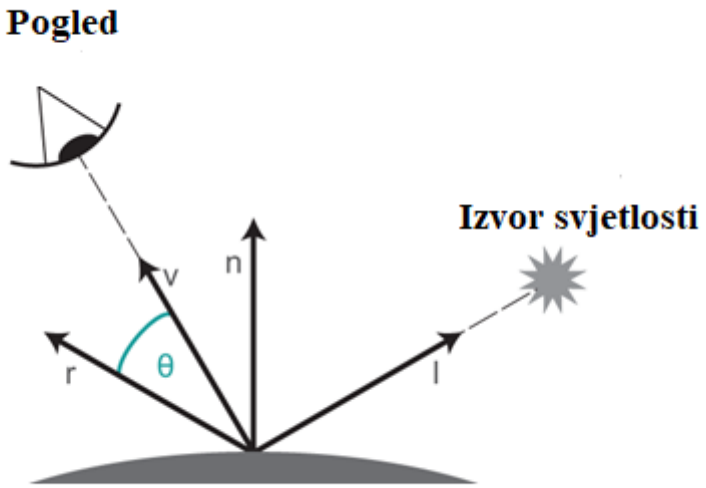
$M_{diffuse}$ ovisi o materijalu. Tehnički ovisi o površini materijala tj. da li je gruba ili glatka. Što je grublja površina to će više doći do izražaja Lambert-ov model sjenčanja.



Slika 6.2 Prikaz lambert-ovog zakona [10]

6.2. Blinn-Phong-ov model sjenčanja

Blinn-Phong model je modifikacija Phong-ova modela. Vrlo je jednostavan, što mu daje popularnost tj. da se često koristi u kompjutorskoj grafici. Slično difuznom sjenčanju, Phong komponenta predstavlja svjetlost koja putuje izravno od izvora svjetlosti do točke zasjenjenja. Zrcalno sjenčanje, za razliku od difuzne komponente, predstavlja svjetlost koja se uglavnom reflektira u savršenom odbijanju, kao od ogledala, čaše vode, itd. To omogućuje površini da izgleda sjajno. U Phong-ovom modelu rasvjete, angažman zrcalnog sjenčanja izračunava se pomoću kosinusa kuta između vektora smjera pogleda (v) i vektora refleksije (r). Kao što je ilustrirano na slici ispod odlomka. Proračuna se umnožak tih dvaju vektora. Rezultat tog umnoška stavlja se na potenciju vrijednosti sjaja materijala (k). To procjenjuje koliko je mala ili velika žarišna točka oko vektora refleksije. Veća vrijednost sjaja stvara manju žarišnu točku, a manja vrijednost veću žarišnu točku. Dobivena vrijednost se na kraju postupka množi s umnoškom materijala površine i zrcalne vrijednosti svjetla.



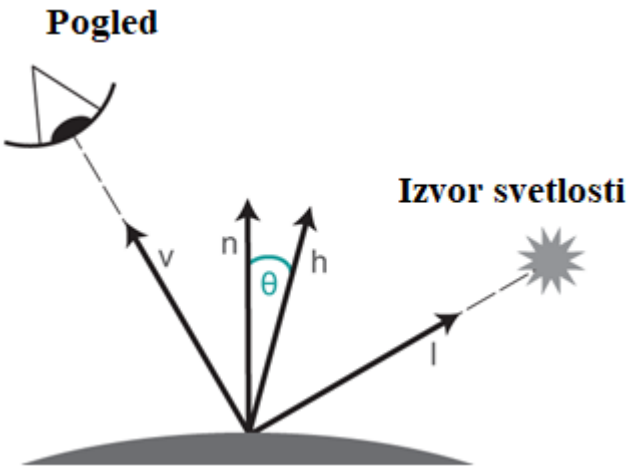
Slika 6.3 Prikaz Phon-ova modela sjenčanja [11]

Formula za Phon-ovo model sjenčanja je:

$$Phong_{Term} = (V \cdot R)^k \tag{6.3}$$

- V zraka prema pogledu
- R smjer reflektirajuće zrake
- k zrcalni koeficijent materijala (sjajnost)

Blinn-ova modifikacija je optimizacija Phon-ove metode tj. postupka kojim se izračunava zrcalno sjenčanje. Kod uporabe Phon-ove metode za izračun se koristi kosinus kuta između vektora smjera pogleda (v) i vektora refleksije (r). Uporabom vrijednosti kosinusa između normale površine (h) i pola vektora (n), Blinn je pojednostavi proračun. Polovični vektor se nalazi u simetričnoj sredini vektora smjera svjetla (r) i vektora smjera pogleda (v). Ovim postupkom zaobilazimo izračunavanje vektora refleksije. Što će učiniti cijeli proces mnogo jednostavnijim i bržim. Također je uveo još jednu optimizaciju, smjer pogleda se može koristiti kao konstantu za objekte koji su vrlo udaljeni naspram promatrača. Naime, vektor smjera svjetlosti se može koristit kao konstantu za objekte koji su vrlo daleko od izvora svjetlosti.



Slika 6.4 Prikaz Blinn-ove modifikacije Phon-onovog modela sjenčanja [12]

Formula za Blinn-ovu modifikaciju je:

$$H = L + V \tag{6.4}$$

$$H = norm(H)$$

- H vektora simetrične sredine između položaja pogleda i izvora svjetlosti
- L izvor svjetlosti
- V položaj pogleda

6.3. Implementacija koda u programskom jeziku Python

U ovom koraku završnog rada u algoritam se uvrste formule za Blinn-Pong model sjenčanja i Lambert-ov model sjenčanja. Pomoću njih prikaz objekta na slici poprimiti će izgled 3D objekta. Lambert-ov model prikazuje koliko se svjetlosti reflektira. Pomoću Blinn-Phon-ove metode koliko je istaknut sjaj. Da bi se ukomponirale ove dvije metode prvo se mora implementirati algoritam za svjetlost. Svjetlost će se također napisati kao klasa. Klasa s bijelom bojom (#FFFFFF), koja je uvezena u main datoteku. Može se izabrati bilo koja boja. Tijekom

implementacije algoritma za motor (engl. engine) upisat će se formule za Blinn-Phong i Lambert-ov model. Primijenit će se model ambijenta u iznosi od 0.05.

```
class Light:

    def __init__(self, position, color=Color.from_hex("#FFFFFF")):
        self.position = position
        self.color = color
```

Slika 6.3 Implementacija svjetlosti u programskom jeziku Python

```
class Material:

    def __init__(
        self, color=Color.from_hex("#FFFFFF"), ambient=0.05, diffuse=1.0,
                                    specular=1.0
    ):
        self.color = color
        self.ambient = ambient
        self.diffuse = diffuse
        self.specular = specular

    def color_at(self, position):
        return self.color
```

Slika 6.4 Implementacija materijala u programskom jeziku Python

```
class Scene:

    def __init__(self, camera, objects, lights, width, height):
        self.camera = camera
        self.objects = objects
        self.lights = lights
        self.width = width
        self.height = height
```

Slika 6.5 Implementacija scene u programskom jeziku Python

```
from math import sqrt

class Sphere:

    def __init__(self, center, radius, material):
        self.center = center
        self.radius = radius
        self.material = material

    def intersects(self, ray):

        sphere_to_ray = ray.origin - self.center
        # a = 1
        b = 2 * ray.direction.dot_product(sphere_to_ray)
        c = sphere_to_ray.dot_product(sphere_to_ray
                                      ) - self.radius * self.radius
        discriminant = b * b - 4 * c

        if discriminant >= 0:
            dist = (-b - sqrt(discriminant)) / 2
            if dist > 0:
                return dist
        return None

    def normal(self, surface_point):
        return (surface_point - self.center).normalize()
```

Slika 6.6 Implementacija sfere u programskom jeziku Python

```
class RenderEngine:
    def render(self, scene):
        width = scene.width
        height = scene.height
        aspect_ratio = float(width) / height
        x0 = -1.0
        x1 = +1.0
        xstep = (x1 - x0) / (width - 1)
        y0 = -1.0 / aspect_ratio
        y1 = +1.0 / aspect_ratio
        ystep = (y1 - y0) / (height - 1)

        camera = scene.camera
        pixels = Image(width, height)

        for j in range(height):
            y = y0 + j * ystep
            for i in range(width):
                x = x0 + i * xstep
                ray = Ray(camera, Point(x, y) - camera)
                pixels.set_pixel(i, j, self.ray_trace(ray, scene))
            print("{:3.0f}%".format(float(j) / float(height) * 100), end="\r")
        return pixels

    def ray_trace(self, ray, scene):
        color = Color(0, 0, 0)
        # Find the nearest object hit by the ray in the scene
        dist_hit, obj_hit = self.find_nearest(ray, scene)
        if obj_hit is None:
            return color
        hit_pos = ray.origin + ray.direction * dist_hit
        hit_normal = obj_hit.normal(hit_pos)
        color += self.color_at(obj_hit, hit_pos, hit_normal, scene)
        return color
```

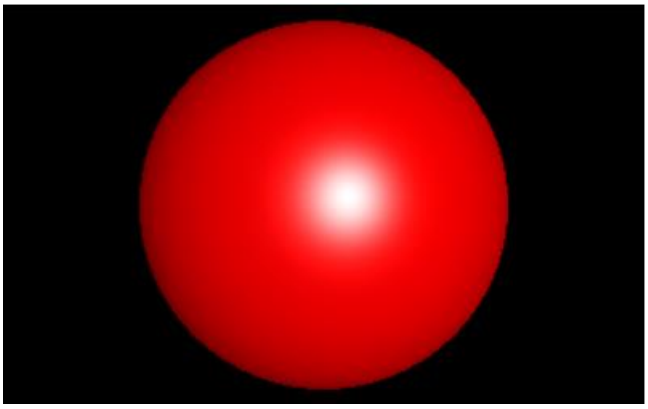
```
    def find_nearest(self, ray, scene):
        dist_min = None
        obj_hit = None
        for obj in scene.objects:
            dist = obj.intersects(ray)
            if dist is not None and (obj_hit is None or dist < dist_min):
                dist_min = dist
                obj_hit = obj
        return (dist_min, obj_hit)

    def color_at(self, obj_hit, hit_pos, normal, scene):
        material = obj_hit.material
        obj_color = material.color_at(hit_pos)
        to_cam = scene.camera - hit_pos
        specular_k = 50
        color = material.ambient * Color.from_hex("#000000")
        # Light calculations
        for light in scene.lights:
            to_light = Ray(hit_pos, light.position - hit_pos)
            # Diffuse shading (Lambert)
            color += (
                obj_color
                * material.diffuse
                * max(normal.dot_product(to_light.direction), 0)
            )
            # Specular shading (Blinn-Phong)
            half_vector = (to_light.direction + to_cam).normalize()
            color += (
                light.color
                * material.specular
                * max(normal.dot_product(half_vector), 0) ** specular_k
            )
        return color
```

Slika 6.7 Implementacija RenderEngine u programskom jeziku Python

```
def main():
    WIDTH = 320
    HEIGHT = 200
    camera = Vector(0, 0, -1)
    objects = [Sphere(
            Point(0, 0, 0), 0.5, Material(Color.from_hex("#FF0000")))]
    lights = [Light(Point(1.5, -0.5, -10.0), Color.from_hex("#FFFFFF"))]
    scene = Scene(camera, objects, lights, WIDTH, HEIGHT)
    engine = RenderEngine()
    image = engine.render(scene)

    with open("test12.ppm", "w") as img_file:
        image.write_ppm(img_file)


if __name__ == "__main__":
```

Slika 6.8 Implementacija glavnog dijela (main) u programskom jeziku Python

Na kraju koraka izrađena je sliku:



Slika 6.9 Slika sfere nakon implementacije modela ambijenta i sjenčanja

7. ZAVRŠNA SCENA

U završnu scenu implementirat će se još jedna sfera i podnožje u stilu šahovnice. Scena na kraju sadrži modele ambijenta, odraza, difuznog i zrcalnog sjenčanja.

7.1. Odraz

Refleksija je jedan od najjednostavnijih oblika interakcije svjetlosti i objekta. Rezultat onog što se događa s fotonima svjetlosti ili upadnom svjetlosnom zrakom nazivamo odraz. Kada svjetlosna zraka udari o površinu objekta, reflektirat će se svjetlost. Ono što se događa svjetlosnoj zraci vrlo je slično onome što se događa bilo kojoj loptici kada udari o pod. Tako će se i svjetlosna zraka odbiti od površine. Stoga odraz nastaje kada reflektirajuća zraka udari o površinu objektan pod kutom refleksije i odbije se od njega. Ako pretpostavi se da je normala okomita linija na površinu (koja reflektira svjetlost) tada je kut refleksije simetričan kutu upada svjetlosti oko normale. Da bi došlo do refleksije zrake površina mora biti glatka i sjajna, kao ogledalo, čaša voda ili materijal od metala. Odraz svjetlosne zrake može se vidjeti samo ako je pogled u istom smjeru kao i smjer putanje reflektirane zrake. Ako se promatrač barem malo pomakne od mjesta gdje zraka reflektira, odraz će se promijeniti. Stoga se može zaključiti da reflektirana slika objekta u sceni mijenja se sa smjerom pogleda. To je razlog zašto se kaže da refleksija svjetlosti ovisi o pogledu, za razliku od difuzne refleksije koja ne ovisi o pogledu jer ne ovisi pod kojim kutom promatrač promatra. Razlog je tome što kut između normale i reflektirajuće zrake mora biti jednaki onome kutu između normale i smjera izvora svjetlosti.

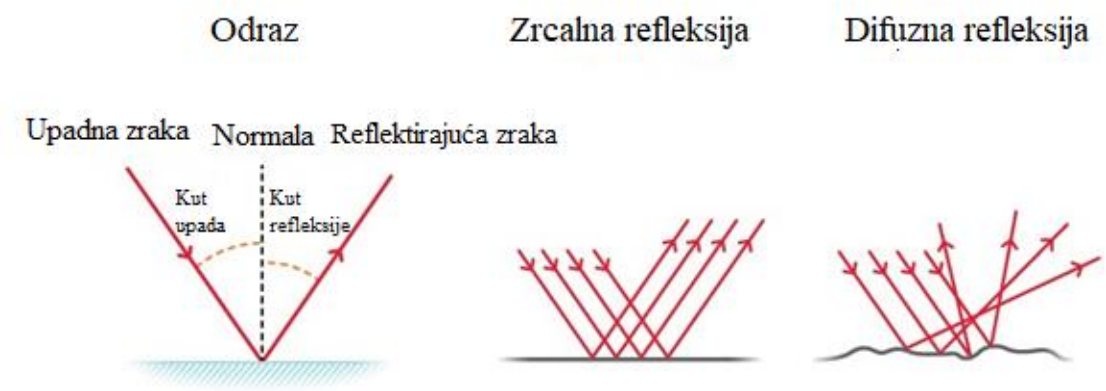
Za sada su se samo slale zrake od kamere do objekta u sceni i pronalazile koje su boje. Zraka se može odbiti od prvotnog objekta zatim pronaći sljedeći objekt. Ako je više objekta, algoritam praćenja zraka svjetlosti može se implementirati tako da se svjetlost odbija od jednog do drugog objekta. Ali nastaje problem ako se svjetlost odbija naprijed-natrag od dva objekta, kao ogledala koja su okrenuta jedno prema drugome. Treba se izračunati kako reflektirajuća zraka izgleda. tj. u kojem smjeru i poziciji će biti ta reflektirajuća zraka.

Formula za pronalaženje reflektirajuće zrake:

$$R_{dir} = L - 2(L \cdot N) \cdot N$$
$$R_{pos} = H_{pos} + N\delta \tag{7.1}$$

- R_{dir} smjer reflektirajuće zrake
- R_{pos} pozicija reflektirajuće zrake
- L vektor zrake
- H_{pos} pozicija udarca zrake
- N normala
- δ koeficijent odraza

Na sljedećoj slici je prikaz odraza zajedno sa zrcalnom refleksijom i difuznom refleksijom iz prošlog poglavlja. Prikazuje na koji način svaka djeluje, koje su im razlike.



Slika 7.1 Razlike između odraza, zrcalne i difuzne refleksije [11]

7.2. Implementacija u programskom jeziku Python

U programskom jeziku Python svaki puta kada se funkcija pozove oduzima se jedan dio stogovne memorije (engl. stack memory). Nakon što se funkcija mnogo puta pozove može se dogoditi „stack overflow“. Zato će se limitirati rekurziju na maksimalnu dubinu(engl. max depth) od pet . Što je prihvatljivo za većinu slika.

Na završnoj slici izraditi će se prikaz poda u stilu šahovnice. Kako izraditi prikaz pod? Zemlja je jedna velika sfera, a ako stojimo na površini zemlje imamo osjećaj da je ravna. Ideja je kreirati jednu veliku sferu koja će predstavljati ravnu površinu na kojoj se nalaze druge dvije sfere. Algoritam se primijenio tako da materijal ima klasu. Ako se želi stvoriti novi uzorak kreirat će se pod-klasa (engl. sub-class) materijala. Kreirat će se još jedna sfera. Isti je princip kao i kod prve sfere. Bit će iste veličine kao i prva. Samo će se promijeniti boju.

```
class Material:
    def __init__(
        self,
        color=Color.from_hex("#FFFFFF"),
        ambient=0.05,
        diffuse=1.0,
        specular=1.0,
        reflection=0.5,
    ):
        self.color = color
        self.ambient = ambient
        self.diffuse = diffuse
        self.specular = specular
        self.reflection = reflection

    def color_at(self, position):
        return self.color

class ChequeredMaterial:

    def __init__(
        self,
        color1=Color.from_hex("#FFFFFF"),
        color2=Color.from_hex("#000000"),
        ambient=0.05,
        diffuse=1.0,
        specular=1.0,
        reflection=0.5,
    ):
        self.color1 = color1
        self.color2 = color2
        self.ambient = ambient
        self.diffuse = diffuse
        self.specular = specular
        self.reflection = reflection

    def color_at(self, position):
        if int((position.x + 5.0) * 3.0) % 2 == int(position.z * 3.0) % 2:
            return self.color1
        else:
            return self.color2
```

Slika 7.1 Implementacija materijala u programskom jeziku Python

```
class RenderEngine:
    """Render 3D objekte u 2D objekte pomocu ray tracing-a"""

    MAX_DEPTH = 5
    MIN_DISPLACE = 0.0001

    def render(self, scene):
        width = scene.width
        height = scene.height
        aspect_ratio = float(width) / height
        x0 = -1.0
        x1 = +1.0
        xstep = (x1 - x0) / (width - 1)
        y0 = -1.0 / aspect_ratio
        y1 = +1.0 / aspect_ratio
        ystep = (y1 - y0) / (height - 1)

        camera = scene.camera
        pixels = Image(width, height)

        for j in range(height):
            y = y0 + j * ystep
            for i in range(width):
                x = x0 + i * xstep
                ray = Ray(camera, Point(x, y) - camera)
                pixels.set_pixel(i, j, self.ray_trace(ray, scene))
            print("{:3.0f}%".format(float(j) / float(height) * 100), end="\r")
        return pixels
```

```
    def ray_trace(self, ray, scene, depth=0):
        color = Color(0, 0, 0)
        # za pronaci najblizi object hit sa zrekom u sceni
        dist_hit, obj_hit = self.find_nearest(ray, scene)
        if obj_hit is None:
            return color
        hit_pos = ray.origin + ray.direction * dist_hit
        hit_normal = obj_hit.normal(hit_pos)
        color += self.color_at(obj_hit, hit_pos, hit_normal, scene)
        if depth < self.MAX_DEPTH:
            new_ray_pos = hit_pos + hit_normal * self.MIN_DISPLACE
            new_ray_dir = (
                ray.direction - 2 * ray.direction.dot_product(
                        hit_normal) * hit_normal
            )
            new_ray = Ray(new_ray_pos, new_ray_dir)
            # za prigušiti refletiranu zreku s koeficijetom reflektiranja
            color += (
                self.ray_trace(
                        new_ray, scene, depth + 1)
                * obj_hit.material.reflection
            )
        return color
```

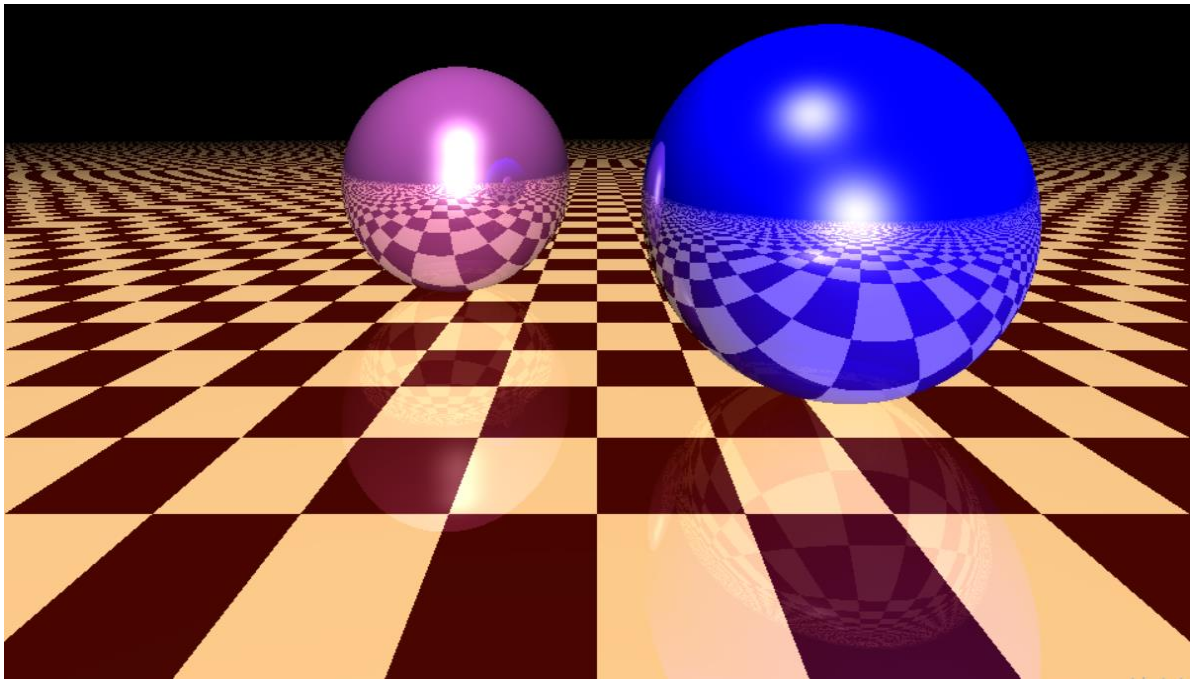
```
    def find_nearest(self, ray, scene):
        dist_min = None
        obj_hit = None
        for obj in scene.objects:
            dist = obj.intersects(ray)
            if dist is not None and (obj_hit is None or dist < dist_min):
                dist_min = dist
                obj_hit = obj
        return (dist_min, obj_hit)

    def color_at(self, obj_hit, hit_pos, normal, scene):
        material = obj_hit.material
        obj_color = material.color_at(hit_pos)
        to_cam = scene.camera - hit_pos
        specular_k = 50
        color = material.ambient * Color.from_hex("#000000")
        # kalkulacije svijetlosti
        for light in scene.lights:
            to_light = Ray(hit_pos, light.position - hit_pos)
            # difunzno sjencaje (Lambert)
            color += (
                obj_color
                * material.diffuse
                * max(normal.dot_product(to_light.direction), 0)
            )
            # speklarno sjencanje (Blinn-Phong)
            half_vector = (to_light.direction + to_cam).normalize()
            color += (
                light.color
                * material.specular
                * max(normal.dot_product(half_vector), 0) ** specular_k
            )
        return color
```

Slika 7.2 Implementacija RenderEngine u programskom jeziku Python

```
def main():
  WIDTH = 960
  HEIGHT = 540
  RENDERED_IMG = "DVIJE_LOPTICE.ppm"
  CAMERA = Vector(0, -0.35, -1)
  OBJECTS = [
    # pod sahovnica
    Sphere(
        Point(0, 10000.5, 1),
        10000.0,
        ChequeredMaterial(
            color1=Color.from_hex("#420500"),
            color2=Color.from_hex("#e6b87d"),
            ambient=0.2,
            reflection=0.2,
        ),
    ),
    # plave lopta
    Sphere(Point(0.75, -0.1, 1), 0.6, Material(Color.from_hex("#0000FF"))),
    # ljubicasta lopta
    Sphere(Point(-0.75, -0.1, 2.25), 0.6, Material(Color.from_hex("#803980")))
  ]
  LIGHTS = [
    Light(Point(1.5, -0.5, -10), Color.from_hex("#FFFFFF")),
    Light(Point(-0.5, -10.5, 0), Color.from_hex("#E6E6E6")),
  ]


  scene = Scene(CAMERA, OBJECTS, LIGHTS, WIDTH, HEIGHT)
  engine = RenderEngine()
  image = engine.render(scene)

  os.chdir(os.path.dirname(os.path.abspath(__file__)))
  with open(RENDERED_IMG, "w") as img_file:
        image.write_ppm(img_file)


if __name__ == "__main__":
    main()
```

Slika 7.3 Implementacija glavnog dijela (main) u programskom jeziku Python

Završetak zadatka:



Slika 7.4 Prikaz slike dviju sfera nakon implementacije modela ambijenta, sjenčanja i odraza

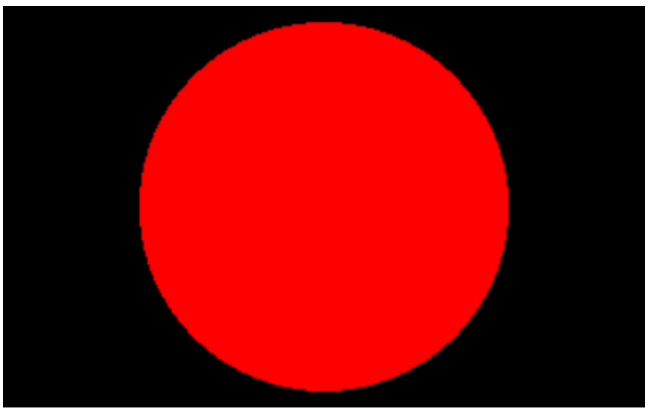
8. ZAKLJUČAK

Praćenje zraka svjetlosti (engl. ray tracing) je algoritam kojim se stvaraju realistične računalne slike. Radi na način tako da imitira prostiranje svjetlost kroz prostor. Taj algoritam je implementirala pomoću programskog jezika Python.

Početak rada je stvaranje najjednostavnijeg zadatka tj. implementacije vektor. Koristila se metoda klase za kreiranje vektora, tako i kod većine ostalih komponenta . Bitna je prvo implementacija vektora jer od njega su proizašle sve ostale komponente.

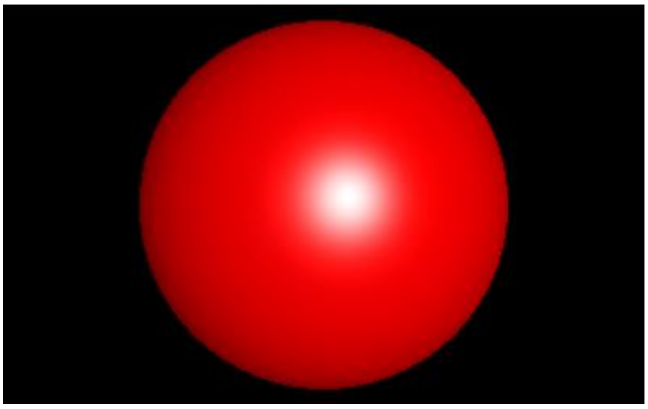
Zatim se krenulo na izradu prikaza (engl. rendering). Primijenile su se nove klase. Klase slika i materijal pomoću klase vektora. Također se kreirala zasebna Python datoteku za izvršenje izrade slike i nazvana je main (glavni dio). Format slike je PPM.

U sljedećem koraku završnog rada u Python datoteci u kojoj je već uvrštena klasa za vektor, sliku i boju, implementirana je zasebna klasa za sferu. Također se primijenila zasebna klasa za scenu i RenderEngin. Klasa scene će sadržavati podatke o visini i širini izrađenog prikaza, objektu i kameri. Također će se implementirati klasa zrake koja će sadržati smjer i početak, kao i klasa sfere sa centrom, polumjerom i materijalom. U sljedećim slika nalazi se implementacija algoritma u programskom jeziku Python.



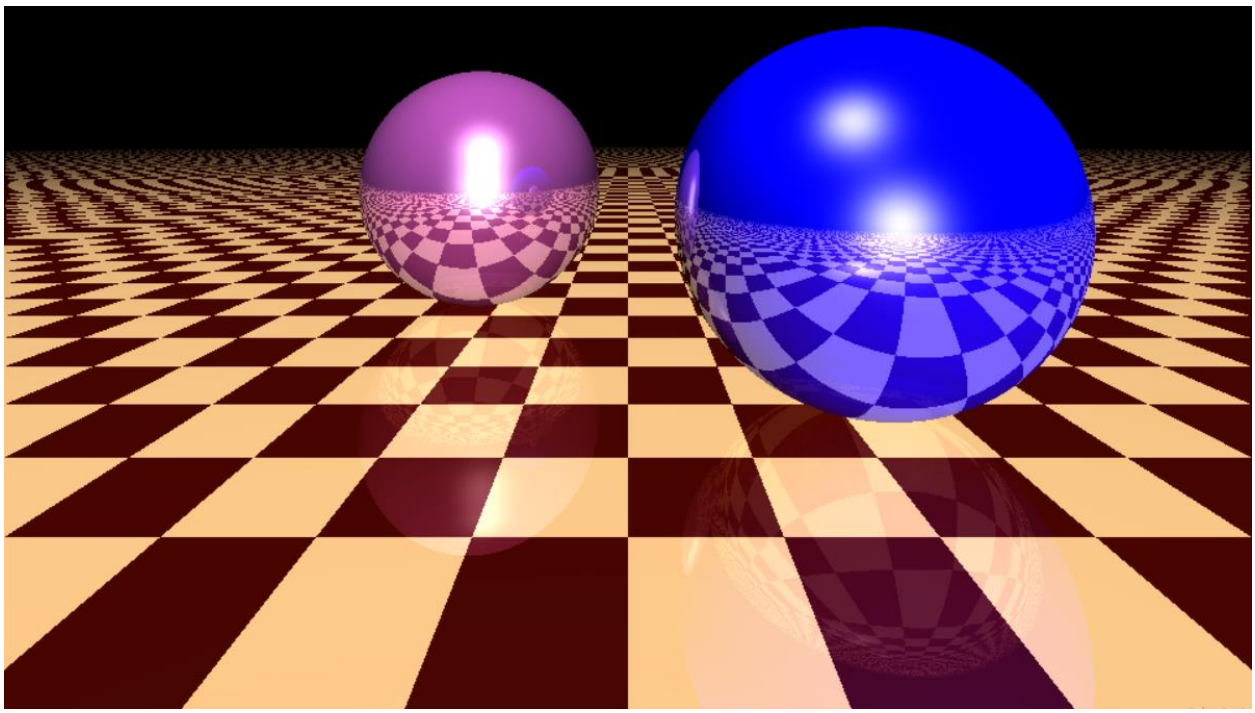
Slika 8.1 Prikaz slike 3D sfere u 2D prostoru

Ambijent, difuzno i zrcalno sjenčanje su načini kako se svijetlost odbija od objekta. Ambijent je pojam kada postoji u prostoru kojeg promatramo barem malo svjetlosti. Dovoljno da se donekle prepoznaju oblici. Difuznim se sjenčanjem svijetlo raspršи u različitim smjerovima pri dodiru neke mat ili drvene površine. Zrcalno sjenčanje je sjajna točka svjetla koja se reflektira od glatke površine. Za izradu ovih troje već navedenih komponenata koristio se Blinn-Phong model sjenčanja i Lambert-ov model sjenčanja. Nakon primjene ovog koraka u algoritam, izradila se slika sfere s komponentama osvjetljenja.



Slika 8.2 Prikaz slike sfere nakon implementacije modela ambijenta i sjenčanja

U završnom koraku izradio se prikaz poda u stilu šahovnice, kao što i još jedna sfera. Isti je princip kao i kod prve sfere. Bit će iste veličine kao i prva. Samo će biti drugačije boje.



Slika 8.4 Prikaz slike dviju sfera nakon implementacije modela ambijenta, sjenčanja i odraza

9. LITERATURA

[1] Henrik: „File:Ray trace diagram.svg“, s Interneta,

https://en.wikipedia.org/wiki/File:Ray_trace_diagram.svg, 12. travnja 2008.

[2] Lu, C.; Roetter, A.; Schultz, A.: „1997 Sophomore College Ray Tracig Site“, s Interneta, https://cs.stanford.edu/people/eroberts/courses/soco/projects/1997-98/ray-tracing/applications.html, 1997

[3] Lu, C.; Roetter, A.; Schultz, A.: „1997 Sophomore College Ray Tracig Site“, s Interneta, https://cs.stanford.edu/people/eroberts/courses/soco/projects/1997-98/ray-tracing/applications.html, 1997

[4] Jefferies, C.: „What Is Ray Tracing? (And What It Means for Pc Gaming)“, s Interneta, https://www.pcmag.com/how-to/what-is-ray-tracing-and-what-it-means-for-pc-gaming, 7. svibnja 2021.

[5] Shirley, P.: „Ray Tracing In One Weekend“, Version 3.2.3., 7. prosinca 2020.

[6] „What is a PPM file“, s Interneta,

https://www.adobe.com/creativecloud/file-types/image/raster/ppm-file.html, 2022.

[7] Shirley, P.: „Ray Tracing In One Weekend“, Version 3.2.3., 7. prosinca 2020.

[8] Shirley, P.: „Ray Tracing In One Weekend“, Version 3.2.3., 7. prosinca 2020.

[9] Scrachapixel: „Computer Graphics for the 'rest of us'“, s Interneta,

https://www.scratchapixel.com, 2022.

[9] Adams, M.: „Lambert's cosin law“, s Interneta,

https://www.researchgate.net/figure/The-variables-which-aaect-diiuse-reeection-according-to-Lamberts-cosine-law_fig2_3430976, travanj 2002.

[10] Kraus, M.: „File:Blinn Vectors.svg“, s Interneta,
https://en.wikipedia.org/wiki/File:Blinn_Vectors.svg, 17. lipnja 2011.

[11] Arien, G.: „Blinn-Phong shading using WebGL“, s Interneta,
https://www.geertarien.com/blog/2017/08/30/blinn-phong-shading-using-webgl/, 30. kolovoza 2017.

[12] Arien, G.: „Blinn-Phong shading using WebGL“, s Interneta, https://www.geertarien.com/blog/2017/08/30/blinn-phong-shading-using-webgl/, 30. kolovoza 2017.

[13] The University of Waikato: „Reflection of light“, s Interneta, https://www.sciencelearn.org.nz/resources/48-reflection-of-light, 27. kolovoza 2020.

10. SAŽETAK I KLJUČNE RIJEČI

Zadatak za završni rad je implementacija algoritma za praćenje zrake svjetlosti u svrhu realističnog prikaza stacionarne scene. Implementirao se algoritam za sve potrebne elemente u programskom jeziku Python da se dobije željena slika. Dala se pažnja materijalu objekta u sceni. Implementirala su se u algoritam dva izvora svjetlosti. Također su se implementirala dva objekta. Korišteni su modeli sjenčanja, odraza, ambijenta. Implementacija je započela od najjednostavnijeg djela tj. vektor. Koristila se metoda klase za stvaranje vektora, i većinu ostalih elemenata. Bitno je prvo primijeniti vektora jer od njega su proizlazile sve ostale komponente za stvaranje zadatka. Zatim se krenulo na izradu prikaza (engl. rendering). Uvrstile su se nove klase. Format slike koji je korišten je PPM. Sljedeće implementirana je formula za sferu u programskom jeziku Python i njezin presjek s zrakom. Ambijent, difuzno i zrcalno sjenčanje su modeli koji prikazuju kako se svjetlost odbija od objekta. Za izradu ovih troje već navedenih modela koristio se Blinn-Phong model sjenčanja i Lambert-ov model sjenčanja. Nakon primjene ovog koraka u algoritam, izrađena je slika sfere s komponentama osvjetljenja. Pri završetku zadatka implementirana je još jedna sfera i pod u stilu šahovnice. U kod se implementirao odraz za realističniji izgled. To je završetak algoritma i završetak rada. Produkt je realistična slika stvorena pomoću praćenja zraka svjetlosti.

Ključne riječi: praćenje, zraka, svjetlost, izrada prikaza

11. ABSTRACT AND KEYWORDS

The task for the final paper is the implementation of an algorithm for ray tracing for the purpose of realistic representation of a stationary scene. An algorithm was implemented for all the necessary elements in the Python programming language to obtain the desired image. Attention was paid to the material of the object in the scene. Two light sources were implemented in the algorithm. Two facilities were also implemented. Shading, reflection, and ambient models were used. The implementation started with the simplest part, i.e. the vector. A class method was used to create the vector, and most of the other elements. It is important to apply the vector first because all the other components for creating the task came from it. Then we started creating rendering. New classes were introduced, using the vector formula. The image format used is PPM. Next, the formula for the sphere in the Python programming language and its intersection with the plane is implemented. Ambient, diffuse and specular shading are models that show how light reflects off an object. The Blinn-Phong shading model and the Lambert shading model were used to create these three aforementioned models. After applying this step in the algorithm, an image of the sphere with lighting components was created. At the end of the task, another sphere and a checkerboard-style floor were implemented. Reflection has been implemented in the code for a more realistic look. This is the end of the algorithm and the end of the work. The product is a realistic image created using light ray tracing.

Keywords: tracing, ray, light, rendering