Implementacija algoritma za praćenje zrake svjetlosti

Grabar, Renco

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Sveučilišni diplomski studij računarstva

Diplomski rad

Implementacija algoritma za praćenje zrake svjetlosti

Rijeka, srpanj 2023.

Renco Grabar
0069078477

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Sveučilišni diplomski studij računarstva

Diplomski rad

Implementacija algoritma za praćenje zrake svjetlosti

Mentor: prof.dr.sc. Jerko Škifić

Rijeka, srpanj 2023.

Renco Grabar
0069078477

Umjesto ove stranice umetnuti zadatak za završni ili diplomski rad

Izjava o samostalnoj izradi rada

Izjavljujem da rad "Implementacija algoritma za praćenje zrake svjetlosti" izradio samostalno uz upotrebu navedene literature i uz pomoć mentora prof.dr.sc Jerka Škifića.

Rijeka, srpanj 2023.

_____
Ime Prezime

Zahvala

Zahvaljujem se mentoru Prof.dr.sc. Jerku Škifiću na savjetima i pomoći tijekom izrade diplomskog rada. Također i svima ostalima koji su me podržavali tijekom izrade rada.

Sadržaj

Popis slika

Popis tablica

Poglavlje 1

Uvod

Računalna grafika odnosi se na područje računarstva koje se bavi prikazom slika na ekranu. Koristi se u mnogim područjima poput video igara, filmova, fotografije, itd. Jedan od glavnih problema prikaza realistične scene je globalno osvjetljenje. Globalno osvjetljenje obuhvaća svjetlost koja ne dolazi samo izravno od izvora svjetla, već se odbija od različitih objekata u sceni. Druga dva problema su realistična simulacija refleksije (sposobnost objekata da odbija svjetlost) i refrakcije (fenomen lomljenja svjetlosti kada prolazi kroz različite materijale).

Prvu modernu implementaciju algoritma za praćenje zrake (engl. ray-tracing) objavio je John Turner Whitted u radu "An Improved Illumination Model for Shaded Display" [1], 1980. godine. U radu je opisana praktična rekurzivna metoda ray-tracing algoritma koja se u nekom obliku koristi i danas. Njegova implementacija algoritma mogla je prikazati scenu s globalnom iluminacijom, refleksijom i refrakcijom. Međutim, velika potreba za resursima još uvijek nije dozvoljavala korištenje algoritma u nekom komercijalnom području. Postoje različite tehnike za ubrzavanje procesa iscrtavanja slike, neke od njih su podjela prostora i višenitnost koje ćemo istražiti u ovom radu. Također modernije implementacije koriste grafičke kartice (GPU) za ubrzavanje procesa.

Ovaj rad proučit će postojeće metode za implementaciju ray-tracing algoritma, ubrzavačkih struktura, simulacije refleksije, refrakcije, sjena kao i različitih ideja vezanih za ray tracing poput materijala, tekstura, itd. Resursi bez kojih ovaj rad ne

bi bio moguć su: The Ray Tracer Challenge [2], Ray Tracing in One Weekend [3].

Drugo poglavlje opisat će osnovnu ideju iza ray-tracing algoritma.

Treće i četvrto poglavlje opisat će matematičke pojmove točke, vektora i transformacijskih matrica koju su potrebni za razumijevanje ostatka rada.

U petom poglavlju prikazat će se matematički izračuni sjecišta zrake i primitivnih tipova, također objasnit će se matematička definicija svakog primitiva. Slično, u šestom poglavlju prikazat će se kako se mogu kombinirati primitivni tipovi u složene cjeline.

Sedmo poglavlje opisat će kameru koja iz 3D prostora generira 2D sliku. Također objasnit će na koji način se generiraju zrake koje odlaze u scenu. Osim toga objasnit će se koncepti focal blura i aliasinga.

Osmo poglavlje opisat će svojstva materijala nekog objekta. Materijal definira na koji način objekt reagira na svijetlo, također koje je boje ili koja tekstura je primijenjena na objekt.

Deveto, deseto i jedanaesto poglavlje fokusirat će se na fizikalna svojstva materijala poput sjaja, prozirnosti, reflektivnih i refraktivnih svojstva. Bit će pokazane sve jednadžbe i izračuni za implementaciju istih.

U dvanaestom poglavlju prikazat će se način na koji se određuje sjena u sceni. Trinaesto poglavlje proširit će sjene definiranjem različitih izvora svjetlosti koji će za rezultat proizvesti meke sjene.

Četrnaesto poglavlje opisat će ubrzavačku strukturu bounding volume hierarchy (BVH)[1] i njezin utjecaj na performanse.

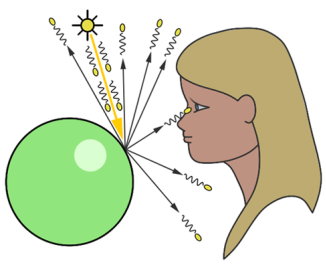
Petnaesto poglavlje fokusirat će se na implementacijske detalje, probleme i poboljšanja. U Šesnaestom poglavlju navest će se neka od mogućih poboljšanja ovog rada na koja sam naišao.

[1]engl. Bounding Volume Hierarchies - ubrzavačka struktura koja grupira objekte u stablo

Poglavlje 2

Opis ray-tracing algoritma

U stvarnome svijetu fotoni dolaze iz različitih izvora svijetla do svih objekta oko nas, prolaze kroz ili se odbijaju od različitih površina i dolaze do očiju. Mozak onda sve različite zrake svijetla pretvara u sliku. Isti scenarij možemo pretočiti u svijet računalne grafike, odnosno simuliramo izvor svijetla i zrake iz izvora pratimo do objekta u sceni. Zraka će se odbiti od objekta u sceni i završiti u oku (kameri). Ova tehnika prati zraku od izvora do oka ili kamere (engl. forward tracing) kao što se vidi sa slike 2.1 samo jako mali broj zraka se odbija od površine i završava u oku (kameri).

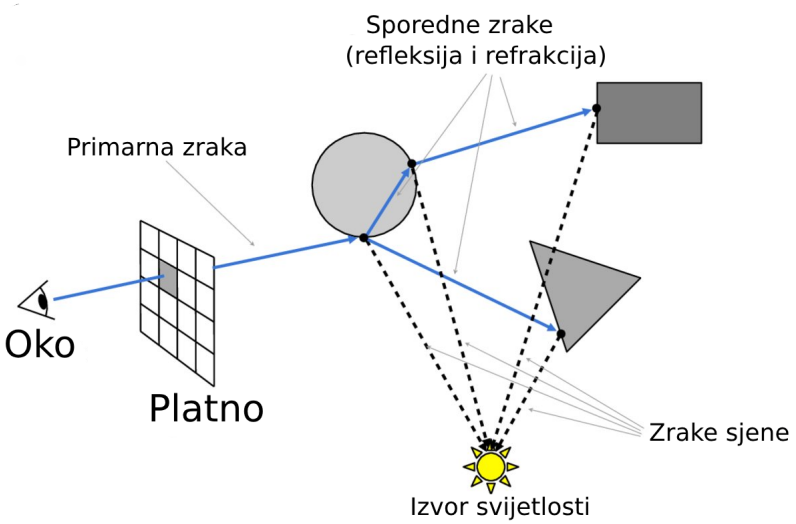


Slika 2.1 Od velikog broja fotona koji dolaze iz izvora svjetlosti, samo neki se odbijaju prema oku[4]

S obzirom na to da simulacije koje prate zraku od svjetla do odredišta nisu efikasne, u simulacijama se zrake češće prate od receptora (kamere ili oka) do izvora svjetla. To nam omogućava da simuliramo puno manji broj zraka u odnosu na prijašnju metodu. Put zrake u simulaciji suprotan je od stvarnosti (engl. backward

tracing).

Backward tracing (koji je implementiran u ovome radu), generira zrake iz oka (kamere), zrake prolaze kroz platno (na platnu će se 3D scena preslikati u 2D scenu) i odlaze dalje u scenu. Na slici 2.2 vidimo da generirana primarna zraka (engl. primary ray) prolazi kroz platno i dolazi do kruga. U točci presjeka zrake i kruga generiraju se sporedne zrake (engl. secondary rays). Na slici 2.2 prikazane su sporedne zrake koje se generiraju za refleksiju, refrakciju i sjene.



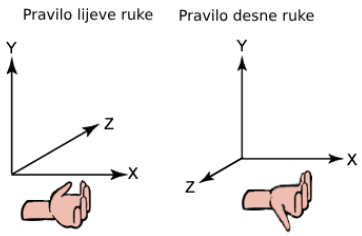
Slika 2.2 Put jedne zrake od oka do izvora svjetlosti [5]

Ukratko ray-tracing generira zrake iz kamere koje prolaze kroz platno dalje u scenu i dolaze do izvora svijetla. Kako zrake prolaze kroz scenu mogu se reflektirati od površine, mogu biti blokirane od strane objekta (znači da imamo sjene) ili mogu prolaziti kroz prozirne/poluprozirne objekte (refrakcija). Sve interakcije se kombiniraju da bi se odredila boja piksela koja će se prikazati na platnu.

Poglavlje 3

Točke, vektori

Ovo poglavlje će služiti kao mali podsjetnik za sve operacije koje se mogu izvesti nad točkama i vektorima. Točke i vektori opisani su s tri parametra x, y i z. Točka je pozicija u 3D prostoru dok vektor ima neku veličinu i smjer. Bitno je napomenuti da postoje i dva tipa koordinatnog sustava (slika 3.1), mogu pratiti pravilo lijeve ili desne ruke. Razlika je smjer u kojem pokazuje Z os. Kod pravila lijeve ruke koje se koristi u ovom radu pozitivan smjer Z osi je "od nas".



Slika 3.1 Koordinatni sustav određen pravilom lijeve i desne ruke [6]

Vrijedi:
točka + vektor -> točka pomaknuta u prostoru za veličinu vektora

$$(P_x, P_y, P_z) + \vec{V} = (P_x + V_x, P_y + V_y, P_z + V_z) \tag{3.1}$$

vektor + vektor -> slično zbrajanju točke i vektora, međutim rezultat je vektor

$$\vec{V} + \vec{M} = \overrightarrow{(V_x + M_x, V_y + M_y, V_z + M_z)} \tag{3.2}$$

točka - točka -> točke nije moguće zbrajati ali ih je moguće oduzimati, što rezultira vektorom koji pokazuje od jedne točke prema drugoj.

$$(P_y, P_y, P_z) - (T_x, T_y, T_z) = \overrightarrow{(P_x - T_x, P_y - T_y, P_z - T_z)} \tag{3.3}$$

točka - vektor -> rezultat je točka

$$(P_x, P_y, P_z) - \vec{V} = (P_x - V_x, P_y - V_y, P_z - V_z) \tag{3.4}$$

vektor - vektor -> rezultat je vektor

$$\vec{V} - \vec{M} = \overrightarrow{(V_x - M_x, V_y - M_y, V_z - M_z)} \tag{3.5}$$

negiranje -> okreće predznak vrijednosti točke ili vektora

$$\begin{aligned} -\vec{P} &= (-P_x, -P_y, -P_z) \\ -\vec{V} &= \overrightarrow{(-V_x, -V_y, -V_z)} \end{aligned} \tag{3.6}$$

Također moguće je točke i vektore množiti sa skalarom (realan broj). U oba slučaja operacije su intuitivne, množenje pomnoži svaku veličinu skalarom, dok se kod dijeljenja svaka veličina dijeli. Za neki skalar n imamo:

$$(P_x, P_y, P_z) \cdot n = (n \cdot P_x, n \cdot P_y, n \cdot P_z) \tag{3.7}$$

$$\vec{V} \cdot n = \overrightarrow{(n \cdot V_x, n \cdot V_y, n \cdot V_z)} \tag{3.8}$$

$$\frac{(P_x, P_y, P_z)}{n} = (\frac{P_x}{n}, \frac{P_y}{n}, \frac{P_z}{n}) \tag{3.9}$$

$$\frac{V}{n} = \overrightarrow{(\frac{V_x}{n}, \frac{V_y}{n}, \frac{V_z}{n})} \tag{3.10}$$

Operacije koje su specifične za vektore su duljina (engl. magnitude), normalizacija, skalarni proizvod (engl. dot product) i vektorsko množenje (engl. cross product). Duljina vektora se računa prema sljedećoj formuli:

$$magnitude = \sqrt{x^2 + y^2 + z^2} \tag{3.11}$$

Vektori čija je duljina 1 zovu se jedinični vektori. Normalizacija je proces pretvaranja vektora u jedinični vektor ali sa zadržanim smjerom. Normaliziran vektor dobijemo dijeljenjem njegovih komponenti s duljinom. Formula za normalizaciju je:

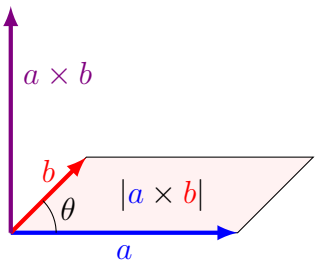
$$nor = \overrightarrow{(\frac{V_x}{mag}, \frac{V_y}{mag}, \frac{V_z}{mag})} \tag{3.12}$$

Skalarni proizvod (engl. Dot product) uzima dva vektora i vraća skalarnu vrijednost. Skalarni proizvod definiran je kao suma umnožaka komponenti svakog vektora.

$$\vec{V} \cdot \vec{M} = Vx \cdot Mx + Vy \cdot My + Vz \cdot Mz \tag{3.13}$$

Vektorsko množenje vraća vektor koji je okomit na oba vektora u operaciji. Također duljina dobivenog vektora odgovara površini koju zatvaraju dva vektora u operaciji. Formula za 3 dimenzionalno vektorsko množenje je:

$$\vec{V} \times \vec{M} = \overrightarrow{Vy \cdot Mz - Vz \cdot My, Vz \cdot Mx - Vx \cdot Mz, Vx \cdot My - Vy \cdot Mx} \tag{3.14}$$



Slika 3.2 Grafički prikaz vektorskog množenja

3.1 Zraka

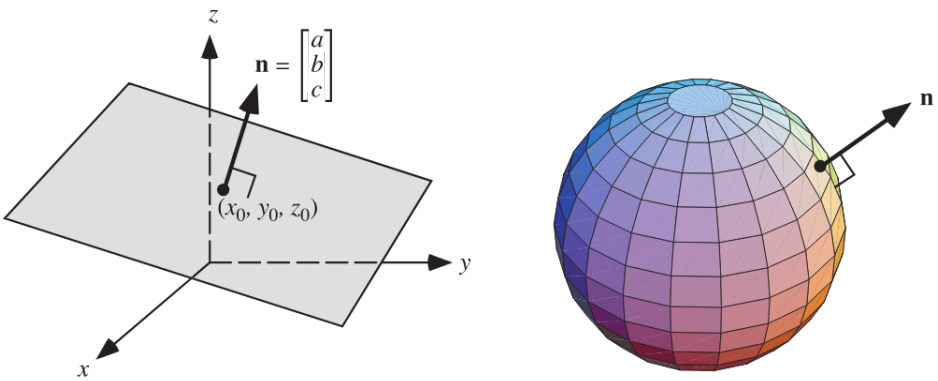
Zraka je opisana ishodištem (O, engl. origin) i smjerom (D, eng. direction). također ima parametar t koji predstavlja vrijeme (engl. time). Parametarska jednadžba zrake je:

$$P(t) = O + tD \tag{3.15}$$

P je 3D pozicija na nekoj 3D liniji u trenutku t, uvrštavanjem realnog broja za t moguće je dobiti bilo koju točku na zraci (za pozitivne vrijednosti dobijemo točke ispred, a za negativne iza zrake).

3.2 Normala

Normala je vektor koji je okomit na neku površinu. Može se izraziti kao vektorski umnožak između dva vektora. Koristi se kod različitih izračuna, na primjer za refleksiju, Phongov model [7], refrakciju, itd.



Slika 3.3 Normala ravnine i sfere [8]

Poglavlje 4

Transformacije

Kako bi izbjegli komplikacije prilikom izrade scene svi objekti se postavljaju u ishodiste i pomiču se pomoću transformacijskih matrica. Da nemamo matrica za npr. pomicanje sfere morali bi joj odrediti radius i centar. Za ostale geometrijske oblike i operacije morali bi primijeniti složene i ne-intuitivne operacije. Transformacije nam omogućavaju da npr. za pomicanje i skaliranje sfere primijenimo transformaciju skaliranja i translacije.

4.1 Translacija

Translacijska matrica pomiče objekt odnosno sve točke objekta u smjeru jedne ili više osi. Translacijska matrica ima jednostavnu formu i primjenom matrice na točku možemo primijetiti da se translacijski vektor (t_x, t_y, t_z) jednostavno dodaje na komponente točke.

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix} \tag{4.1}$$

4.2 Skaliranje

Matrica skaliranja mijenja veličinu objekta širenjem ili sužavanjem točaka u smjeru jedne ili više osi. S_x, S_y i S_z predstavljaju faktore skaliranja u X, Y i Z dimenzijama. Primjenom matrice skaliranja na točku dobijemo točku čije su komponente pomnožene s faktorima skaliranja.

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x \cdot s_x \\ v_y \cdot s_y \\ v_z \cdot s_z \\ 1 \end{pmatrix} \tag{4.2}$$

4.3 Rotacije

Matrice rotacija dosta su kompleksnije od prije navedenih transformacija. Rotacija oko bilo koje osi je kompleksna, a u našem slučaju i nepotrebna, pa su rotacije ograničene na rotiranje oko X, Y i Z osi za neki kut u radijanima. Primjenom matrice rotacije oko X osi na točku dobijemo novu točku zarotiranu za neki kut.
Rotacija oko X osi:

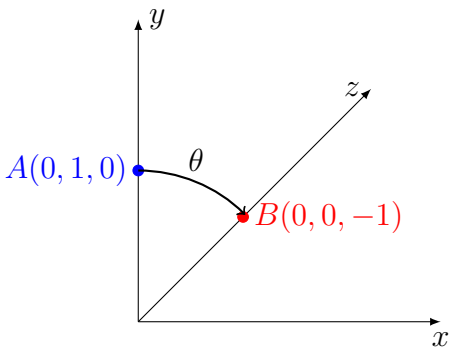
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \cdot cos(\theta) + v_z \cdot -sin(\theta) \\ v_y \cdot sin(\theta) + v_z \cdot cos(\theta) \\ 1 \end{pmatrix} \tag{4.3}$$

Također postoje matrice rotacije za Y i Z os. Koje funkcioniraju na isti način odnosno pomiču točku za neki kut oko odabrane osi.
Rotacija oko Y osi:

$$\begin{pmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

Rotacija oko Z osi:

$$\begin{pmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.5}$$

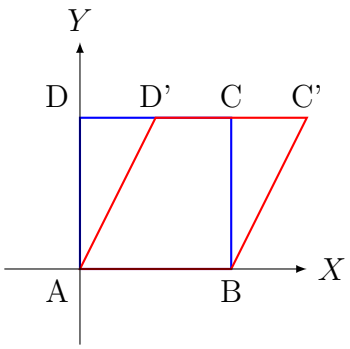


Slika 4.1 Rotacija točke $A(0,1,0)$ u $B(0,0,-1), \theta = \pi/2$

4.4 Smik

Smik matrica se koristi za ukošavanje nekog objekta. Kad se primjeni na neki objekt utječe na točke tako da se svaka točka pomiče ovisno o vrijednostima drugih komponenta vektora. Matrica je oblika:

$$\begin{pmatrix} 1 & x_y & x_z & 0 \\ y_x & 1 & y_z & 0 \\ z_x & z_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.6}$$



Slika 4.2 Smik transformacija kvadrata po X osi

4.5 Ulančavanje transformacija

Transformacije se mogu ulančavati, odnosno umjesto da npr. sferu najprije skaliramo, pa pomaknemo možemo iskoristiti svojstvo asocijativnosti, odnosno pomnožiti translacijsku matricu s matricom skalirana i novu transformaciju primijeniti na sferu. Treba napomenuti da množenje matrica nije komutativno, redoslijed množenja je bitan. Kod transformacija redoslijed operacija se čita s desna na lijevo, što znači da za navedeni primjer skaliranja, pa pomicanja sfere imamo jednadžbu:

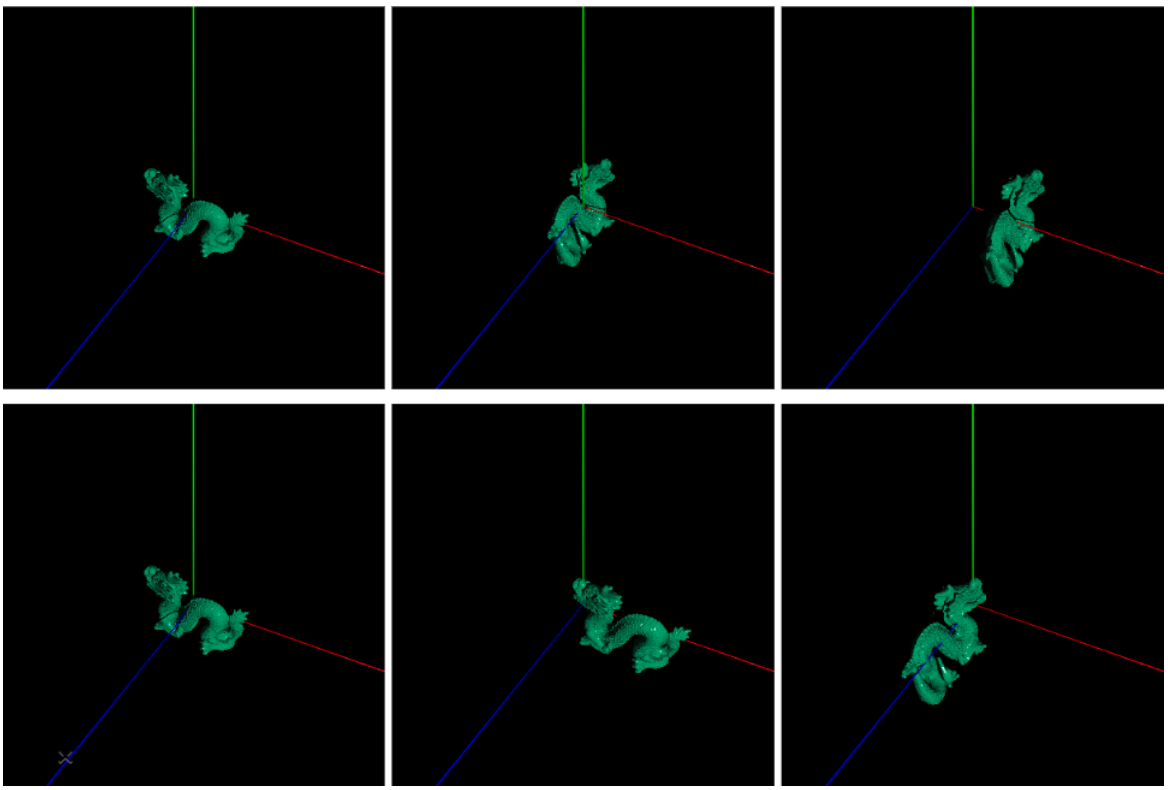
$$novaTransformacija = translacija(x, y, z) \cdot skaliranje(x, y, z) \tag{4.7}$$

Slika 4.3 prikazuje kako različiti redoslijed transformacija, u ovom slučaju primijenjene transformacije su rotacija i translacija, ima drukčiji utjecaj na finalni položaj objekta u sceni. Na slici 4.3 crvena os je X, zelena Y i plava -Z. Prvi red ima sljedeći redoslijed:

$$novaTransformacija = translacija(x, y, z) \cdot rotacijaY(\theta) \tag{4.8}$$

Dok je kod drugog obrnuto:

$$novaTransformacija = rotacijaY(\theta) \cdot translacija(x, y, z) \tag{4.9}$$



Slika 4.3 Različit redoslijed transformacija (slika dobivena vlastitom implementacijom)

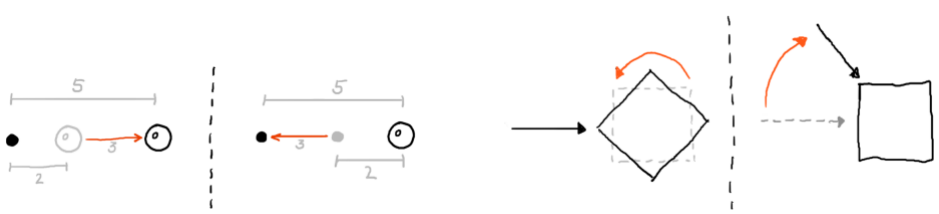
Poglavlje 5

Primitivi

Primitivi su geometrijski oblici za koje je matematički određeno sjecište zrake i oblika. Kombiniranjem primitivnih tipova moguće je stvoriti kompleksne oblike. Najčešći primitivni tipovi su: ravnina, trokut, sfera, kocka, cilindar i stožac.

Sve pokazane jednadžbe oslanjaju se na to da su primitivni tipovi u ishodištu i imaju jediničnu veličinu (npr. sfera ima radius 1). Kako bi pojednostavili računanje umjesto da pomičemo sferu možemo primijeniti inverz transformacije sfere na zraku i time transformirati zraku umjesto sfere. To omogućava da možemo primitivne tipove definirati u ishodištu što znatno pojednostavljuje neke od jednadžbi.

Sa slike 5.1 možemo vidjeti da je svejedno da li translatiramo sferu ili zraku, isto vrijedi i za rotaciju kocke ili pomicanje zrake (također vrijedi za ostale transformacije).



Slika 5.1 Transformacija zrake [2]

5.1 Ravnina

5.1.1 Normala

Ravnina je potpuno ravna površina koja se širi u beskonačnost u dvije dimenzije. Možemo ju postaviti da se prostire u XZ smjeru, pa ako je potrebno uz pomoću matrica transformacija može ju se rotirati u svim smjerovima. S obzirom da ravnina nema nikakvih krivina znamo da ima jednaku normalu u svakoj točci, u slučaju XZ ravnine normala je vektor koji pokazuje u smjeru Y osi, Vektor(0, 1, 0). Matematički normalu možemo odrediti kao vektorski umnožak X i Z osi.

$$\vec{X} =< 1, 0, 0 >, \vec{Z} =< 0, 0, 1 > \tag{5.1}$$

$$\vec{Z} \times \vec{X} =< 0 \cdot 0 - 1 \cdot 0, 1 \cdot 1 - 0 \cdot 0, 0 \cdot 0 - 0 \cdot 1 >=< 0, 1, 0 > \tag{5.2}$$

Redoslijed operacija je bitan, da smo napravili vektorski umnožak obrnutim redoslijedom dobili bi negativnu os Y.

5.1.2 Sjecište

Potrebno je odrediti jednadžbu XZ ravnine. Jednadžba bilo koje 3D ravnine se može zapisati kao:

$$Ax + By + Cz + D = 0 \tag{5.3}$$

gdje su A, B, C koeficijenti normale ravnine, dok je D konstanta. Ako uvrstimo prije dobivene koeficijente normale dobijemo jednadžbu ravnine:

$$\begin{aligned} 0x + 1y + 0z + D &= 0 \\ y + D &= 0 \\ y &= 0 \end{aligned} \tag{5.4}$$

Sjecište zrake i ravnine određujemo samo ako smjer zrake nema y komponentu jednaku ili manju od 0. U suprotnome zraka sigurno ne pogađa površinu, može eventualno ležati direktno na ravnini ali taj slučaj možemo zanemariti pošto je ravnina beskonačno tanka. Ako imamo y komponentu koja je veća od nule imamo i sjecište s

XZ ravninom. Do sjecišta možemo doći na sljedeći način: Parametarski oblik zrake je:

$$P(t) = O + tD \tag{5.5}$$

Gdje je O ishodište zrake, D smjer zrake i t skalar uz pomoću kojega možemo generirati različite točke na zraci. Uvrštavamo u jednadžbu za XZ ravninu i riješimo za t:

$$\begin{aligned} O.y + tD.y &= 0 \\ t &= -O.y/D.y \end{aligned} \tag{5.6}$$

Kako bi dobili sjecište ravnine i zrake potrebno je još uvrstiti dobiven t u parametarsku jednadžbu zrake.

5.1.3 Implementacija

Programski kod 5.1 Sjecište zrake i ravnine, normala

```
Intersections Plane::intersect(const Ray& ray)const {
  if (fabs(ray.direction.y) < EPSILON)
    return {};
  double t = -ray.origin.y / ray.direction.y;

  Intersection* i1 = new Intersection(t, this);

  Intersections inter;
  inter.intersections.insert(i1);

  return inter;
}
Tuple Plane::objectNormal([[maybe_unused]] const Tuple&
   objectPoint, [[maybe_unused]] const Intersection* hit)const
    {
  return Vector(0.0, 1.0, 0.0);
}
```

5.2 Sfera

5.2.1 Sjecište

Definicija jednadžbe koja predstavlja sferu je:

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2 \tag{5.7}$$

S obzirom da ćemo se mi fokusirati na sferu čiji je centar u ishodištu koordinatnog sustava možemo izraz pojednostaviti na:

$$x^2 + y^2 + z^2 = r^2 \tag{5.8}$$

Za *x, y i z* možemo reći da određuju neku točku *P*, pa dobijemo istu jednadžbu u obliku:

$$P^2 - r^2 = 0 \tag{5.9}$$

Ova jednadžba nam govori da postoji set točaka koji definira površinu sfere koja je centrirana u ishodištu i ima radius *r*. Uvrštavanjem jednadžbe za zraku u jednadžbu sfere dobijemo ($O + tD$ definira sve točke na zraci):

$$|O + tD|^2 - r^2 = 0 \tag{5.10}$$

Raspišemo jednadžbu:

$$O^2 + (Dt)^2 + 2ODt - r^2 = O^2 + D^2t^2 + 2ODt - r^2 = 0 \tag{5.11}$$

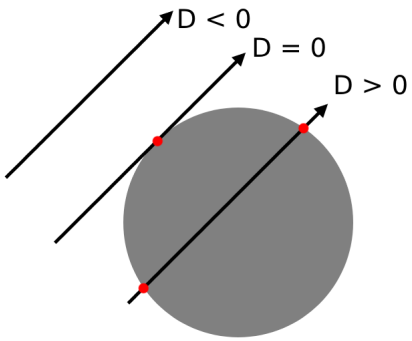
Jednadžba ima oblik kvadratne jednadžbe gdje je:

$$a = D^2, b = 2OD, c = O^2 - r^2 \tag{5.12}$$

Bitno svojstvo kvadratne jednadžbe je determinanta

$$D = b^2 - 4 \cdot a \cdot c \tag{5.13}$$

koja nam govori koliko jednadžba ima rješenja.



Slika 5.2 Determinanta i broj sjecišta

Kad je $D > 0$ imamo dva realna rješenja, kad je $D = 0$ imamo jedno realno rješenje i ako je $D < 0$ nemamo realnih rješenja. Kao i kod ravnine nakon što riješimo kvadratnu jednadžbu rezultat se uvrštava u parametarsku jednadžbu zrake i dobivamo točku sjecišta.

5.2.2 Normala

Normalu sfere računamo tako da od točke sjecišta oduzmemo točku koja predstavlja centar sfere. Kao što je prije navedeno pretpostavljamo da je sfera uvijek u centru koordinatnog sustava, pa treba primijeniti transformacije na točku kako bi dobili točnu normalu. Nakon toga normala se računa prema jednadžbi:

$$normal = sjecište - ishodište \tag{5.14}$$

Gdje za ishodište znamo da je u točci *C(0, 0, 0)*.

5.2.3 Implementacija

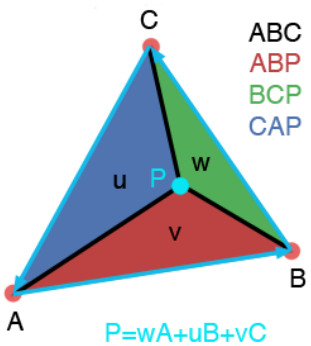
Programski kod 5.2 Sjecište zrake i sfere, normala

```
Intersections Sphere::intersect(const Ray& ray)const {

  auto shapeToRay = ray.origin - Point(0, 0, 0);
  auto a = ray.direction.dotProduct(ray.direction);
  auto b = 2.0 * ray.direction.dotProduct(shapeToRay);
  auto c = shapeToRay.dotProduct(shapeToRay) - 1.0;

  double discriminant = b * b - 4 * a * c;

  if (discriminant < 0)
    return {};

  double t1 = (-b - sqrt(discriminant)) / (2 * a);
  double t2 = (-b + sqrt(discriminant)) / (2 * a);

  Intersection* i1 = new Intersection(t1, this);
  Intersection* i2 = new Intersection(t2, this);

  Intersections inter;
  inter.intersections.insert(i1);
  inter.intersections.insert(i2);

  return inter;
}
Tuple Sphere::objectNormal(const Tuple& objectPoint, [[
    maybe_unused]] const Intersection* hit)const {
  return objectPoint - Point(0, 0, 0);
}
```

5.3 Trokut

5.3.1 Sjecište

Za implementaciju Möller–Trumbore algoritma [9] za sjecište između zrake i trokuta potrebne su baricentrične koordinate. U trokutu ABC određujemo podtrokute CAP za u, ABP za v i BCP za w. Što znači da ih predstavljamo omjerima površine cijelog trokuta i navedenih podtrokuta (slika 5.3).



Slika 5.3 baricentrične koordinate ABC trokuta [10]

$$\begin{aligned} u &= \frac{CAP_{area}}{ABC_{area}} \\ v &= \frac{ABP_{area}}{ABC_{area}} \\ w &= \frac{BCP_{area}}{ABC_{area}} \end{aligned} \tag{5.15}$$

Intuitivno je onda da je: $u + v + w <= 1$. Uz pomoći ovih koordinata moguće je prikazati bilo koju točku P unutar trokuta uz pomoću jednadžbe:

$$P = wA + uB + vC \tag{5.16}$$

Ako je bilo koji od u, v ili w parametara manji od 0 imamo točku koja je izvan trokuta. Kako bi pojednostavili izračun jedan od parametara možemo prikazati

pomoću druga dva:

$$\begin{aligned} P &= (1-u-v)A + uB + vC \\ P &= A - uA - vA + uB + vC \\ P &= A + u(B-A) + v(C-A) \end{aligned} \tag{5.17}$$

Kao u prijašnjim primjerima P možemo zamijeniti s jednadžbom zrake.

$$\begin{aligned} O + tD &= A + u(B-A) + v(C-A) \\ O - A &= -tD + u(B-A) + v(C-A) \end{aligned} \tag{5.18}$$

Kako bi lakše prikazali jednadžbe možemo bridove trokuta označiti sa *E1 = (B - A) E2 = C -A*. Također možemo transformaciju *O - A* koja pomiče trokut iz originalnoga prostora u ishodište koordinatnog sustava prikazati kao: *T*. Bridovi su nam bitni jer koristeći vektorsko množenje možemo dobiti površinu trokuta kojeg zatvaraju, odnosno površinu svakog od podtrokuta. Uz pomoću matričnog zapisa jednadžbe i Cramerovog pravila dobit ćemo sljedeći sustav jednadžbi:

$$\begin{aligned} \begin{bmatrix} -D & (B-A) & (C-A) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} &= O - A \\ \begin{bmatrix} t \\ u \\ v \end{bmatrix} &= \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} \\ \begin{bmatrix} t \\ u \\ v \end{bmatrix} &= \frac{1}{P \cdot E_1} \begin{bmatrix} (Q \cdot E_2) \\ (P \cdot T) \\ (Q \cdot Q) \end{bmatrix} \end{aligned} \tag{5.19}$$

Kao što je prije navedeno moramo provjeriti da su $u > 0$, $v > 0$, $u + v < 1$. Ako su uvjeti zadovoljeni pronašli smo t kojeg je potrebno uvrstiti u početnu jednadžbu zrake i odrediti sjecište s trokutom.

5.3.2 Normala

Za trokut normala se može dobiti iz vektorskog umnoška dvaju bridova trokuta. Bridove trokuta dobijemo tako da oduzmemo parove točaka (vrhovi trokuta).

$$\begin{aligned} \vec{e_1} &= p_2 - p_1 \\ \vec{e_2} &= p_3 - p_1 \\ \overrightarrow{normal} &= \vec{e_1} \times \vec{e_2} \end{aligned} \tag{5.20}$$

Ako učitavamo trokute iz .obj datoteke da bi dobili gladak izgled objekta kao na slici 6.1 iz sljedećeg poglavlja potrebno je interpolirati učitane vrijednosti normala.

To možemo učiniti preko sljedeće jednadžbe:

$$\overrightarrow{normal} = n2 \cdot u + n3 \cdot v + n1 \cdot (1 - u - v) \tag{5.21}$$

Gdje su *n1, n2, n3* vrijednosti normala učitane iz .obj datoteke, a u i v vrijednosti varijabli za mjesto koje je pogođeno zrakom.

5.3.3 Implementacija

Programski kod 5.3 Sjecište zrake i trokuta, normala

```
Intersections Triangle::intersect(const Ray& ray)const {
  auto dirCrossE2 = ray.direction.crossProduct(e2);
  auto det = e1.dotProduct(dirCrossE2);
  if (std::abs(det) < EPSILON)
    return {};

  auto f = 1.0 / det;
  auto p1ToOrigin = ray.origin - p1;
  auto u = f * p1ToOrigin.dotProduct(dirCrossE2);

  if ((u < 0) || (u > 1) || epsilonEqual(0, u) || epsilonEqual
   (1, u))
    return {};
```

```
13
14    auto originCrossE1 = p1ToOrigin.crossProduct(e1);
15    auto v = f * ray.direction.dotProduct(originCrossE1);
16
17    if ((v < 0) || (u + v > 1))
18      return {};
19
20    auto t = f * e2.dotProduct(originCrossE1);
21    Intersection* i1 = new Intersection(t, this);
22    Intersections inter;
23    inter.intersections.insert(i1);
24    return inter;
25  }
26   Tuple Triangle::objectNormal([[maybe_unused]] const Tuple&
       objectPoint, [[maybe_unused]] const Intersection* hit)const
       {
27    return normal;
28  }
29  // SMOOTH TRIANGLE
30  Intersections SmoothTriangle::intersect(const Ray& ray)const {
31    auto dirCrossE2 = ray.direction.crossProduct(e2);
32    auto det = e1.dotProduct(dirCrossE2);
33
34    if (std::abs(det) < EPSILON)
35      return {};
36
37    auto f = 1.0 / det;
38    auto p1ToOrigin = ray.origin - p1;
39    auto u = f * p1ToOrigin.dotProduct(dirCrossE2);
40
41    if ((u < 0) || (u > 1) || epsilonEqual(0, u) || epsilonEqual
       (1, u))
42      return {};
43
44    auto originCrossE1 = p1ToOrigin.crossProduct(e1);
```

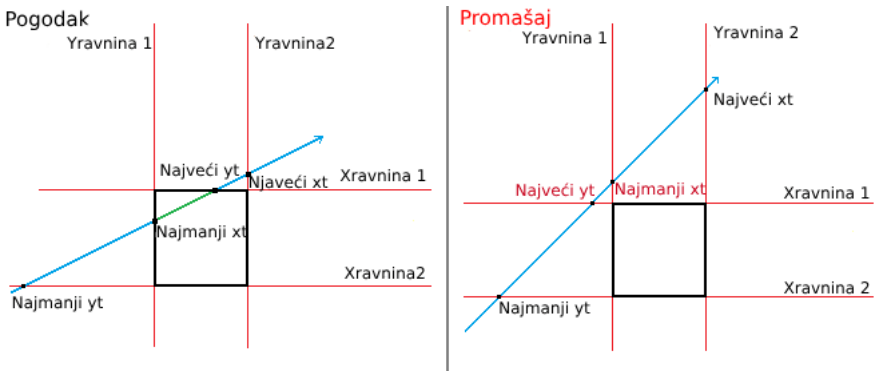
```
45    auto v = f * ray.direction.dotProduct(originCrossE1);
46
47    if ((v < 0) || (u + v > 1))
48      return {};
49
50    auto t = f * e2.dotProduct(originCrossE1);
51
52    Intersection* i1 = new Intersection(t, this, u, v);
53    Intersections inter;
54    inter.intersections.insert(i1);
55    return inter;
56  }
57  Tuple SmoothTriangle::objectNormal([[maybe_unused]] const
       Tuple& worldPoint, const Intersection* hit)const {
58    return n2 * hit->u + n3 * hit->v + n1 * (1 - hit->u - hit->v
       );
59  }
```

5.4 Kocka

5.4.1 Sjecište

Za lakše određivanje presjeka kocke i zrake kao i za ostale oblike koristit ćemo jediničnu kocku. Jedinična kocka je kocka koja je centrirana u ishodištu i prostire se od *Min(-1, - 1, -1) do Max(1, 1, 1)*, poznata je kao axis-aligned bounding box (AABB) kocka. Kako bi još pojednostavili računanje, kocku možemo podijeliti na 6 ravnina. Od kojih su 3 paralelna para, jedan par za svaku stranu kocke. Svaka ravnina određena je s 4 pravca uz pomoću kojih ćemo testirati da li je došlo do sjecišta kocke i zrake. Jednadžbe pravca odgovaraju minimalnoj i maksimalnoj vrijednosti za svaku os. Recimo da imamo jednadžbu pravca $y = minx$ onda možemo odrediti sjecište pravca ubacivanjem vrijednosti u parametarsku jednadžbu zrake:

$$\begin{aligned} O_x + tD_x &= Min_x \\ t_{minx} &= (Min_x - O_x)/D_x \end{aligned} \tag{5.22}$$



Slika 5.4 Sjecište zrake s pravcima koji definiraju jednu ravninu kocke [11]

Ako isti princip upotrijebimo i na ostalim pravcima koji definiraju ravnine dobit ćemo 6 vrijednosti koje predstavljaju sjecište zrake i ravnina.

$$\begin{aligned} t_{maxx} &= (Max_x - O_x)/D_x \\ t_{miny} &= (Min_y - O_y)/D_y \\ t_{maxy} &= (max_y - O_y)/D_y \\ t_{minz} &= (min_z - O_z)/D_z \\ t_{maxz} &= (max_z - O_z)/D_z \end{aligned} \tag{5.23}$$

Potrebno je odrediti najveći minimum i najmanji maksimum za dobivene parametre. U slučaju da je najveći minimum veći od najmanjeg maksimuma nemamo presjek zrake i kocke, u suprotnome sjecište postoji.

Sa slike 5.4 vidimo da je ravnina opisana s 4 pravca koje smo grupirali u 2 para (Y i X parove). Svaki par ima *najmanju* t i *najveću* t vrijednost, *najmanji* t je najmanja vrijednost t za par pravaca, dok je *najveći* t najveća gdje je t točka sjecišta. Kvadrat će se uvijek sjeći u najvećem minimumu i najmanjem maksimumu, zbog čega vrijedi prije navedeno pravilo za određivanje sjecišta.

5.4.2 Normala

S obzirom na to da se kocka u biti sastoji od ravnina za određivanje normale samo moramo znati na kojoj od ravnina se sjecište nalazi. U ovom slučaju imamo kocku

koja se nalazi uvijek u ishodištu i prostire se od *P(-1,-1,-1)* do *E(1, 1, 1)* to znači da će svako sjecište imati jednu komponentu jednaku 1 ili -1. Odnosno ako imamo sjecište *S(1, y, z)* znamo da je sigurno zraka sjekla +x stranu kocke (da je -1 sjekla bi -x, itd). Za normalu onda dobijemo:

$$\begin{aligned} \vec{n} &= \overrightarrow{(\pm x, 0, 0)} \\ \vec{n} &= \overrightarrow{(0, \pm y, 0)} \\ \vec{n} &= \overrightarrow{(0, 0, \pm z)} \end{aligned} \tag{5.24}$$

5.4.3 Implementacija

Programski kod 5.4 Sjecište zrake i kocke, normala

```
void Cube::checkAxis(MinMaxPoint& value, double origin, double
    direction)const {
  auto tminNumerator = (-1 - origin);
  auto tmaxNumerator = (1 - origin);
  if (fabs(direction) >= EPSILON) {
    value.tMin = tminNumerator / direction;
    value.tMax = tmaxNumerator / direction;
  }
  else {
    value.tMin = tminNumerator * INFINITY;
    value.tMax = tmaxNumerator * INFINITY;
  }
  if (value.tMin > value.tMax) {
    auto tmp = value.tMax;
    value.tMax = value.tMin;
    value.tMin = tmp;
  }
}
Intersections Cube::intersect(const Ray& ray)const {
  MinMaxPoint tmpMinMax;
  checkAxis(tmpMinMax, ray.origin.x, ray.direction.x);
```

```
21    auto [xTmin, xTmax] = tmpMinMax;
22
23    checkAxis(tmpMinMax, ray.origin.y, ray.direction.y);
24    auto [yTmin, yTmax] = tmpMinMax;
25
26    checkAxis(tmpMinMax, ray.origin.z, ray.direction.z);
27    auto [zTmin, zTmax] = tmpMinMax;
28
29    auto tmin = std::fmax(xTmin, std::fmax(yTmin, zTmin));
30    auto tmax = std::fmin(xTmax, std::fmin(yTmax, zTmax));
31
32    if (tmin > tmax)
33      return{};
34
35    Intersections inter;
36    inter.intersections.insert(new Intersection(tmin, this));
37    inter.intersections.insert(new Intersection(tmax, this));
38    return inter;
39  }
40  Tuple Cube::objectNormal(const Tuple& objectPoint, [[
       maybe_unused]] const Intersection* hit)const {
41    auto maxc = std::fmax(fabs(objectPoint.x), fabs(std::fmax(
       fabs(objectPoint.y), fabs(objectPoint.z))));
42    if (epsilonEqual(maxc, fabs(objectPoint.x))) {
43      return Vector(objectPoint.x, 0, 0);
44    }
45    else if (epsilonEqual(maxc, fabs(objectPoint.y))) {
46      return Vector(0, objectPoint.y, 0);
47    }
48    return Vector(0, 0, objectPoint.z);
49  }
```

5.5 Valjak

5.5.1 Sjecište

Jednadžba beskonačnog valjka s radijusom 1 oko Y osi je

$$x^2 + z^2 - 1 = 0 \tag{5.25}$$

Uvrstimo vrijednosti iz parametarske jednadžbe zrake.

$$\begin{aligned} (O_x + D_x t)^2 + (O_z + D_z t)^2 - 1 = 0 \\ t^2(D_x^2 + D_z^2) + 2(O_x D_x + O_z D_z)t + (O_x^2 + Z_y^2) - 1 = 0 \end{aligned} \tag{5.26}$$

Ako nakon rješavanja kvadratne jednadžbe dobijemo realna rješenja za t možemo ga uvrstiti u jednadžbu zrake i dobiti sjecište zrake i cilindra. Međutim trenutno valjak je beskonačno dug. Ako želimo ograničiti visinu moramo izračunati vrijednost y točke sjecišta i provjeriti da li je veća ili manja od zadanog minimuma odnosno maksimuma i ovisno o tome tretirati nešto kao sjecište ili ne.

$$\begin{aligned} y0 = O_y + t0 \cdot D_y \\ y1 = O_y + t1 \cdot D_y \end{aligned} \tag{5.27}$$

Smatramo da je sjecište valjano ako je: $minimum < y < maksimum$ i $minimum < y1 < maksimum$. Osim toga cilindar može biti otvoren i zatvoren s obje strane, odnosno omeđen je s dva diska. Kako bi provjerili da li smo pogodili jedan od diskova koristit ćemo sličnu jednadžbu kao kod ravnine, međutim sad je pomaknuta za neki y (minimum i maksimum) u smjeru Y osi. Jednadžbe ravnine tako postaju: $Y = minimum$ *i* $Y = maksimum$.

$$\begin{aligned} y &= minimum \\ O.y + tD.y &= minimum \\ t &= \frac{minimum - O.y}{D.y} \end{aligned} \tag{5.28}$$

Na isti način primijenimo jednadžbe da bi dobili *maksimum*. Kako smo odredili t na beskonačno velikoj ravnini kad uvrstimo t nazad u formulu zrake kako bi dobili točke x, z moramo provjeriti i da je sjecište unutar kruga.

$$\begin{aligned} x &= O.x + t \cdot D.x \\ z &= O.z + t \cdot D.z \\ x^2 \cdot z^2 &<= radius \end{aligned} \tag{5.29}$$

5.5.2 Normala

Za jedinični valjak znamo da je moguće da je sjecište na plaštu valjka ili ako je zatvoren možemo imati presjek s jednom od "ravnina" koja ga zatvara. Ako je pogođen plašt, vraća se vektor s x i z vrijednostima jednakim točci i zanemarujemo y vrijednost. Ako je cilindar zatvoren i pogođen je gornji ili donji dio cilindra onda vraćamo normalu kao da su to ravnine samo mijenjamo predznak y vrijednosti. Znamo da su ravnine pogođene u slučaju da je udaljenost točke od Y osi manja od 1 ($distY = sqrt(x^2 + z^2)$) i ako je y vrijednost jednaka maksimalnoj ili minimalnoj visini cilindra.

Vektori normala su (redom za gornju, donju ravninu i plašt):

$$\begin{aligned} \vec{n} &= \overrightarrow{(0, 1, 0)} \\ \vec{n} &= \overrightarrow{(0, -1, 0)} \\ \vec{n} &= \overrightarrow{(x, 0, z)} \end{aligned} \tag{5.30}$$

5.5.3 Implementacija

Programski kod 5.5 Sjecište zrake i valjka, normala

```
bool Cylinder::checkCap(const Ray& ray, double t)const {
  auto x = ray.origin.x + t * ray.direction.x;
  auto z = ray.origin.z + t * ray.direction.z;
  return(x * x + z * z) <= 1;
}

void Cylinder::intersectCaps(const Ray& ray, Intersections&
   inter)const {
```

```
8   if (!closed || epsilonEqual(ray.direction.y, 0)) {
9     return;
10  }
11  auto t = (minimum - ray.origin.y) / ray.direction.y;
12  if (checkCap(ray, t)) {
13    inter.intersections.insert(new Intersection(t, this));
14  }
15  t = (maximum - ray.origin.y) / ray.direction.y;
16  if (checkCap(ray, t)) {
17    inter.intersections.insert(new Intersection(t, this));
18  }
19  return;
20 }
21
22 Intersections Cylinder::intersect(const Ray& ray)const {
23  bool cylinderBodyIntersection = true;
24  auto a = ray.direction.x * ray.direction.x + ray.direction.z
      * ray.direction.z;
25
26  if (epsilonEqual(a, 0)) {
27    cylinderBodyIntersection = false;
28  }
29  auto b = 2 * ray.origin.x * ray.direction.x + 2 * ray.origin
     .z * ray.direction.z;
30  auto c = ray.origin.x * ray.origin.x + ray.origin.z * ray.
     origin.z - 1;
31  auto disc = b * b - 4 * a * c;
32
33  if (disc < 0)
34    cylinderBodyIntersection = false;
35  Intersections inter;
36  if (cylinderBodyIntersection) {
37    auto t0 = (-b - sqrt(disc)) / (2 * a);
38    auto t1 = (-b + sqrt(disc)) / (2 * a);
39
```

```
40      if (t0 > t1) {
41        std::swap(t0, t1);
42      }
43
44      auto y0 = ray.origin.y + t0 * ray.direction.y;
45      auto y1 = ray.origin.y + t1 * ray.direction.y;
46
47      if (minimum < y0 && y0 < maximum) {
48        inter.intersections.insert(new Intersection(t0, this));
49      }
50      if (minimum < y1 && y1 < maximum) {
51        inter.intersections.insert(new Intersection(t1, this));
52      }
53    }
54    intersectCaps(ray, inter);
55    return inter;
56  }
57
58  Tuple Cylinder::objectNormal(const Tuple& objectPoint, [[
       maybe_unused]] const Intersection* hit)const {
59    auto dist = objectPoint.x * objectPoint.x + objectPoint.z *
       objectPoint.z;
60    if (dist < 1 && objectPoint.y >= maximum - EPSILON)
61      return Vector(0, 1, 0);
62
63    else if (dist < 1 && objectPoint.y <= minimum + EPSILON)
64      return Vector(0, -1, 0);
65
66    return Vector(objectPoint.x, 0, objectPoint.z);
67  }
```

5.6 Stožac

5.6.1 Sjecište

Stožac je poprilično sličan cilindru, oba se prostiru beskonačno uz Y os, mogu se ograničiti i može biti otvoren ili zatvoren. Jednadžba stošca je

$$x^2 + z^2 - y^2 = 0 \tag{5.31}$$

Uvrstimo vrijednosti iz parametarske jednadžbe zrake:

$$\begin{aligned} (O_x + D_x \cdot t)^2 + (O_z + D_z \cdot t)^2 - (O_y + D_y \cdot t)^2 = 0 \\ t^2(D_x^2 + D_z^2 - D_y^2) + 2(O_x D_x + O_z D_z - O_y D_y)t + (O_x^2 + O_z^2 - O_y^2) = 0 \end{aligned} \tag{5.32}$$

Rješavanjem kvadratne jednadžbe dobijemo t i nakon uvrštavanja u parametarsku jednadžbu zrake dobijemo točku sjecišta. Bitno je napomenuti da ako je $a == 0$ *i* b *!=* 0 (gdje je a koeficijent uz t^2 i b koeficijent uz t) imamo samo jedno sjecište sa stošcem, u slučaju da je $a == 0$ *i* $b == 0$ onda nemamo presjek sa stošcem jer zraka prolazi uz površinu stošca. Kao i kod cilindra možemo koristiti dobivenu y vrijednost za ograničiti stožac. Sjecište s bazom se provjerava na sličan način kao i kod cilindra, međutim kod stošca se radius povećava s povećanjem y vrijednosti. Konkretno radius za neki y jednak je apsolutnoj vrijednosti y.

5.6.2 Normala

Normale za baze stošca određujemo jednako kao i kod cilindra. Međutim ako je pogođen plašt onda moramo izračunati udaljenost od Y osi. Provjerimo da li je točka sjecišta veća ili manja od nule (u slučaju da je veća od nule moramo okrenuti predznak udaljenosti kako bi dobili normalu u dobrom smjeru) i koristimo udaljenost kao y vrijednost normale.

$$\begin{aligned} dist &= sqrt(x^2 \cdot z^2) \\ \vec{n} &= \overrightarrow{x, dist, z} \end{aligned} \tag{5.33}$$

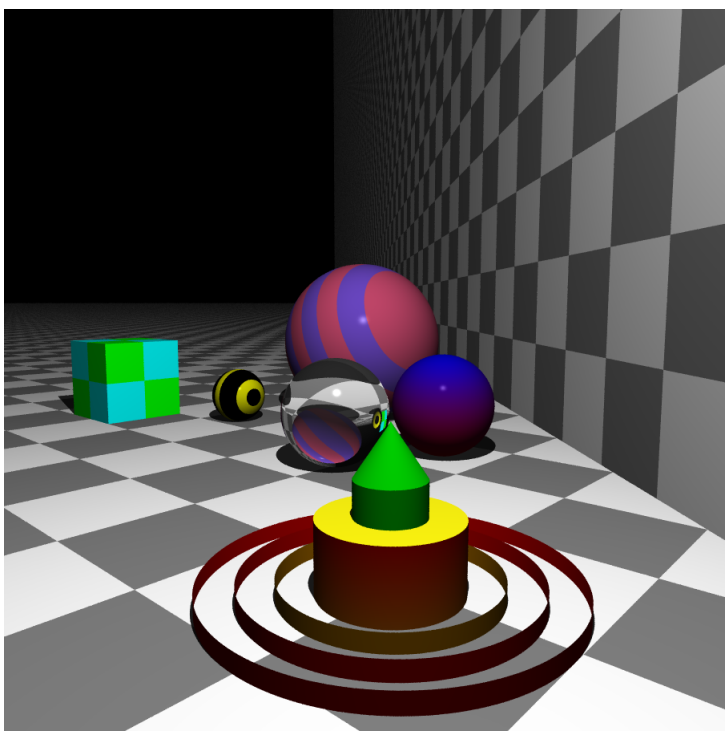
5.6.3 Implementacija

Programski kod 5.6 Sjecište zrake i stošca, normala

```
bool Cone::checkCap(const Ray& ray, const double t, const
    double radius)const {
  auto x = ray.origin.x + t * ray.direction.x;
  auto z = ray.origin.z + t * ray.direction.z;
  return(x * x + z * z) <= radius;
}
void Cone::intersectCaps(const Ray& ray, Intersections& inter)
    const {
  if (!closed || epsilonEqual(ray.direction.y, 0))
    return;
  auto t = (minimum - ray.origin.y) / ray.direction.y;
  if (checkCap(ray, t, std::fabs(minimum)))
    inter.intersections.insert(new Intersection(t, this));
  t = (maximum - ray.origin.y) / ray.direction.y;
  if (checkCap(ray, t, std::fabs(maximum))) {
    inter.intersections.insert(new Intersection(t, this));
  }
}
Intersections Cone::intersect(const Ray& ray)const {
  bool coneBodyIntersection = true;
  auto a = ray.direction.x * ray.direction.x - ray.direction.y
    * ray.direction.y + ray.direction.z * ray.direction.z;
  auto b = 2 * ray.origin.x * ray.direction.x - 2 * ray.origin
    .y * ray.direction.y + 2 * ray.origin.z * ray.direction.z;
  auto c = ray.origin.x * ray.origin.x - ray.origin.y * ray.
    origin.y + ray.origin.z * ray.origin.z;
  auto disc = b * b - 4 * a * c;

  if (epsilonEqual(a, 0) && epsilonEqual(b, 0))
    coneBodyIntersection = false;
  if (disc < 0)
```

```
27      coneBodyIntersection = false;
28    Intersections inter;
29    if (coneBodyIntersection) {
30      if (epsilonEqual(a, 0) && !epsilonEqual(b, 0)) {
31        auto t = -c / (2 * b);
32        inter.intersections.insert(new Intersection(t, this));
33      }
34      else {
35        auto t0 = (-b - sqrt(disc)) / (2 * a);
36        auto t1 = (-b + sqrt(disc)) / (2 * a);
37        if (t0 > t1)
38          std::swap(t0, t1);
39        auto y0 = ray.origin.y + t0 * ray.direction.y;
40        auto y1 = ray.origin.y + t1 * ray.direction.y;
41        if (minimum < y0 && y0 < maximum)
42          inter.intersections.insert(new Intersection(t0, this))
     ;
43        if (minimum < y1 && y1 < maximum)
44          inter.intersections.insert(new Intersection(t1, this))
     ;
45      }
46    }
47    intersectCaps(ray, inter);
48    return inter;
49  }
50  Tuple Cone::objectNormal(const Tuple& objectPoint, [[
     maybe_unused]] const Intersection* hit)const {
51    auto dist = objectPoint.x * objectPoint.x + objectPoint.z *
     objectPoint.z;
52    if (dist < 1 && objectPoint.y >= maximum - EPSILON)
53      return Vector(0, 1, 0);
54    else if (dist < 1 && objectPoint.y <= minimum + EPSILON)
55      return Vector(0, -1, 0);
56    auto y = sqrt(objectPoint.x * objectPoint.x + objectPoint.z
     * objectPoint.z);
```

```
57    if (objectPoint.y > 0)
58      y *= -1;
59    return Vector(objectPoint.x, y, objectPoint.z);
60  }
```



Slika 5.5 Prikaz svih geometrijskih tijela (slika dobivena vlastitom implementacijom)

Poglavlje 6

Kompleksni tipovi

Korištenjem isključivo primitivnih tipova jako je teško kreirati kompleksne objekte kao što su npr. biljke, brodovi, zgrade, itd. Da bi olakšali njihovo modeliranje i prikazivanje koriste se modeli spremljeni u obliku .obj datoteka. Osim toga moguće je kombinirati primitivne tipove korištenjem tehnike "Constructive solid geometry".

6.1 OBJ datoteke

Wavefront OBJ format se koristi za pohranjivanje podataka koji opisuju neki 3D model. Sadrži podatke kao što su vrhovi, koordinate tekstura, normale i način povezivanja točaka. Glavna podjela je na geometrijske podatke i podatke vezane za materijale. Geometrijski podatci su:

- Točke: svaka točka je opisana sa svojim koordinatama u 3D prostoru (x, y, z) u datoteci označena je s *v* (engl. vertex) nakon kojeg slijede koordinate.
- Koordinate tekstura: definiraju kako se tekstura mapira na površinu objekta i označava se sa *vt* (engl. vertex textures). Opisuju se s *u, v* vrijednostima gdje je (0, 0) donji lijevi, a (1, 1) gornji desni kut.
- Normale: normale su vektori koji definiraju orijentaciju. U datoteci označenu su s *VN* (engl. vertex normal) i sadrže komponente *(x, y, z)*.
- Površina: definirana je s početnim slovom *f* nakon kojeg slijede indeksi točaka,

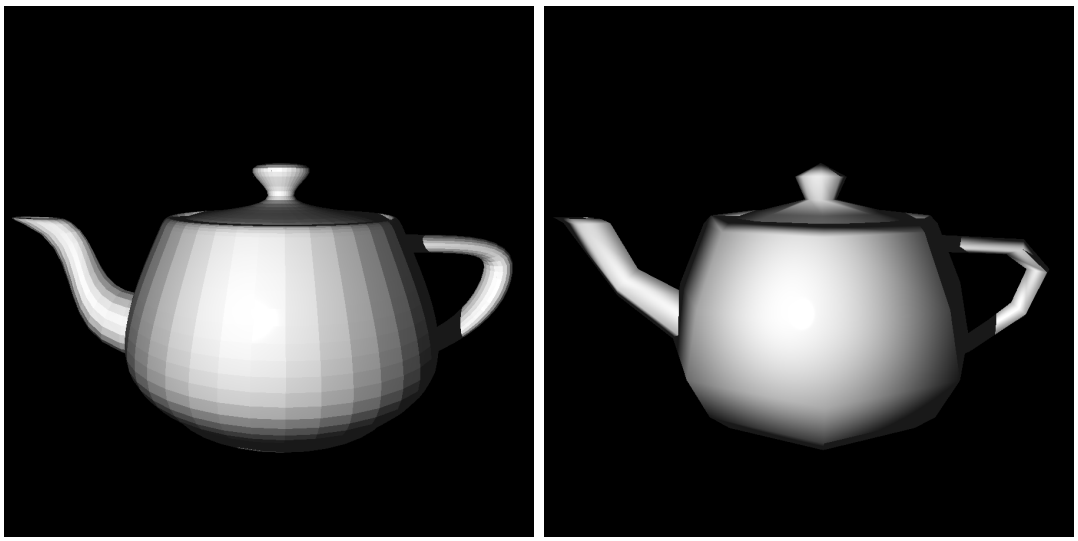
ako su 3 točke onda one određuju jedan trokut, ako ih je više potrebno ih je povezati u više trokuta korištenjem triangulacije (engl. fan triangulation). Također postoji kompleksniji oblik koji sadrži podatke za normale i teksture (*vertex/texture/normal, vertex/texture/normal, vertex/texture/normal*)

- Grupe: komanda *g* nakon koje slijede sve površine (*f*) koje su povezane u jednu cjelinu. Služi za podjelu kompleksnih modela na manje podgrupe (npr. čovjek se može podijeliti na glavu, ruke, itd).

Primjer .obj datoteke (# je oznaka za komentare):

```
# List of geometric vertices, with (x, y, z, [w])
   coordinates, w is optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in (u, [v, w])
   coordinates, these will vary between 0 and 1. v, w are
    optional and default to 0.
vt 0.500 1 [0]
vt ...
...
# List of vertex normals in (x,y,z) form; normals might
   not be unit vectors.
vn 0.707 0.000 0.707
vn ...
...
# Parameter space vertices in (u, [v, w]) form; free
   form geometry statement (see below)
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
```

```
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
...
# Line element (see below)
l 5 8 1 2 4 9
```



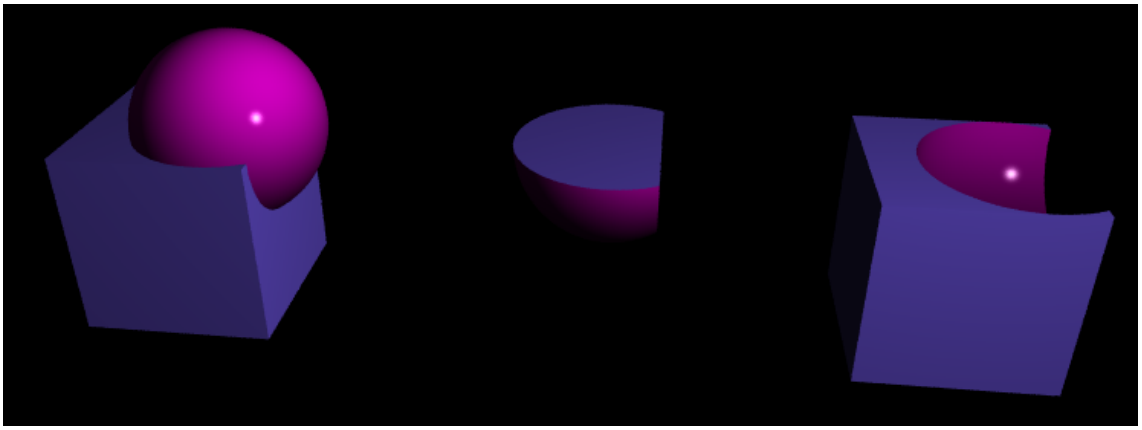
Slika 6.1 Čajnici učitani iz .obj datoteke (slika dobivena vlastitom implementacijom)

6.2 Konstruktivna geometrija tijela

Konstruktivna geometrija tijela (engl. Constructive solid geometry (CSG)) je proces kombiniranja primitivnih tipova u složene cjeline. Glavna prednost CSG-a u odnosu na 3D modele učitane iz .obj datoteka je to da je broj poligona puno manji, za prikaz istog oblika. Također bojanje samih objekata je puno lakše jer ne zahtijeva implementaciju tekstura. Nedostatak je da je puno teže konstruirati složene tipove. Podržane operacije su:

- unija - zadržavaju se oba tijela
- presjek - zadržava se samo presjek između dva tijela

- razlika - zadržava se samo razlika između dva tijela.



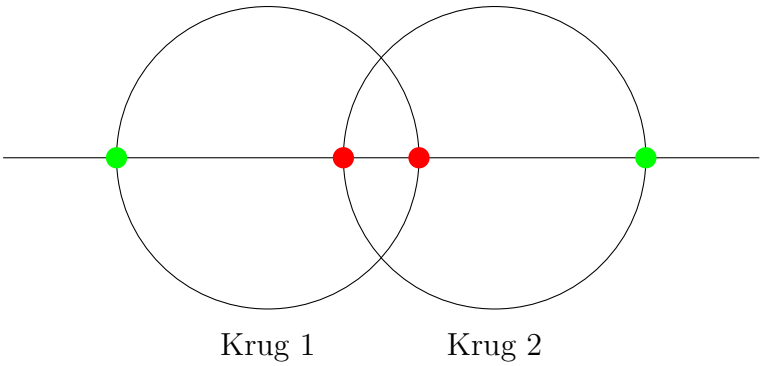
Slika 6.2 Unija, presjek i razlika kocke i sfere (slika dobivena vlastitom implementacijom)

6.2.1 Unija

Sa slike 6.3 vidimo da zraka siječe dva kruga (sfere, ili bilo koja dva tijela) u četiri točke, međutim nas zanimaju samo sjecišta označena zelenom bojom, a ostala mogu biti zanemarena. Kod unije nas zanimaju presjeci koji nisu unutar drugog objekta (ako je pogođen lijevi objekt sjecište ne smije biti unutar desnog, i obrnuto). Presjeke možemo opisati s 3 varijable: *lhit, inl, inr*.

- lhit - vraća `true` kad je lijevi objekt pogođen i `false` ako je desni pogođen
- inl - vraća `true` ako je sjecište unutar lijevog objekta
- inr - vraća `true` ako je sjecište unutar desnog objekta.

Iz uvijeta moguće je napraviti tablicu za uniju. Tablica nam pokazuje da li zadržavamo presjek ili ne. Odnosno Omogućuje nam da odredimo uniju uvjetom: `(lhit and not inr) ||(not lhit and not inl)`



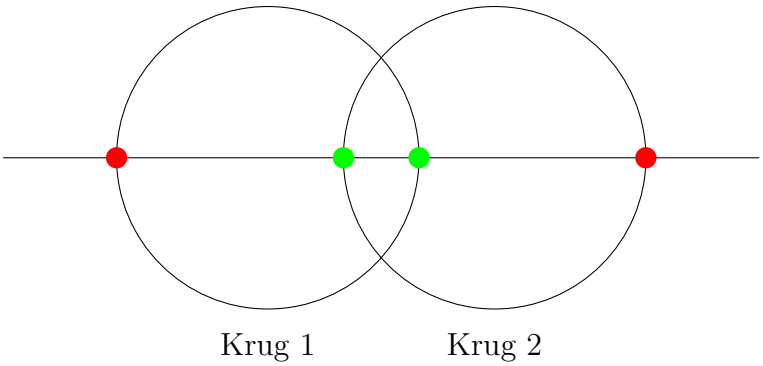
Slika 6.3 Unija - zelena sjecišta zadržavamo, crvena odbacujemo

Tablica 6.1 Unija - pravila za zadržavanje sjecišta.

operacija	lhit	inl	inr	rezultat
unija	true	true	true	false
unija	true	true	false	true
unija	true	false	true	false
unija	true	false	false	true
unija	false	true	true	false
unija	false	true	false	false
unija	false	false	true	true
unija	false	false	false	true

6.2.2 Presjek

Presjek zadržava zajednički dio između dva tijela (dijelove objekta koji se preklapaju). Pravilo za presjek obrnuto je od unije, tj. zanimaju nas presjeci koju pogađaju jedan objekt ali se nalaze unutar drugog (ako je pogođen lijevi krug sjecište mora biti unutar desnog).



Slika 6.4 Presjek - zelena sjecišta zadržavamo, crvena odbacujemo

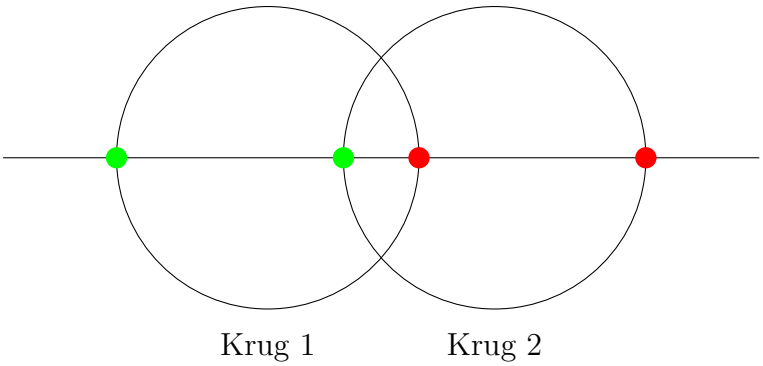
Tablica 6.2 Presjek - pravila za zadržavanje sjecišta

operacija	lhit	inl	inr	rezultat
presjek	true	true	true	true
presjek	true	true	false	false
presjek	true	false	true	true
presjek	true	false	false	false
presjek	false	true	true	true
presjek	false	true	false	true
presjek	false	false	true	false
presjek	false	false	false	false

Uvjet koji određuje presjek je `(lhit and inr) || (not lhit and inl)`

6.2.3 Razlika

Uklanjaju se zajednički dijelovi između dva tijela. Zadržavaju se sva sjecišta koja nisu unutar desnog objekta (ili obrnuto sva sjecišta koja nisu unutar lijevog objekta, a sijeku desni).



Slika 6.5 Razlika - zelena sjecišta zadržavamo, crvena odbacujemo

Tablica 6.3 Razlika - pravila za zadržavanje sjecišta

operacija	lhit	inl	inr	rezultat
razlika	true	true	true	false
razlika	true	true	false	true
razlika	true	false	true	false
razlika	true	false	false	true
razlika	false	true	true	true
razlika	false	true	false	true
razlika	false	false	true	false
razlika	false	false	false	false

Uvjet koji određuje razliku: `(lhit and not inr) || (not lhit and inl)`.

6.3 Normala i sjecište

Kod kompleksnih tipova normala i sjecište određeni su preko jednadžbi za primitivne tipove iz prijašnjeg poglavlja. Ako se koriste modeli iz .obj datoteka onda su to najčešće trokuti. Dok za konstruktivnu geometriju tijela može biti bilo koji primitivan tip, pa bi tako za primjer sa slike 6.2 provjerili sjecište i normalu za kocku i sferu.

6.3.1 Implementacija

Programski kod 6.1 Sjecište zrake i CSG-a i normala

```
bool CSGShape::intersectionAllowed(CSGOperation::OPERATION op,
    bool lhit, bool inl, bool inr)const {
  if (op == CSGOperation::OPERATION::UNION) {
    return (lhit && !inr) || (!lhit && !inl);
  }
  else if (op == CSGOperation::OPERATION::INTERSECTION) {
    return (lhit && inr) || (!lhit && inl);
  }
  else if (op == CSGOperation::OPERATION::DIFFERENCE) {
    return (lhit && !inr) || (!lhit && inl);
  }
  return false;
}
Intersections CSGShape::filterIntersections(const
   Intersections& xs)const {
  auto inl = false;
  auto inr = false;

  auto result = Intersections();

  for (auto intersection : xs.intersections) {
    auto lhit = left->includes((Shape*)intersection->s);
    if (intersectionAllowed(operation, lhit, inl, inr)) {
      result.intersections.insert(intersection);
    }
    if (lhit) {
      inl = !inl;
    }
    else {
      inr = !inr;
    }
```

```
30    }
31    return result;
32  }
33  Intersections CSGShape::intersect(const Ray& ray)const {
34    if (!m_bounds.intersect(ray)) {
35      return {};
36    }
37    auto leftxs = left->shapeIntersect(ray);
38    auto rightxs = right->shapeIntersect(ray);
39
40    auto xs = Intersections();
41
42    xs.intersections.insert(leftxs.intersections.begin(), leftxs
       .intersections.end());
43    xs.intersections.insert(rightxs.intersections.begin(),
       rightxs.intersections.end());
44
45    return filterIntersections(xs);
46  }
```

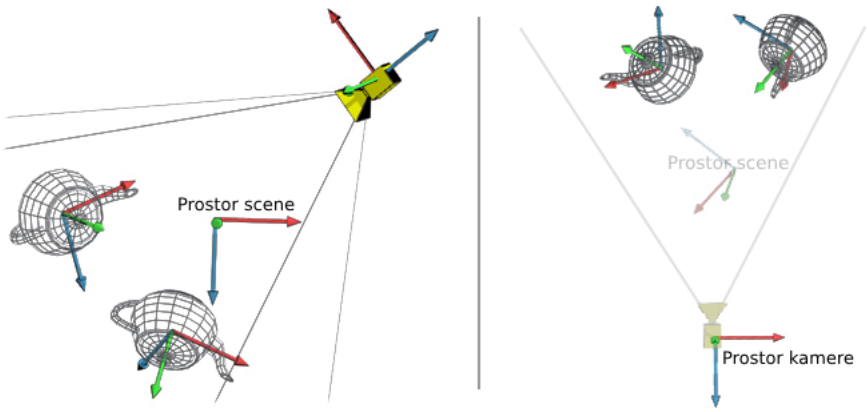
Poglavlje 7

Kamera

Kamera omogućava gledanje objekata u sceni iz različitih udaljenosti, pozicija i kuteva. Glavni cilj kamere je da vjerno prikaže postavljenu 3D scenu u 2D prostoru. Elementi kamere su visina i širina 2D slike, žarišna duljina, kut vidnog polja (engl. field of view (FOV)) i radijus otvora blende. Kamera je definirana s dvije točke (pozicija kamere, i točka u koju kamera gleda) i vektorima koji određuju orijentaciju kamere. Oduzimanjem točke u koju gledamo i točku koja opisuje poziciju kamere dobijemo vektorom koji pokazuje smjer gledišta kamere (pokazuje što se nalazi ispred kamere) imamo još *Up* vektor i *Left* vektor. Uz pomoću navedenih vektora i točaka određuje se matrica transformacije kamere koja postavlja kameru u 3D prostoru.

7.1 Transformacija kamere

Kamera sadrži matricu pogleda koja opisuje način na koji je svijet orijentiran u odnosu na kameru. Matrica kamere ili pogleda je transformacija koja nam pokazuje na koji način je potrebno orijentirati scenu da bi simulirali pomicanje kamere u prostoru. Odnosno pošto je kamera uvijek u ishodištu i gleda u smjeru negativne Z osi potrebno je pomaknuti cijelu scenu u odnosu na kameru.

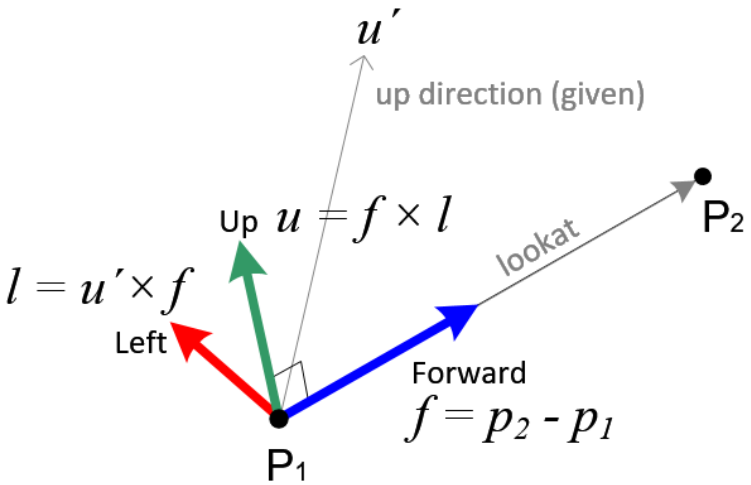


Slika 7.1 Lijevo, čajnici i kamera u prostoru scene; Desno sve transformirano u prostor kamere [12]

Slika 7.1 pokazuje čajnike i kameru u različitim koordinatnim sustavima (prostorima). Prebacivanje između prostora moguće je pomoću transformacijske matrice (ako prebacujemo iz prostora svijeta/scene u prostor kamere) i inverza transformacijske matrice (ako prebacujemo iz prostora kamere u prostor svijeta/scene).

Da bi odredili matricu transformacije kamere potrebno je:

- odrediti vektor od kamere do točke u koju kamera gleda (engl. forward vector)
- odrediti vektor koji pokazuje što je "gore" (engl. up vector) u odnosu na kameru (prema konvenciji često se postavlja vektor (0, 1, 0) i kasnije se računa ispravan vektor) u biti nam treba bilo koji vektor koji leži na ravnini između vektora smjera i pravog up vektora
- odrediti vektor koji pokazuje u lijevo (engl. left vector) u odnosu na kameru (pokazuje u pozitivnom smjeru X osi)
- odrediti samu matricu transformacije.



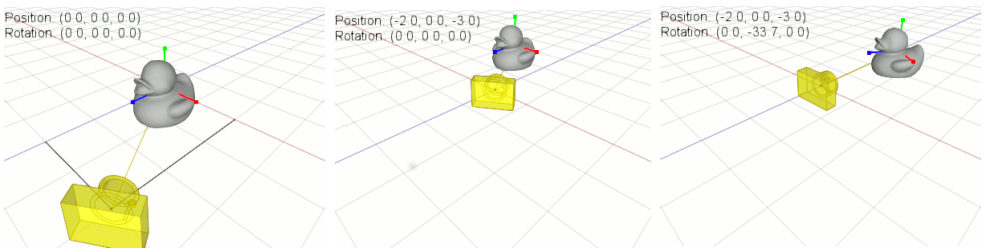
Slika 7.2 Vektori kamere [13]

Forward vector (na slici 7.2 označen sa f) dobijemo oduzimanjem dviju točaka, točke u kojoj se nalazi kamera i točke u koju gleda (na slici 7.2 *P1 i P2*), vektor je potrebno normalizirati. Left vector odredimo uz pomoć vektorskog umnoška normaliziranog up vektora i forward vektora (na slici označen sa 7.2 l). Na kraju još određujemo pravi up vektor preko vektorskog umnoška *forward* i *left* vektora (na slici označen sa 7.2 u). Matricu orijentacije kamere određujemo kao (vektor F negiran je jer imamo lijevi koordinatni sustav):

$$\begin{pmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ -f_x & -f_y & -f_z & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.1}$$

Još je potrebno odrediti matricu transformacije kamere koja postavlja kameru u točnu poziciju u sceni (točnije orijentira svijet u odnosu na kameru). Određuje se kao:

$$\begin{pmatrix} 1 & 0 & 0 & -f_x \\ 0 & 1 & 0 & -f_y \\ 0 & 0 & 1 & -f_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.2}$$



Slika 7.3 Vizualni prikaz transformacije [14]

U transformacijskoj matrici vrijednosti su negirane jer želimo translatirati scenu u suprotnom smjeru u odnosu na mjesto na koje smo postavili kameru (jer je kamera uvijek u ishodištu).

Na Slici 7.3 prikazana je kamera koja se nalazi u točci (2, 0, 3) i gleda u točku (0, 0, 0) u prostoru scene. Kako bi dobili kameru koja je u ishodištu i gleda u $-Z$ os najprije je potrebno translatirati scenu u točku (-2, 0, -3) i potom ju rotirati za otprilike -33.7 stupnjeva oko Y osi.

Na kraju da bi dobili finalnu transformacijsku matricu kamere pomnožimo orijentacijsku i transformacijsku matricu.

7.1.1 Implementacija

Programski kod 7.1 Transformacija kamere

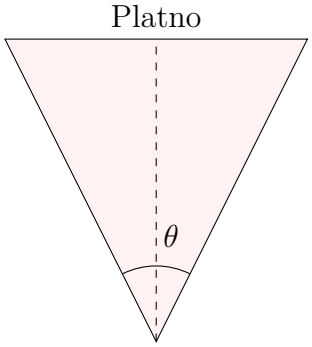
```
1  Matrix viewTransformation(const Tuple& from, const Tuple& to,
       const Tuple& up) {
2    auto forward = to - from;
3    forward = forward.normalize();
4    auto upn = up.normalize();
5    auto left = forward.crossProduct(upn);
6    auto trueUp = left.crossProduct(forward);
7
8    Matrix orientation(4, 4);
9
10   orientation.matrix[0] = left.x;
```

```
11    orientation.matrix[1] = left.y;
12    orientation.matrix[2] = left.z;
13    orientation.matrix[3] = 0;
14    orientation.matrix[4] = trueUp.x;
15    orientation.matrix[5] = trueUp.y;
16    orientation.matrix[6] = trueUp.z;
17    orientation.matrix[7] = 0;
18    orientation.matrix[8] = -forward.x;
19    orientation.matrix[9] = -forward.y;
20    orientation.matrix[10] = -forward.z;
21    orientation.matrix[11] = 0;
22    orientation.matrix[12] = 0;
23    orientation.matrix[13] = 0;
24    orientation.matrix[14] = 0;
25    orientation.matrix[15] = 1;
26
27    return orientation * translate(-from.x, -from.y, -from.z);
28 }
```

7.2 Generiranje zraka

Kod tamne sobe (lat. camera obscura) napravljena je mala rupica prema vanjskom svijetu koja predstavlja leću. Kroz rupicu prolaze zrake koje projiciraju vanjski prostor na pozadinu unutar sobe. S obzirom da je centar projekcije između platna i vanjskog svijeta slika će na pozadini biti obrnuta (rotirana za 180 stupnjeva). Sličan princip koristimo kada postavljamo kameru u prostoru, međutim mi postavljamo svoje "platno" ispred kamere, točnije postavljamo ga za 1 jedinicu udaljenosti u negativan smjer Z osi. Cilj nam je kroz sredinu svakog piksela na platnu ispaliti zraku koja će onda provjeriti da li je nešto pogođeno u sceni, i ako je vratiti boju objekta koji je pogođen. Da bi to napravili moramo odrediti veličinu piksela. Znamo da je platno udaljeno za točno jednu jedinicu od kamere, pa možemo vrlo jednostavno odrediti pola širine platna (i visine jer uvijek pretpostavljamo da je platno kvadrat) koristeći formulu:

$$\text{halfView} = \tan(\frac{\text{fieldOfView}}{2}) \tag{7.3}$$



Slika 7.4 Kamera koja gleda prema platnu, θ je FOV kamere

S obzirom na to da veličina platna u pikselima ne mora imati oblik kvadrata (npr. može biti veličine 1920 x 1080, tj. omjer 16:9) moramo skalirati visinu i širinu platna u jedinicama kamere kako bi zadržali kvadratni oblik piksela. Omjer visine i širine (engl. aspect ratio) primjenjuje se na širinu i visinu platna u jedinicama kamere na sljedeći način:

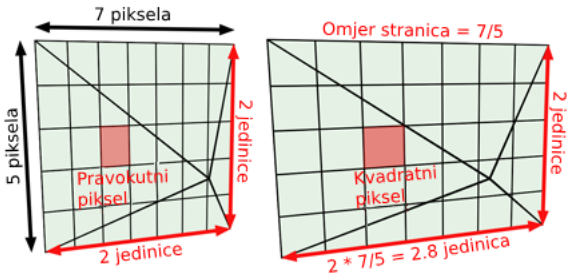
$$\begin{aligned} \text{aspect} &= \frac{hsize}{vsize} \\ \text{halfWidth} &= \begin{cases} \text{halfView}, & \text{if } aspect \geq 1 \\ \text{halfView} \cdot \text{aspect}, & \text{if } aspect < 1 \end{cases} \\ \text{halfHeight} &= \begin{cases} \text{halfView/aspect}, & \text{if } aspect \geq 1 \\ \text{halfView}, & \text{if } aspect < 1 \end{cases} \end{aligned} \tag{7.4}$$

Kako bi dobili veličinu piksela u jedinicama kamere potrebno je još širinu platna u jedinicama kamere podijeliti sa širinom platna u pikselima, odnosno:

$$\text{pixelSizeCamera} = \frac{\text{halfWidth} \cdot 2}{\text{hsize}} \tag{7.5}$$

Sad je moguće odrediti zraku koja počinje u kameri, prolazi kroz centar piksela na platnu i odlazi dalje u scenu. To napravimo tako da najprije pomaknemo zraku u centar piksela i pomnožimo s prije izračunatom varijablom *pixelSizeCamera* i tako dobijemo koordinate u jedinicama kamere.

$$\begin{aligned} \text{xOffset} &= (px + 0.5) \cdot \text{pixelSizeCamera} \\ \text{yOffset} &= (py + 0.5) \cdot \text{pixelSizeCamera} \end{aligned} \tag{7.6}$$



Slika 7.5 Utjecaj omjera stranica na piksele [15]

Još je potrebno dobiti koordinate prostora scene (world coordinates), pošto je platno centrirano na -Z osi, a koordinate kamere počinju od (0, 0) u gornjem lijevom kutu platna, moramo ih pomaknuti za pola širine odnosno duljine platna (treba napomenuti da se zbog smjera gledanja kamere prema -Z osi +X os prostire u "lijevo").

$$\begin{aligned} \text{worldX} &= \text{halfWidht} - \text{xOffset} \\ \text{worldY} &= \text{halfheight} - \text{yOffset} \end{aligned} \tag{7.7}$$

Još je preostalo primijeniti inverz transformacije kamere na dobivenu točku (z koordinata točke se nalazi na -1). Također trebamo transformirati ishodište zrake (kamera je postavljena u centar koordinatnog sustava, pa je točka (0, 0, 0)). Na kraju je još potrebno oduzeti dobivenu transformiranu točku od ishodišta, kako bi dobili smjer ispaljene zrake (zraku je potrebno i normalizirati).

$$\begin{aligned} \text{pixel} &= \text{cameraTransform}^{-1} \cdot \text{Point(worldX, worldY, -1)} \\ \text{origin} &= \text{cameraTransform}^{-1} \cdot \text{Point(0, 0, 0)} \\ \text{direction} &= \frac{\text{pixel} - \text{origin}}{||\text{pixel} - \text{origin}||} \end{aligned} \tag{7.8}$$

Isti postupak se ponavlja za svaki piksel, generira se zraka koja provjerava da li je nešto u sceni pogođeno i boji platno u odgovarajuću boju.

7.2.1 Implementacija

Programski kod 7.2 Generiranje zraka

```
Camera::Camera(int _hSize, int _vSize, double _fieldOfView) :
    hSize(_hSize), vSize(_vSize), fieldOfView(_fieldOfView)
{
  double halfView;
  halfView = tan(fieldOfView / 2.f);
  auto aspect = hSize / (double)vSize;
  if (aspect >= 1) {
    halfWidth = halfView;
    halfheight = halfView / aspect;
  }
  else {
    halfWidth = halfView * aspect;
    halfheight = halfView;
  }
  pixelSize = halfWidth * 2 / hSize;
}
Ray Camera::rayForPixel(double px, double py) {
  auto xOffset = (px + 0.5) * pixelSize;
  auto yOffset = (py + 0.5) * pixelSize;
  auto worldX = halfWidth - xOffset;
  auto worldY = halfheight - yOffset;

  Tuple pixel = Point(0, 0, 0);
  Tuple origin = Point(0, 0, 0);
    pixel = *(transform.inverse()) * Point(worldX, worldY, -1)
   ;
    origin = *(transform.inverse()) * Point(0, 0, 0);

  auto direction = (pixel - origin).normalize();
  return Ray(origin, direction);
}
```

7.3 Focal blur

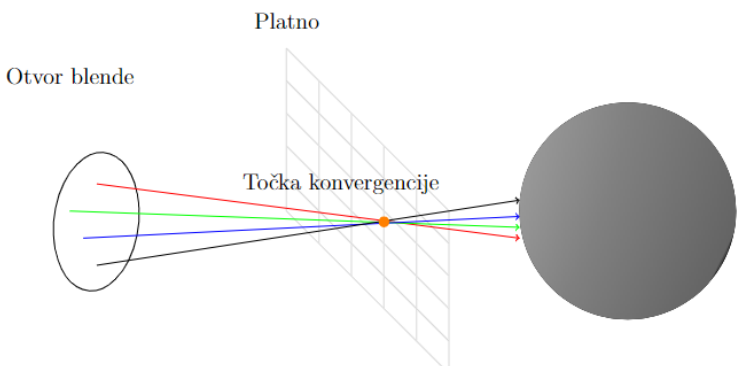
Sve prije navedene operacije zasnivaju se na ideji pinhole kamere odnosno točno jedna zraka svjetlosti može proći kroz otvor blende. Povećanjem otvora blende dozvoljavamo da više zraka svjetlosti prođe do scene, zbog toga slika će postati zamućena. Kako bi to postigli potrebno je za ishodište zrake uzeti neku nasumičnu točku oko ishodišta. Točka će se nalaziti unutar kruga koji predstavlja širinu otvora blende (*Xrand, Yrand, 0*). Pa se tako mijenja prijašnja jednadžba u:

$$\text{origin} = \text{transform}^{-1} \cdot \text{aperturePoint}(); \tag{7.9}$$

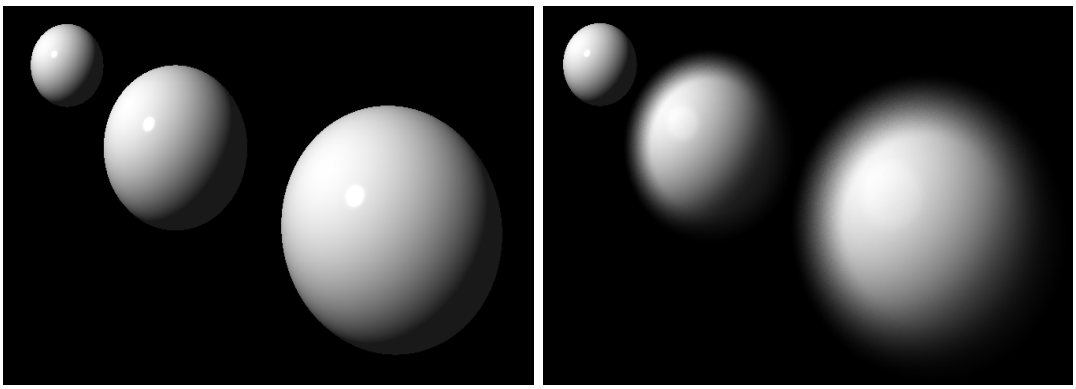
Kako bi omogućili fokusiranje na različite objekte u sceni treba u izračune dodati i fokalnu udaljenost. Udaljenost platna se sad postavlja na fokalnu udaljenost, pa se sad mijenjaju i sljedeće jednadžbe:

$$\begin{aligned} \text{halfView} &= \tan(\text{fieldOfView} \ / \ 2.\text{f}) \cdot \text{focalLenght}; \\ \text{pixel} &= \text{transform}^{-1} \cdot \text{Point}(\text{worldX}, \ \text{worldY}, \ \text{-focalLenght}); \end{aligned} \tag{7.10}$$

Sve ostale jednadžbe ostaju iste. Za svaki piksel slike ispalit će se n zraka iz određenog kruga i kao boja uzima se prosječna vrijednost vraćenih boja.



Slika 7.6 Biranje različitih točaka za izvor zrake (za isti piksel)



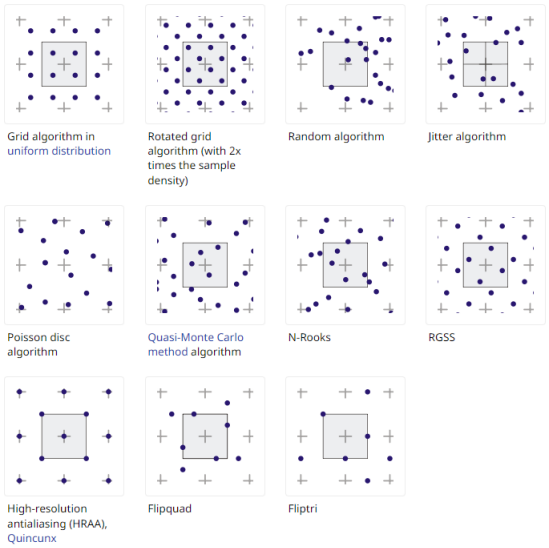
Slika 7.7 Razlika između normalne slike i slike generirane korištenjem focal blur efekta (slika dobivena vlastitom implementacijom)

7.3.1 Implementacija

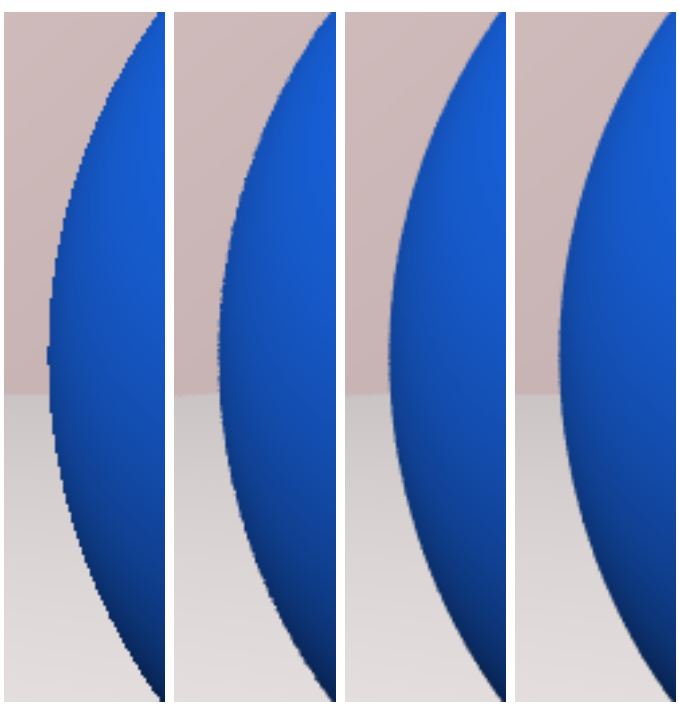
Da bi mogli generirati slike s focal blur efektom potrebno je modificirati prijašnju implementaciju kamere koristeći navedene jednadžbe.

7.4 Aliasing

Na slikama koje su generirane moguće je primijetiti vidljive rubove ili šumove između objekta i njegove pozadine, na samim rubovima objekta. U osnovi anti aliasing (AA) radi na tome da umanjuje efekt nazubljenosti na dijagonalnim ili zakrivljenim linijama. Problem je najlakše riješiti generiranjem puno veće slike, jer je na većim rezolucijama nazubljenost manja, tj rubovi su glađi. To bi zahtijevalo da se slika uvijek generira na većoj rezoluciji od odabrane što nije praktično rješenje. Umjesto toga možemo definirati broj uzoraka po pikselu. Za svaki piksel ispalimo n zraka umjesto jedne, i za svaku zraku definiramo mali pomak unutar piksela. Pomak se može definirani na više načina, u ovome radu uzima se nasumično n uzoraka u odnosu na centar piksela. Potom za svaku zraku dobijemo boju objekta koji je pogođen i uzima se prosječna vrijednost za boju piksela.



Slika 7.8 Različite tehnike za odabir piksela [16]



Slika 7.9 Ugašen AA, 4xAA, 16xAA i 16xAA samo rub (slika dobivena vlastitom implementacijom)

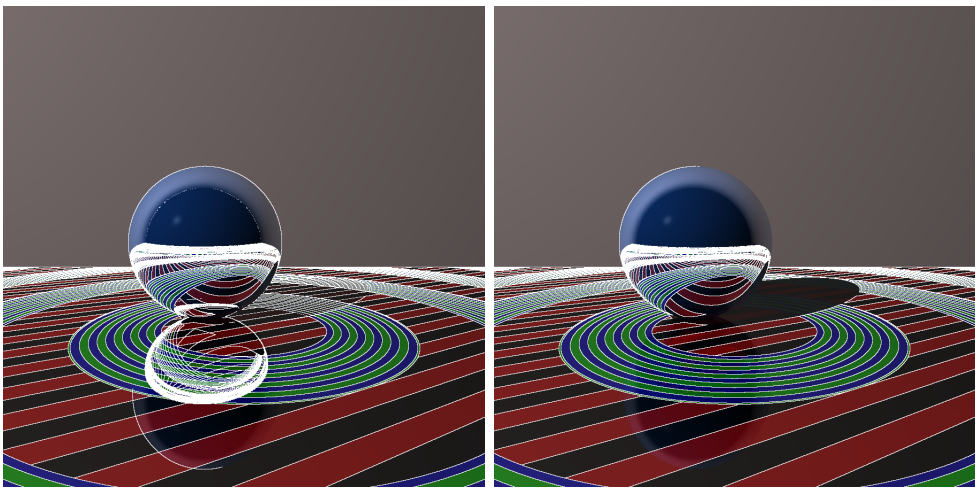
7.4.1 Implementacija

Programski kod 7.3 Anti aliasing

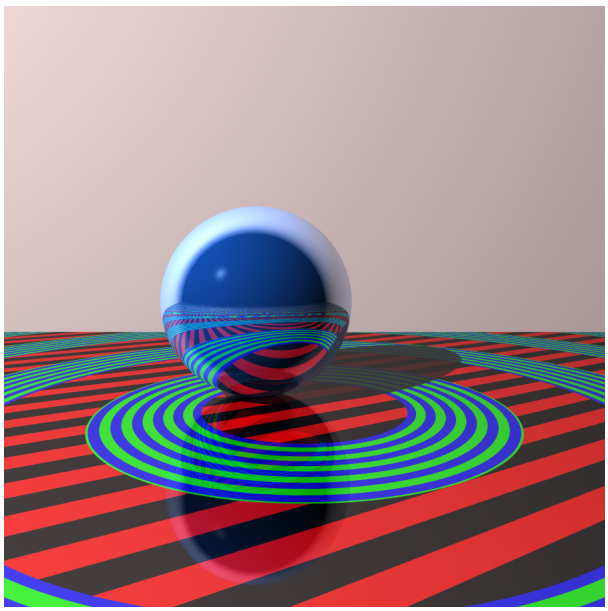
```
1       for (int k = 0; k < aliasingSamples; ++k) {
2
3         u = (y + random_double() - 0.5);
4         v = (x + random_double() - 0.5);
5
6         auto ray = rayForPixel(v, u);
7         pixelColor += world->colorAt(ray);
8       }
9       image->writePixel(x, y, pixelColor / aliasingSamples);
```

7.4.2 Aliasing rubova

Prijašnja metoda će za svaki piksel poslati n zraka i vratiti prosječnu boju. Problem nazubljenih rubova je kao što samo ime govori najčešće prisutan na rubovima, stoga umjesto da se šalje više zraka za svaki piksel bolje je detektirati rubove i poslati više zraka samo za njih. Za detekciju ruba šalje se zraka u svaki kut i centar piksela. Ako je razlika u boji između bilo kojeg kuta i centra veća od neke zadane vrijednosti onda smo pogodili rubni piksel neke površine. Slika 7.10 prikazuje rubove za koje će se poslati dodatne zrake, prva slika detektira rub ako je razlika u boji veća od 0.1 za bilo koju komponentu boje RGB, dok je na drugoj prag za detekciju 0.3. Na slici 7.10 se može primijetiti da kod većeg odstupanja ne dolazi do slanja dodatnih zraka za piksele u sjeni, jer je sijena efektivno zatamnila sve piksele.



Slika 7.10 Prva slika prikazuje rubove na koje će se primijeniti AA za threshold 0.1, druga za threshold 0.3(slika dobivena vlastitom implementacijom)



Slika 7.11 Slika dobivena korištenjem AA samo na rubovima, threshold je 0.1 (slika dobivena vlastitom implementacijom)

7.4.3 Implementacija

Prikazana je implementacija koja šalje zraku samo u gornji lijevi kut piksela, na isti način poslale bi se i provjerile boje piksela u ostalim kutevima.

Programski kod 7.4 Aliasing rubova

```
if (aliasingSamples > 4 && (aliasEdges || edgeAliasHighlights)
    ) {
  // +-0.5 in canvas space moves the ray to the pixel edges
  double topLeftX = v - 0.5;
  double topLeftY = u - 0.5;

  auto rayTopLeft = rayForPixel(topLeftX, topLeftY);
  Color topLeft = world->colorAt(rayTopLeft);

  auto rayCenter = rayForPixel(x, y);
  Color center = world->colorAt(rayCenter);

  if (std::abs(topLeft.r - center.r) > aliasingThreshold ||
   std::abs(topLeft.g - center.g) > aliasingThreshold || std::
   abs(topLeft.b - center.b) > aliasingThreshold)
    isEdge = 1;

  if (isEdge) {
    for (int k = 0; k < aliasingSamples; ++k) {

      u = (y + random_double() - 0.5);
      v = (x + random_double() - 0.5);

      auto ray = rayForPixel(v, u);
      pixelColor += world->colorAt(ray);
    }
    image->writePixel(x, y, pixelColor / aliasingSamples);
  }
}
```

Poglavlje 8

Materijali

Materijali se odnose na svojstva koja se dodjeljuju objektima u sceni. Određuju na koji način objekt interaktira sa svjetlosti. Svaki materijal reagira drukčije na svjetlost, objekti mogu upijati ili odbijati svjetlost, mogu biti prozirni ili imati svojstva refrakcije, itd. Navedena svojstva opisana su sljedećim komponentama:

- Boja materijala
- Ambijentalna komponenta: predstavlja koliko ambijentalnog (svjetlo koje ne dolazi direktno iz izvora nego npr. zbog odbijanja od neke druge površine) svjetla će biti reflektirano od površine
- Difuzna komponenta: označava koliko objekt raspršuje dolazno svijetlo od izvora svjetlosti
- Zrcalna komponenta: označava koliko će naglasaka (sjaja) biti na površini objekta (generalno se može primijetiti kod sjajnih površina)
- Sjaj: određuje veličinu zrcalne komponente
- Refleksija: određuje količinu refleksije okolnog prostora na površini objekta
- Prozirnost: određuje koliko svjetlosti će proći kroz objekt
- Refraktivna komponenta: pokazuje koliko će se svjetlost "saviti" prilikom prolaska kroz objekt.

8.1 Boje

Boje su predstavljene s 3 vrijednosti RGB. R opisuje količinu prisutnosti crvene boje, istu funkciju imaju G (engl. green) i B (engl. blue). Kombinacijom navedenih boja moguće je dobiti sve ostale boje. Boje moraju podržavati operacije: zbrajanja, množena i oduzimanja dviju boja, također množenja i dijeljenja sa skalarom.

Zbrajanje boja:

$$\begin{aligned} b_1 &= (r_1, g_1, b_1) \\ b_2 &= (r_2, g_2, b_2) \\ b_1 + b_2 &= (r_1 + r_2, g_1 + g_2, b_1 + b_2) \end{aligned} \tag{8.1}$$

Oduzimanje boja:

$$\begin{aligned} b_1 &= (r_1, g_1, b_1) \\ b_2 &= (r_2, g_2, b_2) \\ b_1 - b_2 &= (r_1 - r_2, g_1 - g_2, b_1 - b_2) \end{aligned} \tag{8.2}$$

Množenje boja:

$$\begin{aligned} b_1 &= (r_1, g_1, b_1) \\ b_2 &= (r_2, g_2, b_2) \\ b_1 \cdot b_2 &= (r_1 \cdot r_2, g_1 \cdot g_2, b_1 \cdot b_2) \end{aligned} \tag{8.3}$$

Množenje sa skalarom:

$$\begin{aligned} b_1 &= (r_1, g_1, b_1) \\ b_1 \cdot x &= (r_1 \cdot x, g_1 \cdot x, b_1 \cdot x) \end{aligned} \tag{8.4}$$

Dijeljenje sa skalarom:

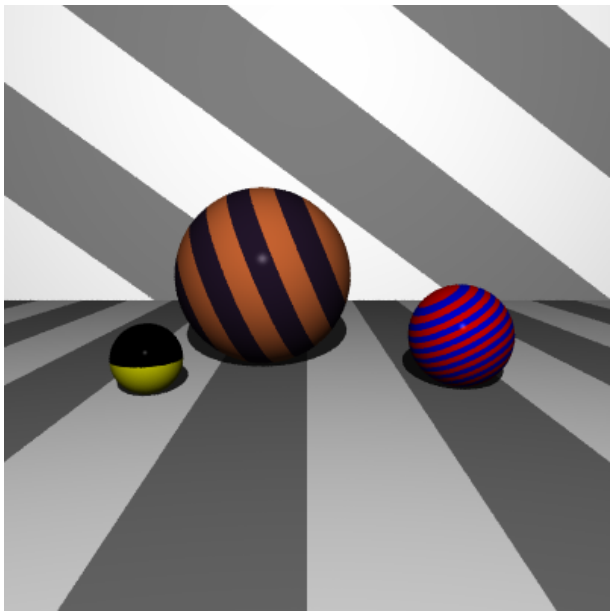
$$\begin{aligned} b_1 &= (r_1, g_1, b_1) \\ \frac{b_1}{x} &= (\frac{r_1}{x}, \frac{g_1}{x}, \frac{b_1}{x}) \end{aligned} \tag{8.5}$$

8.2 Uzorci

Uz pomoć geometrijskih pravila može se odrediti način na koji će se boje prikazati u sceni, odnosno određuju se uzorci koji se koriste za objekte u sceni. Uzorci mogu biti u obliku: pruge, gradijenta, krugova, šahovnice, itd.

8.2.1 Pruge

Uzorci su samo funkcije koje određuju na koji način se neka boja ponavlja u sceni, pa ako određujemo funkciju koja će predstavljati pruge zanima nas samo promjena točke u X osi.



Slika 8.1 Pruge na ravninama i sferama (slika dobivena vlastitom implementacijom)

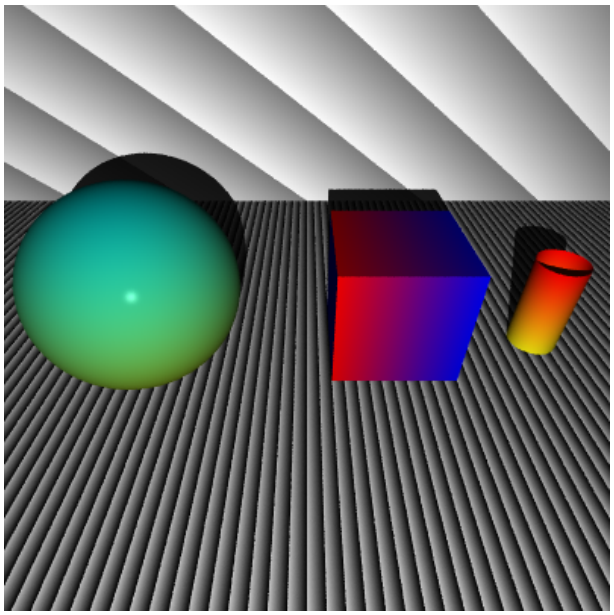
$$color(point, c_a, c_b) = \begin{cases} c_a, & \text{if } \lfloor point_x \rfloor \mod 2 = 0 \\ c_b, & \text{otherwise} \end{cases} \tag{8.6}$$

Uz pomoću transformacija svaki uzorak moguće je rotirati, skalirati, translatirati, itd.

8.2.2 Gradijent

Slično kao i kod pruga, zanima nas samo promjena u X osi. Međutim, za gradijent, potrebno je linearno interpolirati boju kako se mijenja vrijednost točke u X osi. U slučaju kocke, svaka od ravnina mora imati definiranu svoju funkciju za određivanje gradijenta (i ostalih uzoraka) jer se ravnina prostire u drugim osima osim osi X.

$$color(p, c_a, c_b) = c_a + (c_b - c_a) \cdot (p_x - \lfloor p_x \rfloor) \tag{8.7}$$

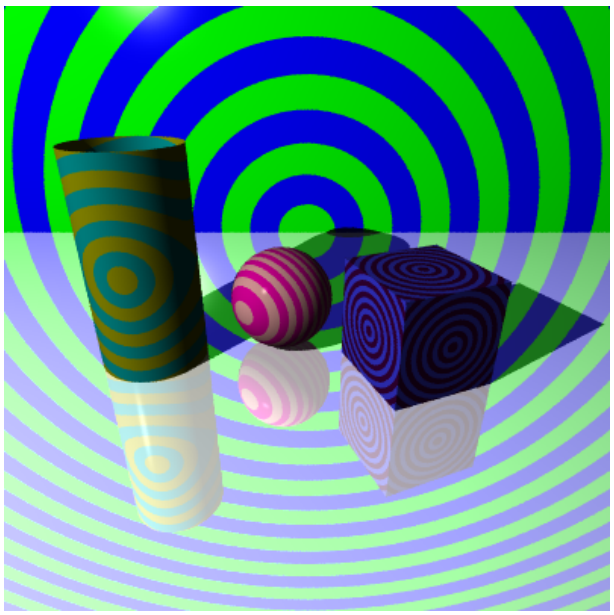


Slika 8.2 Gradijenti na ravnini, sferi, kocki i cilindru (slika dobivena vlastitom implementacijom).

8.2.3 Prsten

Boja prstena ovisi o dvije osi X i Z. Što znači da treba testirati udaljenost točke u odnosu na X i Z os. Jednadžba je:

$$color(point, c_a, c_b) = \begin{cases} c_a, & \text{if } \lfloor \sqrt{point_x} \rfloor \mod 2 = 0 \\ c_b, & \text{otherwise} \end{cases} \tag{8.8}$$



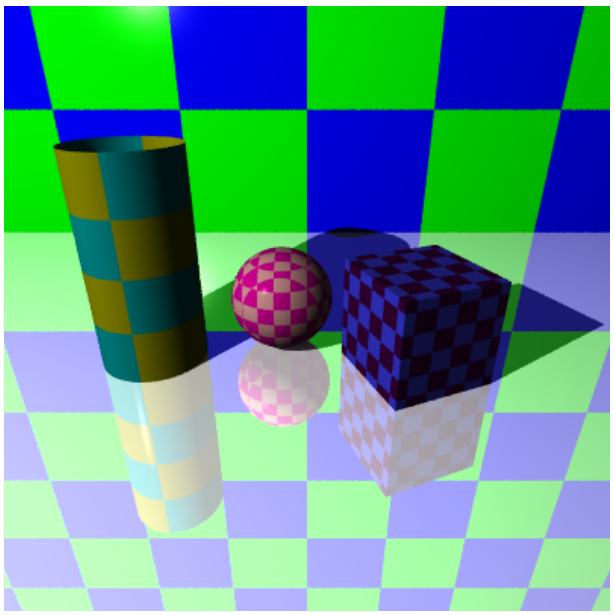
Slika 8.3 Prsteni na ravnini, cilindru, sferi i kocki (slika dobivena vlastitom implementacijom)

8.2.4 Šahovnica

Kako bi imali 3D šahovnicu treba osigurati da dva susjedna kvadrata nikad nisu iste boje. Koristi se slična jednadžba kao kod pruga međutim zanimaju nas sve 3 dimenzije.

$$color(point, c_a, c_b) = \begin{cases} c_a, & \text{if } (\lfloor point_x \rfloor + \lfloor point_y \rfloor + \lfloor point_z \rfloor) \mod 2 = 0 \\ c_b, & \text{otherwise} \end{cases} \tag{8.9}$$

Sa slike 8.4 vidi se da mapiranje na sferu ne radi ispravno. Greška se javlja zbog mapiranja dvo dimenzionalnog uzorka na 3D površinu.



Slika 8.4 Šahovnica na ravnini, cilindru, sferi i kocki (slika dobivena vlastitom implementacijom)

Teže se primijeti na cilindru i kocki, šahovnica izgleda očuvano (na kocki se vidi da mjesta spajanja svake od ravnina koja čini kocku nisu dobra). Taj problem može se riješiti UV mapiranjem iz sljedećeg poglavlja.

8.2.5 Implementacija

Prikazana je implementacija za gradijent, ostali uzorci su implementirani na jednak način, naravno koristeći pripadajuće jednadžbe za svaki uzorak.

Programski kod 8.1 Implementacija gradienta

```
1 Color GradientPattern::patternColorAt(const Tuple& point, [[
     maybe_unused]] const Shape* shape)const {
2   auto distance = b - a;
3   auto fraction = point.x - floor(point.x);
4
5   return a + distance * fraction;
6 }
```

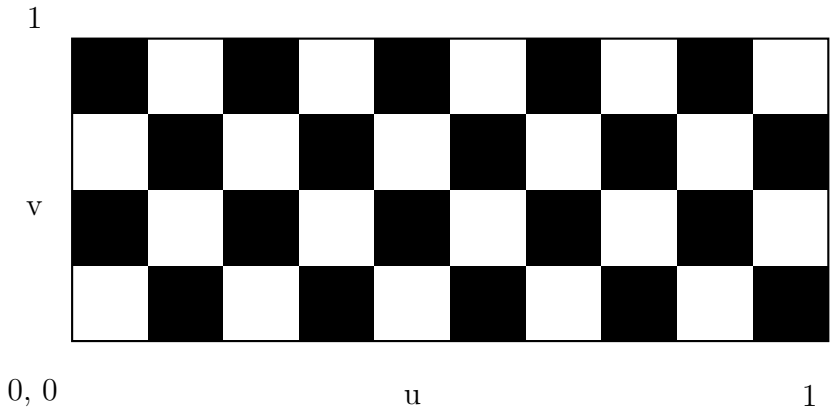
8.3 Teksture

Kako bi teksture ili uzorci bili točno mapirani na sferu, potrebno je "odmotati" sferu (ili bilo koji drugi oblik) da leži u 2D površini, potom povezati nove koordinate (pretvoriti 3D koordinate u 2D prostor) s teksturom. U ovom poglavlju fokusirat ćemo se na teksture koje se mogu primijeniti na primitivne tipove, ne na teksture koje se primjenjuju na .obj modele.

8.3.1 Šahovnica

Šahovnicu se može prikazati u 2D prostoru gdje ima neku duljinu u i širinu v (za njih pretpostavljamo da se uvijek prostiru od 0 do 1). Također definirano je koliko kvadrata šahovnica sadrži uz pomoći varijabli širine i visine. Na slici 8.5 vidimo uzorak sa širinom 10 i visinom 4.

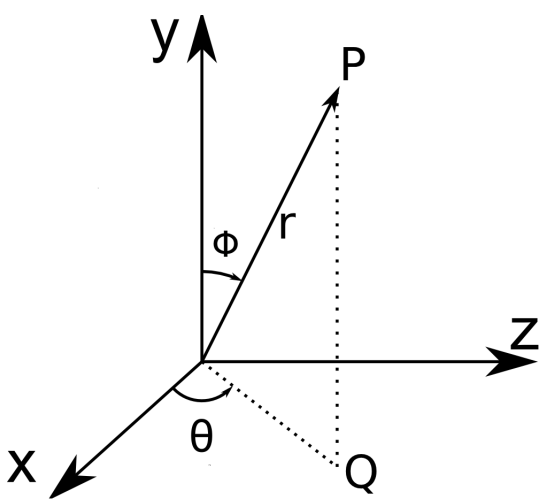
Boju iz *u, v* koordinata određujemo slično kao u prijašnjem poglavlju. Međutim, sad je potrebno dobivenu *u i v* koordinatu pomnožiti sa širinom i visinom uzorka.



Slika 8.5 2D šahovnica prikazana preko u i v varijabli

$$
\begin{aligned}
real_U &= \lfloor (u \cdot \text{width}) \rfloor \\
real_V &= \lfloor (v \cdot \text{height}) \rfloor \\
\text{color}(real_U, real_V) &= \begin{cases} c_a, & \text{if } (real_U + real_V) \mod 2 = 0 \\ c_b, & \text{otherwise} \end{cases}
\end{aligned}
\tag{8.10}
$$

Koordinate (u, v) koje se koriste za dobivanje boje najprije je potrebno odrediti, tj. potrebno ih je prebaciti iz (x, y, z) točke sa sfere u 2D prostor. Ponašanje koje želimo je da se u povećava od 0 do 1 kako se krećemo suprotno u odnosu na smjer sata oko sfere, dok se v povećava od 0 do 1 kako se krećemo od južnog do sjevernog pola sfere. Za pretvorbu koordinata najprije je potrebno izračunati radijus do točke na sferi. Sfera je centrirana u ishodištu, pa je potrebno odrediti vektor iz ishodišta do točke i uzeti njegovu duljinu. Potom se određuje polarni kut ϕ između Y osi i vektora koji spaja ishodište s točkom. Preko ϕ dolazimo do v koordinate. Za odrediti u koordinatu treba odrediti kut θ koji zatvara projekcija vektora na XZ ravninu s X osi (na slici 8.6 Q je projekcija točke P).



Slika 8.6 Sferične koordinate za neku točku P, i vektori i kutevi koji ju određuju

$$\begin{aligned} \Theta &= \arctan 2(p_x, p_z) \\ r &= \|\overrightarrow{p_x, p_y, p_z}\| \\ \phi &= \arccos\left(\frac{p_y}{\text{radius}}\right) \\ raw_U &= \frac{\Theta}{2\pi} \\ u &= 1 - (raw_U + 0.5) \\ v &= 1 - \frac{\phi}{\pi} \end{aligned} \tag{8.11}$$

Iz jednadžbe vidimo da je potrebno još par međukoraka da dobro odredimo u i v. Najprije Θ se povećava u smjeru kazaljke na satu što je suprotno od onog što želimo. Također raw_U je određen za $-0.5 < raw_U <= 0.5$. Zato se u izrazu za u oduzima od jedan i dodaje 0.5 kako bi dobili vrijednosti od 0 do 1 u ispravnom smjeru. Slično da bi imali 0 na južnom dijelu sfere potrebno je oduzeti od 1 i za v.

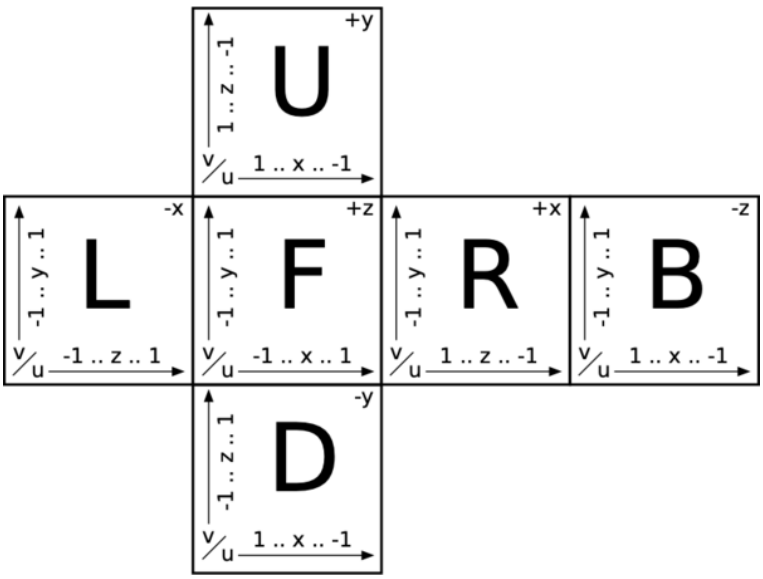
Kako je ravnina već 2D površina za nju se decimalna vrijednost x točke tretira kao u koordinata, a decimalna vrijednost z točke kao v koordinata.

Mapiranje na cilindar pretpostavlja da se uzorak ili tekstura ponavljaju za svako cjelobrojno povećanje u smjeru Y osi. Plašt cilindra mapira se slično kao i sfera, bitno je da se v mapira direktno na decimalni dio y vrijednosti točke.

$$\begin{aligned} \Theta &= \arctan 2(p_x, p_z) \\ raw_U &= \frac{\Theta}{2\pi} \\ u &= 1 - (raw_U + 0.5) \\ v &= p_y \bmod 1 \end{aligned} \tag{8.12}$$

Mapiranje na krajevima cilindra (ako je zatvoren) radi se na sličan način kao mapiranje ravnina kod kocka (pokazano u poglavlju ispod). Za cilindar jednadžbe su:

$$\begin{aligned} u &= fmod(p_x + 1)/2 \\ v &= fmod(1 - p_z)/2 \end{aligned} \tag{8.13}$$



Slika 8.7 Mapiranje strana kocke [17]

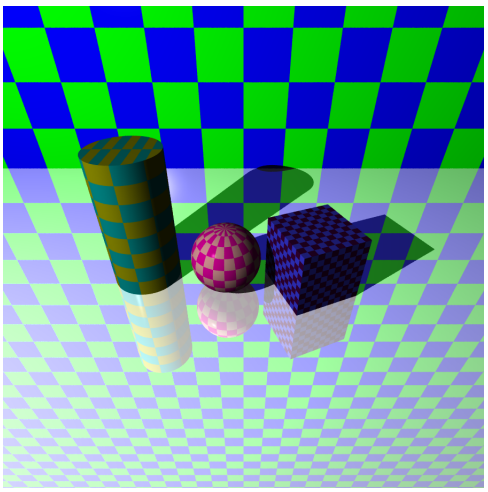
Najsloženije je mapiranje kocke, jer se sastoji od 6 ravnina koje različito mapiraju točke, također orijentacija u odnosu sa susjede je bitna. Mapiranje se obavlja prema slici 8.7 (pretpostavljamo da kocka počinje u točci (-1, -1, -1) i završava u (1, 1, 1) što i odgovara mapi sa slike 8.7), na slici 8.7 u gornjem desnom kutu svakog kvadrata napisana je os uz koju je poravnata strana kocke, također pokazuje da se u i v prostiru od 0 do 1, a x i y za npr. F od -1 do 1. Koraci koji su potrebni za odrediti u i v koordinate su:

- odredimo koja površina je pogođena
- pomaknemo x i y vrijednosti u raspon od 0 do n, i uzmemo decimalni dio dijeljenja raspona s 2 kako bi se uzorak ponavljao
- normaliziramo vrijednost da se nalazi u rasponu od 0 do 1.

Redom za površine F, B, L, R, U, D jednadžbe će glasiti:

$$\begin{aligned}
uF &= fmod((point.x + 1), 2.0)/2.0; \\
vF &= fmod((point.y + 1), 2.0)/2.0; \\
uB &= fmod((1 - point.x), 2.0)/2.0; \\
vB &= fmod((point.y + 1), 2.0)/2.0; \\
uL &= fmod((point.z + 1), 2.0)/2.0; \\
vL &= fmod((point.y + 1), 2.0)/2.0; \\
uR &= fmod((1 - point.z), 2.0)/2.0; \\
vR &= fmod((point.y + 1), 2.0)/2.0; \\
uU &= fmod((point.x + 1), 2.0)/2.0; \\
vU &= fmod((1 - point.z), 2.0)/2.0; \\
uD &= fmod((point.x + 1), 2.0)/2.0; \\
vD &= fmod((point.z + 1), 2.0)/2.0;
\end{aligned} \tag{8.14}$$

Slika 8.8 pokazuje rezultate korištenja 2D UV šahovnice i uv mapiranja (napomena da za sferu treba odabrati širinu duplu od duljine za dobiti pravilne kvadrate, jer u mapira vrijednosti od 1 do 2π, a v od 1 do π).

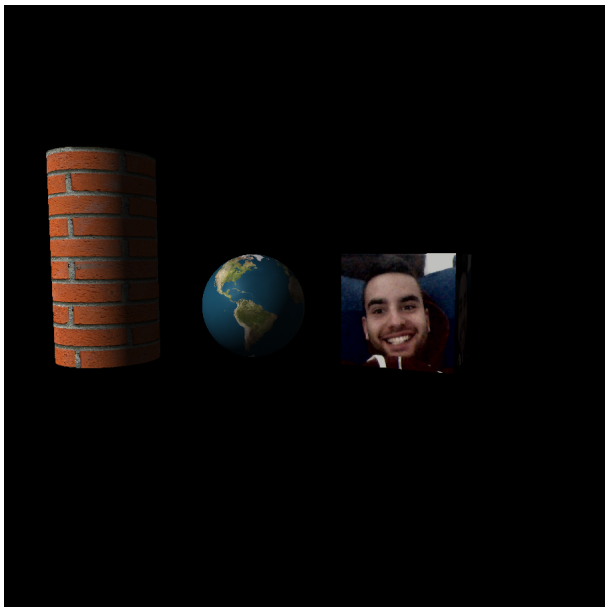


Slika 8.8 Mapiranje šahovnice korištenjem UV mapiranja (slika dobivena vlastitom implementacijom).

8.3.2 Teksture iz PPM datoteke

PPM datoteke su jednostavan način za pohranjivanje slika. U ovom radu podržani su formati P3 i P6 (više u poglavlju 15.1.2). Prije naveden UV uzorak šahovnice je geometrijski uzorak (programski određen) koji koristi matematičke operacije da odredi boju za neke (u, v) koordinate. Kod slika potrebno je samo u, v koordinate za određeni geometrijski oblik (sfera, ravnina, itd) mapirati na piksel slike. Odnosno treba sliku učitati u neko "platno" koje će moći vratiti boju na odgovarajućem indeksu. Da bi odredili x, i y koordinatu slike potrebno je u i v koordinate pomnožiti s visinom i širinom platna i zatražiti boju s izračunatog indeksa. Potrebno je okrenuti v jer UV mapiranje tretira donji lijevi kut kao 0 dok platno tretira gornji lijevi kut kao 0.

$$\begin{aligned} x &= u \cdot (width - 1) \\ y &= (1 - v) \cdot (height - 1) \end{aligned} \tag{8.15}$$



Slika 8.9 Prikaz mapiranja tekstura na primitivne tipove (slika dobivena vlastitom implementacijom)

8.3.3 Implementacija

Programski kod 8.2 UV mapiranje sfere

```
void Sphere::UVmap(const Tuple& p, double* u, double* v) const {
  auto theta = atan2(p.x, p.z);

  Tuple vec = Vector(p.x, p.y, p.z);
  auto radius = vec.magnitude();

  auto phi = acos(p.y / radius);
  auto rawU = theta / (2.0 * acos(-1));

  *u = 1.0 - (rawU + 0.5);
  *v = 1.0 - (phi / acos(-1));
}
```

Programski kod 8.3 UV mapiranje ravnine

```
void Plane::UVmap(const Tuple& p, double* u, double* v) const {

  *u = fmod(p.x, 1);
  *v = fmod(p.z, 1);

  if (*u < 0.0)
    *u += 1;

  if (*v < 0.0)
    *v += 1;
}
```

Programski kod 8.4 UV mapiranje kocke

```
void Cube::UVmap(const Tuple& p, double* u, double* v) const {
  auto face = faceFromPoint(p);

  if (face == cubeFace::LEFT)
    cubeUVLeft(p, u, v);
  else if (face == cubeFace::RIGHT)
    cubeUVRight(p, u, v);
  else if (face == cubeFace::FRONT)
    cubeUVFront(p, u, v);
  else if (face == cubeFace::BACK)
    cubeUVBack(p, u, v);
  else if (face == cubeFace::UP)
    cubeUVUp(p, u, v);
  else
    cubeUVDown(p, u, v);
}
```

Programski kod 8.5 UV mapiranje valjka

```
void Cylinder::UVmap(const Tuple& p, double* u, double* v)
   const {
  if (maximum - EPSILON <= p.y) {
    *u = fmod((p.x + 1.f), 2.f) / 2.f;
    *v = fmod((1.f - p.z), 2.f) / 2.f;
    return;
  }
  auto theta = atan2(p.x, p.z);
  auto rawU = theta / (2 * acos(-1));
  *u = 1 - (rawU + 0.5);
  *v = fmod(p.y, 1);

  if (*v < 0.0)
    *v += 1;
}
```

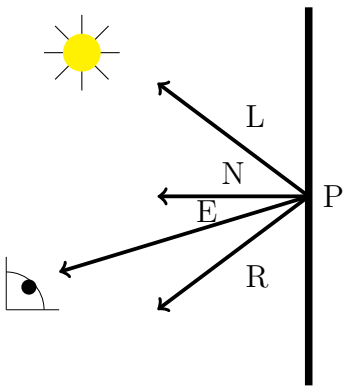
Poglavlje 9

Phongov model osvjetljavanja

Phongov model opisuje refleksiju svijetla na površini objekta. Odnosno aproksimira na koji način svjetlost interaktira s površinom objekta. Prije navedene komponente koje se koriste u Phongovom modelu su: ambijentalna, difuzna, zrcalna komponenta i sjaj.

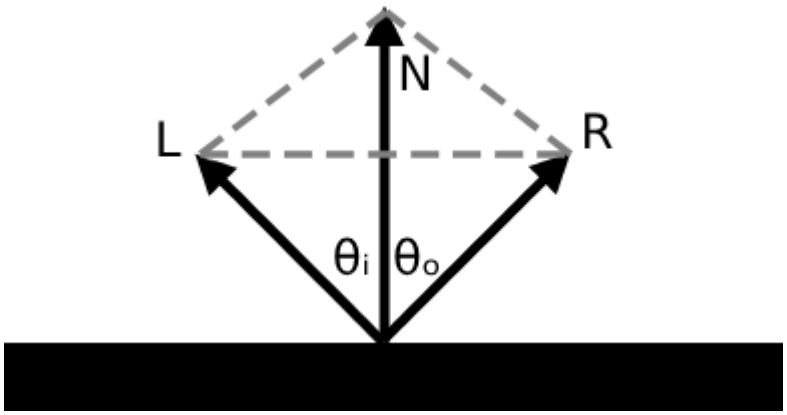
Za Phongov model refleksije bitna su nam četiri vektora koja potiču od točke sjecišta zrake i površine (P), a to su:

- Vektor kamere (engl. eye vektor, E) - vektor od točke sjecišta do oka ili kamere
- vektor svijetla (engl. light vector, L) - vektor koji pokazuje od točke sjecišta do izvora svijetla



Slika 9.1 Vektori korišteni za Phongov model

- normala (engl. surface normal, N) - vektor koji je okomit na površinu (s ishodištem u P)
- vektor refleksije (engl. reflection vector, R) - pokazuje smjeru u kojem se svijetlo odbija od površine.



Slika 9.2 Refleksija zrake oko normale

Vektor E dobije se tako da se negira smjer zrake (opet pokazuje prema izvoru). vektor L dobijemo oduzimanjem pozicije izvora svijetla i točke P. Normalu izračunamo prema postupku navedenom u poglavlju 5. Vektor R je vektor koji predstavlja idealnu zrcalnu refleksiju. R zatvara isti kut s normalom kao ulazni vektor svijetla L. Računa se preko jednadžbe:

$$\begin{aligned} \theta &= \theta_i = \theta_o \\ \vec{L} + \vec{R} &= 2 \cdot \cos\theta \cdot \vec{N} \\ \vec{R} &= -\vec{L} + 2 \cdot (\vec{L} \cdot \vec{N}) \cdot \vec{N} \end{aligned} \tag{9.1}$$

9.1 Ambijentalna komponenta

Phongov model tretira ambijentalnu komponentu kao konstantu koja će pobojati sve točke površine u jednaku boju. Generalno je to nekakva tamnija nijansa boje objekta. Računa se kao umnožak boje objekta, intenziteta svijetla i ambijentalne komponente objekta:

$$ambient = bojaObjekta \cdot intenzitet \cdot ambientalnaKomponenta \tag{9.2}$$

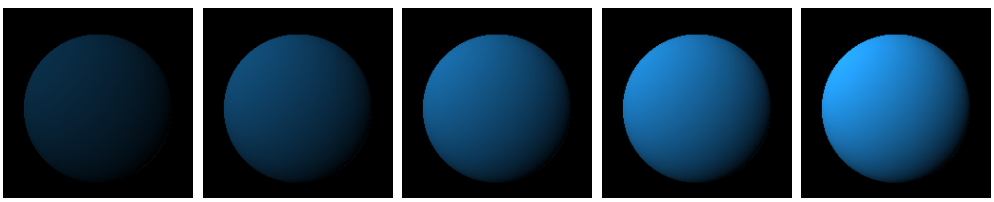
Slika 9.3 Različite vrijednosti ambientalne komponente: 0.1, 0.3, 0.5, 0.7, 0.9 (slika dobivena vlastitom implementacijom)

9.2 Difuzna komponenta

Difuzna komponenta ovisi o kutu kojeg zatvaraju vektor prema izvoru svijetla i normala pogođene površine. Računa se prema jednadžbi:

$$\begin{aligned} kut &= \vec{L} \cdot \vec{N} \\ bojaPovrsine &= bojaObjekta \cdot intenzitet \\ diffuse &= bojaPovrsine \cdot difuznaKomponenta \cdot kut \end{aligned} \tag{9.3}$$

Napomena: ako je kut manji od 0 difuzna boja je RGB(0, 0, 0)

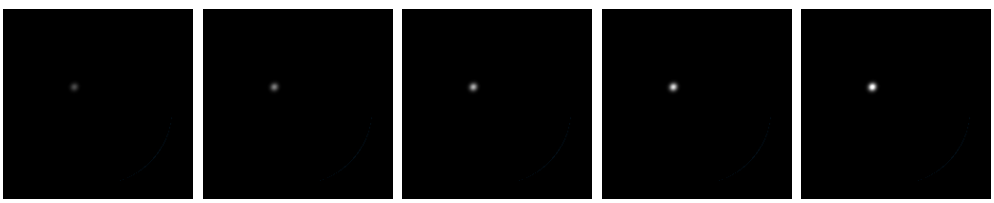


Slika 9.4 Različite vrijednosti difuzne komponente: 0.1, 0.3, 0.5, 0.7, 0.9 (slika dobivena vlastitom implementacijom)

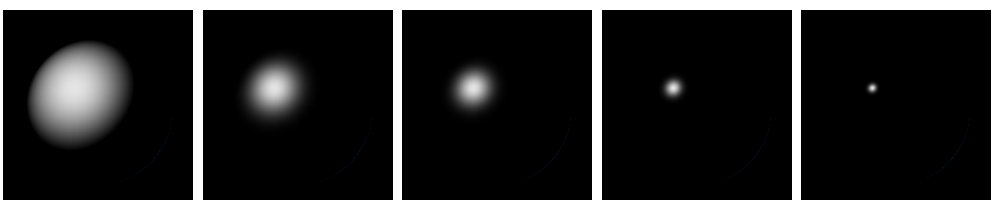
9.3 Zrcalna komponenta

Zrcalna komponenta ovisi o kutu između vektora refleksije i vektora prema oku (kameri), također ovisi o zrcalnoj komponenti materijala koja predstavlja intenzitet sjaja i komponenti sjaja koja opisuje veličinu sjaja.

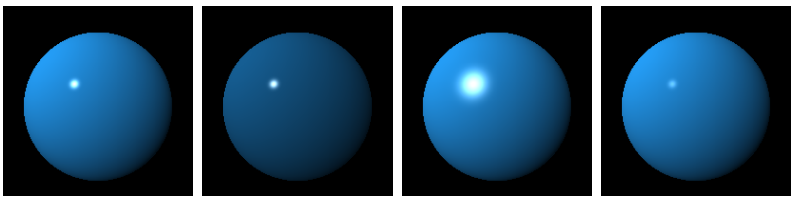
$$\begin{aligned} kut &= \vec{R} \cdot \vec{E} \\ faktor &= kut^{komponentaSjaja} \\ spekular &= intenzitet \cdot zrcalnaKomponenta \cdot faktor \end{aligned} \tag{9.4}$$



Slika 9.5 Različite vrijednosti zrcalne komponente: 0.1, 0.3, 0.5, 0.7, 0.9 (slika dobivena vlastitom implementacijom)



Slika 9.6 Različite vrijednosti komponente sjaja: 1, 10, 25, 100, 300 (slika dobivena vlastitom implementacijom)



Slika 9.7 Sfere dobivene kombinacijom različitih vrijednosti za parametre (slika dobivena vlastitom implementacijom)

Za dobiti finalnu boju potrebno je samo zbrojiti dobivene boje iz komponenti.

$$boja = ambijentalna + difuzna + zrcalna \tag{9.5}$$

9.4 Implementacija

Programski kod 9.1 Phongov model osvjetljavanja

```
1  Color Light::lighting(Material& material, Shape* object, const
       Tuple& point, const Tuple& eyev, const Tuple& normalv,
     const double intensityAt) {
2
3    auto effectiveColor = material.color * intensity;
4    auto ambientColor = effectiveColor * object->material.
     ambient;
5
6      Color diffuseColor(0, 0, 0);
7      Color specularColor(0, 0, 0);
8
9      auto lightv = (position - point).normalize();
10     auto lightDotNormal = lightv.dotProduct(normalv);
11
12     if (lightDotNormal < 0) {
13         diffuseColor = Color(0, 0, 0);
14         specularColor = Color(0, 0, 0);
```

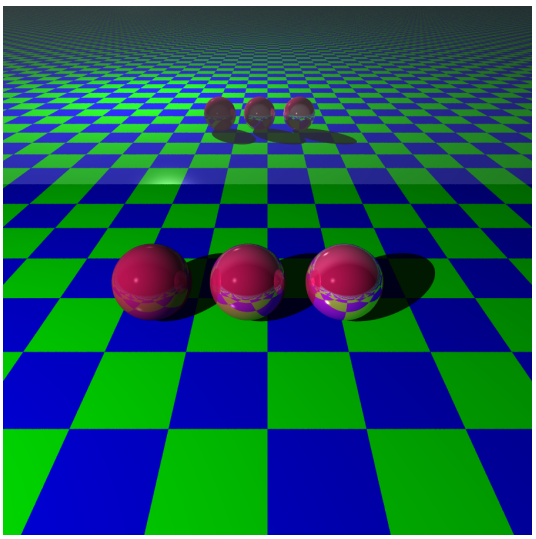
```
15      }
16      else {
17          diffuseColor = effectiveColor * object->material.
    diffuse * lightDotNormal;
18          auto reflectv = -lightv.reflect(normalv);
19          auto reflectDotEye = reflectv.dotProduct(eyev);
20
21          if (reflectDotEye <= 0)
22              specularColor = Color(0, 0, 0);
23          else {
24              auto factor = pow(reflectDotEye, object->material.
    shininess);
25              specularColor = intensity * object->material.
    specular * factor;
26          }
27      }
28
29    return ambientColor + diffuse + specular;
30  }
```

Poglavlje 10

Refleksija

Refleksija je svojstvo materijala da odbija zrake od površine. Materijali imaju komponentu refleksije koja određuje reflektivnost površine (ako je 0 površina ne reflektira svjetlost, ako je 1 onda imamo savršeno ogledalo, za sve između imamo parcijalnu refleksiju). Slično kao i kod Phongovog modela treba odrediti vektor refleksije, međutim ovdje reflektiramo zraku, a ne svijetlo. Potrebno je u jednadžbu 9.1 umjesto vektora L (vektor od sjecišta do svijetla) uvrstiti vektor E (vektor od sjecišta do kamere). Ako je pogođena površina koja je reflektivna potrebno je stvoriti novu reflektiranu zraku oko normale i pratiti novonastalu zraku do sljedećeg sjecišta.

Ako reflektirana zraka pogađa neki drugi objekt u sceni uzimamo boju sjecišta i množimo ju s reflektivnom komponentom. Kod refleksije bitno je ograničiti broj mogućih odbijanja od objekta, u suprotnome doći će do beskonačnog odbijanja između npr. dva zrcala.



Slika 10.1 Sfere s različitim reflektivnim svojstvima (slika dobivena vlastitom implementacijom)

10.1 Implementacija

Programski kod 10.1 Implementacija refleksije

```
Color World::reflectedColor(const Precomputations& comps, int&
    remaining, int remainingRefraction)const {
  if (remaining <= 0)
    return Color(0, 0, 0);

  if (epsilonEqual(comps.shape->material.reflective, 0))
    return Color(0, 0, 0);

  auto reflectedRay = Ray(comps.overPoint, comps.reflectv);
  auto color = colorAt(reflectedRay, remaining - 1,
   remainingRefraction);

  return color * comps.shape->material.reflective;
}
```

Poglavlje 11

Prozirnost i refrakcija

Refrakcija opisuje način na koji se svijetlo "savija" kad pogodi proziran materijal. Kao što je prije opisano materijal sadrži komponente refrakcije i prozirnosti. Kao i kod refleksije, potrebno je stvoriti novu zraku u sjecištu, pošaljemo ju u nekom smjeru i vratimo boju pogođenog objekta. Kad zraka prelazi iz jednog prozirnog objekta prema drugom lomi se pod kutem koji je određen Snellovim zakonom.

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1} \tag{11.1}$$

Nas zanima smjer zrake nakon loma, pa Snellov zakon možemo zapisati u vektorskom obliku:

$$n_1(i \times n) = n_2(t \times n) \tag{11.2}$$

Preko vektorskog oblika moguće je dobiti smjer vektora preko jednadžbe:

$$n_1(i \times n) = n_2(t \times n) \tag{11.3}$$

Iz čega se može izvući smjer zrake refrakcije preko jednadžbe:

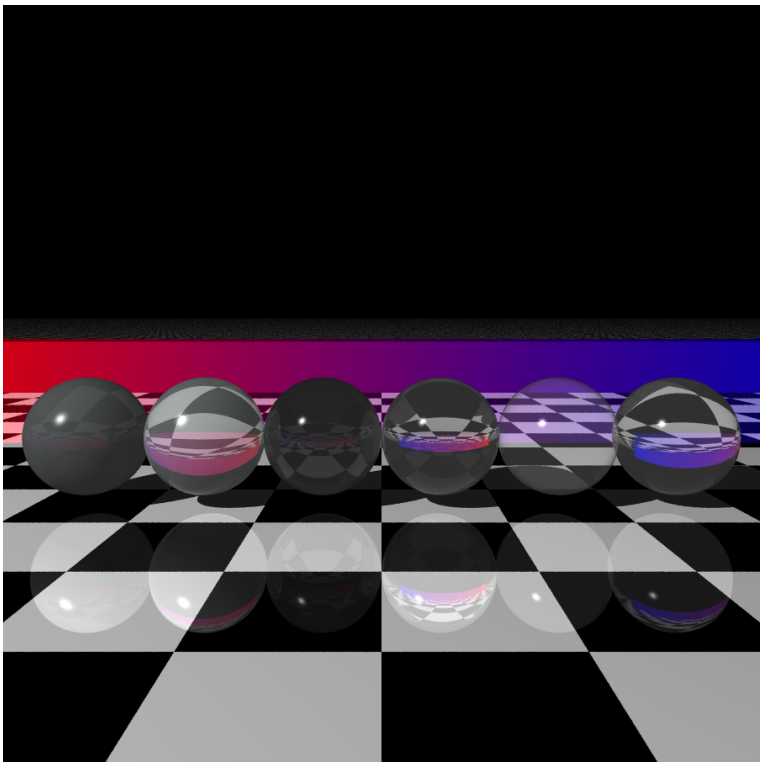
$$t = \frac{n_1}{n_2}((\sqrt{(n \cdot i)^2 + (\frac{n_2}{n_1})^2 - 1} - n \cdot i) \cdot n + i) \tag{11.4}$$

Kod refleksije ponekad dolazi do totalne unutarnje refleksije. U slučaju kad zraka prolazi iz sredstva s većim refraktivnim indeksom u sredstvo s manjim (npr. iz vode

u zrak) ako je kut θ_1 dovoljno malen zraka se samo odbiti od površine (ne dolazi do refrakcije). Do totalne unutarnje refleksije će doći kad je:

$$(n \cdot i)^2 > 1 - (\frac{n_2}{n_1})^2 \tag{11.5}$$

Kao i kod refleksije novo definira zraka prati se do novog sjecišta nakon čega smo odredili boju, i ovdje je potrebno ograničiti rekurziju.



Slika 11.1 Sfere s različitim refraktivnim svojstvima (slika dobivena vlastitom implementacijom)

11.1 Implementacija

Programski kod 11.1 Implementacija refrakcije

```
1 Color World::refractedColor(const Precomputations& comps, int&
      remaining, int remainingRefraction) const {
```

```
  if (remainingRefraction <= 0)
    return Color(0, 0, 0);
  if (epsilonEqual(comps.shape->material.transparency, 0))
    return Color(0, 0, 0);

  auto nRatio = comps.n1 / comps.n2;
  auto cosI = comps.eyev.dotProduct(comps.normalv);
  auto sin2T = nRatio * nRatio * (1 - cosI * cosI);

  if (sin2T > 1)
    return Color(0, 0, 0);

  auto cosT = sqrt(1.0 - sin2T);
  auto direction = comps.normalv * (nRatio * cosI - cosT) -
   comps.eyev * nRatio;

  auto refractRay = Ray(comps.underPoint, direction);
  auto refractColor = colorAt(refractRay, remaining,
   remainingRefraction - 1) * comps.shape->material.
   transparency;
  return refractColor;
}
```

11.2 Fresnelov efekt

Fresnelov efekt je svojstvo materijala da drukčije reagira na svjetlost ovisno o kutu gledanja. Na primjer ako promatramo more za velike kuteve gledanja (ako gledamo okomito u vodu) prozirnost je puno veća nego ako promatramo vodu pod manjim kutem (voda na većoj udaljenosti), u tom slučaju voda je puno reflektivnija. Fresnelov efekt, aproksimira se pomoću Schlickove aproksimacije. Schlickova aproksimacija, tj zrcalni koeficijent refleksije računa se prema jednadžbi:

$$R(\theta) = R_0 + (1 - R_0)(1 - cos(\theta))^5 \tag{11.6}$$

Gdje je

$$R_0 = (\frac{n_1 - n_2}{n_1 + n_2})^2 \tag{11.7}$$

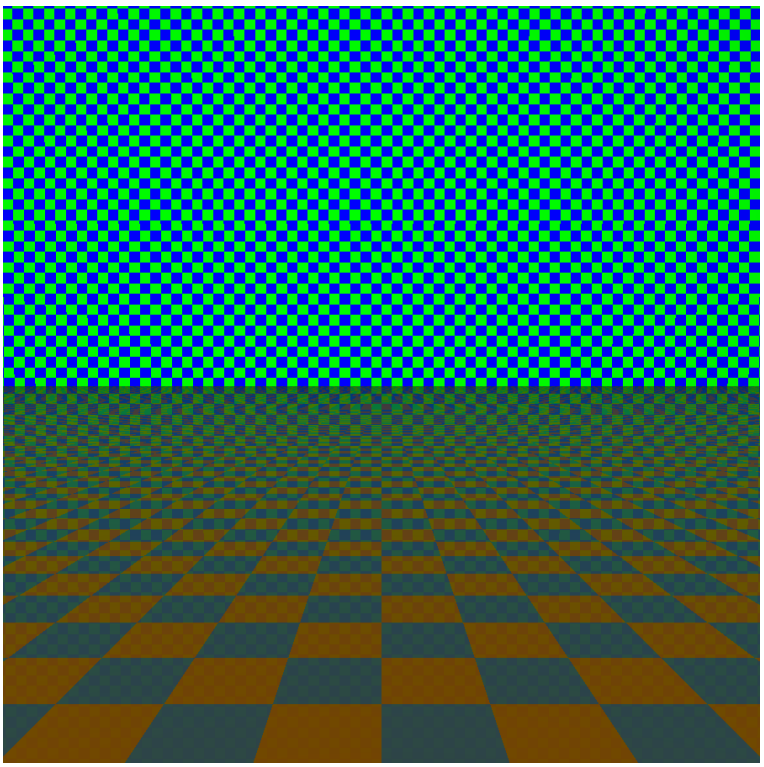
U slučaju da je materijal i reflektivan i proziran boja površine računa se kao $P + R \cdot R_0 + RF \cdot (1 - R_0)$ Gdje je: P: Phongova kontribucija za boju, R boja dobivena praćenjem reflektirane zrake, R_0 Schlikov koeficijent i RF boja dobivena praćenjem zrake refrakcije.

Ako površina nije i reflektivna i prozirna boja se računa kao suma: $P + R + RF$

11.3 Implementacija

Programski kod 11.2 Schlickova aproksimacija

```
double Precomputations::schlick()const {
  auto cos = eyev.dotProduct(normalv);
  if (n1 > n2) {
    auto n = n1 / n2;
    auto sin2T = n * n * (1.0 - cos * cos);

    if (sin2T > 1.0)
      return 1.0;
    cos = sqrt(1.0 - sin2T);
  }
  auto r0 = pow(((n1 - n2) / (n1 + n2)), 2);
  return r0 + (1 - r0) * pow(1 - cos, 5);
}
```



Slika 11.2 Fresnelov efekt; simulacija vode (slika dobivena vlastitom implementacijom)

Poglavlje 12

Sjene

Sjene postoje ako se neki objekt nalazi između izvora svjetlosti i točke u koju gledamo. Kao što je navedeno u poglavlju 8 difuzna i zrcalna kontribucija boja objekta ovise o izvoru svjetlosti, stoga ako je točka u sjeni oni se ignoriraju i dodaje se samo ambijentalna komponenta materijala. Da bi odredili da li je točka u sjeni potrebno je iz točke sjecišta odrediti novu zraku kojoj je ishodište sjecište s objektom, a usmjerena je prema izvoru svijetla. Novu zraku pratimo do izvora svijetla, i ako je na putu pogođen objekt onda imamo sijenu. S obzirom na to da se objekt može nalaziti i iza svijetla moramo i provjeriti da li je udaljenost do objekta manja od udaljenosti do svijetla.

$$\begin{aligned} v &= P - S \\ D &= ||v|| \\ sjena &= \begin{cases} 1, & \text{if } D > DH \\ 0, & \text{inače} \end{cases} \end{aligned} \tag{12.1}$$

Gdje je v vektor od sjecišta S do pozicije svijetla P, D udaljenost do svijetla i DH udaljenost do objekta ako je neki objekt pogođen.

Poglavlje 13

Izvori svjetlosti

U svakom prijašnjem primjeru korišten je točkasti izvor svijetla, to je izvor koji nema veličinu, postoji u jednoj točci u prostoru i osvjetljava prostor jednako u svim smjerovima. Naravno u stvarnosti takav izvor ne postoji, svaki izvor svjetlosti ima neku veličinu, intenzitet i smjer prema kojem putuju zrake svjetlosti. Glavna razlika u prikazu slike je u sjenama, točkasti izvor uvijek binarno odgovara na pitanje "da li je to ova točka u sjeni?" zbog čega su sjene uvijek jako oštre i uglavnom neprirodne.

13.1 Kružni reflektor

Postupak određivanja sjene jednak je kao kod točkastog izvora, kod točkastog izvora generirala se zraka koja je putovala od točke sjecišta do točke na kojoj se nalazi izvor. Kako točkasti izvor nema veličinu, zraka se uvijek vraćala na isto mjesto zbog čega je sijena bila binarno određena. Za razliku od točkastog izvora, kružni reflektor ima nekakvu veličinu (radijus), pa umjesto da testiramo tako što generiramo zraku od sjecišta do centra kruga (da to radimo imali bi isti rezultat kao kod točkastog izvora), pomicat ćemo točku prema kojoj zraka putuje. Potrebno je za n zraka provjeriti da li se točka nalazi u sjeni, i onda uzimamo prosjek vrijednosti koji predstavlja intenzitet sjene (ako je npr. 3 od 8 zraka stigne do izvora svjetla intenzitet je $\frac{3}{8}$). Kružno svijetlo definirano je sljedećim parametrima:

- Intenzitet - intenzitet svijetla koje dolazi iz reflektora

- Pozicija - točka u kojoj se nalazi centar reflektora
- Smjer - smjer u kojem reflektor osvjetljava prostor
- Kut - kut koji određuje širinu osvijetljenog prostora
- Broj uzoraka - koliko točaka na reflektoru ispitujemo
- Radius - veličina reflektora.

13.1.1 Biranje nasumične točke

Da bi mogli generirati nasumičnu točku na reflektoru, potrebno je najprije nasumično odabrati točku unutar jediničnog kruga.

$$\begin{aligned} x &= \sqrt{r} \cdot \cos\phi \\ y &= \sqrt{r} \cdot \cos\phi \end{aligned} \tag{13.1}$$

Gdje je $r \in [0, 1]$ i $\phi \in [0, 2\pi)$.

Potom određujemo dva vektora koji će definirati ravninu na kojoj se nalazi reflektor. Oba vektora trebaju biti okomita na normalu ravnine (normalu ravnine dobijemo oduzimanjem točke u kojoj se nalazi centar reflektora i točke u koju gleda) i međusobno okomiti. Da bi to postigli potrebno je odrediti nasumični vektor, i napraviti vektorski umnožak s normalnom ravine. Nakon što smo odredili prvi vektor treba još jednom uzeti vektorski umnožak novo dobivenoga vektora i normale. Svi vektori trebaju biti normalizirani.

$$\begin{aligned} N &= D - P \\ ON &= RND \times N \\ OB &= ON \times N \end{aligned} \tag{13.2}$$

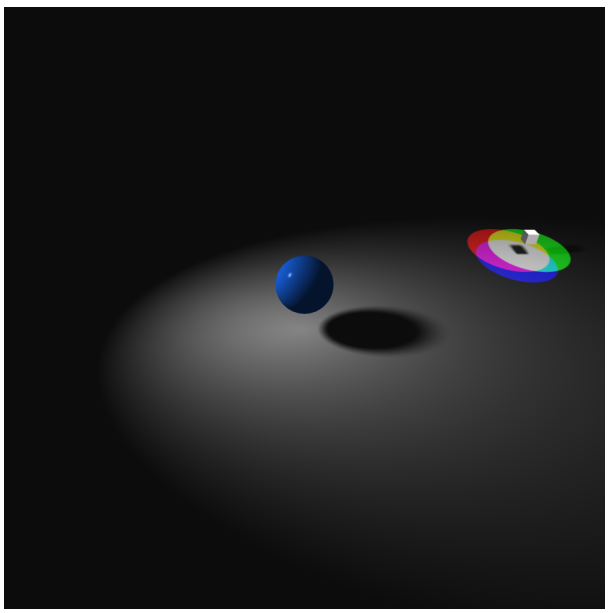
Gdje su N normala ravnine reflektora, D smjer, P centar reflektor, RND vektor s nasumičnim (x, y, z), ON vektor ortogonalan na normalu i OB vektor ortogonalan i na normalu i na vektor ON.

Nakon što smo odredili vektore, potrebno je još odrediti nasumičnu točku na reflektoru na koju će se vraćati zraka iz sjecišta. Pomicanjem vektora ON za prije

određenu vrijednost x i pomicanjem vektora OB za određeni y, te njihovim zbrajanjem dobijemo jedinični vektor koji pokazuje na nasumičnu vektor unutar jediničnog kruga. Napokon možemo dobiveni vektor pomnožiti s radijusom reflektora i zbrojiti s centrom reflektora da dobijemo nasumičnu točku na reflektoru. Isti postupak se ponavlja za još n točki ili uzoraka. U poglavlju 11 navedeno je da za sjenu uzimamo samo ambijentalnu vrijednost, pošto su kod kružnog reflektora neke zrake uspjele doći od sjecišta do izvora svijetla, to znači da imamo i difuznu i zrcalnu kontribuciju. Boja će se sada računati prema sljedećoj formuli:

$$boja = A + (\frac{sum}{n}) \cdot I \tag{13.3}$$

Gdje je A ambijentalna kontribucija, sum suma difuzne i zrcalne komponente za svaki uzorak n na i I intenzitet sjene. Implementaciju iz poglavlja 9.0.4 potrebno je prilagoditi tako da se za svaki uzorak na svijetlu računaju difuzna i zrcalna komponenta (sum u jednadžbi 13.3). I konačna boja se određuje prema jednadžbi 13.3.



Slika 13.1 Primjer scene generirane korištenjem kružnog reflektora (slika dobivena vlastitom implementacijom)

13.2 Pravokutni reflektor

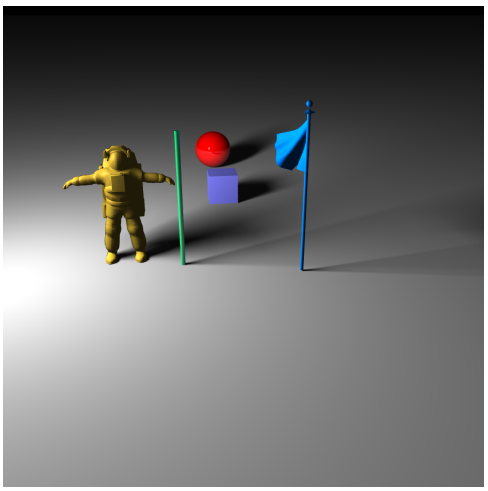
Kod pravokutnog reflektora mijenja se način na koji dobivamo nasumične točke na reflektoru, sve ostalo je jednako kao i kod kružnog reflektora, tj intenzitet i boju određujemo na isti način. Parametri koji određuju pravokutno svijetlo su:

- kut - točka koja se nalazi na donjem lijevom kutu reflektora
- fuvec - vektor koji određuje smjer i duljinu ruba (širinu)
- fvvec - vektor koji određuje smjer i duljinu riba (visinu)
- usteps - broj ćelija u smjeru u vektora
- vsteps - broj ćelija u smjeru v vektora
- uvec - vektor koji predstavlja širinu jedne ćelije ($fuvec/usteps$)
- vvec - vektor koji predstavlja visinu jedne ćelije ($vvec/vsteps$).

Za pravokutni izvor lakše je definirati točku prema kojoj će zraka putovati jer već imamo definirane vektore koji određuju ravninu na kojoj se reflektor prostire. Znamo da imamo $vsteps \cdot usteps$ ćelija, što znači da ćemo za svaki par (x, y) gdje je $x \in [0, usteps)$ i $y \in [0, vsteps)$ odrediti n nasumičnih točaka.

$$P = K + uvec \cdot (x + RND) + vvec \cdot (y \cdot RND) \tag{13.4}$$

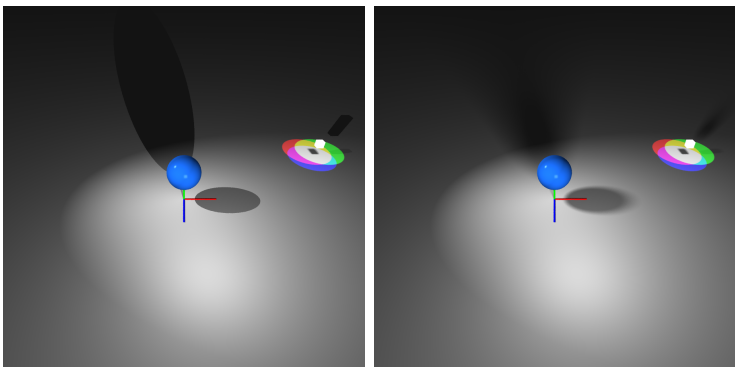
Gdje je P točka koju određujemo, K donji lijevi kut reflektora i $RND \in [0, 1]$



Slika 13.2 Scena dobivena korištenjem pravokutnog reflektora (slika dobivena vlastitom implementacijom)

13.3 Više izvora svjetlosti

Ako imamo više izvora svjetlosti za svaki izvor svjetlosti posebno izračunamo boju površine i sjene i nadodamo na ukupnu vrijednost boja. Refleksiju i refrakciju računamo jednom za sva svjetla.



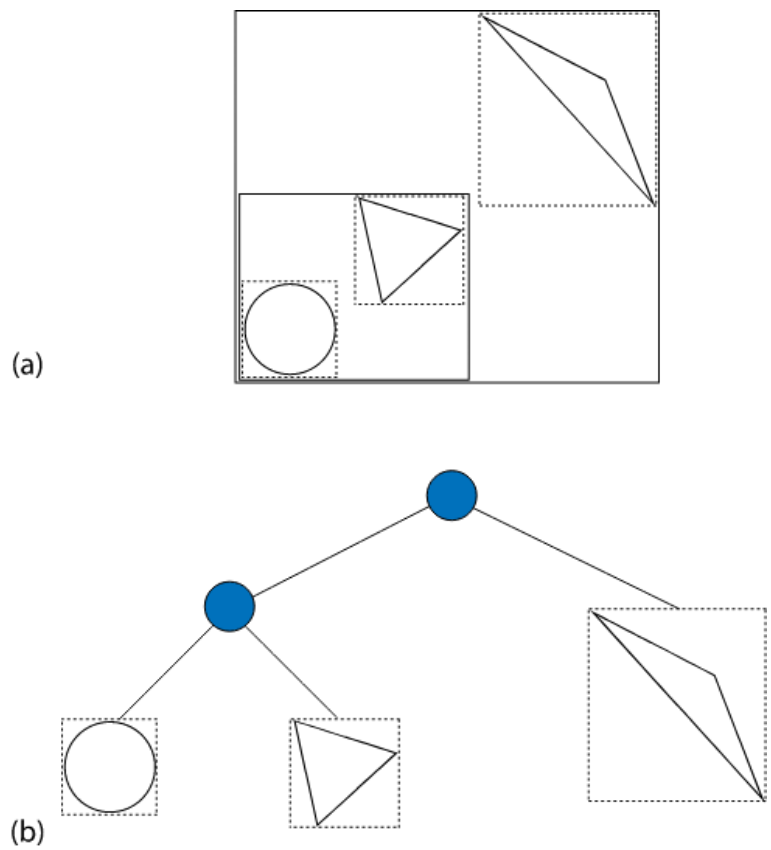
Slika 13.3 Scena dobivena korištenjem i kružnog i pravokutnog reflektora (na lijevoj slici ugašene su meke sjene) (slika dobivena vlastitom implementacijom).

Poglavlje 14

BVH

Svaki put kad je zraka ispaljena u prostor potrebno je provjeriti da li je došlo do sjecišta sa svakim od objekata u sceni. Za scene koje sadrže malo objekata to nije problem, međutim s porastom kompleksnosti scene, npr. korištenja složenih modela koji se sastoje od tisuća trokuta vrlo brzo dolazi do velikog usporavanja programa. Osim sjecišta samog objekta također se računaju zrake za sjene refrakciju, itd., što također zahtjeva dodatne resurse. BVH dijeli primitivne tipove u listove i povezuje ih u stablo. Za razliku od tehnika za podjelu prostora, BVH ne zahtijeva nužno da se dijeli prostor već on samo grupira objekte (moguća je čak nasumična podjela, no to ne bi bilo efikasno). Grupiranje objekata omogućuje provjeru da li zraka sječe grupu i ako ne onda možemo preskočiti provjeru za sve ostale objekte unutar grupe.

Često se kao "kutija" u koju ubacujemo primitivne tipove koji će činiti grupu koriste kocke. Može se koristiti bilo koji primitivni tip, međutim kocka najbliže opisuje svaki od primitivnih tipova, na slici 14.1 možemo vidjeti da nije savršeno i neki oblici imaju veća odstupanja od drugih.



Slika 14.1 BVH za jednostavnu scenu. (a) prikazuje kako su objekti opisani kvadratima (u 3D prostoru kockama) (b) Dobiveni BVH u obliku stabla [18]

14.1 Opisivanje primitivnih tipova

Kako bi mogli svrstati sve primitivne tipove u kocke za svaki od njih potrebno je odrediti kocku koja ga opisuje s najmanjim mogućim odstupanjem.

14.1.1 Sfera

S obzirom da su sfere centrirane u ishodištu i imaju radius 1, onda znamo da će kocka koja opisuje sferu biti centrirana u ishodištu i prostirat će se od *Min(-1, -1, -1), Max(1, 1, 1)*

14.1.2 Kocka

Kocka se očito može savršeno opisati drugom kockom, pa za nju uvijek dobijemo savršene granice. Za kocku koja ju opisuje potrebno je samo odrediti iste dimenzije. Za jediničnu kocku to je *Min(-1, -1, -1), Max(1, 1, 1)*

14.1.3 Valjak

Jedinični valjak se prostire u beskonačnost u Y osi. Zbog rotacija može se desiti da se prostire u svim smjerovima u beskonačnost, pa opisivanje s kockom neće biti korisno. Svejedno za jedinični cilindar, možemo ga opisati s kockom *Min(-1, -inf, -1), Max(1, inf, 1).* Puno više smisla ima ograničiti valjak koji ima određen minimum i maksimum. Za cilindar kojemu su određeni *Min = -5, Max = 5* imamo određenu kocku koja ga opisuje sa *Min(-1, min, -1), Max(1, max, 1)*

14.1.4 Stožac

Kod jediničnog stošca opisivanje kockom ima još manje smisla jer se on prostire u beskonačnost u svim dimenzijama. Međutim svejedno je moguće to napraviti kockom koja se prostire od *Min(-inf, -inf, -inf) do Max(inf, inf, inf).* Kao i kod valjka opisivanje stošca kockom puno je efikasnije kad su određeni minimum i maksimum stošca, pa tako za stožac koji ima određene npr. *Min(-5), Max(3)*, možemo odrediti kocku na sljedeći način:

$$\begin{aligned} a &= |cone.min| = 5 \\ b &= |cone.max| = 3 \\ limit &= max(a, b) = 5 \end{aligned} \tag{14.1}$$

Za kocku onda imamo određene sljedeće točke minimuma i maksimuma: *Min(-limit, cone.max, -limit), Max(limit, cone.max, limit).*

14.1.5 Ravnina

Slično kao i valjak i stožac ravnina se prostire u beskonačnost kroz dvije osi (XZ), međutim njoj ne možemo ograničiti veličinu. Svejedno radi izbjegavanja rubnih slučajeva, definira se kocka *Min(-inf, 0, -inf), Max(inf, 0, inf).*

14.1.6 Trokuti

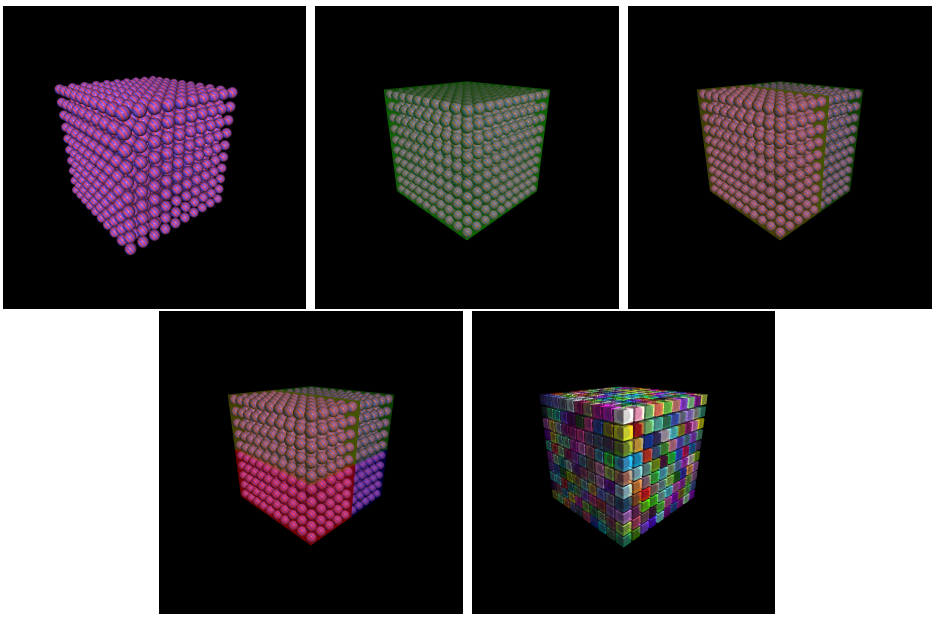
Za pronaći kocku koja opisuje trokut dovoljno je odrediti najmanju i najveću x, y i z komponentu triju točaka koje određuju trokut. Pa tako ako imamo točke trokuta:

$$\begin{aligned} p_1 &= (-3, 7, 2) \\ p_2 &= (6, 2, -4) \\ p_3 &= (2, -1, -1) \end{aligned} \tag{14.2}$$

Dobijemo kocku kojoj su minimum i maksimum *Min(-3, -1, -4), Max(6, 7, 2).*

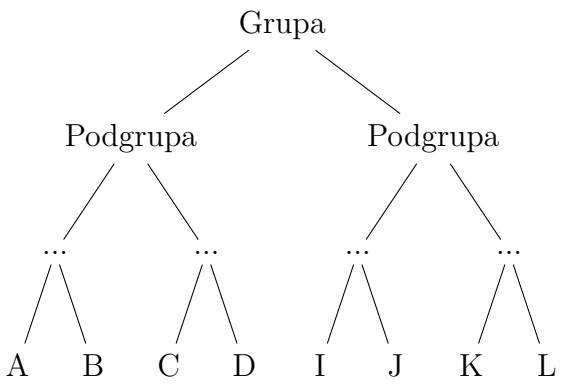
14.1.7 Grupe

BVH ne opisuje samo jedan objekt u sceni, već on grupira sve objekte u jednu kutiju, koja se onda dijeli na podgrupe odnosno na manje kutije u koje se stavljaju objekti. Proces podjele radi tako da najprije odredimo kutiju koja opisuje sve objekte u sceni, potom geometrijski podijelimo kutiju po najvećoj osi (ako su jednake možemo odabrati bilo koju), sve objekte ubacimo podijelimo u nove manje konstruirane "kutije", objekte na rubu možemo ostaviti u najvećoj "kutiji". Na slici 14.2 vidi se najprije inicijalni raspored sfera. Potom vidimo da se sve sfere nalaze u unutar jedne velike kocke, pa je kocka prepolovljena na dvije manje po X osi, pa onda još jednom prepolovljena po Y osi. Iterativnim postupkom dolazimo do zadnje slike koja prikazuje svaki sferu u svojoj kocki.



Slika 14.2 Inkrementalna podjela kocke na manje segmente (slika dobivena vlastitom implementacijom)

Ili ako istu sliku prikažemo u obliku stabla dobijemo:



Slika 14.3 Stablo (BVH) dobiveno podjelom prostora.

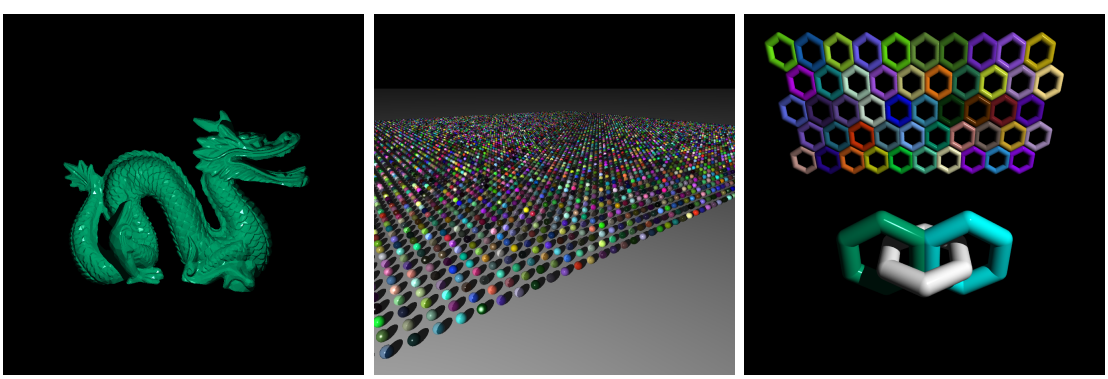
Iz stabla očito je da ako želimo odrediti koji je objekt pogođen najprije provjerimo ako je pogodio grupu, ako je provjerimo koju od podgrupa, itd. dok ne dođemo do listova i samog objekta koji je pogođen. Za BVH koji pohranjuje primitivne tipove

u listove potrebno je 2n - 1 čvorova gdje je n broj primitivnih tipova. Postojat će n listova i n - 1 unutarnjih čvorova (grupa). Što znači da ćemo za npr. 100 elemenata ako zraka promaši grupu uštedjeti na testiraju svih 100 objekata zasebno, u slučaju da je neka sfera ili objekt pogođen, da bi odredili koji trebat će nam otprilike $log_2 100$ odnosno 6 (otprilike jer ovisi kako će se objekti rasporediti u stablu, nije svaki put moguća savršena raspodjela kao u primjeru), u svakom slučaju puno manje od 100. Tablica prikazuje vrijeme potrebno za prikaz scene ako se koristi ili ne koristi BVH.

Tablica 14.1 Brzina izvođenja s i bez BVH optimizacije.

Scena	**Broj elemenata**	**Vrijeme (ms), s BVH**	**Vrijeme (ms), bez BVH**
Many Spheres	10000	18,456	1,133,146
Dragon	24000	8,222	189,877
SphereCube	1000	8,134	35,503
Hexagon	636	1,878	1,916

Iz tablice vidimo da je prikaz scene značajno brži ako je veći broj poligona u sceni. Dok je za scene s malim brojem poligona približno jednako brzo. Scena "Many Spheres" sporija je od scene "Dragon" iako ima manje elemenata jer sfere mogu biti i reflektivne i refraktivne što znači da je potrebno generirati dodatne zrake.



Slika 14.4 Kompleksije scene za koje je poželjno imati BVH optimizaciju (slika dobivena vlastitom implementacijom)

Poglavlje 15

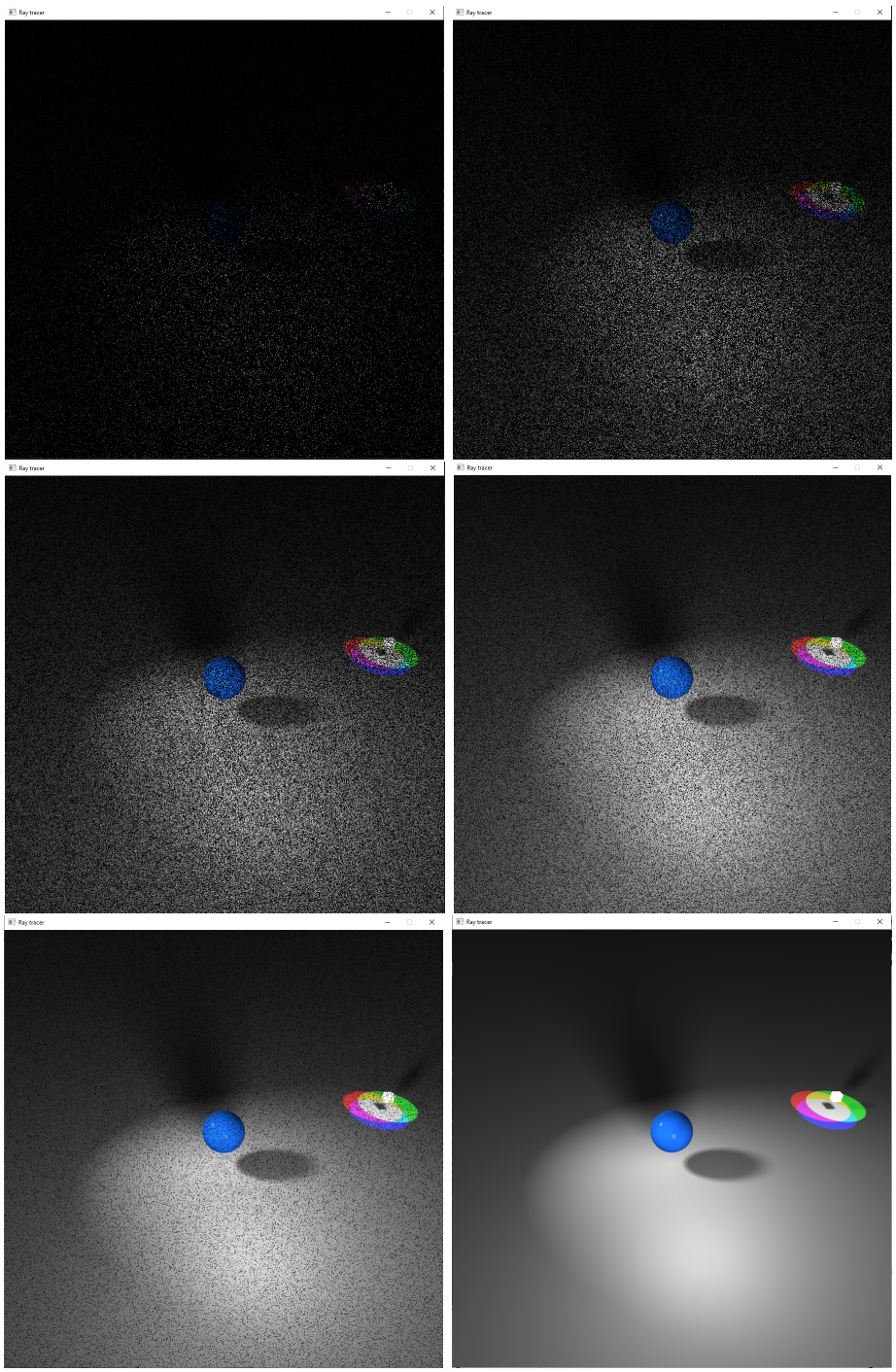
Implementacija

Dobar dio implementacije prikazan je u prijašnjim poglavljima, pa će se ovo poglavlje više fokusirati na optimizaciju i razne probleme koji su javili prilikom implementacije, te na prikaz slike.

15.1 Prikaz slike

15.1.1 RayLib

Kako ne bi trebali čekati da svi izračuni završe za prikaz trenutnog stanja slike koristi se RayLib [19]. RayLib je knjižica koja se koristi za grafičke aplikacije i video igre. Kad se boja piksela odredi prosljeđuje se RayLibu koji osvježi sliku koja se prikazuje.



Slika 15.1 Prikaz scene u različitim trenutcima (5, 25, 50, 75, 90 i 100% svih piksela) (slika dobivena vlastitom implementacijom)

Na slici 15.1 prikazana je scena u različitim trenutcima. Redom na slikama je prikazano 5, 25, 50, 75, 90 i 100% od ukupnih piksela.

15.1.2 PPM

Nakon što je slika gotova pohranjuje se u PPM formatu. PPM datoteke počinju najprije oznakom formata (npr. P2, P3, itd) nakon čega slijede širina i visina slike (u ovom radu pohranjuju se u P3 formatu). Potom vrijednost koja predstavlja najveću moguću vrijednost boje, i onda podatci koji opisuju sliku. Za P3 podatci su zapisani u ASCII formatu gdje trojke brojeva predstavljaju jedan pixel (R, G, B). Format P6 je isti samo su podatci u binarnom formatu. Drugim riječima kako bi dobili sliku u PPM formatu za svaki piksel pohranjuje se njegova RGB vrijednost.

15.2 Optimizacije i performanse

15.2.1 Višenitnost

Značajno poboljšanje u performansama došlo je nakon implementacije višenitnosti. Prilikom računanja vrijednosti za svaki piksel možemo primijetiti da je svaki piksel odnosno svaka boja neovisna od svih ostalih piksela u sceni. Iz čega zaključujemo da je Ray Tracing embarrassingly parallel problem, to jest moguće je podijeliti posao (paralelizirati) na više dretvi uz minimalne promjene. Konkretno u kodu se najprije pomiješaju sve točke (svi pikseli) unutar jednog vektora (u ovom radu *shufflePoints*) koji se onda predaje dretvama. Svaka dretva onda uzima jedan piksel i računa boju.

Programski kod 15.1 Implementacija višenitnosti

```
for (auto y = 0; y < vSize; ++y) {
    for (auto x = 0; x < hSize; ++x) {
        shufflePoints.emplace_back(x, y);
    }
}
....
for (int i = 0; i < (int)numOfThreads; ++i) {
```

```
8       // give a bit of time for each tread to start otherwise
        some pixels get skipped
9       busyLoop(10);
10      threads.push_back(std::thread(&Camera::splitArray, this, &
        world, &image));
11  }
12   ...
13  void Camera::splitArray(World* world, Canvas* image) {
14
15      int x = shufflePoints[curIndex].first;
16      int y = shufflePoints[curIndex].second;
17      ...
18      auto ray = rayForPixel(x, y);
19      pixelColor += world->colorAt(ray);
20      image->writePixel(x, y, pixelColor);
21  }
```

15.2.2 Caching

Inverz transformacije koristi se za prebacivanje zrake iz prostora svijeta u prostor objekta, također koristi se za prebacivanje točke u prostor objekta kod računanja normale i obrnuto. Što znači da imamo nekoliko poziva funkcije za inverz za svaku ispaljenu zraku. Inverz također poziva funkciju za izračun determinante, koji koristi funkciju za cofactor, itd. Umjesto da svaki put računamo inverz i determinantu oni se mogu cacheirati što znatno ubrzava rad programa.

Programski kod 15.2 Implementacija cacheiranja

```
1  double Matrix::determinant()const {
2
3      if (!(isnan(cacheDeterminant)))
4          return cacheDeterminant;
5
6      if(w == 2)
7          return matrix[0] * matrix[3] - matrix[1] * matrix[2];
```

```
8
9       double det = 0;
10      for (int i = 0; i < w; ++i) {
11          det += matrix[i] * cofactor(0, i);
12      }
13      cacheDeterminant = det;
14      return det;
15  }
16  ...
17  Matrix* Matrix::inverse() {
18      if (!invertible()) {
19          std::cout << "Not invertible!\n";
20      }
21
22      if (cachedMatrix == nullptr) {
23          cachedMatrix = new Matrix(w, h);
24          for (int i = 0; i < w; i++) {
25              for (int j = 0; j < h; ++j) {
26                  double c = cofactor(i, j);
27                  cachedMatrix->matrix[j * w + i] = c /
    determinant();
28              }
29          }
30          return cachedMatrix;
31      }
32      return cachedMatrix;
33  }
```

15.2.3 Rezultati optimizacija

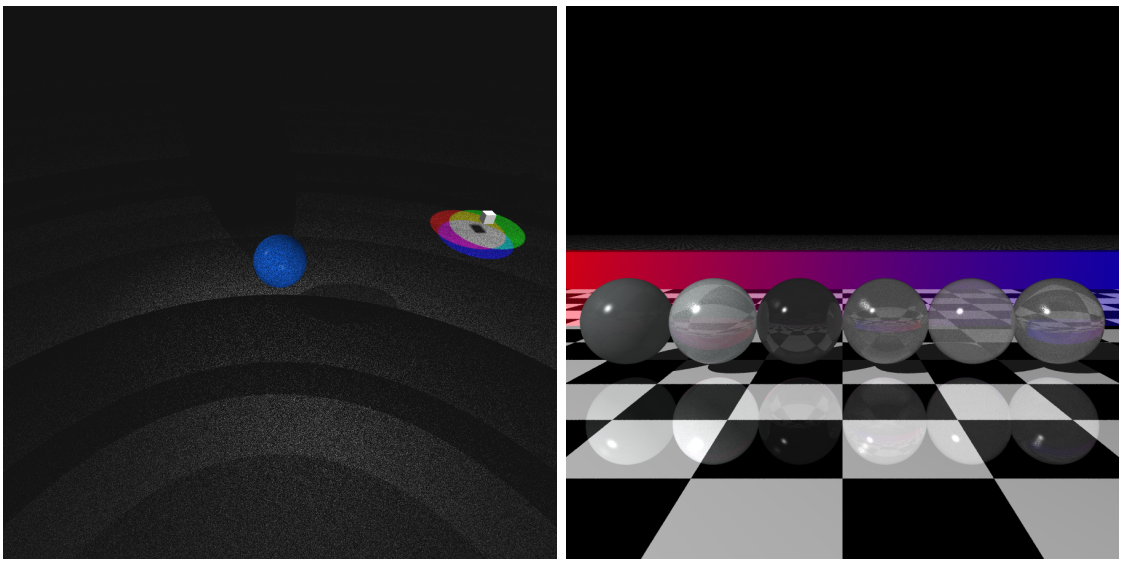
Slika 15.2 prikazuje rezultate optimizacija na jednostavnoj sceni koja sadrži jednu sferu i ravninu.

Slika 15.2 Utjecaj optimizacija i višenitnosti na vrijeme izvođenja programa

Sa slike 15.2 vidimo da je cacheiranja determinante i inverza ekstremno bitno, čak više ubrzava program od korištenja 12 više dretvi. Međutim i višedretvenost ima značajan utjecaj na vrijeme izvođenja.

15.3 Akne

Reprezentacija floating point brojeva nije u potpunosti točna, tj može se pojaviti mala greška kod računanja. Problem se javlja kod izračuna sjena i refrakcije. Konkretno može se desiti da je zbog greške sjecište sa zrakom pomaknuto malo ispod površine objekta, zbog čega objekt baca sijenu sam na sebe u točci sjecišta.



Slika 15.3 Scene koje imaju akne zbog greške u izračunu (slika dobivena vlastitom implementacijom)

Na slici 15.3 vidimo kako slika izgleda zbog greške kod računanja. Rješenje je da pomaknemo točku sjecišta u smjeru normale, normala se skalira kako bi pomak bio minimalan.

Programski kod 15.3 Pomak točke u smjeru normale

```
...
const double EPSILON = 1.e-011;
...
overPoint = point + normalv * (EPSILON);
underPoint = point - normalv * (EPSILON);
}
```

OverPoint koristi se kada želimo točku pomaknuti iznad površine (za sjene) objekta, a *underPoint* kada želimo točku pomaknuti malo ispod površine (za refrakciju).

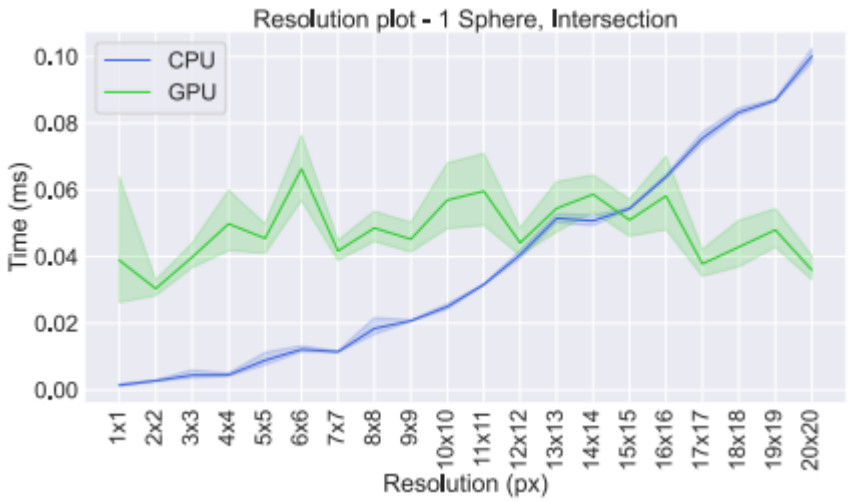
Poglavlje 16

Poboljšanja

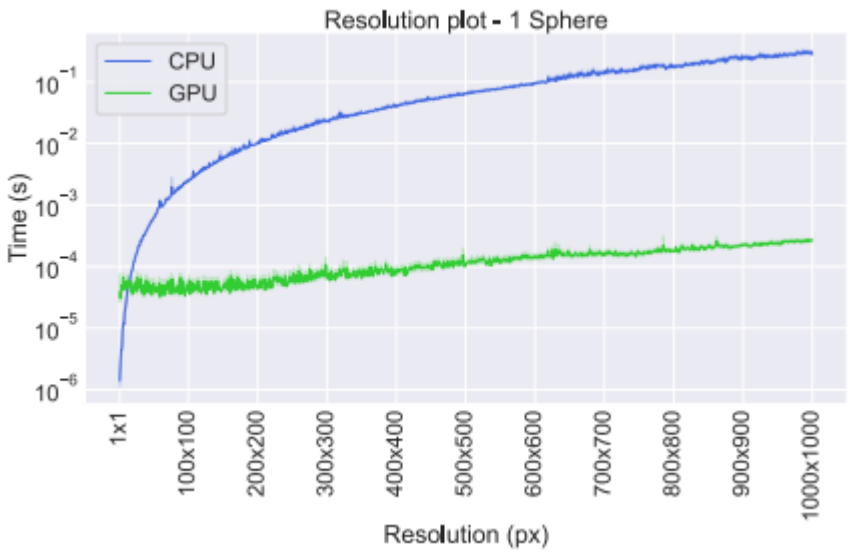
Naravno vrlo vjerojatno postoji mnogo poboljšanja koja bi se mogla napraviti počevši od same strukture koda, do smanjenja broja alokacija, korištenja boljih algoritama, itd. U ovom poglavlju navest ću neka od poboljšanja ili ideja na koja sam naišao dok sam istraživao različitu literaturu vezanu za ovaj rad.

16.1 GPU ray tracing

S nedavnim poboljšanjima u hardware-u specifično s novim NVIDIA RTX grafičkim karticama sve je više ray-tracing algoritama modificirano da radi uz pomoću grafičkih kartica. Grafičke kartice su bolje u paraleliziranju zadataka što rezultira značajno boljim performansama. Još uvijek postoje neka ograničenja kod jako velikih scena koje se ne mogu učitati u memoriju grafičke kartice, međutim s napredcima u tehnologiji ograničenja su sve manja i korištenje GPUs postaje raširenije.



Slika 16.1 CPU vs GPU za male rezolucije [20]



Slika 16.2 CPU vs GPU za veće rezolucije [20]

Sa slike 16.1 vidi se da je za vrlo male rezolucije CPU ray tracing brži, međutim kod većih rezolucija (slika 16.2) (za scene koje je moguće učitati u memoriju GPUa) korištenje GPU-a ima značajno bolje rezultate.

16.2 OBJ teksture

U ovom radu teksture je moguće koristiti na primitivnim tipovima, međutim nije podržano učitavanje tekstura koje se prikazuju na .obj modelima. Dodavanje tekstura na .obj modele omogućilo bi prikazivanje modela kako ih je dizajner originalno zamislio.

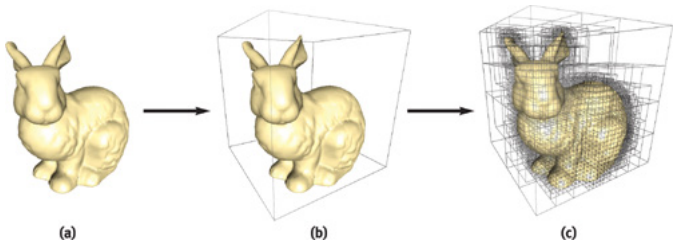
Slika 16.3 Primjer modela s i bez tekstura [21]

16.3 Memory arena

U kodu se kod npr. svakog sjecišta sa sferom stvara novi objekt "Intersection" na hrpi (engl. heap) što znači da je za svako sjecište potrebno ponovno alocirati memoriju što usporava rad programa, također istu memoriju je potrebno i dealocirati. Za razliku od toga arene alociraju svu memoriju na početku programa (zauzima se neka količina memorije koja će biti dovoljna za vrijeme izvršavanja cijelog programa) što znači da će se sve alokacije dešavati instantno.

16.4 Podjela prostora

Iako nije nužno poboljšanje bilo bi zanimljivo usporediti performance algoritma za podjelu prostora (npr. octree) u odnosu na BVH.



Slika 16.4 Podjela prostora koristeći Octree [22]

Poglavlje 17

Zaključak

Ovaj rad predstavio je osnovnu implementaciju ray tracing algoritma, optimizacija (BVH, caching), višenitnosti, materijala, modela refleksije, refrakcije, sjena itd. Naravno moderni ray tracing alati poput Pixarovog RenderMan-a, Autodeskovog Arnold-a, V-ray-a puno su brži, nude više opcija, koriste brže algoritme za izračun sjecišta, podjelu prostora, itd. Podržavaju više tehnika ray-tracing, path-tracing, rasterizaciju, itd. Kako autor ovog rada nema prijašnjeg iskustva s razvojem ray-tracing algoritma, a i razvoj kompleksnih alata zahtjeva jako puno vremena i resursa i nije posao za jednog čovjeka, nikad ni nije bio cilj konkurirati modernim alatima.

Međutim, smatram da je projekt bio uspješan iako kao što je navedeno u poglavlju 15 postoji mnogo prostora za poboljšanja i još uvijek imam želju vratiti se ovom projektu i implementirati spomenute ideje. Stekao sam dobro razumijevanje o tome što je ray-tracing, kako radi, koji su problemi, kako je moguće ubrzati izvršavanje programa i koja je budućnost za ray-tracing u stvarnome vremenu (engl. real time ray-tracing). Područje ray-tracinga i simulacija osvjetljenja jako je široko i kompleksno, ali smatram da ovaj rad služi kao dobar uvod u ovo područje i implementira sve osnovne elemente jednog ray-tracing alata.

Bibliografija

[1] Whitted, T: *An improved illumination model for shaded display*, *Communications of the ACM*, 23(6), 343–349, 1980, doi: 10.1145/358876.358882.

[2] Jamis Buck, *The Ray Tracer Challenge: A Test-Driven Guide to Your First 3D Renderer*, Pragmatic Bookshelf, 2019, ISBN: 978-1680502715.

[3] Shirley, P., *Ray Tracing in One Weekend*, Amazon Kindle books, 2018.

[4] *Raytracing Algorithm in a Nutshell*, Nepoznati autor, Scratchapixel, `https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/raytracing-algorithm-in-a-nutshell.html`, pristupljeno: 18.6.2023.

[5] *CS184 Lecture: Acceleration - Slide 8*, Ren Ng, University of California, Berkeley, 2016, `https://cs184.eecs.berkeley.edu/sp16/lecture/acceleration/slide_008`, pristupljeno: 18.6.2023.

[6] *Windows Desktop App Development*, Microsoft, Microsoft Documentation, `https://learn.microsoft.com/en-us/previous-versions/windows/desktop/bb324490(v=vs.85)`, pristupljeno: 18.6.2023.

[7] Bui Tuong Phong, *Illumination for computer generated pictures*, Communications of the ACM, 18(6), 311–317, 1975, doi: 10.1145/360825.360839.

[8] Eric W. Weisstein, *Normal Vector*, MathWorld–A Wolfram Web Resource, `https://mathworld.wolfram.com/NormalVector.html`, pristupljeno: 18.6.2023.

[9] Möller, T., Trumbore, B. *Fast, Minimum Storage Ray/Triangle Intersection.* Journal of Graphics Tools, 2(1):21-28, 1997.

[10] Nepoznati autor, *Barycentric Coordinates*, Scratchapixel, URL: `https://www.scratchapixel.com/lessons/3d-basic-rendering/`

ray-tracing-rendering-a-triangle/barycentric-coordinates.html, pristupljeno: 18.6.2023.

[11] Nepoznati autor, *Raycasting Against Axis-Aligned Bounding Boxes*, GDBooks, URL: https://gdbooks.gitbooks.io/3dcollisions/content/Chapter3/raycast_aabb.html, pristupljeno: 18.6.2023.

[12] Marco Alamia, *Understanding the World-View-Projection Matrix*, Coding Labs, URL: http://www.codinglabs.net/article_world_view_projection_matrix.aspx, pristupljeno: 18.6.2023.

[13] Song Ho Ahn, *OpenGL: Look At Matrix and the View Matrix*, Song Ho Ahn's website, URL: https://www.songho.ca/opengl/gl_lookattoaxes.html, pristupljeno: 18.6.2023.

[14] Song Ho Ahn, *OpenGL: Camera*, Song Ho Ahn's website, URL: https://www.songho.ca/opengl/gl_camera.html, pristupljeno: 18.6.2023.

[15] Nepoznati autor, *Generating Camera Rays*, Scratchapixel, URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays.html, pristupljeno: 18.6.2023.

[16] Nepoznati autor, *Supersampling*, Wikiwand, URL: https://www.wikiwand.com/en/Supersampling, pristupljeno: 18.6.2023.

[17] James Buck, *Texture Mapping*, Ray Tracer Challenge, URL: http://raytracerchallenge.com/bonus/texture-mapping.html, pristupljeno: 18.6.2023.

[18] Matt Pharr, Wenzel Jakob, and Greg Humphreys, *Bounding Volume Hierarchies*, *Physically Based Rendering*, 3rd Edition, 2018, URL: https://pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies, pristupljeno: 18.6.2023.

[19] Ramon Santamaria, *Raylib: A simple and easy-to-use library to enjoy videogames programming*, URL: https://www.raylib.com/, pristupljeno: 18.6.2023.

[20] Robin Nordmark and Tim Olsén, *A Ray Tracing Implementation Performance Comparison between the CPU and the GPU*, Degree Project in Computer Science and Engineering, School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, pristupljeno: 18.6.2023.

[21] Nepoznati autor, *Hand-Painted Textures on 3D Models using Photoshop*, Stylized Station, URL: `https://stylizedstation.com/tutorials/hand-painted-textures-on-3d-models-using-photoshop/`., pristupljeno: 18.6.2023.

[22] Fernando, R., *Octree Textures on the GPU*, In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, edited by Pharr, M., Fernando, R., Addison-Wesley Professional, 2005, Chapter 37, URL: `https://developer.nvidia.com/gpugems/gpugems2/part-v-image-oriented-computing/chapter-37-octree-textures-gpu`.

Pojmovnik

AA anti aliasing. 54

AABB axis-aligned bounding box. 24

BVH bounding volume hierarchy. 2

CSG Constructive solid geometry. 38

FOV field of view. 45

GPU grafičke kartice. 1

Sažetak

Ray-tracing je algoritam koji stvara fotorealistične slike simulirajući interakciju svjetlosti s objektima u sceni. U ovom radu detaljno je opisan cijeli proces algoritma, počevši od generiranja zrake pa sve do određivanja sjecišta i simulacije različitih fizikalnih pojava poput refleksije, refrakcije, sjena, sjaja i drugih. Poseban naglasak stavljen je na upotrebu ubrzavajuće strukture poznate kao BVH (Bounding Volume Hierarchy) te na utjecaj paralelizacije na performanse algoritma. U praktičnom dijelu rada razvijen je program koji koristi ray-tracing za generiranje fotorealističnih slika. Kroz primjere prikazane su slike generirane algoritmom s različitim postavkama i scenama. Na kraju rada, dan je kratak osvrt na moguća poboljšanja.

Ključne riječi — **Ray-tracing, BVH, refleksija, refrakcija, sjene**

Abstract

Ray-tracing is an algorithm that produces photorealistic images by simulating light and the interaction between light and objects in a scene. This paper provides a detailed description of the entire algorithm, starting from ray generation to intersection determination and the simulation of various physical phenomena such as reflection, refraction, shadows, highlights, and more. Special emphasis is placed on the use of an acceleration structure known as BVH (Bounding Volume Hierarchy) and the impact of parallelization on the algorithm's performance. In the practical part of the paper, a program is developed that utilizes ray-tracing to generate photorealistic images. Examples are presented to showcase images generated by the algorithm with different settings and scenes. Finally, the paper concludes with a brief discussion on potential improvements.

Keywords — **Ray-tracing, BVH, reflection, refractions, shadows**

Dodatak A

Upute za preuzimanje i pokretanje programskog koda

U oba slučaja (Windows i Linux) potrebno je imati neki cpp prevodioc, podržani su MSVC, gcc, g++ i clang. Dodatno za Linux je potrebno imati make.

A.1 Windows

Za pokretanje programa potrebno je preuzeti knjižicu RayLib i postaviti ju u direktorij programskog koda priloženog uz ovaj rad. Potom se može buildati i pokrenuti program korištenjem jedne od priloženih skripti (buildClang.sh, buildgcc.sh). Također je moguće koristiti Visual Studio u tom slučaju nije potrebno preuzimati RayLib već se on može dodati kao NuGet paket. Nakon instalacije potrebno je otvoriti .snl rješenje što će podesiti sve unutar Visual studia.

A.2 Linux

Potrebno je preuzeti RayLib i postaviti ga u direktorij programskog koda. Kod Linuksa je u mom slučaju bilo potrebno instalirati i dodatne knjižice uz pomoć komande: `sudo apt install libasound2-dev mesa-common-dev libx11-dev libxrandr-dev`

`libxi-dev xorg-dev libgl1-mesa-dev libglu1-mesa-dev`. Međutim preporučio bih da pogledate službenu RayLib dokumentaciju. Kao i kod Windowsa još je potrebno pokrenuti jednu od skripti (buildClang.sh, buildgcc.sh) za buildanje projekta.

A.3 Odabir scene

Upisivanjem komande: `RayTracer -help` za windows ili ./RayTracer –help za linux prikazati će se sve dostupne scene i opcije (aliasing, dimenzije slike, fokalna udaljenost, broj dretvi, itd) koje se mogu koristiti.

Komanda je oblika `[Ime scene] --[opcija] --[opcija]...`

Primjer komande za pokretanje (Windows):`RayTracer Aliasing --aliasing 8`

Primjer komande za pokretanje (Linux):`./RayTracer Aliasing --aliasing 8`