SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6644

PRIKAZ PROZIRNIH I REFLEKTIRAJUĆIH POVRŠINA

Filip Husnjak

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6644

PRIKAZ PROZIRNIH I REFLEKTIRAJUĆIH POVRŠINA

Filip Husnjak

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

ZAVRŠNI ZADATAK br. 6644

Pristupnik:	**Filip Husnjak (0036506711)**
Studij:	Računarstvo
Modul:	Računarska znanost
Mentor:	prof. dr. sc. Željka Mihajlović

Zadatak: **Prikaz prozirnih i reflektirajućih površina**

Opis zadatka:

Proučiti fizikalne osnove vezane uz prikaz prozirnih i reflektirajućih površina. Proučiti programsko grafičko sučelje Vulkan te platformu Nvidia RTX. Razraditi mogućnosti prikaza prozirnih i reflektirajućih površina upotrebom sučelja Vulkan i platforme Nvidia RTX. Ostvariti niz primjera koji demonstriraju dobivene rezultate te usporediti sa sličnim objektima u stvarnom svijetu. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++ i grafičko programsko sučelje Vulkan. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 12. lipnja 2020.

SADRŽAJ

1. Uvod

Napretkom tehnologije i razvojem grafičke opreme dolazi do ideje o novom načinu stvaranja slike virtualnog svijeta na ekranu. Ideja je imitirani način prostiranja svjetlosti kroz prostor, a pošto je pretpostavka da je putanja svjetlosti pravac, algoritam se zove *Algoritam praćenja zrake* (engl. *Ray tracing algorithm*). To je jedan od algoritama globalnog osvjetljenja, a rezultat su slike visokog stupnja realističnosti. Fizikalni pristup problemu je praćenje zrake iz svakog izvora svjetlosti u sceni promatrajući njezinu interakciju s objektima i stvaranje konačne slike koju vidi promatrač. Međutim ovaj način vrlo je nepogodan za računala prvenstveno zato što izvor stvara beskonačan broj zraka što bi se trebalo na neki način diskretizirati. Puno pogodniji način je pratiti zraku u obrnutom smjeru, odnosno započeti zraku u očima promatrača. Time već dobivamo potrebnu diskretizaciju pošto se slika na ekranu prikazuje uz pomoć konačnog broja slikovnih elemenata. Iz ovoga proizlazi glavna ideja algoritma: za svaki slikovni element odredi se pripadna zraka, prati se njezin put kroz scenu te u konačnici odredi boja slikovnog elementa. Ovaj način stvaranja slike daje nam mogućnost prikaza sjena, reflektirajućih i refraktirajućih površina na vrlo jednostavan i intuitivan način što je za uobičajen način stvaranja slike uz pomoć rasterizacije predstavljao problem. Glavni nedostatak *Algoritma praćenja zrake* je sporost. U usporedbi s rasterizacijom je vremenski puno skuplji pa se do sada uglavnom koristio za primjene koje ne zahtijevaju izvođenje u stvarnom vremenu. Napretkom tehnologije dolazi do razvoja nove arhitekture grafičkih kartica kako bi se omogućilo stvaranje slike uz pomoć *Algoritma praćenja zrake* u stvarnom vremenu. Arhitekturu je osmislila američka tvrtka NVIDIA Corporation, a platforma za rad postupkom praćenja zrake nazvana je *Nvidia RTX*. Platformu je moguće koristiti u sklopu programskih grafičkih sučelja *Vulkan*, *DirectX* i *Optix*. Ovaj rad usredotočen je na implementaciju Algoritma praćenja zrake uz pomoć RTX platforme kao i integraciju uz korištenje grafičkog programskog sučelja Vulkan. Uz pomoć navedene tehnologije napravljen je fizikalan prikaz reflektirajućih i refraktirajućih površina s različitim parametrima te uspoređen s primjerima iz stvarnog svijeta.

2. Korištene tehnologije i alati

Program je implementiran koristeći jezik C++, grafičko programsko sučelje *Vulkan* te tehnologiju *Nvidia RTX*. Za stvaranje prozora i dohvaćanje korisničih ulaza i događaja korištena je biblioteka *GLFW* koja pruža jednostavnu integraciju s Vulkanom.

2.1. Vulkan

Vulkan je grafičko programsko sučelje prilagođeno modernim grafičkim karticama koje je u mogućnosti na vrlo efikasan način iskoristiti resurse koje pruža moderna grafička oprema, za razliku od već zastarjelog sučelja OpenGL. Razvijen je od strane neprofitne organizacije Khronos group s ciljem poboljšanja performansi grafičkih aplikacija. Otpočetka je zamišljen kao iznimno verbozno sučelje na vrlo niskoj razini kako bi se programeru dala što veća kontrola nad resursima i omogućio visok stupanj optimizacije. Međutim, iz istih razloga Vulkan zahtjeva dobro poznavanje grafičkog protočnog sustava kao i znanje kako upravljati memorijom. Grafičko programsko sučelje Vulkan može se koristiti na različitim operacijskim sustavima kao što su *Windows*, *Linux* ili *Android*.

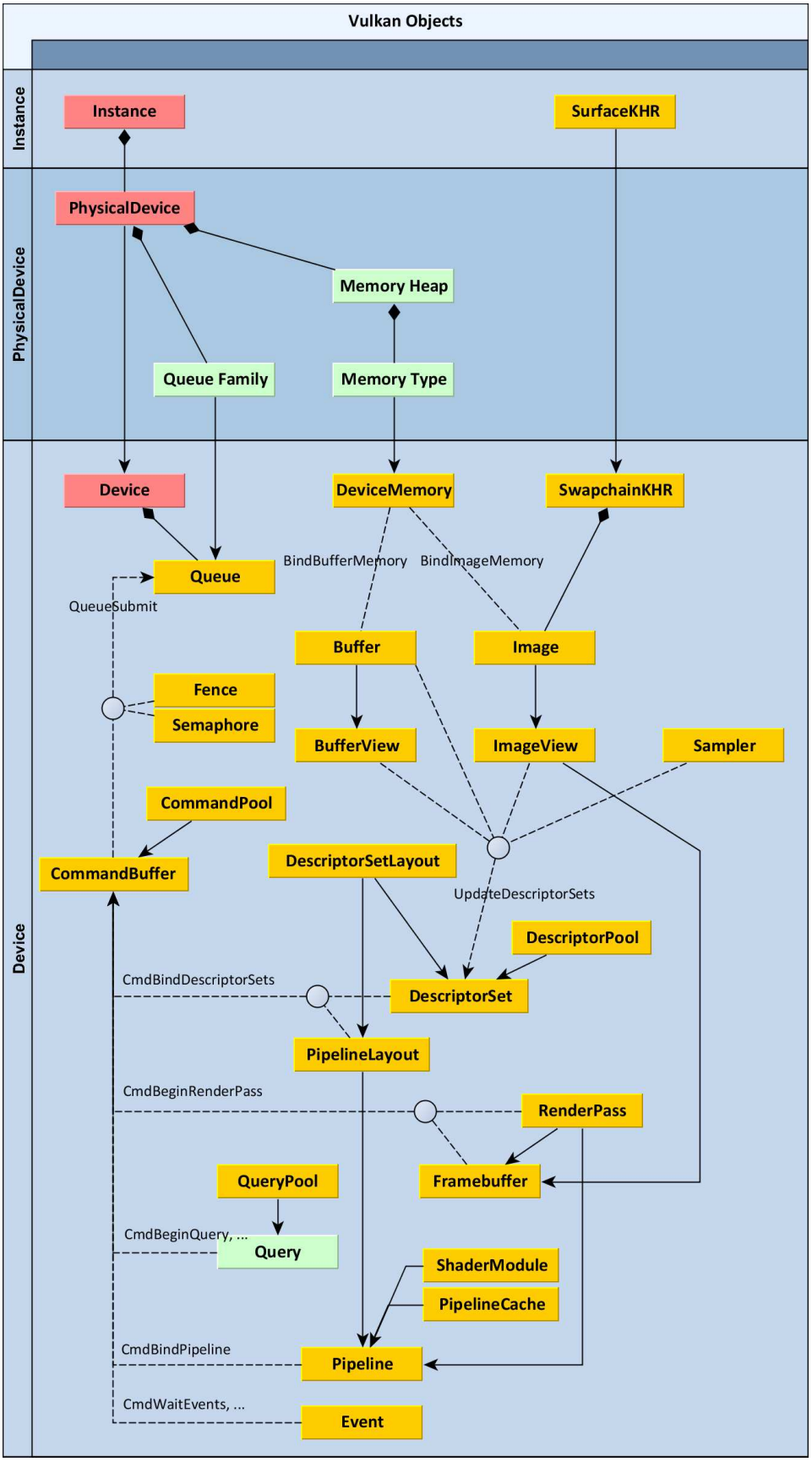
2.1.1. Arhitektura

Alokacija i pristup resursima te korištenje funkcionalnostima u Vulkanu se ostvaruju pomoću odgovarajućih objekata. Prikaz osnovnih objekata koji su sastavni dio gotovo svake aplikacije napisane u Vulkanu dan je na slici 2.1.
Vršni objekt *Instance* odnosi se na samu aplikaciju te sadrži osnovne informacije kao što su verzija i naziv.
Preko objekta *PhysicalDevice* omogućen je pristup fizičkim uređajima koji imaju mogućnost izvođenja aplikacija napisanih koristeći Vulkan. Svaki uređaj ima dobro definirane podržane funkcionalnosti i produžetke (engl. *extensions*) na temelju kojih programer može izabrati uređaj koji podržava željene funkcionalnosti aplikacije.

Vulkan preko objekta *Device* pruža sučelje prema odabranom fizičkom uređaju. Sučelje se sastoji od jednog ili više redova (engl. *queue*) u koje se pohranjuju naredbe koje se žele izvršiti na grafičkoj kartici. Svaki red se sastoji od jedne ili više familija (engl. *queue family*). Svaka familija predstavlja skup naredbi vezan uz određenu funkcionalnost, npr. prikaz slike na ekranu *Present queue family* ili rasterizacija objekata pomoću grafičkog protočnog sustava *Graphics queue family*. Naredbe se stvaraju i pohranjuju uz pomoć objekta *CommandBuffer* koji se instancira pomoću strukture *CommandPool*. Vulkan na ovaj način nastoji maksimalno učinkovito iskoristiti memoriju za stvaranje i brisanje pojedinih naredbi pošto to ovisi o grafičkoj opremi i tipu grafičke kartice. Naredbe poslane grafičkoj kartici preko sučelja *Device* izvršavaju se asinkrono neovisno o tome pripadaju li istom ili različitim redovima te je programer zadužen za njihovu ispravnu sinkronizaciju.
SwapChain objekt predstavlja polje slika koje se izmjenjuju na zaslonu. Resursi ovog objekta su u potpunosti u vlasništvu Vulkana dok aplikacija može zatražiti sliku na koju želi iscrtati željeni prikaz. Dohvaćanje slike također je asinkroni proces stoga ostale naredbe vezane uz iscrtavanje treba sinkronizirati.
Grafički protočni sustav potrebno je stvoriti od nule, a rezultat je objekt tipa *Pipeline*. Proces se započinje stvaranjem objekta *PipelineLayout* pomoću kojeg se definira raspored parametara u memoriji koji se prenose sjenčarima (engl. *Shaders*). Idući korak je definiranje raznih parametara vezanih uz pojedine faze grafičkog protočnog sustava kao što stu:

- broj i vrsta sjenčara,
- izvodiv kod sjenčara,
- uklanjanje poligona: može imati vrijednost isključeno, uklanjanje stražnjih poligona ili uklanjanje prednjih poligona,
- način zapisa atributa vrhova (engl. *vertex attributes*).

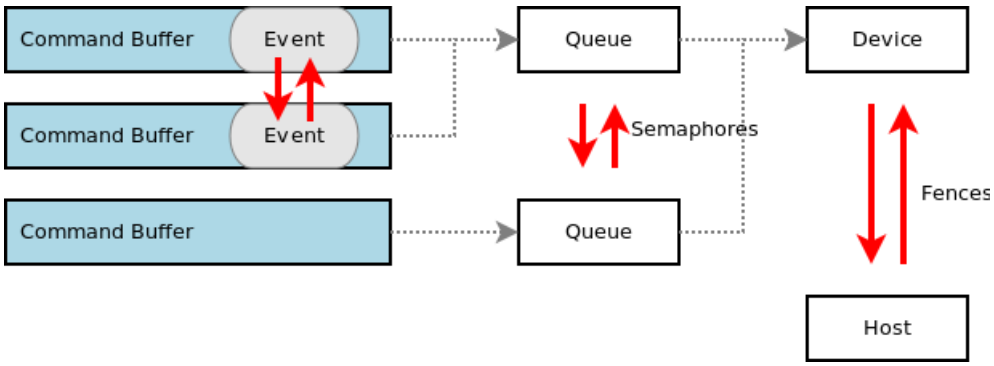
Parametri se definiraju odgovarajućim strukturama te se pomoću njih stvara objekt tipa *Pipeline*.
Resursi se sjenčarima prosljeđuju pomoću opisnika (engl. *Descriptor*). Opisnici su dio objekta tipa *DescriptorSet* koji predstavlja skup opisnika. Memorija za pohranu opisnika rezervira se na sličan način kao i memorija za pohranu naredbi, odnosno uz pomoć odgovarajućeg bazena resursa, u ovom slučaju pomoću strukture *DescriptorPool*. Prije stvaranja objekta tipa *DescriptorPool* potrebno je definirati raspored parametara u memoriji koje opisnici prenose, kako bi Vulkan mogao maksimalno optimizirati stvaranje i uništavanje resursa. To se ostvaruje strukturom *DescriptorSetLayout*.



Slika 2.1: Arhitektura grafičkog programskog sučelja Vulkan

2.1.2. Sinkronizacija

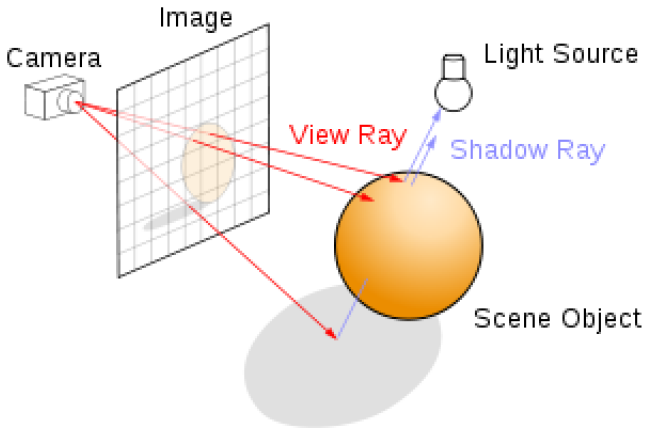
Tri osnovna mehanizma sinkronizacije u Vulkanu su semafori (engl. *Semaphores*), ograde (engl. *Fences*) i događaji (engl. *Events*). Na slici 2.2 dan je njihov prikaz. Ograde se koriste prilikom sinkronizacije procesora i grafičke kartice, npr. prilikom učitavanja resursa u memoriju grafičke kartice. Semafori služe za sinkronizaciju naredbi između različitih redova, dok događaji omogućavaju sinkronizaciju naredbi unutar istog reda. Osim spomenutih mehanizama postoje još i barijere koje omogućavaju vrlo preciznu sinkronizaciju više pristupa istom dijelu memorije, te sinkronizaciju između različitih faza grafičkog protočnog sustava.



Slika 2.2: Prikaz sinkronizacijskih objekata

2.2. Nvidia RTX

Nvidia RTX je razvojna platforma kojoj je glavni cilj omogućiti stvaranje slike *Algoritmom praćenja zrake* u stvarnom vremenu. Kao što je već spomenuto u uvodu glavna ideja je za svaki slikovni element na ekranu izračunati pripadnu zraku, odrediti najbliže sjecište s objektom ako postoji, pratiti zrake iz sjecišta prema svim izvorima te na temelju parametara lokalnog osvjetljenja odrediti boju slikovnog elementa, što je prikazano na slici 2.3.



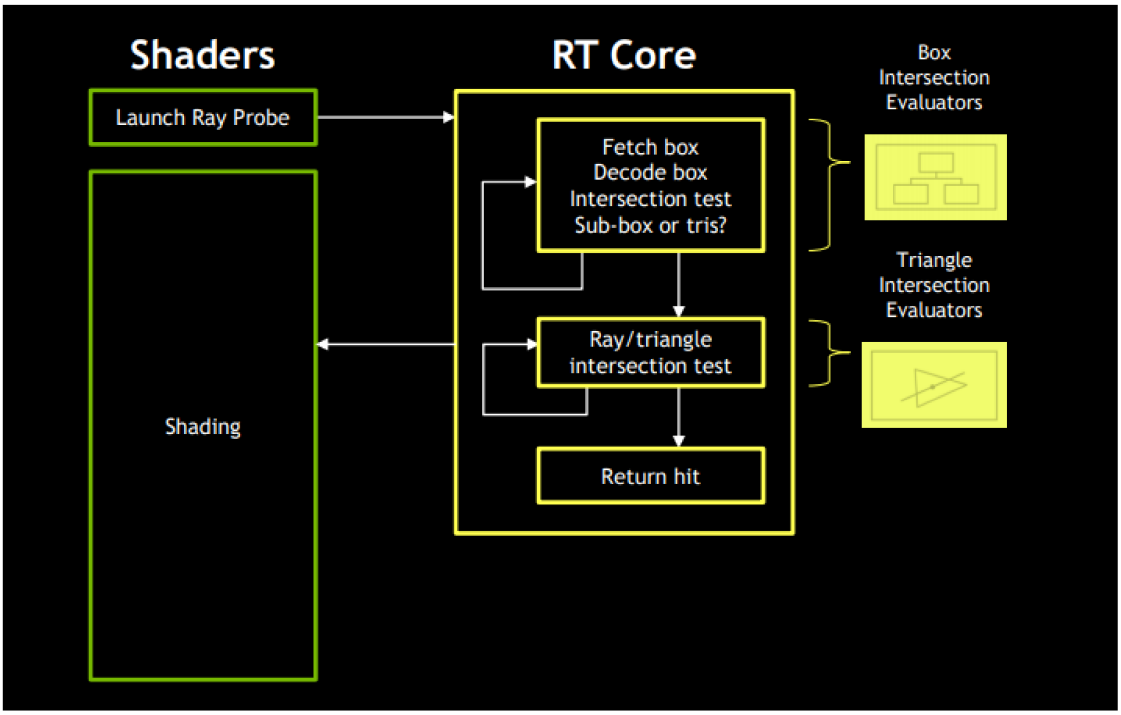
Slika 2.3: Algoritam praćenja zrake

Kako su danas zasloni vrlo visoke rezolucije potrebno je pratiti veliki broj zraka, a pošto se u sceni može nalaziti puno objekata nalaženje najbližeg sjecišta postaje poprilično skupo. Ovo naravno nepovoljno utječe na brzinu izvođenja aplikacije te donedavno ovaj postupak nije bilo moguće provoditi u stvarnom vremenu. *Nvidia RTX* platforma nastoji na brojne načine optimizirati algoritam te sklopovski ubrzati proces stvaranja i praćenja zrake. Uvodi se novi tip jezgri na grafičkoj kartici koje se nazivaju *RT jezgre* (engl. *RT cores*), a koje primarno služe za paralelno praćenje više zraka u sceni. Svaka jezgra u mogućnosti je za danu zraku i definirane objekte u sceni izračunati sjecište. Pokazuje se da ova sklopovska optimizacija ubrzava izvođenje algoritma praćenja zrake do deset puta.

2.2.1. Pohrana objekata u sceni

Prilikom iscrtavanja slike postupkom praćenja zrake potrebno je unaprijed poznavati sve objekte u sceni i učitati ih u memoriju grafičke kartice. Jedna mogućnost pohrane objekata je jednostavna lista. Međutim ovakav način pohrane rezultira izrazito skupim

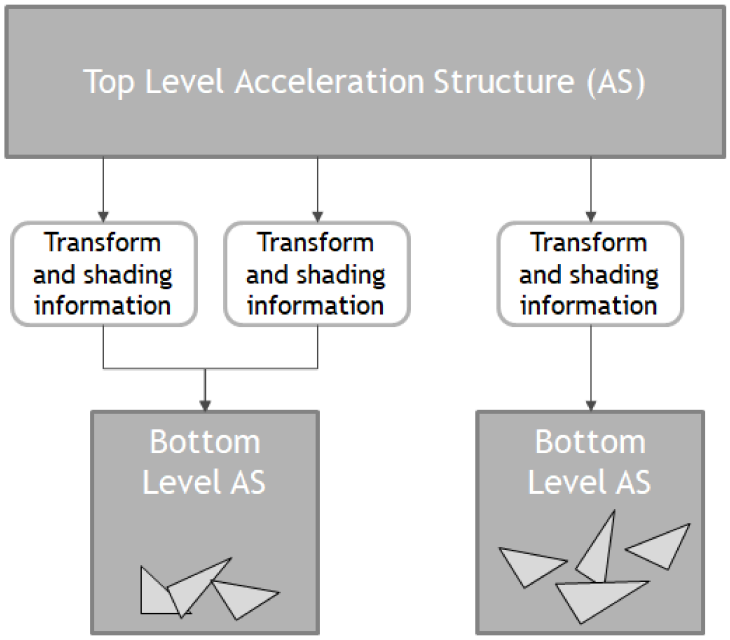
pronalaženjem najbližeg sjecišta. Svaki put kada bismo tražili sjecište bilo bi potrebno proći kroz cijelu listu što može biti poprilično skupo ako imamo puno objekata u sceni. *Nvidia RTX* platforma stoga koristi optimzacijsku strategiju grupiranja objekata u veće grupe. Svaki objekt reprezentira se svojim obuhvaćajućim volumenom koji je obično jednostavnog oblika, najčešće kvadar, te se prije pronalaženja sjecišta zrake s objektom pronalazi sjecište zrake s pripadnim obuhvaćajućim volumenom. Pronalaženje sjecišta s obuhvaćajućim volumenom znatno je jednostavnija operacija od pronalaženja sjecišta s objektom. Ukoliko sjecište s obuhvaćajućim volumenom ne postoji objekt se preskaće. Lako je uočiti da je ovo znatna optimizacija u odnosu na početni način ispitivanja pošto sjecište najćešće neće postojati. Slijedeći korak optimizacije intiutivno proizlazi iz prethodnog, a to je grupiranje više objekata u veći obuhvaćajući volumen, čime se mogu preskakati grupe objekata ukoliko sjecište ne postoji. Ovim načinom nastaje struktura podataka koja se zove hijerarhija obuhvaćajućih volumena BVH (engl. *Bounding Volume Hierarchy*). Općenit algoritam koji provodi *Nvidia RTX* platforma prikazan je na slici 2.4.



Slika 2.4: Postupak praćenja zrake pomoću platforme Nvidia RTX

Nvidia RTX platforma za pohranu objekata koristi dvije vrste akceleracijskih struktura koje su prikazane na slici 2.5. BLAS (Bottom level acceleration structure) pohranjuje vrhove objekata u lokalnom koordinatnom sustavu dok TLAS (Top level acceleration structure) sadrži pokazivače na BLAS akceleracijske strukture koje mu pripadaju

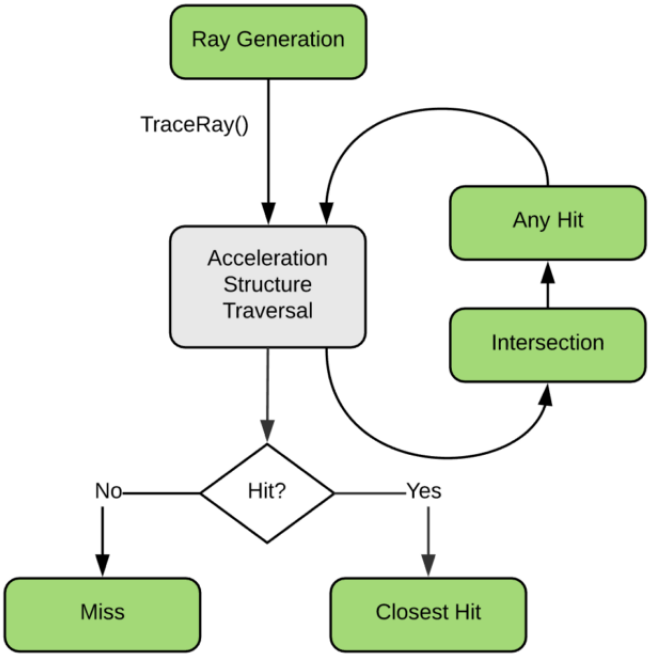
te informacije o njihovom položaju u globalnom koordinatnom sustavu. Poželjno je da se objekti grupiraju u BLAS akceleracijske stukture s obzirom na prostorni lokalitet. Ovakav način pohrane osim već spomenutih prednosti omogućava jednostavno ažuriranje pozicije. Potrebno je samo ažurirati odgovarajuću transformacijsku matricu koju posjeduje TLAS akceleracijska struktura.



Slika 2.5: Akceleracijske strukture

2.2.2. Protočni sustav

Rasterizacijom se unaprijed zna koji objekt se iscrtava te se prije njegovog iscrtavanja mogu učitati odgovarajući parametri i sjenčari specifični za taj objekt. Kod postupka iscrtavanja algoritmom praćenja zrake to nije moguće. Ne možemo unaprijed znati koji objekt će zraka pogoditi i učitati odgovarajući sjenčar pa se zbog toga protočni sustav kod ovog postupka razlikuje od klasičnog. Prije iscrtavanja slike potrebno je učitati sve objekte i sjenčare koji će se koristiti. Faze protočnog sustava prikazane su na slici 2.6. Zeleno obojano prikazane su faze koje se izvršavaju u odgovarajućim sjenčarima.



Slika 2.6: Protočni sustav Nvidia RTX platforme

2.2.3. Sjenčari

Vrste sjenčara koje se koriste kod postupka praćenja zrake u okviru *Nvidia RTX* platforme su:

- Sjenčar za generiranje zraka (*Ray generation shader*)
- Sjenčar za presjek (*Intersection shader*)
- Sjenčar za bilo kakav sudar (*Any hit shader*)
- Sjenčar za najbliži sudar (*Closest hit shader*)
- Sjenčar za promašaj (*Miss Shader*)

Ray generation shader je početna faza protočnog sustava. Poziva se za svaki slikovni element na ekranu i tipično se koristi za izračunavanje i slanje zrake iz očišta kroz slikovni element. Kako bi se mogla pozvati odgovarajuća funkcija za praćenje zrake potrebno je definirati smjer i izvor zrake u globalnom koordinatnom sustavu. Izvor zrake jednostavno se može odrediti translacijom i rotacijom ishodista inverznom matricom pogleda.

```
vec4 origin = camera.viewInverse * vec4(0, 0, 0, 1)
```

Za izračun vektora smjera zrake najprije je potrebno normirati poziciju slikovnog elementa na raspon [-1, 1].

```
vec2 pos2D = vec2(pixel.x / screenWidth, pixel.y / screenHeight)
pos2D *= 2
pos2D -= 1
```

Zatim se množenjem inverzom matrice projekcije dobije pozicija s obzirom na ishodište. Nakon toga potrebno je još samo zarotirati dobiven vektor pomoću inverza matrice pogleda te je konačan rezultat vektor smjera zrake.

```
vec4 target = camera.projectionInverse * vec4(pos2D.x, pos2D.y, 1, 1)
vec4 direction = camera.viewInverse * vec4(normalize(target.xyz), 0)
```

Intersection shader idući je korak protočnog sustava. Zadaća ovog sjenčara je pronalazak potencijalnih presjeka zrake sa scenom. Ovisno o tipu geometrije implementacija ovog sjenčara će se razlikovati. U okviru *Nvidia RTX* platforme za trokute već postoji ugrađen optimalan sjenčar ovog tipa.

Any hit shader poziva se za svaki sudar zrake sa scenom prilikom traženja najbližeg sudara. Jednostavan sjenčar ovog tipa također je dio *Nvidia RTX* platforme.

Closest hit shader poziva se prilikom pronalaženja najbližeg objekta koji se nalazi na putu promatrane zrake, ako takav objekt postoji. Tipično se koristi za izračunavanje osvjetljenja. Ova vrsta sjenčara se poziva ovisno o pronađenom objektu kako bi se ostvarila fleksibilnost i mogućnost pozivanja različitih sjenčara za različite objekte ukoliko je to potrebno.

Miss Shader poziva se ukoliko zraka ne sječe niti jedan objekt u sceni. Tipično se koristi za prikaz iluzije neba (engl. *skybox*) tako što se na temelju zrake izračuna odgovarajuća koordinata na teksturi neba i odredi boja.

3. Blinn-Phongov model osvjetljenja

U računalnoj grafici uobičajeno je svjetlost rastaviti na komponente te odvojeno izračunati utjecaj svake od komponenti a zatim i njihovu interakciju. Primarne komponente kojima se svjetlost opisuje su ambijentalna, difuzna i zrcalna. Postoji više modela osvjetljenja koji na različite načine računaju utjecaj i interakciju ovih komponenti, a u ovom radu opisan je Blinn-Phongov model.

3.1. Ambijentalna komponenta

Ambijentalna komponenta rezultat je interakcije svjetlosti sa svim objektima u sceni. Pretpostavlja se da je reflektirana svjetlost koja dolazi do neke površine s drugih površina konstantnog iznosa. U stvarnom svijetu ta pretpostavka naravno ne mora vrijediti, ali se pokazalo da dovoljno dobro opisuje stvarno ponašanje svjetlosti. Prema Blinn-Phongovom modelu osvjetljenja ambijentalna komponenta računa se prema izrazu (3.1).

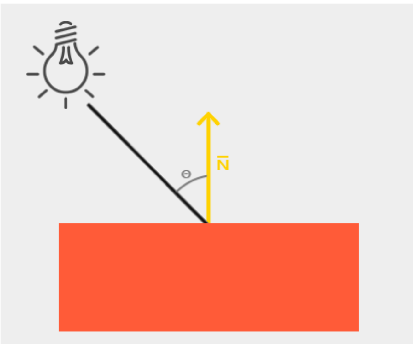
$$I_g = k_a \cdot I_a. \tag{3.1}$$

Koeficijent k_a nalazi se u rasponu $[0, 1]$ i određuje koliko svjetlosti I_a neka površina apsorbira. Ova komponenta osigurava da su objekti vidljivi iako se možda nalaze u potpunom mraku. Rezultat je prikazan na slici 3.1.

Slika 3.1: Ambijentalna komponenta

3.2. Difuzna komponenta

Nakon što zraka svjetlosti dođe do objekta dio svjetlosti se reflektira. Reflektirana boja koju vidimo ovisi o svojstvima podloge i kutu upada zrake. To je aproksimirano upravo difuznom komponentom. Iz fizike je poznato da je intezitet reflektirane svjetlosti jači što je zraka svjetlosti okomitija na površinu. Ta se ovisnost kod Blinn-Phongovog modela aproksimira kosinusom kuta između normale u promatranoj točki tijela i zrake svjetlosti, kao što je prikazano na slici 3.2.



Slika 3.2: Promatrani kut kod difuzne komponente

Formula za difuznu komponentu glasi:

$$I_d = I_i \cdot k_d \cdot cos(\theta). \tag{3.2}$$

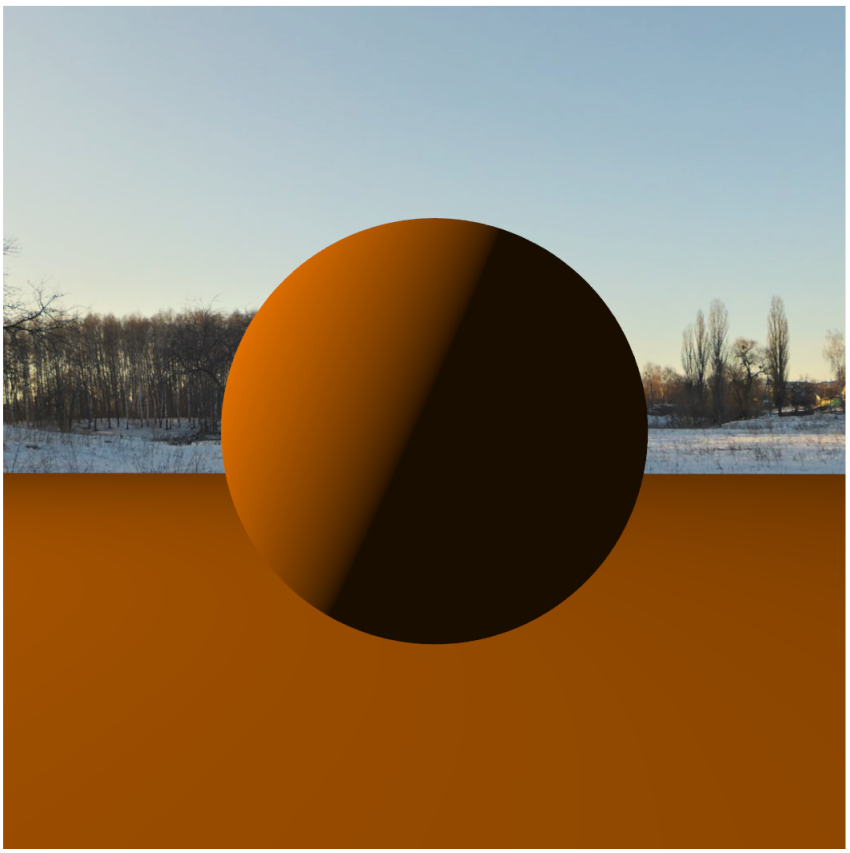
I_i predstavlja intezitet točkastog izvora, a k_d je empirijski koeficijent refleksije koji ovisi o valnoj duljini svjetlosti. Kosinus kuta računa se preko skalarnog produkta vektora smjera upadne zrake i normale u promatranoj točki $\vec{l} \cdot \vec{n}$. Međutim, moguće je da je izvor svjetlost tako postavljen da kosinus kuta postane negativan. U tom slučaju za iznos difuzne komponente uzima se vrijednost 0. Konačni izraz za izračun difuzne komponente onda glasi:

$$I_d = I_i \cdot k_d \cdot max(0, \vec{l} \cdot \vec{n}). \tag{3.3}$$

Ukoliko u sceni postoji više izvora difuzna komponenta u promatranoj točki se akumulira prema izrazu:

$$I_d = k_d \cdot \sum_m I_{i,m} \cdot max(0, \vec{l} \cdot \vec{n}). \tag{3.4}$$

Rezultat dodavanja difuzne komponente na ambijentalnu prikazan je na slici 3.3.



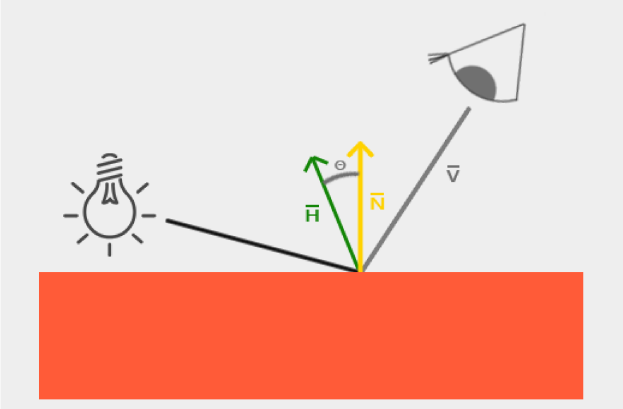
Slika 3.3: Difuzna i ambijentalna komponenta

3.3. Zrcalna komponenta

Zrcalna komponenta naglašava dijelove objekta gdje se svjetlost iz izvora savršeno reflektira prema promatraču. Kod Blinn-Phongovog modela ova komponenta ovisi o normali u promatranoj točki $\vec{n}$, vektoru prema promatraču $\vec{v}$ kao i vektoru prema izvoru $\vec{l}$. Što je kut između vektora reflektirane svjetlosti iz izvora i vektora prema promatraču manji to će intezitet ove komponente biti jači. Kod aproksimacije tog utjecaja koristi se vektor koji se nalazi točno na sredini između vektora prema promatraču i vektora prema izvoru, a računa se prema izrazu:.

$$\vec{h} = \frac{\vec{l} + \vec{v}}{||\vec{l} + \vec{v}||}. \tag{3.5}$$

Zrcalna komponenta poprima najjači intezitet ukoliko se vektor $\vec{h}$ i vektor normale $\vec{n}$ poklapaju, a sve slabiji što je kut među njima veći. To se aproksimira kosinusom kuta između vektora $\vec{h}$ i normale $\vec{n}$, koji je prikazan na slici 3.4. Ukoliko su vektori normalizirani kosinus se može izračunati njihovim skalarnim produktom $\vec{h} \cdot \vec{n}$.

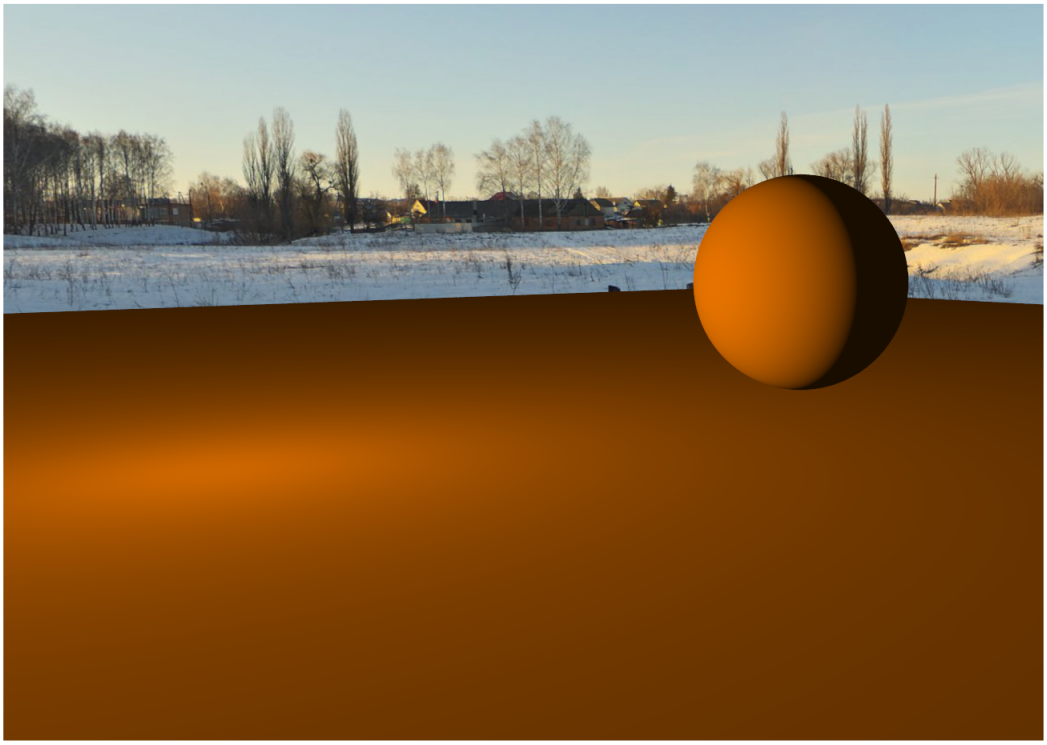


Slika 3.4: Promatrani kut kod zrcalne komponente

Konačan izraz za izračun zrcalne komponente glasi:

$$I_s = I_i \cdot k_s \cdot cos(\theta) = I_i \cdot k_s \cdot \vec{h} \cdot \vec{n}. \tag{3.6}$$

I_i predstavlja intezitet točkastog izvora, dok je k_s parametar koji opisuje svojstva materijala. Konačan rezultat kada se uzmu u obzir sve 3 komponente prikazan je na slici 3.5.



Slika 3.5: Promatrani kut kod zrcalne komponente

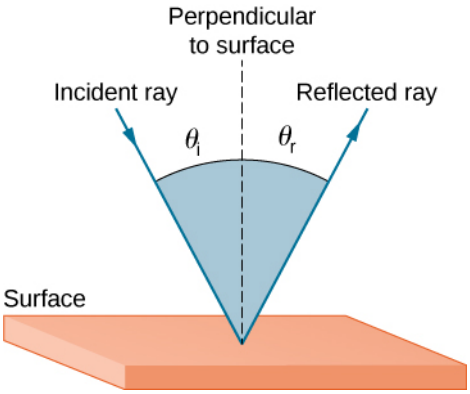
4. Fizikalni model svjetlosti

Do danas su razvijene različite interpretacije koje opisuju gibanje svjetlosti kroz medij i interakciju s površinom. Tri su osnovne fizikalne interpretacije: geometrijska optika, fizikalna optika i kvantna optika. Fizikalna optika shvaća svjetlost kao širenje progresivnog, transferzalnog elektromagnetskog vala, kvantna optika otkriva čestičnu prirodu svjetlosti temeljenu na česticama zvanim fotonima, dok geometrijska optika predstavlja svjetlost skupom svjetlosnih zraka. Algoritam praćenja zrake temelji se na trećoj interpretaciji svjetlosti, odnosno na geometrijskoj optici. Geometrijska optika zasnovana je na četiri osnovna zakona:

- pravocrtno širenje svjetlosti
- neovisnost svjetlosnih zraka
- zakon odbijanja (refleksije)
- zakon loma (refrakcije).

4.1. Refleksija svjetlosti

Zraka svjetlosti koja pada na neku ravninu reflektira se tako da je upadni kut jednak kutu refleksije, a upadna i reflektirana zraka leže u istoj ravnini kao što je prikazano na slici 4.1. Reflektirana svjetlost uvijek je manjeg inteziteta nego upadna jer dio energije apsorbira sredstvo.

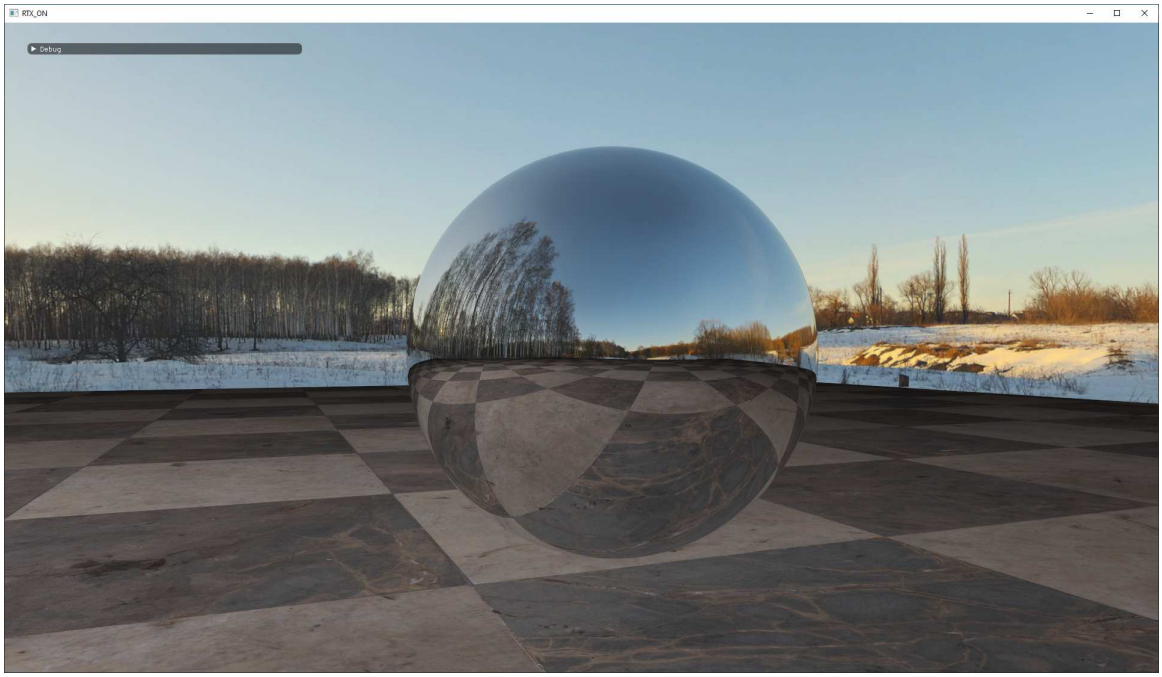


Slika 4.1: Promatrani kut kod zrcalne komponente

Poznavajući smjer upadne zrake $\vec{l}$ i normalu na površinu $\vec{n}$ može se izračunati smjer reflektirane zrake $\vec{r}$ prema izrazu:

$$\vec{r} = \vec{l} - 2(\vec{l} \cdot \vec{n})\vec{n}. \tag{4.1}$$

Na slici 4.2 prikazana je totalna refleksija na kugli.



Slika 4.2: Totalna refleksija na kugli

4.1.1. Fresnelova jednadžba

Intezitet reflektirane zrake ovisi kutu pogleda, što je on manji to je refleksivnost podloge izraženija. Fresnelova jednadžba daje vjerojatnost odbijanja svjetlosti, odnosno izračunava udio reflektirane u odnosu na refraktiranu ili apsorbiranu svjetlost. Pošto su Fresnelove jednadžbe kompliciranе u računalnoj grafici koristi se Fresnel-Schlick aproksimacija pa se vjerojatnost računa prema izrazu:

$$F = F_0 + (1 - F_0)(1 - (\vec{h} \cdot \vec{v}))^5.$$

$\vec{v}$ je vektor prema promatraču dok je $\vec{h}$ već prethodno spomenut vektor između vektora prema promatraču i vektora prema izvoru svjetlosti. F_0 predstavlja minimalnu reflekssivnost podloge ukoliko svjetlost upada pod pravim kutem, te ovisi o vrsti podloge. Parametar F_0 za staklo iznosi $(0.08, 0.08, 0.08)$, a na slici 4.3 prikazana je staklena sfera. Uz sam rub sfere može se vidjeti pojačana refleksija zbog toga što je kut pogleda blizu 0, dok je refleksija najmanje izražena na sredini.

Slika 4.3: Refleksija i refrakcija na staklenoj sferi uz Fresnelov koeficijent

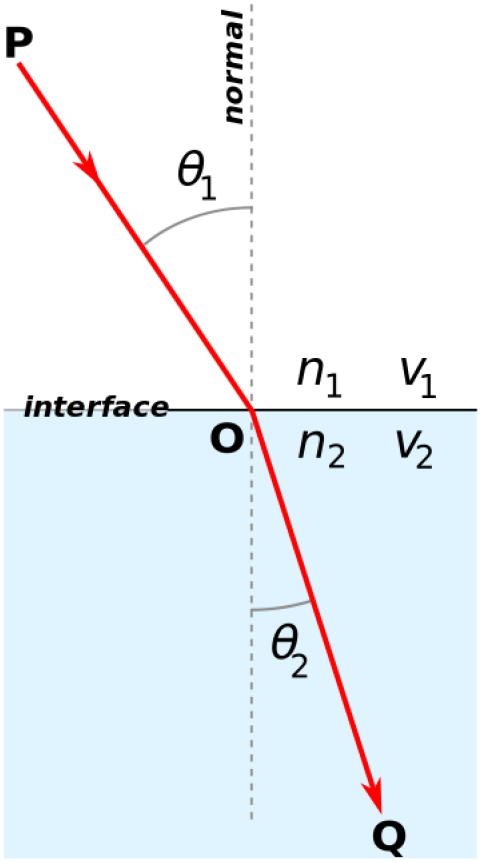
4.2. Refrakcija svjetlosti

Refrakcija ili lom svjetlosti je promjena smjera svjetlosnih zraka pri prijelazu iz jednog sredstva u drugo, kao što je prikazano na slici 4.4. Zakon refrakcije (Snellov zakon)

kaže da za kut između lomljene zrake i normale θ_2 te upadni kut zrake θ_1 vrijedi:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{v_1}{v_2} = \frac{n_2}{n_1}. \tag{4.2}$$

Parametri v_1 i v_2 odnose se na brzinu svjetlosti u sredstvima dok n_1 i n_2 predstavljaju indeks loma svjetlosti za pojedino sredstvo.



Slika 4.4: Snellov zakon

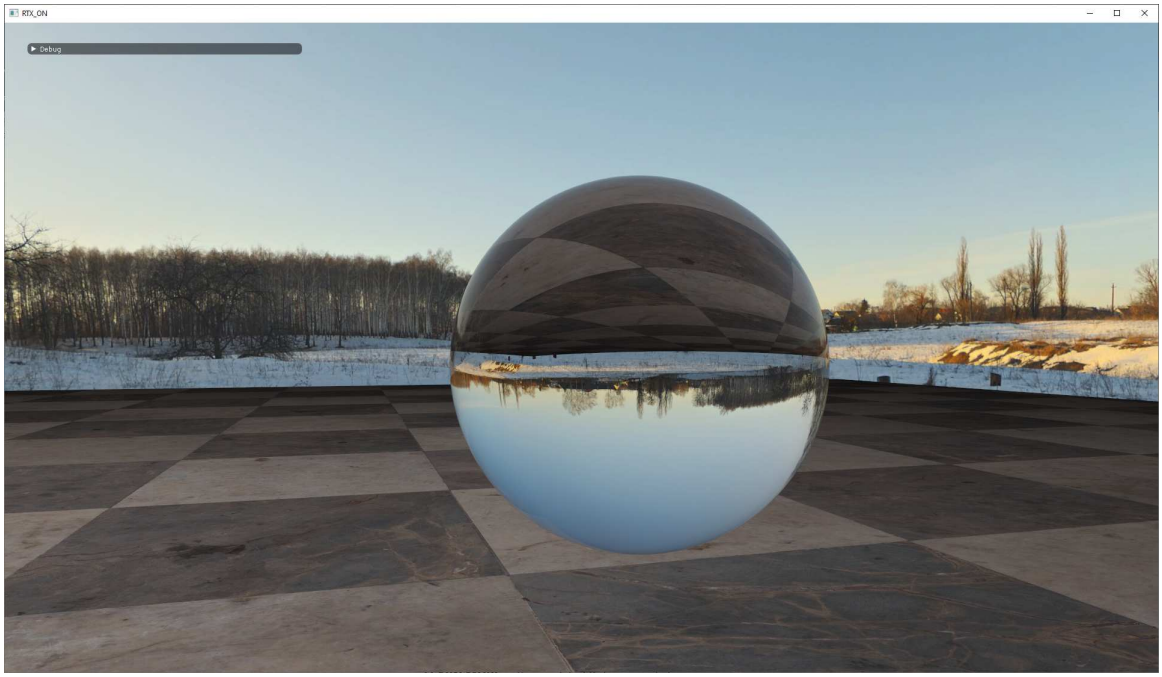
Indeks loma računa se po formuli:

$$n = \frac{c}{v} \tag{4.3}$$

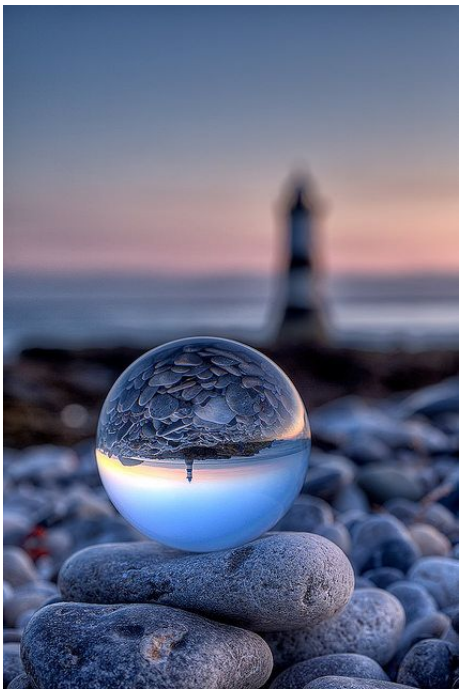
Parametar c je brzina svjetosti u vakumu, dok v predstavlja brzinu svjetlosti u pripadnom sredstvu. Lako se može zaključiti da će indeks loma uvijek biti veći od 1.

4.2.1. Refrakcija na staklenoj kugli

Prilikom refrakcije na staklenoj kugli dolazi do zanimljive pojave. Na slici 4.5 je prikazana refrakcija uz pomoć algoritma praćenja zrake i zakona geometrijske optike, dok je na slici 4.6 prikazan primjer iz stvarnog svijeta.



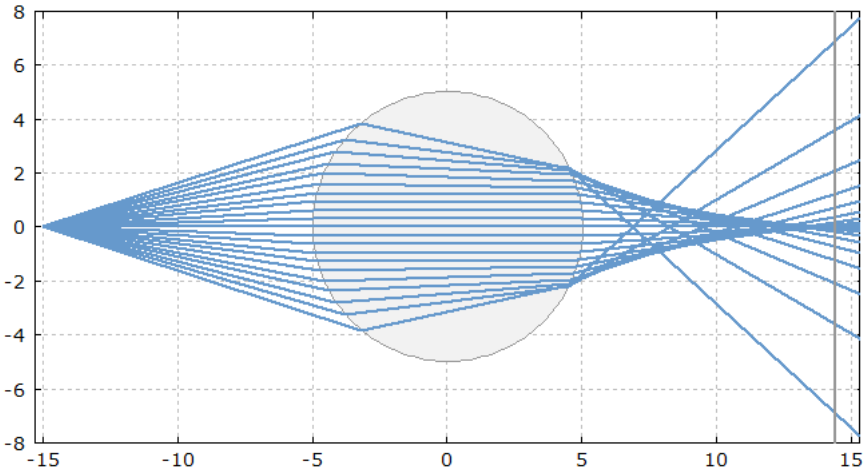
Slika 4.5: Refrakcija na staklenoj kugli uz pomoć programske implementacije



Slika 4.6: Prikaz refrakcije na staklenoj kugli iz stvarnog svijeta

Kao što se može uočiti na oba prikaza slika koja nastaje na staklenoj kugli je obrnuta. Razlog proizlazi iz Snellovog zakona i činjenice da staklo ima indeks loma veći od 1, točnije 1.517. Na slici 4.7 prikazane su zrake iz očišta i njihov put kroz staklenu kuglu. Do refrakcije dolazi prilikom ulaska svjetlosti u kuglu kao i izslaska. Ako

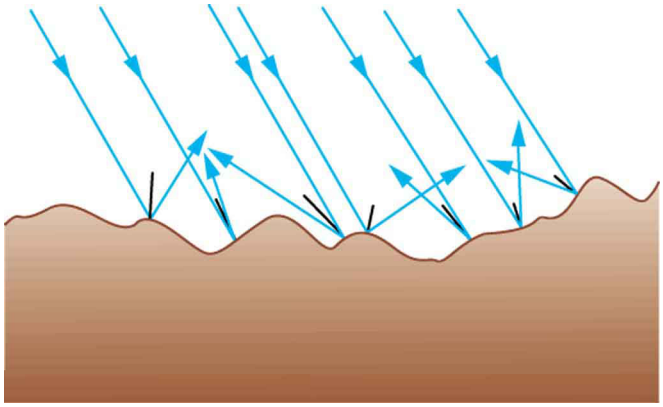
se prati jedna zraka može se uočiti da ona prilikom refrakcije na izlazu iz kugle ima suprotan smjer u odnosu na početni što je ujedno i razlog obrnute slike koja nastaje.



Slika 4.7: Prikaz zraka prilikom refrakcije na staklenoj kugli

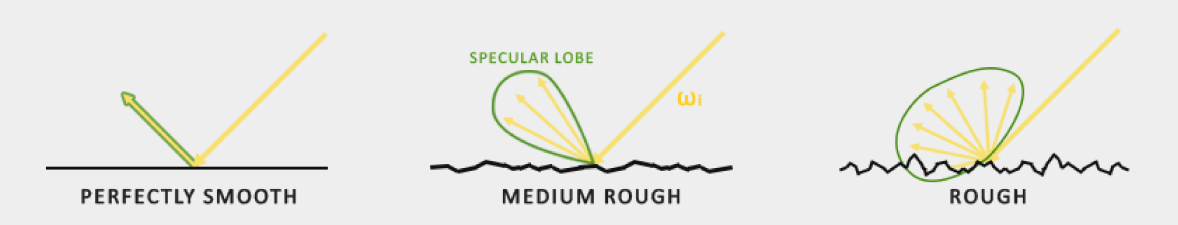
4.3. Prikaz grubih površina

Grubost površine jedno je od svojstva svake podloge. Ponašanje svjetlosti prilikom kontakta s grubim površinama razlikuje se od glatkih. Na slici 4.8 prikazano je odbijanje svjetlosnih zraka na gruboj površini.



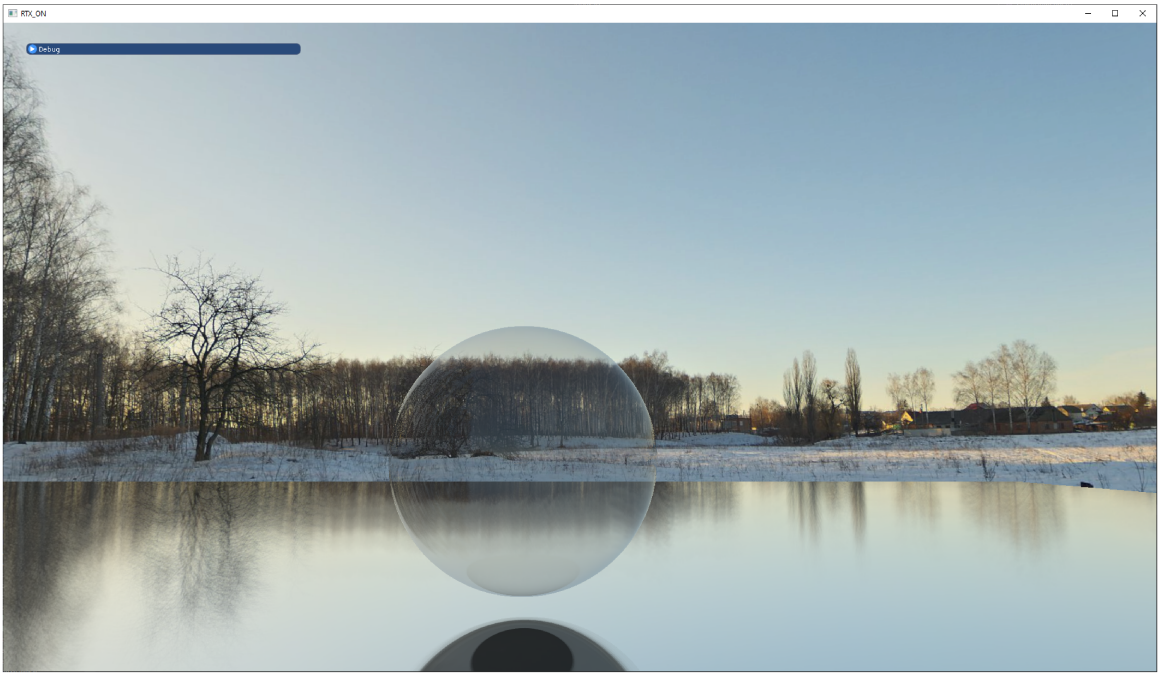
Slika 4.8: Refleksija svjetlosnih zraka na gruboj površini

Prikaz refleksija na grubim površinama algoritmom praćenja zrake temelji se na uzorkovanju većeg broja reflektiranih zraka od kojih je svaka malo zarotirana u odnosu na savršenu refleksiju na glatkoj površini. Što je površina grublja to je i raspon iz kojeg se uzimaju uzorci veći kao što je prikazano na slici 4.9.



Slika 4.9: Izračun zraka za grube površine

Pošto je broj uzoraka koji izaberemo ograničen i negativno utječe na performanse može se izabrati tek manji broj uzoraka pa je potrebno osigurati da odabrani uzorci budu uniformo raspodijeljeni kako bi se dobio realističan prikaz. Kao rezultat odbijanja svjetlosnih zraka na grubim površinama nastaju mutne refleksije kao što je prikazano na slici 4.10 uz korištenje 25 uzorka.



Slika 4.10: Prikaz refleksija na gruboj površini algoritmom praćenja zrake

5. Zaključak

Postupak praćenja zrake dugi je niz godina bio rezerviran isključivo za aplikacije koje ne zahtjevaju iscrtavanje u stvarnom vremenu. Međutim razvojem tehnologije i računalnih igara razvila se potreba za provođenjem ovog načina iscrtavanja u stvarnom vremenu. *Nvidia RTX* platforma razvijena je upravo s tim ciljem te se već koristi u brojnim igrama kao što su *Metro Exodus*, *Battlefield 5*, *Call od duty*. Unutar platforme razvijene su brojne optimizacijske strategije te je uz pomoć sklopovskog ubrzanja izvođenje čak do 10 puta brže. Međutim kompleksnost igara raste strahovito brzo te nije moguće u potpunosti se oslanjati na ovaj postupak već se algoritam praćenja zrake koristi isključivo za stvaranje učinaka koje nije moguće stvoriti rasterizacijom kao npr. dinamičke refleksije ili refrakcije. Simuliranje tih pojava uz pomoć rasterizacije porpilično je ograničeno, a nekada nije niti moguće dobiti željeni rezultat, stoga se tu pokazuje prava vrijednost algoritma praćenja zrake i potreba njegovog provođenja u stvarnom vremenu.

LITERATURA

[1] Alexander Overvoorde. Vulkan tutorial, 2020. URL https://vulkan-tutorial.com/Overview.

[2] Joey de Vries. URL https://learnopengl.com/Introduction.

[3] Željka Mihajlović Marko Čupić. *Interaktivna računalna grafika kroz primjere u OpenGL-u*. Sveučilišni udžbenik, Zagreb, 2019.

[4] NVIDIA. *Vulkan Ray Tracing Tutorial*, 2019. URL https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/.

[5] The Khronos® Vulkan Working Group. *Vulkan® 1.2.141 - A Specification*, 2020. URL https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html.

[6] The Khronos® Vulkan Working Group. *Ray Tracing In Vulkan*, 2020. URL https://www.khronos.org/blog/ray-tracing-in-vulkan#raytracing2a3.

Prikaz prozirnih i reflektirajućih površina

Sažetak

U sklopu ovog rada objašnjen je algoritam praćenja zrake te koje su prednosti tog postupka iscrtavanja. Objašnjena je složenost ovog postupka te potrebne optimizacije i strukture podataka kako bi se ovaj algoritam mogao izvoditi u realnom vremenu i kako je to implementirano unutar platforme Nvidia RTX. Rad daje kratak uvod u grafičko programsko sučelje Vulkan te predstavlja različite koncepte i modele globalnog osvjetljenja koji se tipično koriste uz algoritam praćenja zrake. Algoritam je implementiran koristeći već spomenuto grafičko programsko sučelje Vulkan i Nvidia RTX platformu. Kao rezultat implementacije dani su primjeri refleksije, refrakcije i refleksije na grubim površinama te uspoređeni sa stvarnim ponašanjem svjetlosti.

Ključne riječi: Algoritam praćenja zrake, Vulkan, Nvidia RTX, Blinn-Phong, refleksija, refrakcija

Transparent and Reflective Surfaces Rendering

Abstract

This paper explores raytracing algorithm and why it is used. It is explained which optimisations and data structures are required for real time raytracing and how it is solved inside Nvidia RTX platform. It gives a brief introduction to modern graphics API Vulkan and introduces different concepts and global illumination models typically used by raytracers. The raytracer is implemented using already mentioned graphics API Vulkan and Nvidia RTX platform. As part of implementation, several examples of light reflections, refractions and rough surface reflections using described algorithms are presented and compared to real light behavior.

Keywords: Raytracing algorithm, Vulkan, Nvidia RTX, Blinn-Phong, reflection, refraction