SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 131

POSTUPAK PRAĆENJA ZRAKE NA GRAFIČKOM PROCESORU

Luka Piljek

Zagreb, siječanj 2011.

Sadržaj

Uvod

Potaknuti nezasitnom potražnjom na tržištu 3D grafike u stvarnom vremenu, programirljivi GPU-ovi razvili su se u visoko paralelni, višedretveni, višejezgreni procesor s iznimnom računalnom snagom i vrlo velikom memorijskom propusnošću. Ovakvi procesori prikladni su za rješavanje problema koji se mogu postaviti kao algoritmi s paralelizmom na razini podataka (eng. *data-parallel*) tj. tako da se istim algoritmom paralelno obrađuje mnogo različitih podataka. Postoje mnoge primjene ovakvog načina obrade podataka, npr. pri iscrtavanju slike veliki skupovi elemenata slike (eng. *pixel*) ili vrhova (eng. *vertex*) mogu se obraditi paralelno tako da se svaki element obrađuje jednom dretvom. Na sličan način može biti ubrzana i obrada slike, kodiranje i dekodiranje videa, prepoznavanje uzoraka te mnogu drugi algoritmi čak i iz područja koja nisu vezana uz računalnu grafiku, od obrade signala, sortiranja i pretraživanja podataka, fizičkih simulacija do računanja u financijama i računskih modela u biologiji.

Unatoč tome, temeljna je namjena grafičkih kartica još uvijek iscrtavanje rasterizacijom i one to rade vrlo učinkovito te u realnom vremenu mogu obraditi vrlo složene scene. S druge strane metoda praćenja zrake se, iako vrlo popularna, donedavno smatrala prezahtjevnom za iscrtavanje u stvarnom vremenu [1]. Međutim, ta metoda ima mnoge prednosti. Ona relativno jednostavnim algoritmom lako postiže vizualne učinke koje je teško ili nemoguće ispravno dobiti klasičnom metodom rasterizacije kao što su lom i zrcaljenje svjetlosti.

Postavlja se pitanje mogu li se današnji programirljivi grafički procesori učinkovito iskoristiti za iscrtavanje slike postupcima temeljenima na bacanju i praćenju zrake.

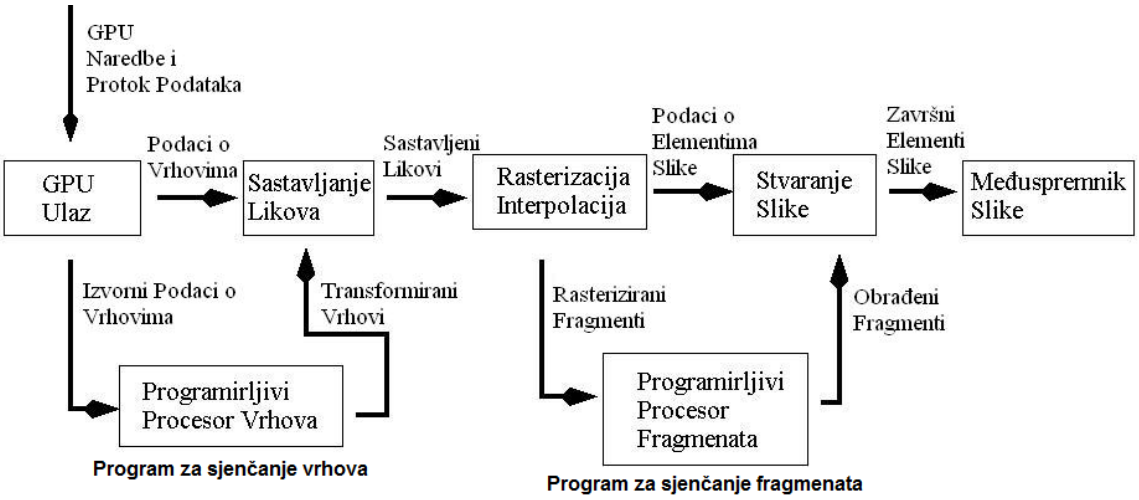
Cilj je ovog rada proučiti arhitekturu jednog takvog procesora, razraditi algoritme praćenja zrake i osnovne vizualne učinke koji se njima mogu postići te istražiti neke optimizacije pri implementaciji tih algoritama na GPU-u.

1. Arhitektura grafičkih procesora

GPU je tradicionalno imao vrlo usku namjenu i koristio se isključivo za ubrzavanje pojedinih dijelova grafičkog protočnog sustava, no kako su se zahtjevi sve više povećavali, a tehnologija sazrijevala grafičke su kartice s vremenom preuzele cijeli grafički protočni sustav.

Razvoj grafičkih kartica odvijao se u dva smjera. S jedne strane cilj je bio ubrzati obradu podataka u protočnom sustavu, dok se s druge strane proširivala funkcionalnost grafičkih kartica. Upravo je ovaj drugi cilj doveo do ugrađivanja programirljivih sklopova u pojedine dijelove protočnog sustava.

Prvotno se radilo o dvije etape protočnog sustava: etapa transformacije geometrijskih primitiva te etapa bojanja elemenata slike. Pojednostavljeni model takve arhitekture prikazan je slikom (Slika 1). Opis scene koji obično uključuje niz podataka o vrhovima (koordinate u prostoru, normale, koordinate tekstura), podatke o osvjetljenju i materijalima na sceni šalje se iz glavnog programa grafičkoj kartici koja zatim prvo koristeći program za sjenčanje vrhova (eng. *Vertex shader*) transformira i obradi podatke na razini vrhova te tako obrađene podatke nakon rasterizacije prosljeđuje programu za sjenčanje fragmenata (eng. *Fragment shader*) koji na izlazu daje završnu boju fragmenta.



Slika 1: Klasični grafički protočni sustav

Iako u početku vrlo ograničeni, ovi programirljivi sklopovi omogućili su programerima implementaciju velikog broja posebnih učinaka koji se prije nisu mogli izvesti na GPU. Daljnjim napretkom tehnologije mogućnosti jedinica za sjenčanje višestruko su se povećale što je ubrzo dovelo do razine programirljivosti koja je gotovo jednaka onoj središnjeg procesora. To se prvenstveno odnosi na podržane tipove podataka, koji uključuju tipove s pomičnim zarezom, naprednu kontrolu toka izvođenja programa te pristup RAM memoriji na grafičkoj kartici.

Sljedeći je korak bio odvojene etape protočnog sustava objediniti i zamijeniti jedinstvenom arhitekturom sačinjenom od velikog broja identičnih jezgri na kojima su se mogli izvoditi svi programi za sjenčanje bez obzira na njihov zadatak u protočnom sustavu. Iako su ovime sklopovi (jezgre) izgubili jednostavnost specifične namjene te su morali biti prilagođeni svim operacijama koje su potrebne za iscrtavanje, mnoge od tih operacija bile su zajedničke svim fazama te se tako omogućilo svojevrsno „recikliranje“ sredstava grafičkog sklopovlja te uravnoteživanje opterećenja pri obradi podataka.

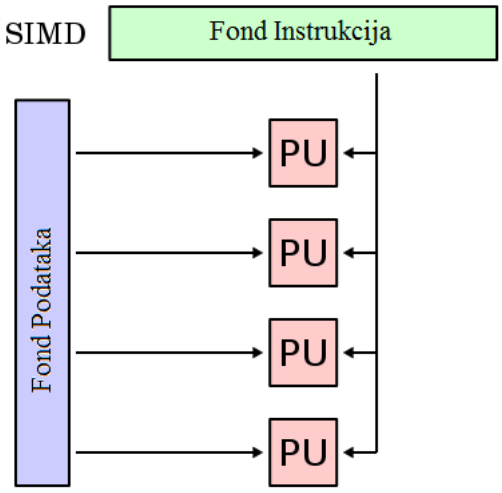
Takvi grafički procesori imaju mogućnost čitanja i pisanja u grafičku memoriju te podržavaju kontrolu toka i osnovne aritmetičke i logičke operacije nad cjelobrojnim podacima i brojevima s pomičnim zarezom. Ovo je dovelo do trenda korištenja grafičkog procesora za razvoj programske potpore koja koristi masovnu paralelnu obradu podataka za namijene izvan grafičkog protočnog sustava, tehnika nazvana GPGPU (eng. *General-purpose computing on graphics processing units*).

GPGPU primjena grafičkih kartica počela je još za vrijeme programa za sjenčanje, ali nakon prelaska na objedinjenu arhitekturu dobila je potpuno novu dimenziju. Programeri su sada mogli pisati programe u poznatim programskim jezicima (npr. C) i nisu više morali brinuti o specifičnostima grafičkog protočnog sustava i mnogim ograničenjima pojedinih programa za sjenčanje već su svim resursima grafičke kartice mogli pristupiti iz jednog mjesta koristeći API-je visoke razine.

Za razumijevanje stvarnih prednosti i nedostataka suvremenih grafičkih procesora u usporedbi sa središnjim procesorima, potrebno je razumjeti paradigmu SIMD (eng. *single instruction, multiple data*) te paralelnu obradu toka podataka (eng. *stream processing*).

Pretpostavimo da imamo skup podataka iz kojeg svaki element moramo obraditi na sličan način koristeći isti algoritam. Ovakav problem vrlo se često javlja u računalnoj grafici gdje npr. veliki broj vrhova ili elemenata slike moramo obraditi istom funkcijom. Funkcija kojom obrađujemo pojedini element naziva se jezgrena funkcija (eng. *kernel function*) i može se primijeniti paralelno na više neovisnih elementa iz toka podataka tj. koristi se paralelizam na razini podataka.

To je upravo i ideja iza paradigme SIMD na kojoj se temelji arhitektura današnjih grafičkih procesora. Takav procesor ima više manjih jedinica za obradu koje rade paralelno pa nakon dohvaćanja skupa podataka iz memorije, a može jednom instrukcijom obraditi cijeli skup (Slika 2).



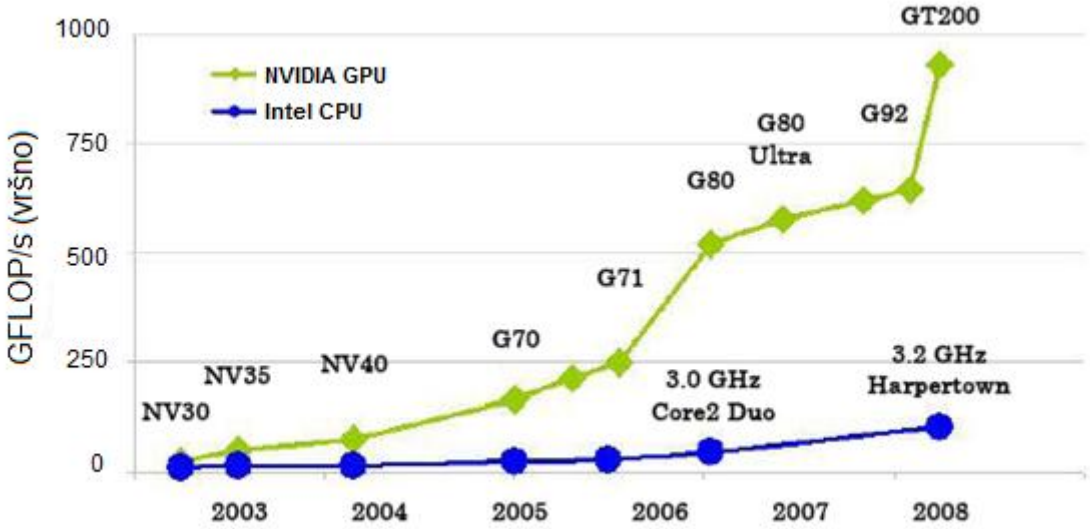
Slika 2: SIMD obrada podataka [2]

Ovakav pristup ipak nameće neka bitna ograničenja. Prije svega, paralelna obrada podataka podrazumijeva da su podaci međusobno neovisni, tj. da obrada jednog elementa ne ovisi o drugima. Nadalje, budući da se više podataka obrađuje jednom instrukcijom, slijed instrukcija treba biti jednak za sve podatke pri obradi. Konkretno to znači da kontrola toka u jezgrenoj funkciji ne bi smjela ovisiti o pojedinačnom podatku koji se obrađuje.

Također treba uzeti u obzir činjenicu da dohvaćanje većeg skupa podataka iz memorije često može biti i do nekoliko redova veličine vremenski zahtjevnije od pojedinih

instrukcija za obradu pa je u takvim slučajevima poželjno da u jezgrenoj funkciji bude veliki omjer aritmetičkih operacija u odnosu na ulazno-izlazne operacije.

Zbog paralelnog načina obrade podataka, današnji grafički procesori uvelike premašuju središnje procesore po broju mogućih operacija nad brojevima s pomičnim zarezom po sekundi (Slika 3).



Slika 3: Snaga GPU i CPU u posljednjih nekoliko godina [3]

Ovo se ipak ne može uzeti kao općenito mjerilo učinkovitosti procesora jer postizanje tih teoretskih mogućnosti u praksi ovisi o tome koliko se dobro pojedini algoritam može prilagoditi paralelnom izvođenju i ograničenjima GPU-a.

1.1. CUDA

Kao odgovor na trend korištenja grafičkih procesora u općenite svrhe računanja, Nvidia je pokrenula CUDA-u, arhitekturu i pripadajući API visoke razine kako bi razvijateljima omogućila što jednostavniji rad u poznatom razvojnom okruženju.

CUDA (eng. *Compute Unified Device Architecture*) je arhitektura grafičkih kartica za paralelno računanje koja omogućava pristup grafičkom sklopovlju i razvoj programske potpore za GPU pomoću programskog jezika C te radi na svim Nvidijinim grafičkim karticama od serije G8X [4].

CUDA uvodi neke mogućnosti koje joj daju prednost nad dotadašnjim GPGPU arhitekturama [5]. CUDA omogućava raspršeno čitanje memorije tj. čitanje iz proizvoljnih adresa u grafičkoj memoriji, uvodi potpunu potporu za operacije nad cjelobrojnim podacima i nad bitovima te implementira napredno upravljanje dretvama na grafičkom uređaju.

Dretvama koje se izvode na GPU omogućava pristup dijeljenoj memoriji koja može poslužiti kao priručna memorija čime se ostvaruje veća propusnost nego bi to bilo moguće koristeći dohvaćanje podataka iz globalne memorije, a podržan je i mehanizam sinkronizacije dretvi.

U usporedbi sa standardnim programskim jezikom C i razvojem programske potpore za klasično računalo, programsko sučelje CUDA ima i neke bitne nedostatke.

CUDA u verziji 1.x ne podržava rekurzije i pokazivače na funkcije te uvodi neka proširenja na standardni programski jezik C.

Brojevi s pomičnim zarezom dvostruke preciznosti podržani su samo na novijim serijama uz određeno odstupanje od IEEE 754 standarda. Na primjer jedini podržani tip zaokruživana decimalnog broja je zaokruživanje na najbliži cijeli broj. Brojevi s pomičnim zarezom jednostruke preciznosti također nisu po IEEE standardu pa tako npr. nisu podržane NaN (eng. *Not a Number*) signalne vrijednosti. Osim toga, propusnost sabirnice između CPU i GPU može u nekim situacijama kada je potrebno prenositi velike količine podataka iz grafičke memorije u glavnu i obratno biti usko grlo, a algoritam koji se treba

izvršavati na GPU potrebno je prvo prilagoditi paralelnom izvršavanju. Najučinkovitije djelovanje postiže se kada je ukupan broj dretvi reda veličine 1000 ili više jer se tek tada zauzimaju sva dostupna sredstva na grafičkoj kartici (zauzeće sredstava može se izračunati Nvidijinim tabličnim kalkulatorom zauzeća).

Grananje u programskom kodu može imati veliki utjecaj na učinkovitost izvođenja kada dretve imaju različiti tok izvođenja programa.

Za razliku od nekih drugih API-ja, CUDA je podržana samo na Nvidijinim grafičkim karticama.

Neki od ovih prednosti i nedostataka bit će detaljnije proučeni u narednim poglavljima.

Nvidia je nastavila s unapređivanjem ove tehnologije pa tako njihova najnovija arhitektura kodnog naziva „Fermi“ ima bolju podršku za brojeve s pomičnim zarezom dvostruke preciznosti, više CUDA jezgri (reda veličine 100), više dijeljene memorije itd.

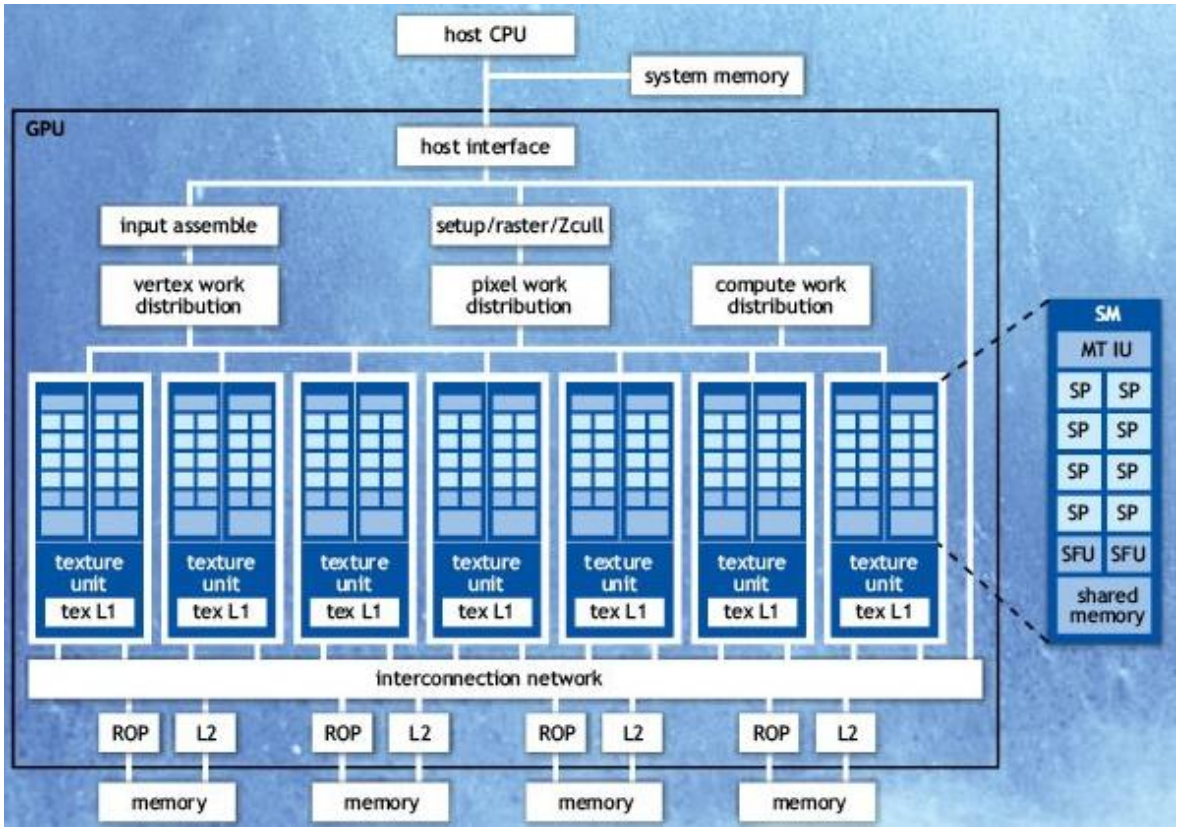
Različite grafičke kartice imaju različite mogućnosti računanja (eng. *compute capability*) te iako CUDA daje zajednički API za sve kartice s arhitekturom CUDA, neke su funkcionalnosti dostupne samo na novijem sklopovlju pa o tome treba voditi pažnju.

Opisi arhitekture CUDA u daljnjem tekstu temeljit će se na verziji arhitekture Tesla osim ako nije drugačije navedeno.

1.1.1. Sklopovska implementacija

Temelj arhitekture Tesla je podesivo polje višedretvenih višestrukih procesora toka (eng. s*treaming multiprocesor*, skraćeno SM) [5]. Na slici (Slika 4) prikazan je GPU s 14 SM-ova. Svaki SM sadrži 8 manjih jezgri procesora toka (eng. *streaming processor*, skraćeno SP), dvije jedinice za posebne funkcije (eng. *special function unit*, skraćeno SFU) te višedretvenu instrukcijsku jedinicu (eng. *multithreaded instruction unit*, skraćeno MT IU).

SM procesori također imaju skup 32-bitnih registara po svakom SP procesoru, dijeljenu memoriju (eng. *shared memory*) te priručnu memoriju za konstante i priručnu memoriju za teksture iz kojih može samo čitati.



Slika 4: Nvidia Tesla GPU [6]

SP jedinice mogu obavljati osnovne aritmetičke i logičke operacije nad cjelobrojnim brojevima te brojevima s pomičnim zarezom dok SFU jedinice služe za izvođenje nekih složenijih operacija kao što su trigonometrijske funkcije i logaritmi. MT IU se brine o dohvaćanju i izvršavanju instrukcija pojedinih skupina dretvi.

SM procesori grupirani su u parove pri čemu svaki par ima svoju jedinicu za uzorkovanje tekstura (eng. *texture unit*) koja teksture iz glavne memorije dohvaća preko L1 priručne memorije čime se ubrzava čitanje tekstura. SM procesori također mogu čitati i pisati u glavnu DRAM memoriju, ali je pristup istoj mnogo sporiji od pristupa dijeljenoj memoriji i registrima.

Kada računalo domaćin želi dodijeliti neki posao grafičkom procesoru, prvo se prenesu potrebni podaci i instrukcije za njihovu obradu iz glavne memorije na računalu u memoriju na grafičkom uređaju. Nakon što CPU pokrene izvršavanje, na GPU se istovremeno pokreće veliki broj dretvi, a jedinica za raspodjelu posla na CWD (eng. *compute work distribution*) na GPU ih dinamički raspoređuje po dostupnim SM procesorima.

Svaka od istovremeno pokrenutih dretvi na SM procesoru ima rezervirane vlastite registre u kojima se pohranjuje trenutni kontekst izvršavanja pa je prebacivanje izvršavanja s jedne dretve na drugu vrlo brzo jer nema zamijene konteksta kao što je to slučaj kod CPU-a.

Osim toga, jedan SM može koristeći sve SP jezgre istovremeno izvršiti određenu instrukciju nad cijelom skupinom dretvi (veličine 8 u ovom slučaju).

Mogu se primijetiti očite sličnosti između arhitekture SM procesora i SIMD paradigme, međutim Nvidia ovakav pristup naziva SIMT (eng. *single instruction, multiple thread*) jer SIMD model proširuje naprednim upravljanjem dretvama o čemu će više riječi biti u sljedećem poglavlju.

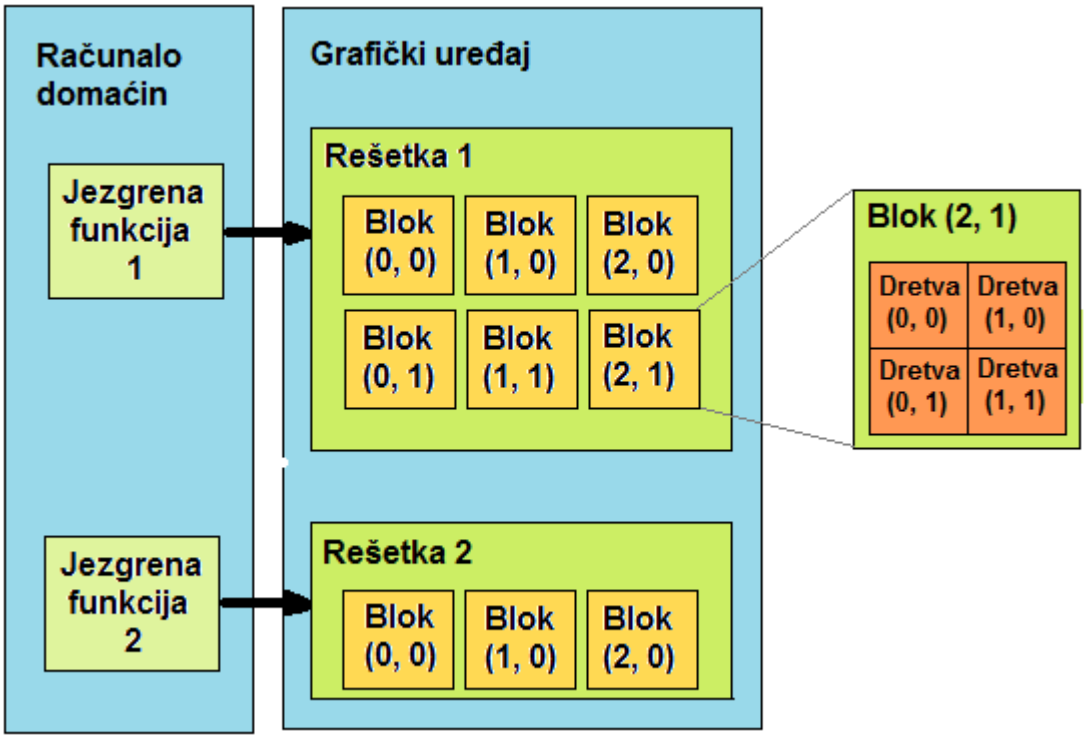
1.1.2. Logička organizacija

Kako bi se pojednostavio problem istovremenog izvršavanja velikog broja dretvi, CUDA uvodi poseban način hijerarhijskog grupiranja dretvi koji olakšava raspodjelu cjelokupnog posla unutar GPU-a.

Dretve su prije svega raspoređene u blokove. Blok je skup dretvi čija je veličina određena s jednom dvije ili tri dimenzije te unutar kojeg se dretve mogu sinkronizirati i međusobno komunicirati pomoću dijeljene memorije. Dretve unutar pojedinog bloka uvijek se izvode na istom SM procesoru, ali jednom SM procesoru može biti dodijeljeno više blokova.

Blokovi su raspoređeni u dvodimenzionalnu rešetku (eng. *grid*) koja predstavlja logičku raspodjelu jednog zadatka određenog jezgrenom funkcijom. To znači da sve dretve u blokovima unutar jedne rešetke izvršavaju istu jezgrenu funkciju. Dretve iz različitih blokova ne mogu međusobno komunicirati niti se uskladiti pri izvršavanju.

Programer mora organizirati posao po blokovima i rešetkama te u glavnom programu odrediti njihove dimenzije pri pokretanju jezgrene funkcije. GPU zatim instancira jezgrenu funkciju na rešetku paralelnih blokova dretvi. Svaka dretva unutar bloka izvršava instancu jezgrene funkcije te ima svoj ID koji označava njezinu poziciju u bloku. Blok također ima svoj ID unutar rešetke.



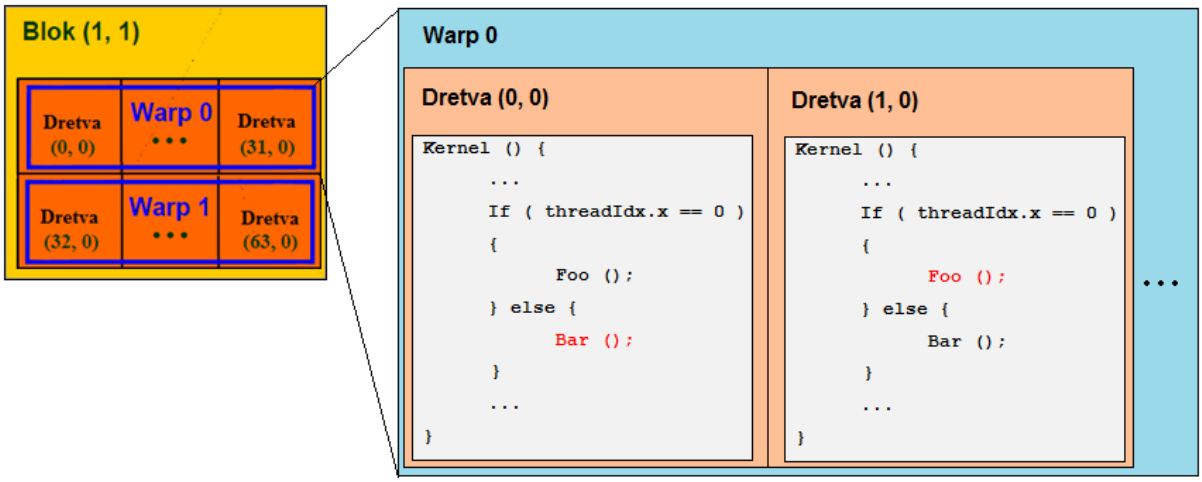
Slika 5: Logička organizacija dretvi po blokovima i rešetkama

Ovakva organizacija dretvi omogućava prilagodljivu raspodjelu poslova na GPU. Na primjer ako podijelimo rešetku na 8 blokova, grafički uređaj s dvije jezgre može svakoj dodijeliti 4 bloka, dok bi grafički uređaj s 4 jezgre svakoj mogao dodijeliti 2 bloka. Programer dakle treba samo logički organizirati posao, dok će stvarnu raspodjelu posla po jezgrama ovisno o dostupnim sredstvima obaviti GPU, preciznije CWD jedinica.

Paralelno izvršavanje i upravljanje dretvama obavlja se automatski. Stvaranjem dretvi, vremenskim upravljanjem i prekidom izvršavanja rukovodi CUDA sustav izravno na sklopovlju i programer se o tome ne mora brinuti.

Iako se to sa stajališta programera ne može vidjeti, blokovi su pri izvršavanju podijeljeni na skupine dretvi točno određene veličine (koja na trenutnim verzijama arhitekture iznosi 32 dretve) koje Nvidia naziva *warpovima*. Sve se dretve iz istog *warpa* uvijek izvode fizički paralelno što znači da u svakom dodijeljenom taktu procesora one moraju obaviti istu instrukciju i nijedna dretva ne može nastaviti s izvršavanjem dok ju sve ne obave.

Ovdje do izražaja dolazi ranije spomenuti SIMT model procesora koji omogućuje učinkovito upravljanje stotinama dretvi u izvođenju. SM procesor pridružuje svakoj dretvi jednu SP jezgru i svaka se dretva izvršava neovisno, s vlastitim stanjem registara i adresom trenutne instrukcije. SM stvara, upravlja i dodjeljuje taktove *warpovima*. Dretve iz jednog *warpa* počinju izvođenje od iste adrese u programu, ali se kasnije pri izvođenju mogu razilaziti.



Slika 6: Razilaženje dretvi

U svakom taktu, SIMT jedinica odabire *warp* koji je spreman za izvođenje te izdaje sljedeću instrukciju svim aktivnim dretvama *warpa*.

Ako različite dretve istog *warpa* zbog grananja u programskom kodu imaju različiti tok izvođenja, pojedine grane se moraju izvoditi slijedno dok se sve dretve opet ne sastanu na

istoj adresi u programu. Ta se pojava naziva razilaženje (eng. *divergence*) i od iznimnog je utjecaja na učinkovitost izvođenja paralelnih algoritama na GPU-u.

Očito je da se najveća učinkovitost postiže kada sve dretve iz istog *warpa* imaju isti slijed izvođenja.

U gornjem primjeru (Slika 6) zbog razilaženja u izvršavanju, prvo će dretva 0 obaviti funkciju `Foo()` dok će ostale dretve *warpa* čekati, a zatim će ostale dretve obaviti funkciju `Bar()` dok će dretva 0 čekati i tek će se nakon završetka uvjetnog grananja dretve opet sastati i nastaviti paralelno izvršavanje.

Pisanjem jezgrene funkcije na način da dolazi do što manjeg razilaženja dretvi može se postići mnogo bolja učinkovitost izvođenja, međutim to često nije lak zadatak.

Ovaj je problem ograničen samo na dretve unutar istog *warpa*, dok se različiti *warpovi* izvršavaju neovisno o tome izvršavaju li istu ili različitu granu programa. Paralelno izvođenje *warpova* unutar bloka je stoga samo logičko.

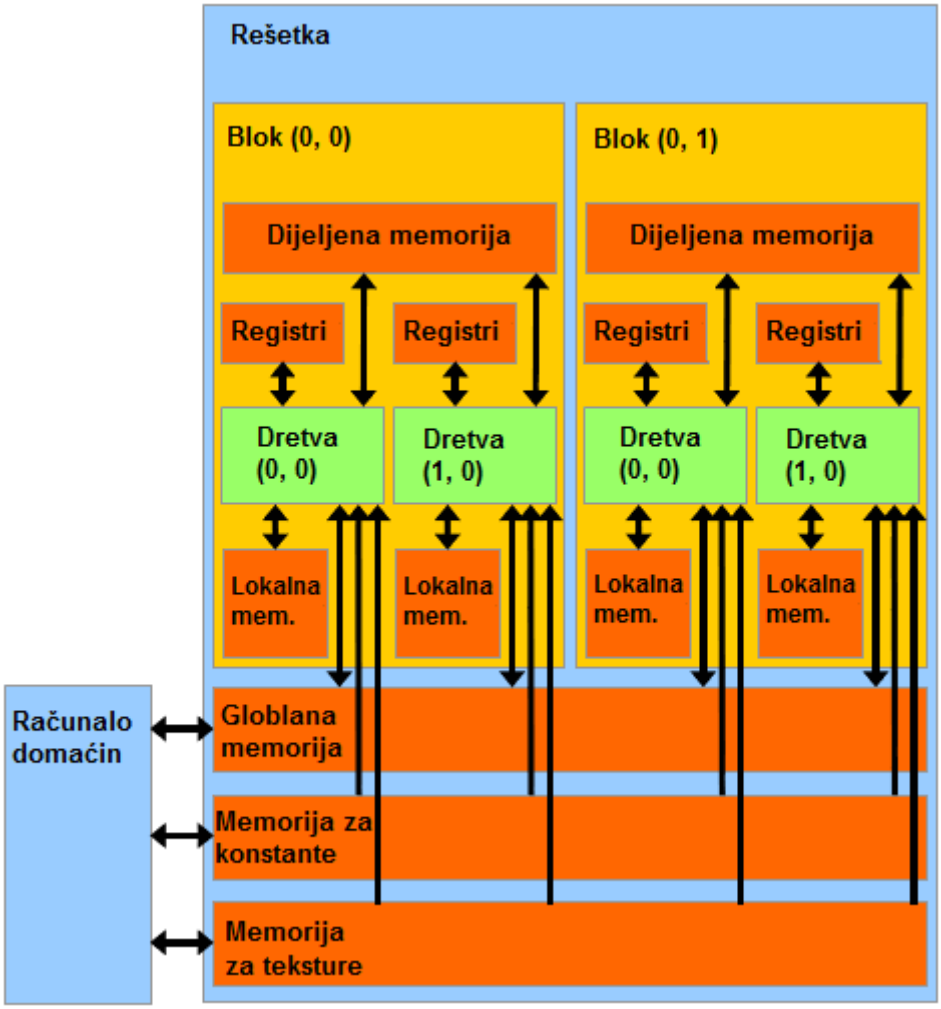
Tijekom izvršavanja, dretve mogu pristupiti različitim memorijskim prostorima. Svaka dretva na raspolaganju ima određeni broj registara, lokalnu memoriju, dijeljenu memoriju bloka te pristup memoriji za konstante, memoriji za teksture i globalnoj memoriji. Lokalna memorija koristi se za pomoćne varijable koje ne stanu u registre dretve. Dijeljena memorija bloka vidljiva je svim dretvama u bloku i obično ima mnogo manje vrijeme kašnjenja (eng. *latency*) od globalne memorije pa se koristi kao priručna memorija bloka te može poslužiti za ubrzavanje izvršavanja i učinkovitu komunikaciju među dretvama bloka.

Dretve u pojedinom bloku mogu se sinkronizirati pozivom ugrađenih funkcija za sinkronizaciju čime se osigurava da nijedna dretva neće nastaviti s izvršavanjem dok sve dretve nisu došle do sinkronizacijske granice. Sinkronizacija je neophodna pri korištenju dijeljene memorije. Nakon prolaza sinkronizacijske granice, sve dretve mogu u dijeljenoj memoriji bloka vidjeti memorijske zapise ostalih dretvi bloka koje su napravljene prije sinkronizacije i na taj način mogu međusobno komunicirati.

Globalna memorija zajednička je dretvama svih blokova u rešetki i koristi se za pribavljanje ulaznih podataka i zapisivanje krajnjih rezultata.

Memorijski prostor za konstante, teksture te lokalna i globalna memorija fizički se nalaze u DRAM memoriji na grafičkom uređaju, ali konstantama i teksturama se pristupa preko priručne memorije te se tako ubrzava njihovo dohvaćanje.

Dretve koje se izvode na GPU iz memorija za konstante i teksture mogu samo čitati dok u ostale memorije mogu i pisati.



Slika 7: Logička organizacija memorije na grafičkom uređaju [7]

Računalo domaćin može pomoću programskog sučelja čitati i pisati u globalnu memoriju te memorije za konstante i teksture.

2. Programsko sučelje (API)

Pri razvoju programske potpore, grafičkom uređaju temeljenom na arhitekturi CUDA može se pristupiti pomoću nekoliko različitih programskih sučelja (eng. *Application Programming Interface*, skraćeno API).

Jedan od mogućih izbora je Microsoftov Direct3D koji od verzije 11 uvodi novi *DirectCompute* API te time službeno podržava GPGPU programiranje na operativnim sustavima Windows Vista i Windows 7. Iako je prvi put predstavljan u specifikacijama DX11 verzije API-ja, *DirectCompute* podržan je i na DX10 kompatibilnom grafičkom sklopovlju.

Druga popularna platforma za GPGPU razvoj je OpenCL (Open Computing Language), konkurencija otvorenog standarda (analogno OpenGL i OpenAL) koju razvija grupa Khronos. OpenCL omogućava razvoj programa koji se mogu izvršavati paralelno na različitim CPU i GPU sustavima. Uključuje jezik temeljen na C-u koji se koristi za pisanje jezgrenih funkcija i API za definiranje i kontrolu platformi. Podržava paralelno računanje koristeći paralelizam temeljen na podacima i na zadacima.

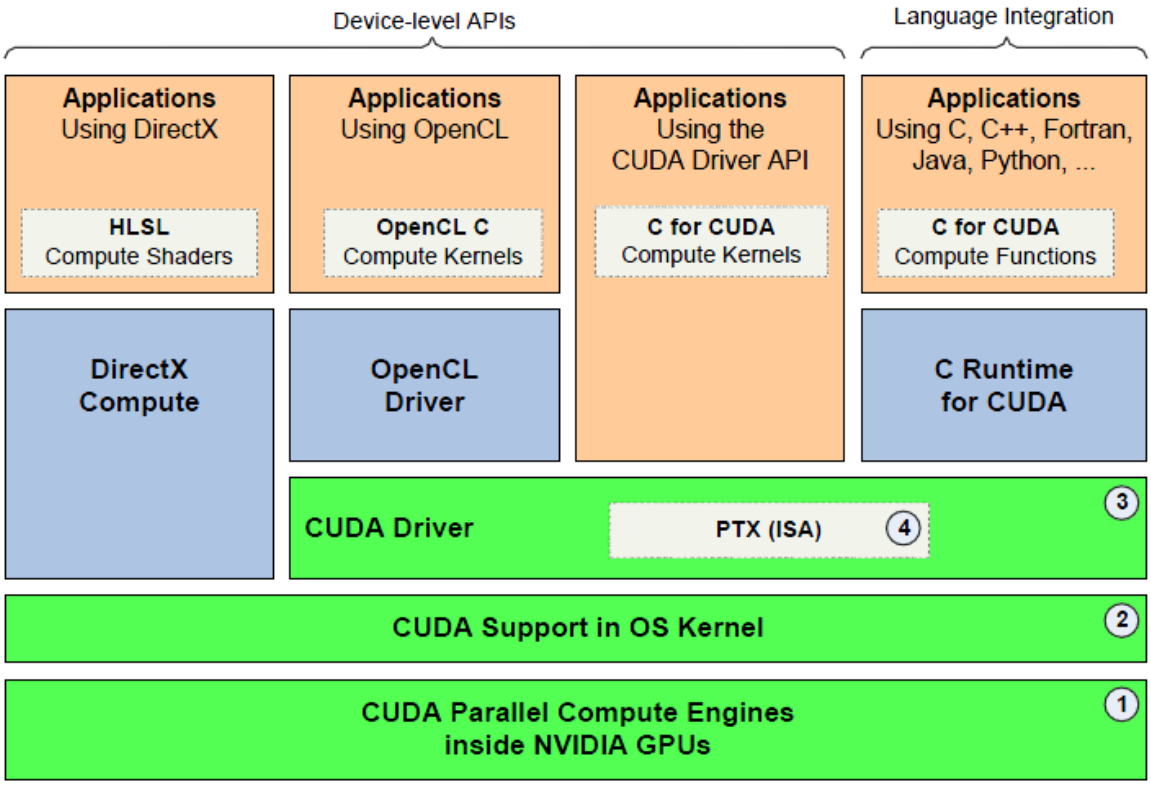
Nvidijin konkurent AMD odbacio je svoj izvorni GPGPU API *Close to Metal* te umjesto toga odlučio izravno podržati OpenCL (i kasnije *DirectCompute*).

Treća mogućnost je koristiti CUDA API koji Nvidia razvija paralelno uz arhitekturu CUDA i koji omogućava relativno jednostavnu integraciju s projektima temeljenim na programskom jeziku C i C++.

2.1. CUDA API i razvojno okruženje

Sučelje arhitekture CUDA može se podijeliti na nekoliko razina (Slika 8):

1. Pokretači za paralelno računanje unutar samog GPU-a
2. Podrška za inicijalizaciju sklopovlja na razini jezgre sustava
3. Upravljački program u korisničkom načinu rada koji pruža API za razvijatelje
4. PTX skup asemblerskih instrukcija arhitekture za paralelno izvršavanje



Slika 8: Upravljački programi i programsko sučelje arhitekture CUDA [6]

CUDA API implementiran je na dvije razine.

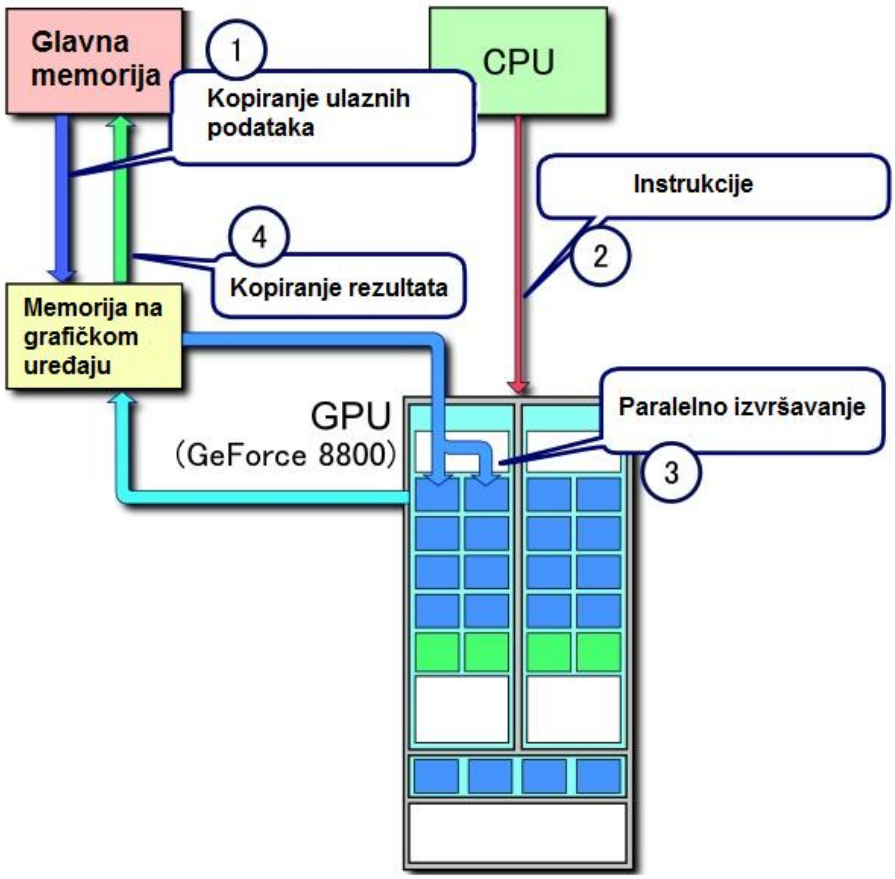
Prvo je sučelje na razini grafičkog uređaja u kojem aplikacija koristi *DirectCompute*, *OpenCL* ili *CUDA Driver API* izravno za podešavanje GPU-a, pokretanje jezgrenih funkcija i čitanje rezultata.

Kada se koristi ovo sučelje, potrebno je napisati jezgrene funkcije u zasebnim datotekama koristeći programski jezik koje odabrani API podržava. Za *DirectCompute* to je HLSL, *OpenCL* funkcije pišu se u jeziku sličnom C-u nazvanom „OpenCL C“, a *CUDA Driver API* prihvaća funkcije napisane u C-u ili PTX asemblerskom jeziku.

Drugo je programsko sučelje na razini integracije s programskim jezikom u kojem aplikacija koristi *C Runtime for CUDA API* te programeri pomoću malog skupa proširenja standardnog C-a određuju koje funkcije se trebaju izvoditi na GPU i kako. Kada se koristi ovo sučelje programeri pišu funkcije u C-u, a *CUDA Runtime API* automatski postavlja GPU i upravlja izvršavanjem funkcija

Rad sa grafičkim procesorom uglavnom se odvija u nekoliko osnovnih koraka (Slika 9):

1. Kopiranje svih potrebnih podataka koji se trebaju obraditi iz glavne memorije računala u memoriju na grafičkoj kartici
2. CPU nizom instrukcija upućuje GPU kako podatke treba obraditi, te započinje proces obrade
3. GPU obavlja obradu podataka paralelnim izvršavanjem jezgrene funkcije
4. Rezultat se kopira iz memorije na grafičkoj kartici u glavnu memoriju računala



Slika 9: Tipičan slijed izvođenja pri radu s arhitekturom CUDA [5]

U daljnjem tekstu bit će ukratko opisan *CUDA Runtime API* te kako se koristeći to sučelje mogu obaviti gore navedeni koraci.

2.1.1. CUDA Runtime API

Upravljanje uređajem

Kako bi mogli koristiti GPU moramo prije svega u glavnom programu odabrati i postaviti željeni grafički uređaj.

Funkcija `cudaGetDeviceCount()` vraća broj dostupnih CUDA uređaja, a funkcija `cudaGetDeviceProperties()` daje popis značajki i mogućnosti navedenog uređaja kao što su veličine pojedinih memorija, podržane mogućnosti računanja, najveće dozvoljene dimenzije blokova i rešetki itd.

Za pomoć pri odabiru uređaja možemo koristiti funkciju `cudaChooseDevice()` koja za zadani popis željenih značajki pronalazi uređaji koji najbolje odgovara opisu.

Kada smo odabrali željeni uređaj, postavljamo ga funkcijom `cudaSetDevice()`.

Upravljanje grafičkom memorijom

Nakon postavljanja uređaja, potrebno mu je prenijeti ulazne podatke koje želimo obraditi. U tu svrhu koriste se funkcije vrlo slične funkcijama za upravljanje memorijom u standardnom C-u.

Za početno alociranje memorije na uređaju koristi se funkcija `cudaMalloc()` koja postavlja zadani pokazivač na početnu adresu u memoriji. Funkcijom `cudaMemcpy()` se pomoću tog pokazivača mogu kopirati podaci u alociranu memoriju iz glavne memorije ili iz druge adrese u memoriji na uređaju, a mogu se prenositi i podaci u drugom smjeru tj. iz memorije uređaja u glavnu memoriju računala. Kako prevoditelj ne može znati u kojoj memoriji je adresa na koju pokazuje pokazivač, smjer kopiranja se mora eksplicitno odrediti posebnim parametrom u navedenoj funkciji.

Kada memorija više nije potrebna, treba ju osloboditi pozivom funkcije `cudaFree()`.

Upravljanje teksturama

Osim pristupa globalnoj memoriji, iz glavnog programa možemo i povezati određeni blok alocirane memorije s referencom za teksturu pomoću koje se iz GPU-a toj memoriji može pristupiti kao teksturi. Povezivanje će se obaviti funkcijom `cudaBindTexture()`.

Pristup teksturi izvodi se preko priručne memorije, a omogućava i neke dodatne opcije kao što je interpolacija elemenata teksture. Dretve na GPU mogu samo čitati iz teksture, a pisanje nije dozvoljeno.

Upravljanje dretvama

Pokretanje jezgrene funkcije iz glavnog programa je asinkrono što znači da će CPU samo poslati naredbu GPU-u da započne s izvršavanjem, ali neće čekati da se izvršavanje dovrši nego će nastaviti s glavnim programom.

Ako je potrebno, može se iz glavnog programa pričekati kraj izvršavanja svih poslova na GPU funkcijom `cudaThreadSynchronize()` i na taj način sinkronizirati izvođenje.

2.1.2. C for CUDA proširenje standardnog jezika C

Kvalifikatori (oznake tipa) funkcija

Jedno od proširenja standardnog C-a koje CUDA uvodi su posebne oznake tipa funkcija koje određuju hoće li se pojedina funkcija izvoditi na središnjem ili grafičkom procesoru.

Dostupne oznake su: `__global__`, `__device__` i `__host__`.

Kvalifikator `__global__` označava jezgrenu funkciju koja se izvršava na GPU, a može se pozvati samo iz glavnog programa.

Kvalifikator `__device__` označava funkciju koja se izvodi na GPU i može se pozvati iz GPU-a, a kvalifikator `__host__` označava funkciju koja se izvodi na CPU i ne može se pozvati iz GPU-a. Kvalifikatori `__device__` i `__host__` mogu se kombinirati pa se u tom slučaju označena funkcija prevodi i za CPU i za GPU.

Funkcije koje se izvršavaju na GPU ne podržavaju rekurziju ni statičke varijable unutar funkcije te ne mogu imati promjenjiv broj argumenata kao što je to inače dozvoljeno u standardnom C-u.

Kvalifikatori varijabli

Varijable također mogu imati posebne oznake koje označavaju u kojem memorijskom prostoru se nalaze. Moguće oznake su:

Kvalifikator `__device__` označava da se varijabla nalazi u globalnoj memoriji na uređaju i dostupna je svim dretvama rešetke te glavnom računalu preko API-ja.

Kvalifikator `__constant__` označava da se varijabla nalazi u memoriji za konstante i dostupna je svim dretvama rešetke, ali samo za čitanje. Postaviti se mora iz glavnog programa na računalu.

Kvalifikator `__shared__` označava da se varijabla nalazi u dijeljenoj memoriji bloka i mogu joj pristupiti samo dretve u istom bloku.

Lokalne varijable deklarirane u funkcijama koje se izvršavaju na grafičkom procesoru, najčešće se nalaze u registrima, ali prevoditelj ih može staviti i u lokalnu memoriju ako nema dovoljno dostupnih registara.

Ugrađeni vektorski tipovi podataka

Kako se GPU najčešće koristi za obradu vektorskih podataka, a to mu je i izvorna namjena, CUDA definira nekoliko vektorskih tipova podataka i osnovne operacije nad njima.

Ti su tipovi izvedeni kao strukture sastavljene od 1, 2, 3 ili 4 člana standardnog tipa podatka. Na primjer, za cjelobrojni tip podatka (`int`) postoje vektorski tipovi: `int1`, `int2`, `int3` i `int4`, a za tip podatka s pomičnim zarezom (`float`) izvedeni su vektorski tipovi: `float1`, `float2`, `float3` i `float4`.

Vektori se inicijaliziraju pomoću funkcije oblika `make_<tip vektora>` i mogu se nad njima koristiti operatori množenja, zbrajanja i oduzimanja kao i nad standardnim tipovima podataka npr:

```
float3 u = make_float3(1.0f, 1.0f, 1.0f);
float3 v = u * 2.0f;
float3 t = u + v;
```

Operacije se zapravo provode po pojedinim komponentama vektora, npr. pri zbrajanju se svaka komponenta jednog vektora zbroji sa pripadajućom komponentom drugog.

Ugrađene varijable

Dretve pri izvršavanju na grafičkom procesoru imaju pristup ugrađenim varijablama pomoću kojih se mogu identificirati. Varijabla `threadIdx` je trodimenzionalni vektor koji označava logički položaj dretve unutar bloka, a varijabla `blockIdx` identificira blok unutar rešetke. Dimenzije bloka dane su varijablom `blockDim`, a dimenzije rešetke varijablom `gridDim`. Može se doznati još i veličina *warpa* varijablom `warpSize`.

Pretpostavimo da je potrebno na GPU obraditi sliku koja je u memoriji zadana kao dvodimenzionalno polje dimenzija 256*256. Ako se pojedini elementi slike mogu obrađivati neovisno o drugima, nabolje rješenje je pokrenuti po jednu dretvu za svaki element. To ćemo učiniti podešavanjem dimenzija rešetke i blokova tako da ukupni broj dretvi odgovara dimenzijama slike. Prvo odabiremo prikladnu veličinu bloka npr. 8*8 dretvi te zatim popunimo rešetku s potrebnim brojem blokova, u našem slučaju to je 256/8=32, dakle dimenzije rešetki su 32*32 blokova.

Svaka dretva sada može pomoću ugrađenih varijabli na sljedeći način točno odrediti koji element slike treba obraditi:

```
x = blockIdx.x * blockDim.x + threadIdx.x
y = blockIdx.y * blockDim.y + threadIdx.y
```

Traženi element je dakle na poziciji (x, y) u slici.

Ugrađene funkcije

Ugrađene funkcije su funkcije za koje postoji posebna sklopovska podrška na grafičkom procesoru i mogu se pozivati samo u kodu kojeg izvodi GPU.

Jedan od mehanizama za koje postoji sklopovska podrška je sinkronizacija dretvi unutar bloka. Sinkronizacija se provodi funkcijom `__syncthreads()` nakon koje će se dretva koja ju poziva zaustaviti dok sve dretve iz tog bloka ne pozovu istu funkciju. Ovdje je bitno primijetiti da ako se sinkronizacija provodi unutar uvjetnog grananja u kodu te ako sve dretve jednog *warpa* ne odaberu isti tok izvršavanja tj. neke zatraže sinkronizaciju, a neke ne, doći će to potpunog zastoja jer se pojedine grane izvode slijedno pa će dretve koje zatraže sinkronizaciju zaustaviti i sve ostale dretve u *warpu*.

Druga skupina ugrađenih funkcija su posebne matematičke koje izvodi jedinica SFU na grafičkom procesoru. To uključuje trigonometrijske funkcije (`__sinf()`, `__cosf()`,

`__tanf()`), funkcije za logaritmiranje i potenciranje (`__logf()`, `__powf()`) te funkciju za dijeljenje dvaju brojeva (`__fdividef()`).

Te se funkcije, zbog sklopovskog ubrzanja, izvršavaju brže od ekvivalentnih funkcija u standardnom C-u, ali zato imaju nešto manju preciznost pa je na programeru da odluči koju verziju funkcije će upotrijebiti.

GPU također podržava nedjeljive funkcije koje u jednoj operaciji mogu pročitati, urediti i zapisati neki podatak u glavnu ili dijeljenu memoriju. Sklopovski je osigurano da za vrijeme obavljanja takve operacije nijedna druga dretva ne može pristupiti istoj memorijskoj adresi. Primjeri takvih funkcija su `atomicAdd()` i `atomicSub()` za zbrajanje i oduzimanje vrijednosti.

Na kraju, tu su i funkcije za uzorkovanje tekstura iz memorije za teksture, npr. funkcija `tex1Dfetch()`.

Pozivanje jezgrene funkcije

Paralelne jezgrene funkcije pokreću se iz glavnog programa proširenom sintaksom za pozivanje funkcija:

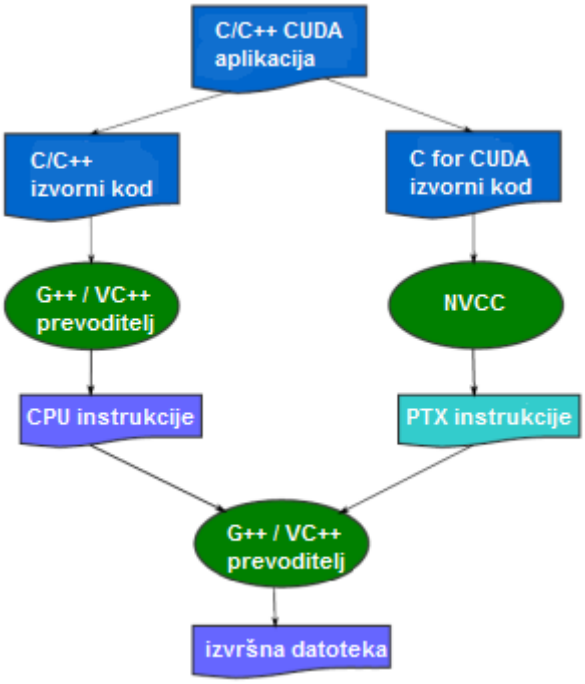
```
kernel<<<dimGrid, dimBlock>>>(... lista parametara ...);
```

Pri čemu su `dimGrid` i `dimBlock` trodimenzionalni vektori tipa `dim3` koji određuju dimenzije rešetke izražene brojem blokova te dimenzije blokova izražene brojem dretvi. Dimenzije koje nisu eksplicitno zadane bit će implicitno postavljene na 1.

GPU će zatim asinkrono pokrenuti veliki broj dretvi određen zadanim dimenzijama od kojih će svaka neovisno izvoditi instancu jezgrene funkcije.

2.1.3. Prevođenje

Jezici više razine kao što su *C for CUDA* prevode se u PTX (eng. *Parallel Thread Execution*) asemblerske instrukcije koje su zatim optimizirane i prevedene u strojni kod sklopovlja na kojem se program treba izvoditi.



Slika 10: Prevođenje CUDA programa

Nvidijin *C for CUDA* prevoditelj (skraćeno NVCC) lako se integrira u postojeće razvojne okoline koje podržavaju programski jezik C/C++. Svaka datoteka koja sadrži *C for CUDA* proširenja jezika mora se prevesti prevoditeljem NVCC, dok se ostale datoteke mogu prevoditi standardnim C/C++ prevoditeljima. Izvorni kod se zatim podijeli na kod za GPU te kod za CPU te se odvojeno prevodi.

Nakon prevođenja odvojeni dijelovi se opet spajaju u jednu izvršnu datoteku.

3. Postupak praćenja zrake

Metoda praćenja zrake i srodne metode vrlo su popularni postupci iscrtavanja slike. Postoje mnoge inačice takvih algoritama, ali sve se temelje na istom principu: simulaciji i aproksimaciji načina na koji zrake svjetla putuju po sceni. Ti su modeli zapravo pojednostavljeni fizikalni modeli svjetlosti poznati iz geometrijske optike.

Širenje svjetlosti na sceni može se zamisliti kao mnoštvo zraka koje polaze od izvora svjetla te se pri sudaru s objektima na sceni mogu zrcaliti, lomiti, apsorbirati ili raspršiti. Neke od tih zraka će naposljetku doći i do oka promatrača i omogućiti mu da stvori sliku scene. Ovakav model primjenjuje se u metodi praćenja puta (eng. *path tracing*) s tom razlikom da se put koji zrake svjetlosti trebaju preći da bi došle od izvora do promatrača pokušava pronaći u obrnutom smjeru tj. od promatrača prema sceni dok se ne dođe do izvora svjetla. Prednost „obratnog“ praćenja je što se ne gubi vrijeme na praćenje zraka koje možda nikad ne bi ni došle do promatrača.

Postupak praćenja puta daje vrlo vjerni prikaz scene i prirodno simulira mnoge optičke učinke koje je drugim metodama teško dobiti kao što su meke sjene, defokusiranje, kaustika svjetla, neizravno osvjetljenje itd. Međutim, ova je metoda računski iznimno zahtjevna pa se koristi samo kada je kvaliteta slike mnogo bitnija od vremena iscrtavanja.

Praćenje puta može se pojednostaviti tako da se u obzir uzme samo izravno osvjetljenje, a raspršenje svjetla se zanemari. Primarne zrake se prate od promatrača do objekata na sceni i kada se pronađe najbliži presjek zrake i objekta, provjeri se je li objekt u sjeni bacanjem zrake prema izvoru svjetla, a osvjetljenje se računa po lokalnom modelu osvjetljenja kao što je npr. Phongov model. Dodatno se stvaraju još i zrake za praćenje zrcaljenja i loma svjetlosti. Taj se postupak naziva metodom praćenja zrake (eng. *ray tracing*). Ovako pojednostavljeni model daje vrlo oštre slike, a učinci koji nastaju raspršenjem svjetlosti se gotovo u potpunosti gube, ali se mogu aproksimirati dodatnim izmjenama u postupku.

Daljnje pojednostavljenje navedenih metoda je postupak bacanja zrake (eng. *ray casting*) koji u obzir uzima samo početne zrake od promatrača prema sceni.

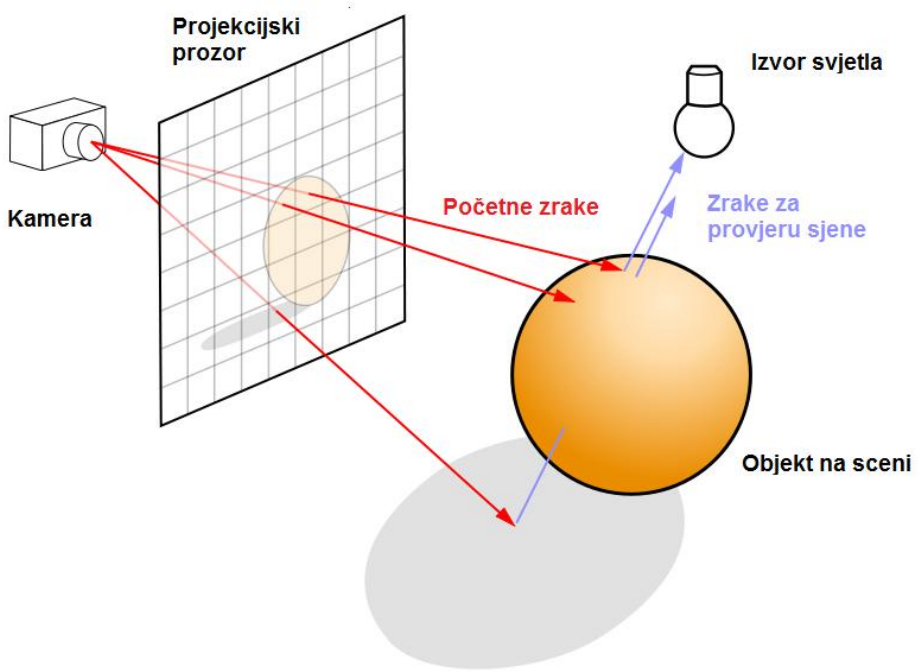
Jasno je da iako se svakim pojednostavljenjem izvornog postupka gubi na kvaliteti i vjernosti slike, istovremeno se dobiva na brzini iscrtavanja. Stoga će izbor pojedine metode prije svega ovisiti o svrsi iscrtavanja.

Metode temeljene na praćenju zrake, najčešće se koriste za tzv. „*off-line*“ iscrtavanje, dok se za iscrtavanje u stvarnom vremenu danas gotovo isključivo koristi metoda rasterizacije.

Bitna razlika između metode praćenja zrake i metode rasterizacije je u tome što metoda praćenja zrake za svaki pojedini element slike prolazi kroz objekte na sceni te traži presjek, dok se pri rasterizaciji ide obratnim postupkom: za svaki objekt na sceni pronalaze se elementi slike na kojima je taj objekt vidljiv.

3.1. Osnovni princip praćenja zrake

U svojoj osnovnoj inačici, postupak praćenja zrake vrlo je jednostavan. Zamislimo sliku koja se iscrtava kao projekciju scene na projekcijskom prozoru (Slika 11).



Slika 11: Praćenje zrake kroz projekcijski prozor [9]

Za svaki element slike potrebno je baciti početnu zraku te ju nakon toga pratiti po sceni. Zrake su jednoznačno određene izvorištem i vektorom smjera.

Početne zrake imaju izvorište u točki u kojoj se nalazi virtualna kamera (promatrač), a vektor smjera zrake dobivamo oduzimanjem pozicije izvorišta od pozicije elementa slike na projekcijskom prozoru.

Za svaku tako dobivenu početnu zraku zatim tražimo najbliži presjek sa objektima na sceni te ako ga nađemo, izračunamo osvjetljenje u točki presjeka, stvaramo zraku za provjeru sjene te zrake zrcaljenja i loma svjetlosti za koje se postupak zatim rekurzivno ponavlja.

```
Za svaki element slike {
      Izračunaj početnu zraku R
      Boja_elementa_slike = Prati_zraku(R)
}

Prati_zraku(R) {
      Pronađi točku T najbližeg presjeka zrake R sa scenom
      Ako nema presjeka vrati boju pozadine
      Inače izračunaj boju Cloc lokalnim modelom osvjetljenja
      Izračunaj zrcaljenu zraku Rzrc i lomljenu zraku Rlom
      Boja Czrc = Prati_zraku(Rzrc)
      Boja Clom = Prati_zraku(Rlom)
      Vrati kombinaciju Cloc, Czrc i Clom
}
```

Kod 1: Pseudokod postupka praćenja zrake [1]

Gore navedenim pseudokodom opisan je osnovni algoritam praćenja zrake. U praksi se dubina rekurzije mora ipak ograničiti pa se u implementaciji dodaje neki mehanizam izlaza iz rekurzije. Funkcija koja računa lokalno osvjetljenje može prvo provjeriti je li točka presjeka u sjeni praćenjem zrake od točke presjeka do izvora svjetla.

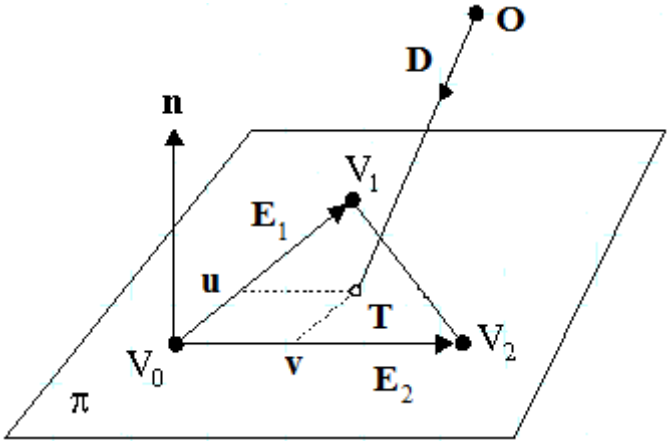
Budući da arhitektura CUDA ne podržava rekurzije, ovaj se algoritam mora prilagoditi tako da rekurziju zamijenimo petljom.

3.2. Pronalazak presjeka zrake i objekata na sceni

Funkcija kojom se pronalazi presjek zrake i objekata na sceni ovisit će o načinu na koji su ti objekti zapisani. Objekti mogu npr. biti zapisani parametarski (npr. kugla određena središtem i promjerom), volumno ili kao skupine grafičkih primitiva.

Za potrebe ovog rada, pretpostavit ćemo da je scena zapisana kao skup trokuta (eng. *poyligon*).

Svaki trokut jednoznačno je određen s tri vrha (V_0, V_1, V_2) i leži na ravnini koja je određena jednom točkom trokuta i njegovom normalom (Slika 12).



Slika 12: Presjek zrake i trokuta

Sve točke koje leže na istoj ravnini kao i promatrani trokut, mogu se zapisati kao linearna kombinacija jednog vrha (V_0) i dvaju vektora koji označavaju njemu prileježeće bridove.

$$T(u, v) = V_0 + uE_1 + vE_2 \tag{1}$$

Pri čemu su u i v baricentrične koordinate točke T, a vektori E_1 i E_2 dani su sljedećim izrazima:

$$E_1 = V_1 - V_0 \quad , \quad E_2 = V_2 - V_0 \tag{2}$$

Točka T nalazi se unutar trokuta ako i samo ako su zadovoljeni uvjeti:

$$0 \leq u \leq 1 \quad , \quad 0 \leq v \leq 1 \quad , \quad 0 \leq (u + v) \leq 1 \tag{3}$$

Točka T može se zapisati i kao translacija od izvorišta zrake O u smjeru zrake koji je određen jediničnim vektorom D.

$$T(u, v) = O + tD \tag{4}$$

Ovdje je *t* udaljenost točke *T* od izvorišta zrake *O*.

Kombinacijom izraza (1) i (4) dobijemo:

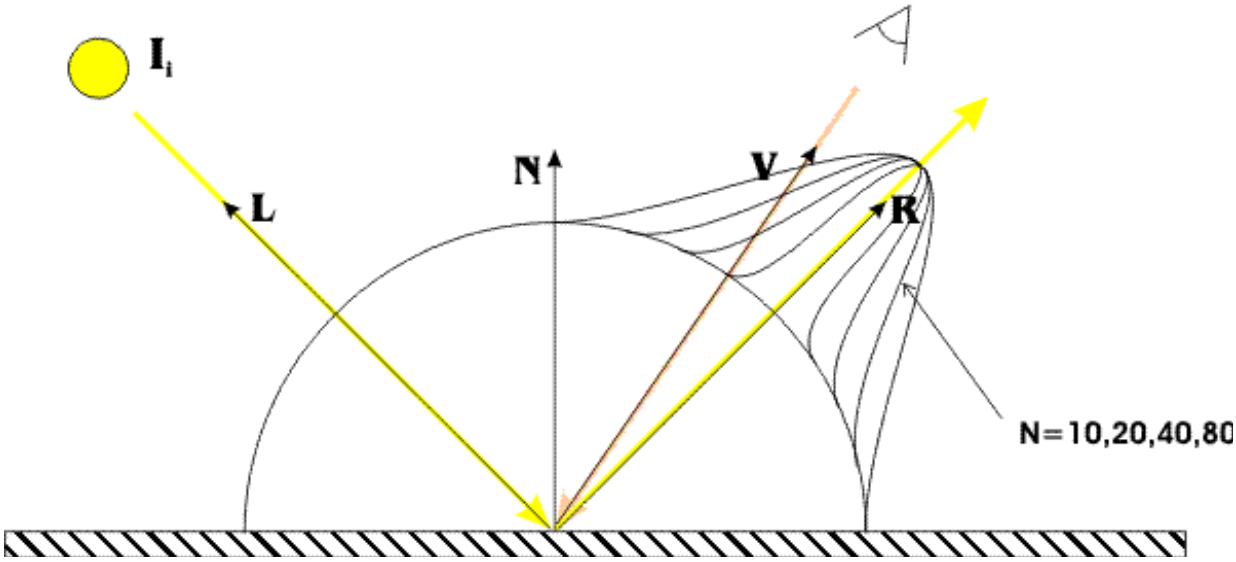
$$O + tD = V_0 + uE_1 + vE_2 \tag{5}$$

Rješenje te jednadžbe dali su Möller i Trumbore [10] te opisali učinkoviti algoritam za pronalazak presjeka zrake i trokuta.

U svrhu ubrzavanja izvođenja, bridne vektore E_1 i E_2 izračunat ćemo za sve trokute u glavnom programu te tako pripremljene podatke poslati grafičkoj kartici.

3.3. Model lokalnog osvjetljenja

Kada pronađemo najbližu točku presjeka zrake i scene, potrebno je odrediti osvjetljenje u toj točki. Lokalno osvjetljenje možemo računati po Phongovom modelu koji svjetlost rastavlja na tri komponente: svjetlost iz okoline (eng. *ambient*), raspršenu svjetlost (eng. *diffuse*) i zrcaljenu svjetlost (eng. *speccular*).



Slika 13: Phongov model osvjetljenja

Prema Phongovom modelu osvjetljenje je najjače u smjeru zrcaljenja te zatim opada s kutom.

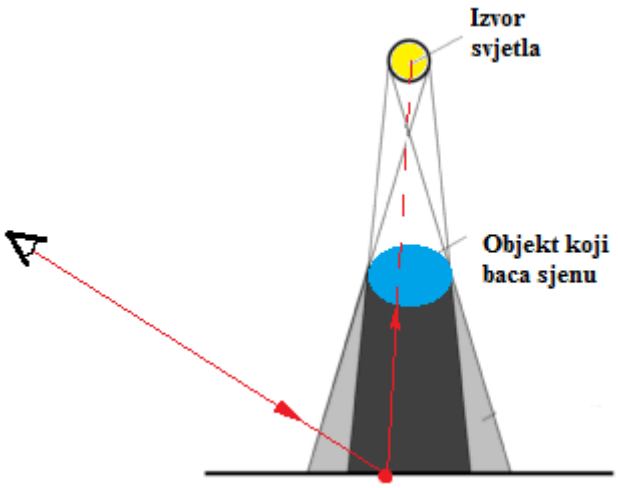
Potpuna jednadžba osvjetljenja glasi:

$$I = I_a k_a + I_d k_d (L \cdot N) + I_s k_s (R \cdot V)^n \tag{6}$$

Pri čemu su s *I* označeni intenziteti svjetlosti (intenziteti pojedinih komponenti, svojstvo izvora svjetlosti), *k* koeficijenti odbijanja svjetlosti (svojstvo materijala), *L* vektor smjera svjetla, *N* vektor normale, *R* vektor smjera zrcaljenja, *V* vektor pogleda te faktor sjajnosti *n*. Svi vektori moraju biti prethodno normirani na jediničnu veličinu.

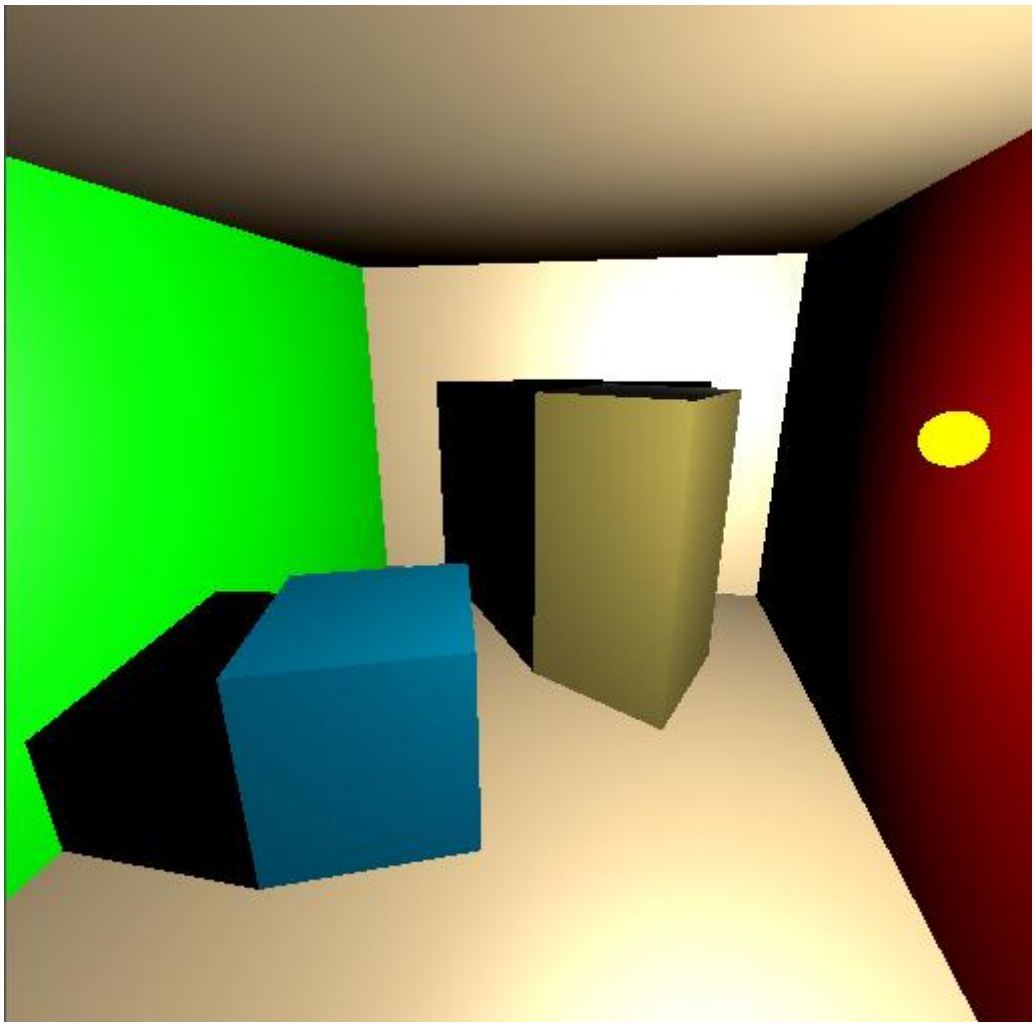
3.4. Sjene

U sklopu računanja lokalnog osvjetljenja, trebalo bi se provjeriti je li pronađena točka u sjeni. To se obično rješava zrakama za ispitivanje sjene koje se šalju od trenutne točke prema izvoru svjetla te se algoritmom za pronalazak presjeka provjeri postoji li objekt koji zaklanja svjetlost na putu prema izvoru.



Slika 14: Sjena i polusjena

Na taj se način dobiju oštre sjene (Slika 15). No u stvarnosti sjene su često „meke“ tj. nemaju oštru granicu jer izvori svjetla nisu savršene točke. Za ostvarenje učinka mekih sjena treba baciti više zraka za ispitivanje sjena s malim odstupanjima u položaju izvora svjetla te dobivene rezultate interpolirati.

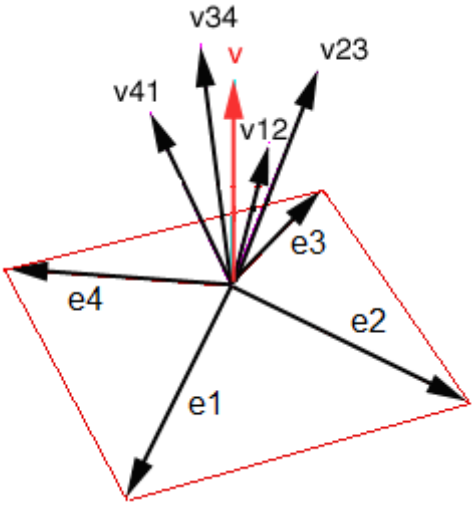


Slika 15: Scena iscrtana postupkom praćenja zrake na GPU

3.5. Interpolacija normala

Pri računanju lokalnog osvjetljenja te smjera zrcaljenja i loma svjetlosti koristimo vektor normale na površinu trokuta. Taj vektor određuje smjer u kojem je okrenuto lice trokuta i može se izračunati kao vektorski produkt bridnih vektora trokuta. Međutim na taj će način svi trokuti izgledati kao potpuno ravne površine. Ponekad trokutima želimo aproksimirati zakrivljenu površinu te bi bilo poželjno normale računati tako da simuliraju tu zakrivljenost.

To ćemo učiniti tako da pri učitavanju scene izračunamo normale po svakom vrhu koje dobijemo kao srednje vrijednosti normala svih trokuta koji dijele taj vrh (Slika 16).



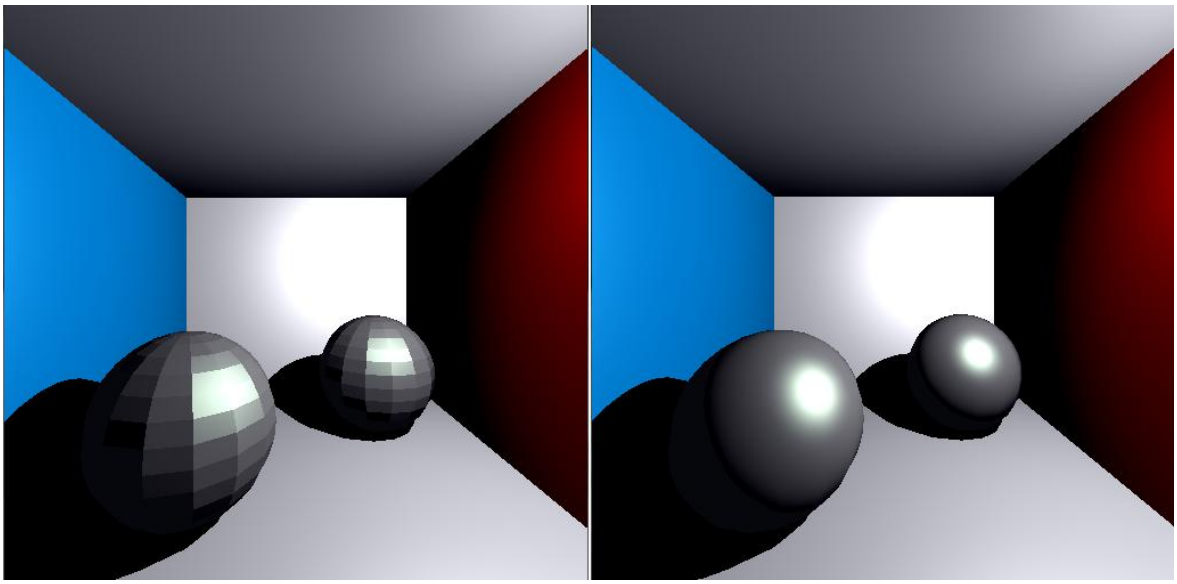
Slika 16: Računanje normale vrha

Normale vrhova jednog trokuta se zatim interpoliraju ovisno o položaju točke presjeka unutar trokuta. Tome će nam poslužiti baricentrične koordinate dobivene algoritmom za pronalazak presjeka.

Normala u točki presjeka određena je izrazom:

$$N = (1-u-v)N_1 + uN_2 + vN_3 \tag{7}$$

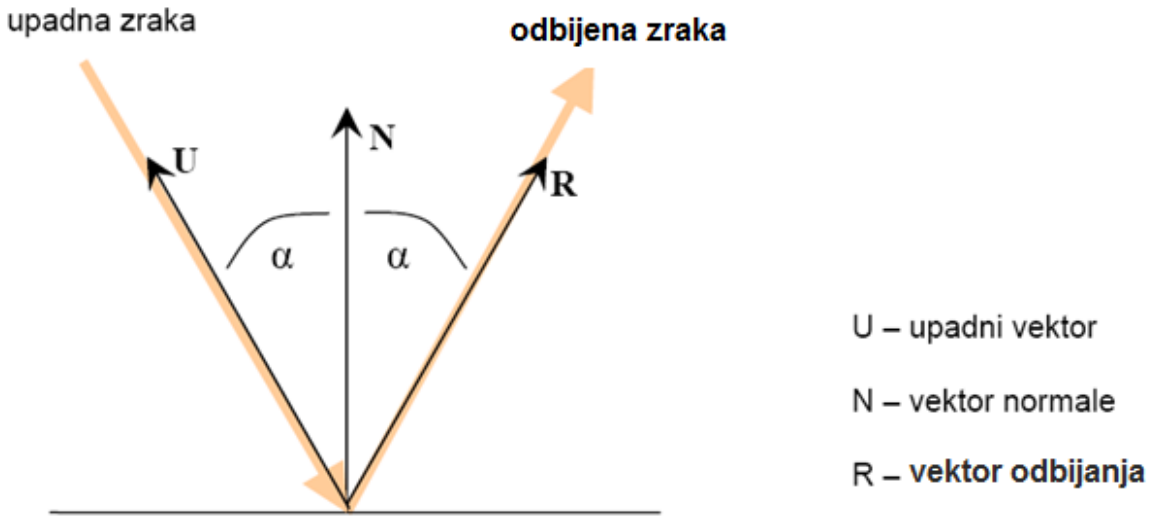
gdje su N_1, N_2 i N_3 normale u pojedinačnim vrhovima trokuta, a u i v su baricentrične koordinate točke presjeka zrake i trokuta.



Slika 17: Usporedba učinaka dobivenih različitim računanjem normala:
lijevo – bez interpolacije, desno – sa interpolacijom

3.6. Zrcaljenje svjetlosti

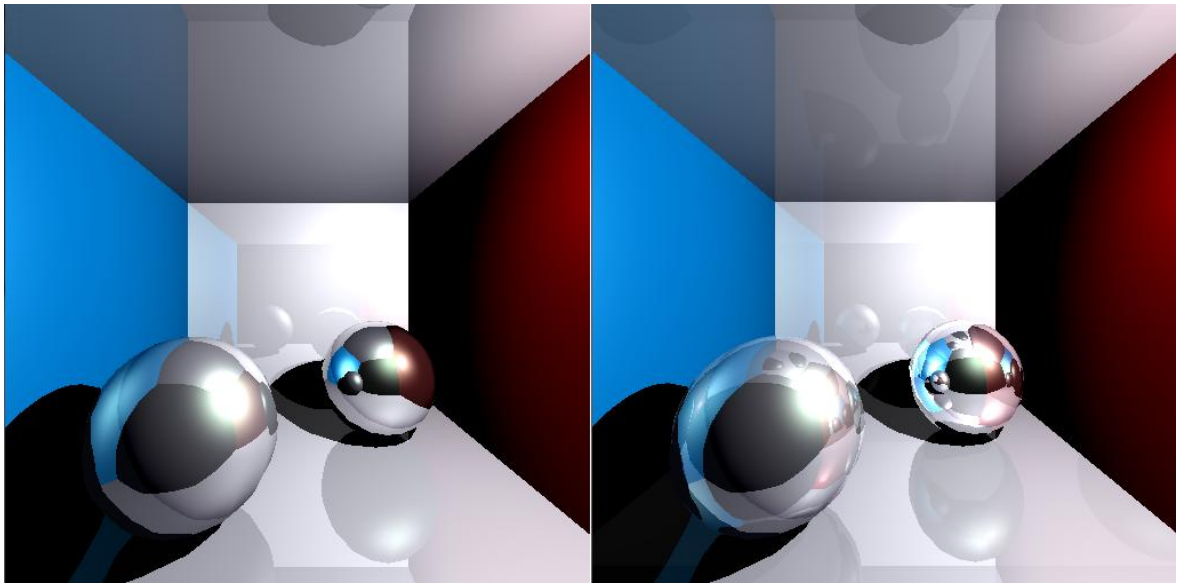
Nakon što smo izračunali lokalno osvjetljenje, računaju se zrake zrcaljenja i loma. Zraku zrcaljenja jednostavno je izračunati jer je kut odbijanja svjetlosti uvijek jednak upadnom kutu samo na suprotnu stranu od normale zrcalne površine.



Slika 18: Upadna i odbijena zraka

Ako je poznat vektor normale i upadni vektor, uz uvjet da su ti vektori prethodno normirani, odbijenu zraku računamo po formuli:

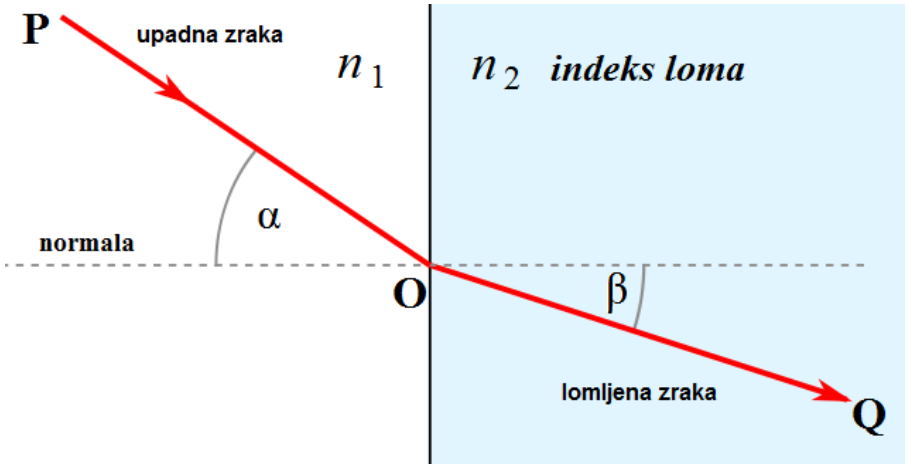
$$R = (2U \bullet N) * N - U \tag{8}$$



Slika 19: Zrcaljenje svjetlosti, lijevo – jedna iteracija, desno – dvije iteracije

3.7. Lom svjetlosti

Svjetlost se pri prelasku iz jednog materijala u drugi lomi prema Snellovom zakonu koji kaže da su sinusi kutova upada i loma u odnosu na normalu proporcionalni indeksima loma materijala između kojih se zraka lomi.



Slika 20: Lom svjetlosti

Ako znamo indekse loma materijala η_1 i η_2 te upadni kut α, kut loma β možemo odrediti izrazom:

$$\eta_1 * \sin(\alpha) = \eta_2 * \sin(\beta) \tag{9}$$

Kako bi odredili vektor lomljene zrake svjetlosti postavljamo lomljeni vektor R kao linearnu kombinaciju upadnog vektora U i vektora normale N [1].

$$R = -a * N - b * U \tag{10}$$

Koeficijenti *a* i *b* dobiju se izrazima:

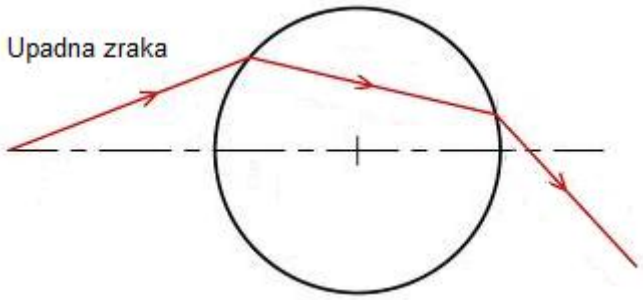
$$b = \frac{\eta_1}{\eta_2} \tag{11}$$

$$a = \frac{(-2 * b * \cos\alpha + \sqrt{D})}{2} \tag{12}$$

pri čemu je

$$D = 4 * (b^2 * (\cos\alpha)^2 - b^2 + 1) \tag{13}$$

Za pravilan prikaz loma svjetlosti najčešće je potrebno pratiti dvije uzastopne zrake: jednu pri ulazu u prozirni objekt te jednu pri izlazu iz objekta (Slika 21).



Slika 21: Lom pri ulazu i izlazu zrake iz objekta

Ovdje je pretpostavljeno da unutar prozirnog objekta nema drugih objekata.

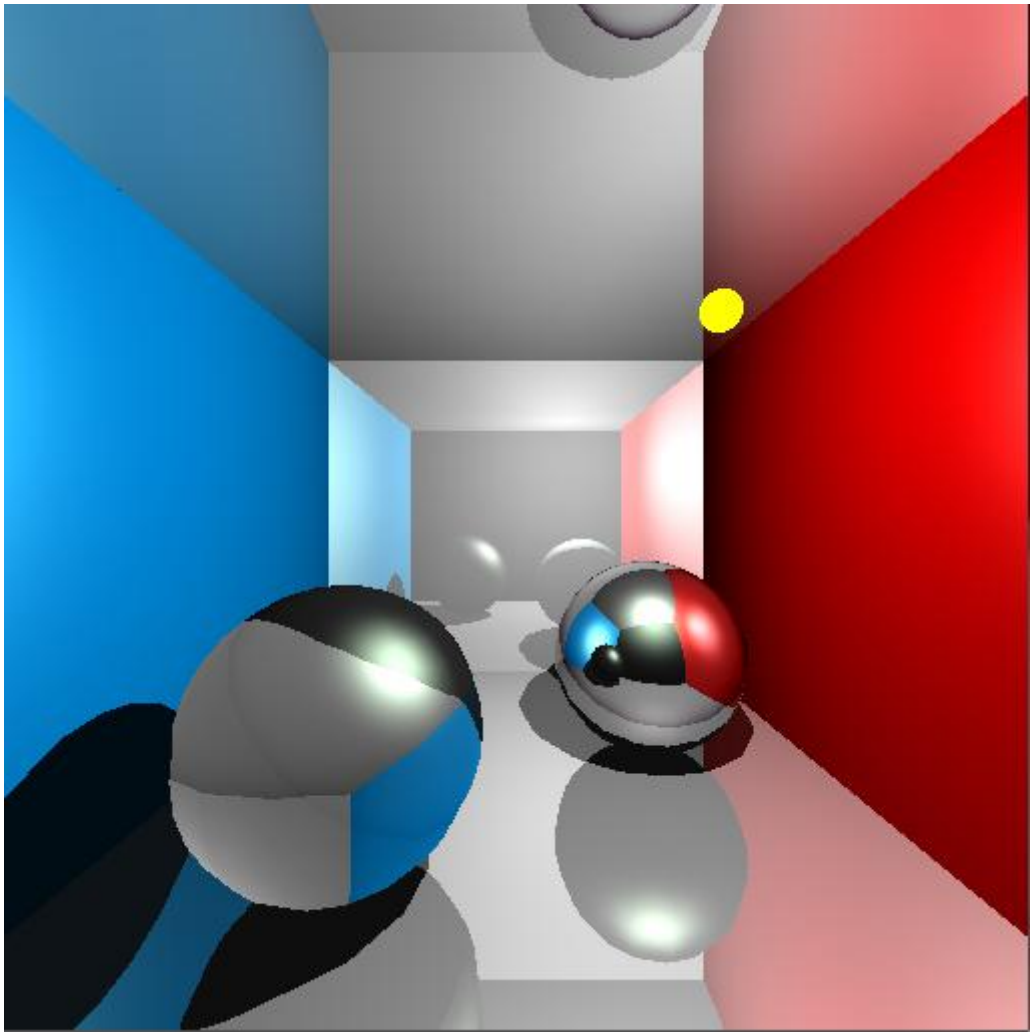
3.8. Kombiniranje lokalnog osvjetljenja sa zrcaljenim i lomljenim svjetlom

U prethodnim poglavljima objašnjeno je kako izračunati doprinose lokalnog, zrcaljenog i lomljenog svjetla. Tako izračunate doprinose moramo na neki način spojiti u ukupno osvjetljenje u točki loma tj. u završnu boju elementa slike.

Najjednostavniji način je uvesti koeficijente zrcaljenja i loma te boju izračunati kao linearnu kombinaciju lokalnog, zrcaljenog i lomljenog svjetla na sljedeći način:

$$I = I_{local} + k_1 I_{refl} + k_2 I_{refr} \tag{14}$$

Pri čemu su I_{local} , I_{refl} i I_{refr} redom doprinosi lokalnog, zrcaljenog i lomljenog svjetla, a k_1 i k_2 koeficijenti zrcaljenja i loma.



Slika 22: Zrcaljenje (desna kugla) i lom (lijeva kugla) svjetlosti

Složeniji i zahtjevniji, ali fizikalno ispravan način kombiniranja svjetlosti loma i zrcaljenja bio bi korištenjem Fresnelovih jednadžbi.

4. Implementacija

Pri implementaciji postupka praćenja zrake na GPU, potrebno je prilagoditi postupak paralelnom izvršavanju te određenim ograničenjima kao što je nedostatak rekurzije i relativno mala količina brze memorije.

4.1. Računanje početnih zraka

Pri pokretanju jezgrene funkcije, svakom je elementu slike pridružena jedna dretva koja računa njegovu boju. Prvo što dretva mora napraviti je izračunati početnu zraku za pridruženi element slike. U glavnom se programu pomoću pozicije i orijentacije kamere izračuna pozicija prvog elementa slike na projekcijskom prozoru te vertikalni i horizontalni pomak između susjednih elemenata. Tu je bitno uzeti u obzir razlučivost te omjer visine i širine slike.

Ti se podaci zatim prenose u grafičku memoriju pa svaka dretva može pomoću njih u kombinaciji s ugrađenim varijablama odrediti koji element slike treba obraditi i izračunati početnu zraku.

```
// odredi element slike
int x = blockIdx.x*blockDim.x + threadIdx.x;
int y = blockIdx.y*blockDim.y + threadIdx.y;

// položaj točke u prostoru koja odgovara zadanom elementu
// slike na projekcijskom prozoru
float3 tocka = prva + x*pomakDesno + y*pomakGore;

// izvorište početne zrake je u točki kamere
float3 izvor = polozajKamere;
// smjer zrake
float3 smjer = tocka – polozajKamere;
```

Kod 2: Računanje početne zrake u jezgrenoj funkciji

U gornjem primjeru varijable položaj kamere (`polozajKamere`), položaj prve točke (`prva`) te horizontalni i vertikalni pomak između dvije susjedne točke na projekcijskom

prozoru (`pomakDesno`, `pomakGore`) izračunati su u glavnom programu te preneseni u memoriju za konstante na grafičkoj kartici.

4.2. Prilagođena inačica postupka praćenja zrake

Kao što je u 3. poglavlju objašnjeno, postupak praćenja zrake temelji se na rekurzivnom algoritmu. Ovo je veliki problem jer arhitektura CUDA ne podržava rekurziju pa se algoritam mora ostvariti iterativno.

Postoji nekoliko različitih pristupa tom problemu. Ako računamo samo jednu dodatnu zraku za svaki pronađeni presjek, npr. samo zrcaljenje, postupak se može napisati pomoću jedne programske petlje koja izvršava onoliko iteracija kolika je bila dubina rekurzije u izvornom algoritmu.

```
// postaji zraku kao pocetnu
zraka = izracunajPocetnuZraku();
// postavi boju na boju pozadine
boja = boja_pozadine;
// postavi faktor doprinosa zrcaljenja
reflKoef = 1.0;

dok(reflKoef > 0.01 && i<dubina)
{
   presjek = pronadiPresjek(zraka);
   ako(pronađen_presjek){
      normala = izracunajNormalu(presjek);
      mat = dohvatiMaterijal(presjek);
      boja += reflKoef * lokalnoOsvjetljenje(normala, mat);
      reflKoef *= mat.reflKoef;

      zraka = zrcali(zraka, normala);
      ++i;
   }
   inace izadi iz petlje;
}
```

Kod 3: Iterativno praćenje zrake: inačica za zrcaljenje

Gore navedeni algoritam vrlo učinkovito zamjenjuje rekurziju, ali računa samo zrcaljenje. Kako bi dodali i računanje loma svjetlosti možemo napraviti vlastiti stog u memoriji grafičkog uređaja te na njega pohranjivati potrebne podatke o zrakama i pronađenim presjecima u svakoj iteraciji. Međutim korištenje globalne memorije je vrlo sporo pa bi to postalo usko grlo u izvođenju jezgrene funkcije, a brze memorije kao što su registri i dijeljena memorija ima vrlo malo. Pokazalo se da je spremanje svih podataka potrebnih za računanje loma i zrcaljenja na stog memorijski vrlo zahtjevno pa je inačica programa koja računa lom svjetlosti izvedena slijedno i ograničena na jednu zrcaljenu i dvije lomljene zrake pri čemu je jedna lomljena zraka za ulaz, a druga za izlaz iz prozirnog objekta.

Algoritam korišten za računanje lomljene zrake iz upadne, vektora normale i indeksa loma opisan je u radu [11] i glasi:

```
float3 refract(float3 normala, float3 upadni,
                float n1, float n2)
{
   float n = n1 / n2;
   // kosinus se računa pomoću skalarnog produkta dot()
   float cosI = dot(normala, upadni);
   float sinT2 = n * n * (1.0 - cosI * cosI);
   if (sinT2 > 1.0)
   {
      // došlo je do unutarnjeg zrcaljenja
   }
   return n * upadni - (n + sqrt(1.0 - sinT2)) * normala;
}
```

Kod 4: Funkcija za računanje lomljene zrake

4.3. Zauzeće sredstava grafičkog uređaja

Prije pokretanja jezgrene funkcije na GPU, svi podaci o sceni preneseni su u različite memorijske prostore na grafičkoj kartici. Koordinate vrhova spremljene su u memoriju za teksture jer se često dohvaćaju pri određivanju presjeka zrake sa scenom pa je ključno da pristup tim podacima bude brz što je osigurano priručnom memorijom za teksture.

Normale i podaci o prostornoj podjeli su također spremljeni u memoriju za teksture. Ostali podaci o sceni, kao što su svjetla i materijali spremljeni su u memoriju za konstante čije je zauzeće bilo oko 3kB.

Dretve su pokretane u blokovima dimenzija 2*64 za što je eksperimentalno utvrđeno da daje najbolje rezultate. Prema Nvidijinom tabličnom kalkulatoru zauzeća, ovime je postignuto tek 33% zauzeća grafičkog procesora, ali je taj broj ograničen brojem korištenih registara.

Po dretvi je korišteno 32 registra pa budući da je po jedinici SM dostupno 8192 registara, to znači da se istovremeno može izvršavati 256 dretvi, tj. 8 *warpova* po SM-u. Zauzeće lokalne memorije ovisi o inačici jezgrene funkcije i dubini praćenja zrake, a iznosi od 40B do 104B.

Dijeljena memorija korištena je u inačici jezgrene funkcije koja računa zrcaljenje i lom svjetla te iznosi oko 6kB.

5. Optimizacija

Proces optimizacije možemo ugrubo podijeliti na optimizaciju izravno vezanu uz arhitekturu grafičkog procesora te općenitu optimizaciju algoritma praćenja zrake. Iako su te dvije vrste optimizacija usko vezane jedna uz drugu, u prvu grupu možemo npr. staviti optimizacije nad pristupom memoriji i rasporedom grananja u programu, dok bi u drugoj bile tehnike kao što su podjela prostora i strukture za ubrzavanje pretraživanja scene.

5.1. Optimizacija vezana uz arhitekturu

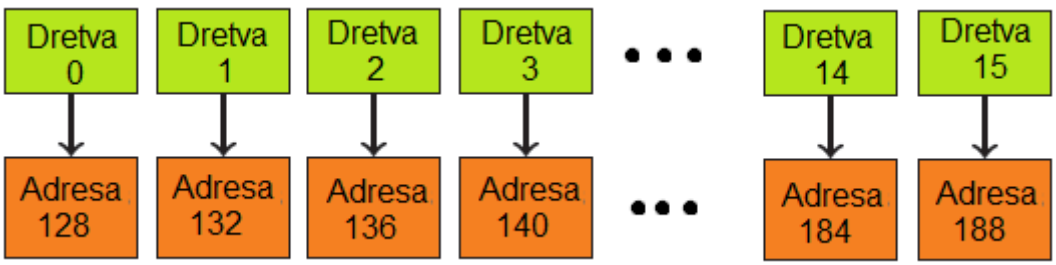
Prilikom pisanja jezgrenih funkcija vrlo je bitno uzeti u obzir sklopovsku implementaciju arhitekture CUDA. Neke od najbitnijih posebnosti takve arhitekture su način na koji se rješava grananje u programskom kodu te pristup različitim memorijskim prostorima.

5.1.1. Združeni pristup globalnoj memoriji

Pristup globalnoj memoriji grafičko sklopovlje obavlja u diskretnim blokovima memorije veličine 128, 256 ili 512 okteta, ovisno o korištenom tipu podataka.

Pristup globalnoj memoriji može imati kašnjenje od 400 do 600 ciklusa, svako čitanje iz dohvaća cijeli blok, neovisno o tome koliko je podataka stvarno potrebno.

Zbog toga se čitanje iz memorije treba obavljati združeno (eng. *coalesced*) na način da dretve iz jednog *warpa* istovremeno zatraže podatke iz istog bloka memorije.



Slika 23: Združeni pristup globalnoj memoriji

Početna adresa bloka u memoriji poravnata je s veličinom bloka. Kako bi dretve nekog *warpa* iskoristile združeni pristup memoriji, n-ta dretva mora zatražiti pristup n-tom elementu bloka memorije.

Ako ti uvjeti nisu zadovoljeni, svaka će dretva morati zasebno pristupiti memoriji čime se gubi paralelizam izvođenja.

Združeni pristup memoriji teško je osigurati u postupku praćenja zrake jer nije unaprijed poznato koji će podaci biti potrebni budući da to ovisi o samom putu zrake kroz scenu.

Međutim čak i jednostavno poravnavanje pristupa memoriji korištenjem ugrađenih tipova podataka za koje je združeno čitanje podržano (`float`, `float2` i `float4`) ubrzalo je iscrtavanje za oko 2.5 puta.

Dodatno ubrzanje postignuto je prebacivanjem podataka o sceni (popis vrhova, normala itd.) u memorijski prostor za teksture koji ima priručnu memoriju te se na taj način gotovo uklonio zastoj koji nastaje zbog čekanja na pristup memoriji.

5.2. Podjela prostora i strukture za ubrzavanje

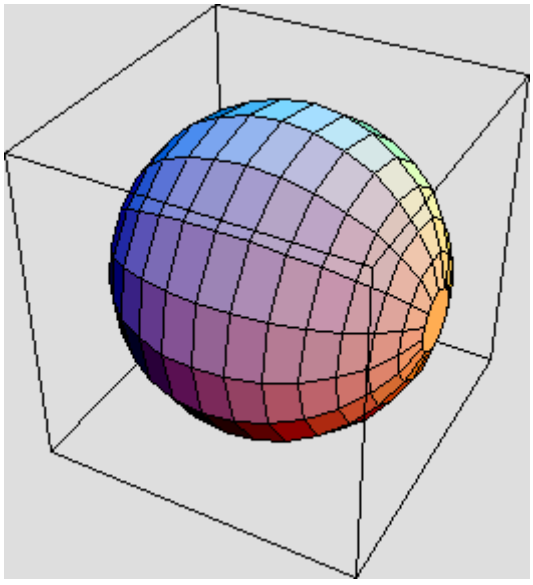
Presjek zrake i scene traži se tako da se ispita presijeca li zraka bilo koji trokut na sceni. Najjednostavnije rješenje je proći kroz sve trokute na sceni te za svaki pozvati funkciju za ispitivanje presjeka zrake i trokuta.

Na taj se način za složenije scene većina vremena troši na računanje presjeka s trokutima koji nisu na putu zrake. Iz tog je razloga poželjno scenu prethodno prostorno podijeliti kako bi se veći dijelovi scene koji nisu na putu trenutne zrake mogli brzo odbaciti. Za to se koriste strukture za ubrzavanje pretraživanja prostora kao što su hijerarhije obujmica, kd-stabla, oktalna stabla i druge.

5.2.1. Obujmice

Obujmica (eng. *bounding volume*) je zatvoreni geometrijski oblik koji omeđuju jedan ili više objekata scene. Poželjno je da su obujmice što jednostavnijeg oblika kako bi provjera presijecanja s njima bila što brža. Zato se često kao obujmica koristi kugla određena samo središtem i promjerom unutar kojeg se nalazi objekt.

Drugi često korišteni oblik je kvadar poravnat s osima (eng. *axis-aligned bounding box*, skraćeno AABB).



Slika 24: Objekt (kugla) i njegova obujmica

Postupak korištenja obujmica u algoritmu praćenja zrake je sljedeći:

```
Za sve obujmice na sceni
      Ispitaj presjek sa zrakom R
      Ako postoji presjek
            Ispitaj presjek sa svim trokutima unutar obujmice
            I zapamti najbliži
      Inače nastavi pretragu
```

Kod 5: Korištenje obujmica

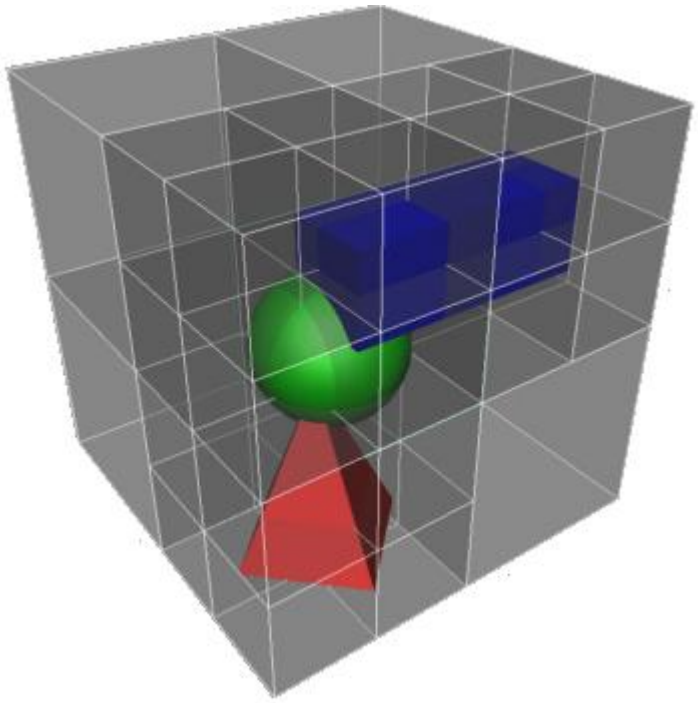
Za implementaciju u ovom radu odabrane su obujmice AABB pri čemu je za ispitivanje presjeka zrake i obujmice korišten algoritam opisan u radu [12].

5.2.2. Hijerarhijske strukture

Kada je u pitanju optimizacija pretrage prostora, neizbježne su hijerarhijske strukture. To su strukture koje prostor dijele rekurzivno, stvarajući tako svojevrsno stablo gdje svaki čvor predstavlja manji dio ukupnog prostora scene. Prostor se može podijeliti pravilno, tako da svaka podjela dijeli prostor na jednake dijelove ili nepravilno gdje se prostor može dijeliti na različite dijelove.

Prednost pravilnih podjela prostora je to što se unaprijed zna kako će prostor biti podijeljen pa se algoritam za pretragu može posebno prilagoditi toj strukturi.

Kod nepravilnih struktura scena se najčešće dijeli tako da u svakoj grani bude podjednak broj elemenata. Zbog toga je takvo stablo uglavnom bolje uravnoteženo i kao posljedica toga, manje dubine.

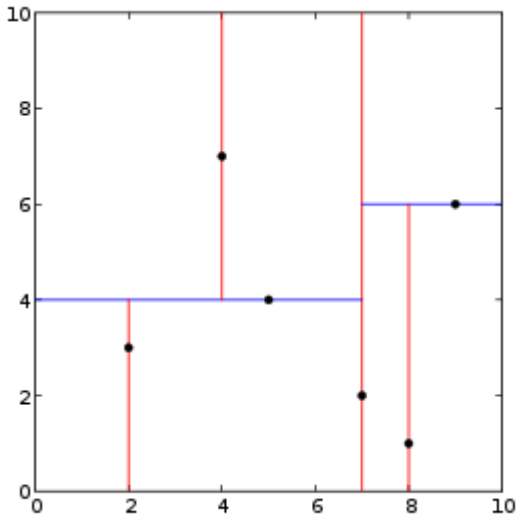


Slika 25: Oktalno stablo

Problem koji se često javlja kod podjele prostora je što učiniti sa objektima (ili trokutima) koji se nalaze na granici između različitih ćelija (Slika 25). U tom se slučaju objekt može ili pridodati u obje ćelije ili podijeliti. Oba rješenja stvaraju dodatne podatke i povećavaju zauzeće memorije.

Pri optimizaciji za GPU jedna od najzastupljenijih struktura za ubrzavanje je kd-stablo.

Kd-stablo u svakom čvoru dijeli prostor na dva dijela po jednoj osi. U svakoj se razini stabla odabire druga os unaprijed određenim redoslijedom i tako u krug dok se ne dođe do listova (Slika 26).

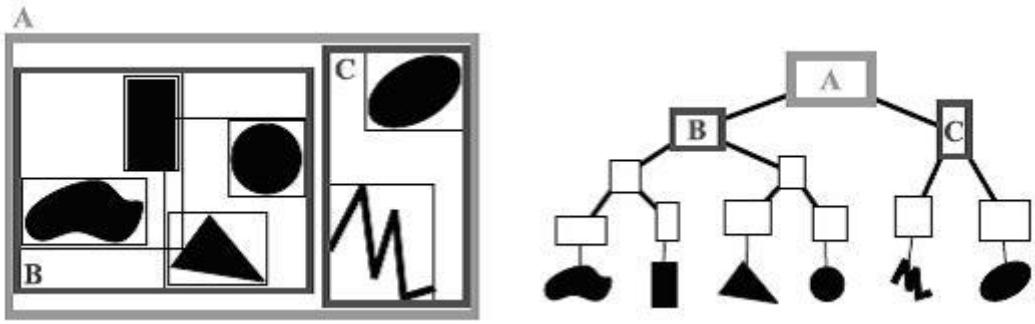


Slika 26: Podjela prostora kd-stablom u dvije dimenzije [13]

Za učinkovito pretraživanje ovakvog stabla potreban je stog pa se za implementaciju na GPU algoritmi pretraživanja posebno prilagođavaju.

U radu [14] kd-stablo implementirano je na nekoliko načina s ciljem smanjenja potrebnog stoga te se među ostalim navodi kako je njihovim najučinkovitijim algoritmom za scenu sa 282 tisuće trokuta ostvareno iscrtavanje 15.2 milijuna početnih zraka na grafičkoj kartici ATI Radeon X1900 XTX. Na razlučivosti koju su koristili (1024*1024) to je oko 14.5 slika po sekundi.

Osim kd-stabla, za optimizaciju praćenja zrake na GPU često se koristi i hijerarhija obujmica (eng. *Bounding Volume Hierachy*, skraćeno BVH). BHV kreće od ranije opisane ideje obujmica, ali ju dorađuje hijerarhijskim grupiranjem obujmica.



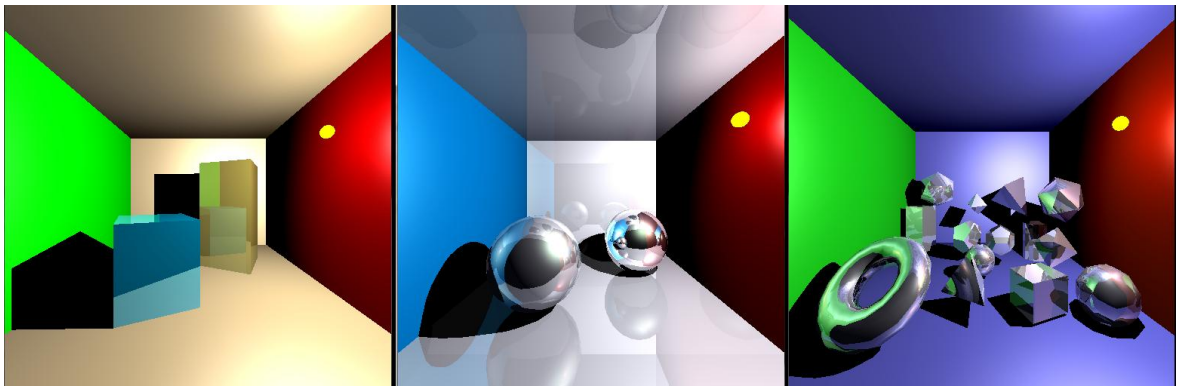
Slika 27: Hijerarhija obujmica [15]

Struktura BVH uvodi vrlo nepravilnu podjelu prostora i susjedni čvorovi na istoj razini stabla se mogu i preklapati pa je cilj izgraditi stablo koje minimizira obujam obujmica što nije lak zadatak.

Koristeći strukturu BVH, Alia, T. i Laine, S. [16] ostvarili su iscrtavanje ranije navedene scene od 282 tisuće trokuta u čak 142.2 milijuna početnih zraka po sekundi na grafičkoj kartici GeForce GTX285.

6. Rezultati

Testiranje postupka praćenja zrake obavljeno je na tri različite scene (Slika 28) čija su bitnija obilježja dana tablicom (Tablica 1) na razlučivosti 512*512 elemenata slike tj. 262144 početne zrake.



Slika 28: Scene za testiranje, s lijeva na desno: *CornellBox, CornellSphere, PlatonicSolids*

Tablica 1: Obilježja scena za testiranje

Scena	*CornellBox*	*CornellSphere*	*PlatonicSolids*
Broj trokuta	34	970	1342
Broj vrhova	102	2910	4026
Broj objekata	8	8	20

Mjerenje vremena iscrtavanja obavljeno je iz tri različita pogleda na scenu te je zatim uzeta njihova srednja vrijednost.

Sva su mjerenja obavljena na grafičkoj kartici NVIDIA GeForce 8800 GT čija su obilježja dana u tablici (Tablica 2).

Tablica 2: Obilježja grafičke kartice korištene za testiranje

GeForce 880 GT			
Mogućnosti računanja	1.1	Globalna memorija	512 MB
Broj SM procesora	14	Memorija za konstante	64 kB
Registara po bloku	8192	Najveći broj dretvi po bloku	512
Dijeljena memorija po bloku	16 kB	Veličina *warpa*	32

Obavljena su četiri različita testa: iscrtavanje scene samo sa početnim zrakama bez uporabe obujmica, iscrtavanje scene samo sa početnim zrakama sa uporabom obujmica, iscrtavanje scene sa uključenim zrcaljenjem bez uporabe obujmica te iscrtavanje scene sa uključenim zrcaljenjem i uporabom obujmica.

Rezultati su izraženi u prosječnom vremenu potrebnom za iscrtavanje jedne slike u milisekundama (Tablica 3).

Tablica 3: Rezultati testova

Tip testa \ Testirana scena	*CornellBox*	*CornellSphere*	*PlatonicSolids*
Početne zrake, bez AABB	5,6 ms	138,2 ms	207,5 ms
Početne zrake sa AABB	5,5 ms	109.0 ms	164,4 ms
Zrcaljenje, bez AABB	7,6 ms	295,2 ms	362,4 ms
Zrcaljenje, sa AABB	9,8 ms	252,8 ms	323,8 ms

Primjećujemo da je pri iscrtavanju samo početnih zraka ispitivanje obujmica dalo ubrzanje od oko 26 % na složenijim scenama dok na jednostavnoj sceni s malo trokuta nije imalo previše utjecaja. To je i očekivano budući da je ispitivanje obujmica najučinkovitije kada se malim brojem testova može odbaciti veliki broj trokuta.

Pri iscrtavanju s uključenim zrcaljenjem ispitivanje obujmica pokazalo je lošije rezultate. Najveće ubrzanje bilo je oko 17% na drugoj sceni dok se iscrtavanje prve scene čak i usporilo.

Zrcaljene zrake su mnogo raspršenije od početnih zraka pa ih je teže paralelno obrađivati.

Ovakvi rezultati ukazuju na to da se implementirano ispitivanje obujmica slabo nosi sa razilaženjem dretvi u izvršavanju.

Zaključak

Iako još uvijek imaju neke nedostatke u usporedbi s tipičnim središnjim procesorima, grafičke se kartice pokazuju kao bolji izbor za rješavanje sve većeg skupa problema. Proizvođači grafičkih kartica su toga svjesni pa se polako okreću razvoju sklopovlja s ciljem općenite uporabe. Jedan od prvih koraka u tom smjeru svakako je Nvidijina arhitektura CUDA koja omogućava pisanje programske potpore za izvođenje na GPU u programskom jeziku C uz neke manje izmjene. Nvidija je na taj način izgradila veliku GPGPU zajednicu te danas postoje brojni radovi i članci koji opisuju korištenje grafičkog procesora u raznolike svrhe.

Postupak praćenja zrake relativno je popularna metoda iscrtavanja slike, ali se donedavno koristio isključivo za *„off-line“* iscrtavanje. Međutim, u zadnje se vrijeme provodi mnogo istraživanja s ciljem prilagodbe te metode za izvršavanje na GPU čime se potencijalno postižu brzine iscrtavanja dovoljne za korištenje i u stvarnom vremenu.

Kako bi se to ostvarilo, mnogo se truda ulaže u optimizaciju i implementaciju struktura za ubrzavanje pretraživanja prostora na grafičkom sklopovlju.

Nije još jasno može li postupak praćenja zrake ili neka srodna metoda u skoroj budućnosti zamijeniti metodu rasterizacije kao glavnu metodu za iscrtavanje u stvarnom vremenu, ali svakako je ostvaren veliki napredak u tom smjeru.

Literatura

[1] PANDŽIĆ, I. S. Virtualna Okruženja: Računalna grafika u stvarnom vremenu i njene primjene, Zagreb, 2004

[2] WIKIPEDIA, SIMD, studeni 2010. http://en.wikipedia.org/wiki/SIMD

[3] NVIDIA CORPORATION, NVIDIA CUDA Programming Guide, veljača 2010.

[4] NVIDIA CORPORATION, NVIDIA's Next Generation CUDA Compute Architecture: Fermi, *Whitepaper*, 2009.

[5] WIKIPEDIA, CUDA, siječanj 2011. http://en.wikipedia.org/wiki/CUDA

[6] NICKOLLS, J., BUCK, I., GARLAND, M., SKADRON, K., Scalable Parallel Programming with CUDA, 28. travnja 2008.

[7] CYRIL ZELLER, Tutorial CUDA, *NVIDIA Developer Technology*, travanj 2008.

[8] NVIDIA CORPORATION, NVIDIA CUDA Architecture: Introduction & Overview, travanj 2009.

[9] WIKIPEDIA, Ray tracing (graphics), prosinac 2010. http://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29

[10] MOLLER, T. , TRUMBORE, B., Fast, Minimum Storage Ray / Triangle Intersection, 1997.

[11] DE GREVE, B., Reflections and Refractions in Ray Tracing

[12] WILLIAMS, A., BARRUS, S., MORLEY, R.K., SHIRLEY, P., An Efficient and Robust Ray–Box Intersection Algorithm, *University of Utah*

[13] WIKIPEDIA, Kd-tree, siječanj 2011. http://en.wikipedia.org/wiki/Kd-tree

[14] HORN, D., R., SUGERMAN, J., HOUSTON, M., HANRAHAN, P., Interactive k-D Tree GPU Raytracing, *University of Utah*

[15] WIKIPEDIA, Bounding volume hierarchy, lipanj 2010. http://en.wikipedia.org/wiki/Bounding_volume_hierarchy

[16] AILA, T., LAINE, S., Understanding the Efficiency of Ray Traversal on GPUs, *NVIDIA Research*

Sažetak

Postupak praćenja zrake na grafičkom procesoru

U ovom je radu proučena arhitektura suvremenih grafičkih procesora s posebnim naglaskom na arhitekturu CUDA. Arhitektura CUDA objašnjena je sa sklopovske razine i s razine programskog sučelja te su razrađene najbitnije prednosti i nedostaci iste.

Također je obrađena metoda praćenja zrake i osnovni učinci koji se tom metodom postižu, kao što su zrcaljenje i lom svjetlosti.

Implementacijom postupaka praćenja zrake na GPU na konkretnom je primjeru razmotreno kako se i koliko učinkovito različiti problemi mogu prilagoditi i optimizirati za izvođenje na grafičkom procesoru.

Ključne riječi: CUDA, GPU, GPGPU, praćenje zrake, bacanje zrake, zrcaljenje, lom svjetlosti, strukture za ubrzavanje

Abstract

Ray tracing on graphics processor

This paper examines the architecture of modern graphics processors with special attention devoted to CUDA architecture. CUDA is explained from both hardware and API aspects and its advantages and disadvantages are further analyzed.

This paper also covers the basic ray tracing method and special effects that it can produce, e.g. reflection and refraction.

By implementing the ray tracing algorithm on GPU a real example is given of how efficiently different problems can be solved and optimized for GPU execution.

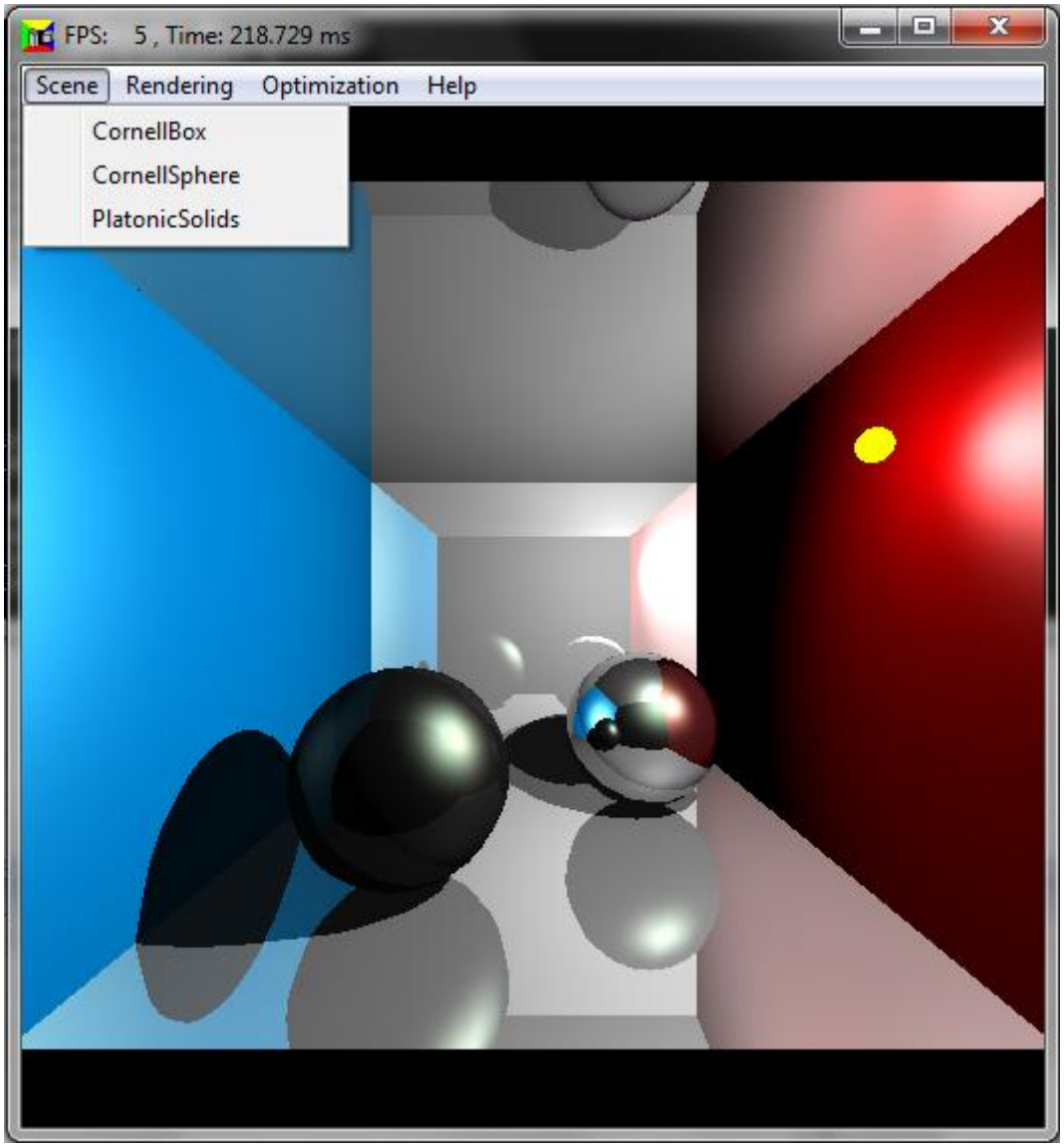
Keywords: CUDA, GPU, GPGPU, ray tracing, ray casting, reflection, refraction, acceleration structures

Privitak A

Programsko rješenje

Uz diplomski rad priloženo je programsko rješenje metode praćenja zrake na GPU.

Program radi na grafičkom sučelju Direct3D10 i grafičkim karticama s podrškom za arhitekturu CUDA.



Slika 29: Korisničko sučelje programa

Program omogućava odabir jedne od 3 predodređene scene iz izbornika *„Scene“*.

U izborniku *„Rendering“* odabire se način iscrtavanja koji može biti iscrtavanje sa zrcaljenjem svjetla (dubine 1, 2 ili 3) te iscrtavanje sa zrcaljenjem i lomom svjetla.

I na kraju moguće je odabrati metodu optimizacije s obujmicama iz izbornika *„Optimization“*.

Pogled kamere može se mijenjati strelicama gore, dolje, lijevo i desno, te tipkama 'W', 'S', 'A', 'D', 'Q' i 'E'.