



UNIVERZITET U NOVOM SADU,
PRIRODNO-MATEMATIČKI FAKULTET

DEPARTMAN ZA
MATEMATIKU I
INFORMATIKU

Trg Dositeja Obradovića 4, 21000 Novi Sad



Implementacija i poređenje *raytracing* algoritama za renderovanje vokselnih objekata

Master rad

Mentor:

dr Marko Savić

Student:

dr Nikola Vetnić 82m/23

Novi Sad, 2024

Predgovor

Predmet ovog rada je istraživanje efikasnijih načina renderovanja velikih vokselnih objekata, i to počevši od *brute force* pristupa, preko inkrementalno optimizovanih rešenja, sve do algoritama koji primenjuju *grid marching* tehniku i onih koji koriste oktalna stabla kao način čuvanja informacije o modelima. Svi opisani pristupi biće implementirani i testirani na različitim vokselnim objektima visoke rezolucije i vremena renderovanja svake od njih biće upoređena jedna sa drugim.

Očekivanje istraživanja je da će predloženi algoritmi višestruko skratiti vreme prikaza trodimenzionalnih vokselnih objekata. Stečeno znanje primenjivo je pre svega u obrazovanju, naročito u slučaju studenata osnovnih i master akademskih studija koji su se pretežno bavili Java programskim jezikom, ali i svih ostalih koji su zainteresovani za računarsku grafiku, pisanje *ray tracing* programa i generisanje i renderovanje objekata opisanim vokselima.

Želim da se ovim putem zahvalim supruzi Viktoriji i sinovima Vsevolodu i Istoku za podršku, stpljenje i ljubav u svakom trenutku, a naročito tokom poslednjih pet godina od kada sam odlučio da se vratim prirodnim naukama, zatim majci Štefaniji i ocu Vojislavu, kojem smatram da ovakav rad već predugo dugujem, i konačno dragom prijatelju i mentoru dr Marku Saviću, kojeg sam imao sreću da upoznam već na početku studija i koji me je i naučio da programiram.

SADRŽAJ

1 Uvod	7
1.1 Object Order i Image Order algoritmi	7
1.2 Ray Tracing algoritam	9
1.3 Vokselni prostor	9
2 Osnove ray tracing algoritma	11
2.1 Opis zraka i računanje preseka sa geometrijskim telima	12
2.1.1 Presek zraka sa sferom. Analitičko rešenje	12
2.1.2 Presek zraka sa sferom. Geometrijsko rešenje	13
2.1.3 Poređenje analitičkog i geometrijskog rešenja	15
2.2 Presek zraka sa pravougaonom kutijom. Geometrijsko rešenje	15
2.3 Vokseli kao pravougaone kutije jednakih ivica.....	17
3 VoxelWorld	19
3.1 Opis vokselnih tela.....	19
3.1.1 Klasa BaseM.....	19
3.1.2 Klasa BasePF.....	21
3.1.3 Klasa BaseF	22
3.2 Opis preseka zraka sa vokselom	22
3.2.1 Klasa HitVoxel.....	23
3.2.2 Prenos informacije o boji pogodenog voksela.....	24
3.3 Opis algoritama za renderovanje vokselnih tela	25
3.3.1 Brute force algoritam i klasa BruteForce	25
3.3.2 Direction array algoritam i klase DirArray i DirArrayO	28
3.3.3 Grid march algoritam i klasa GridMarch1	32
3.3.4 Grid march algoritam i klase GridMarch2 i GridMarch2O.....	36
3.3.5 Octree brute force algoritam i klasa OctreeBF	42
3.3.6 Octree brute force algoritam i klase OctreeRec i OctreeRecO.....	45
4 Rezultati testiranja	49
4.1 Nasumično generisana vokselna kocka (6.25% popunjenošti).....	49
4.2 Nasumično generisana vokselna kocka (0.19% popunjenošti).....	54
4.3 Nasumično generisana vokselna kocka (96.87% popunjenošti).....	58
4.4 Mars MGS MOLA DEM, izometrijski detalj doline Ares	62

4.5 Mars MGS MOLA DEM, zapadni deo četvorougla Oxia Palus.....	69
5 Zaključak	71
Literatura	73
Spisak slika i grafika	75
Biografija	77
Ključna dokumentacijska informacija	79

Uvod

Računarska grafika je pojam koji se prema [1] u najširem smislu koristi za opis "bilo čega na računaru što nije tekst ili zvuk". Danas ovaj termin najčešće obuhvata različite tehnologije koje učestvuju u stvaranju i manipulaciji slika, metode digitalne sinteze i manipulacije slikovnim sadržajem, i konačno prikaz, odnosno manipulaciju slikom predstavljenom u vidu podataka na računaru. Kako se navodi u [1], poslednjoj grupi pripada i tzv. "crtanje na računaru", odnosno renderovanje (eng. *rendering*), iliti prikazivanje, različitih trodimenzionalnih objekata na računarskom ekranu.

Postupak renderovanja uključuje učitavanje scene sačinjene od geometrijskih tela pozicioniranih u trodimenzionalnom prostoru i potonje generisanje dvodimenzionalne slike koja predstavlja pogled na tu scenu iz određene tačke. Na ovaj način se primenom matematičkog i fizičkog modelovanja kao rezultat dobija vredni prikaz objekata u prostoru uzimajući u obzir kako svojstva samih objekata tako i sredine u kojoj se nalaze.

1.1 Object Order i Image Order algoritmi

Prema Maršneru (*Steve Marschner*) i Širliju (*Peter Shirley*) [2, pp. 79], renderovanje u užem smislu podrazumeva obradu i pretvaranje ulaznih podataka datih u vidu kolekcije objekata u prostoru u dvodimenzionalnu matricu piksela na izlazu koja se prikazuje kao slika na ekranu. Proces se u suštini zasniva na utvrđivanju koliko svaki objekt ponaosob utiče na svaki od piksela, što je problem kojem se istorijski prilazio na dva načina.

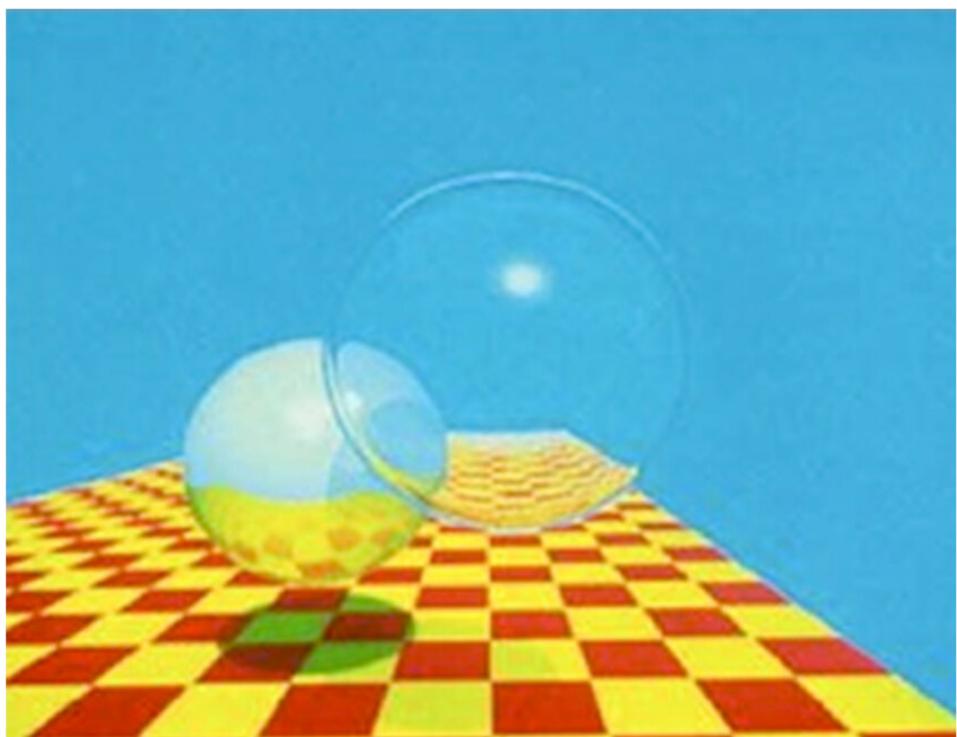
Prvi način iznedrio je grupu algoritama koji prvenstvo daju obradi čitavih objekata i njihovom prikazu na ekranu pre nego što se pređe na sledeći u kolekciji (eng. *object order algorithms*). Na ovaj način se tokom renderovanja scene elementi pojavljuju jedan po jedan, od najdaljeg ka najbližem, tako da ukoliko dođe do preklapanja dva objekta na dvodimenzionalnoj slici vrednost piksela izračunata tokom renderovanja prednjeg objekta prepisuje do tada upisanu vrednost.

Šreder (*Will Schroeder*), Martin (*Ken Martin*) i Lorensen (*Bill Lorensen*) navode u [3, pp. 35] da su se istorijski posmatrano algoritmi koji pripadaju ovoj grupi pojavili prvi, delimično i zbog toga što je ovakav pristup prirodniji prilikom kreiranja likovnih prikaza na tradicionalan način, poput crtanja na papiru ili slikanja na platnu. S druge strane, rani vektorski ekrani bili su veoma pogodni za ovakav način renderovanja, pa je zato bilo prirodno preneti već postojeća rešenja na rasterske izlazne uređaje kada su oni postali pristupačni i dovoljno rasprostranjeni. Danas se ovi algoritmi i dalje koriste u *real-time* grafici, ali ređe za fotorealistično renderovanje. Razlog tome leži u činjenici da su opravdani samo u specifičnim slučajevima, poput situacije kada se na sceni nalazi značajno manje objekata nego piksela u izlaznoj dvodimenzionalnoj matrici, ili kada se koristi namenski hardver za renderovanje.

Drugi način podrazumeva iteriranje po pikselima izlazne matrice (eng. *image order algorithms*), koji izračunavaju uticaj svakog od prisutnih objekata na trenutni piksel i tako ažuriraju matricu. Problem pozicioniranja objekata po dubini scene se u ovom slučaju rešava posmatranjem

trenutaka preseka zraka sa svakim od objekata, što je, kako navode Maršner i Širli u [2, pp. 79], najčešće implementirano kroz prekid iteriranja već pri prvom preseku zraka sa nekim objektom.

Kako je snaga računara rasla sa razvojem hardvera, postajalo je sve jasnije da mogućnosti koje postaju dostupne višestruko prevazilaze jednostavnu grafiku ranih vektorskih ekrana zasnovanu pretežno na linijama. Januara 1980. objavljen je rad Tarnera Viteda (*Turner Whitted*) na temu alternativnog pristupa renderovanju trodimenzionalne grafike. Vitedov rad se zasnivao na konstruisanju stabla zraka koji se rasprostiru od posmatrača do površine prvog objekta na sceni sa kojim se sudaraju, i koje uzima u obzir globalnu informaciju o osvetljenju, odnosno meru toga koliko se svetlosti reflektuje ka posmatraču sa vidljive tačke na površini tela, pri računanju vrednosti svakog piksela [4]. Vited je u radu opisao implementaciju programa koji verno simulira refleksiju, senke i refrakciju, uz primenu anti-alijasinga (eng. *anti-aliasing*)¹ na tela koja se sastoje kako od krivih površina, tako i poligona. Slika jednog od proizvedenih renderinga prikazana je na slici 1.



Slika 1. Vitedov originalni rendering kao rezultat algoritma opisanog u radu iz 1980. godine, prikazan u [5].

Algoritmi obe grupe u stanju su da renderuju identične slike, iako se značajno razlikuju u načinu implementacije pojedinih efekata i performansama prilikom izvršavanja. Prema Maršneru i Širliju [2, pp. 79], algoritmi koji prednost daju objektima i dalje su popularni prvenstveno zahvaljujući postojanju širokog spektra specijalizovanog hardvera dizajniranog za brzo renderovanje objekata. Algoritmi koji iteriraju po pikselima, što uključuje i one zasnovane na praćenju zraka (eng. *ray tracing*), ne zahtevaju namenski hardver, uopšte uvezvi su lakši za implementaciju i prilagodljiviji različitim efektima, ali su za uzvrat zahtevniji i značajno se sporije izvršavaju. [4]

¹ U obradi signala i srodnih disciplina, alijasing je efekat koji čini da različiti signali budu isti tokom diskretizovanja signala. Takođe se odnosi na izobličenje ili grešku koja proističe kada se signal rekonstruiše iz uzorka koji se razlikuju od originalnog kontinuiranog signala.

1.2 Ray Tracing algoritam

Ubrzo nakon objavljivanja Vitedovog rada dolazi do značajnog napretka računarskog hardvera, kao i dalje optimizacije *ray tracing* algoritama, što je za rezultat imalo približavanje računarske grafike širokim krugovima entuzijasta i konzumenata i apsolutnu dominaciju ovog pristupa u domenu renderovanja trodimenzionalne grafike. Kako navodi Safern (*Kevin Suffern*) u [6, pp. ix], rani rezultati *ray tracing* renderinga bili su karakteristični po izobilju najrazličitijih lopti, budući da je sferu od svih tela bilo najlakše opisati te je stoga bila i najjeftinija za renderovanje u pogledu korišćenja računarskih resursa.

Prema rečima Maršnera i Širlija u [2, pp. 80], program koji primenjuje *ray tracing* algoritam prolazi kroz izlaznu matricu i izračunava boju piksela jedan po jedan. Za svaki piksel potrebno je pronaći objekat koji se vidi sa njegove pozicije na slici, odnosno potrebno je pronaći presek zraka koji izlazi iz oka posmatrača, odnosno kamere, i kreće se u pravcu piksela na izlaznoj matrici. Traženi objekat uvek je prvi sa kojim se zrak sudara, budući da on zaklanja od kamere bilo koji objekat sa kojim bi se zrak dalje sudarao. Nakon pronalaženja objekta i tačke preseka, računa se nijansa boje tela za date parametre uzimajući u obzir poziciju tačke na površini objekta, normale na površinu u tački preseka, i druge informacije.

Prilikom traženja preseka od značaja su način predstavljanja kako zraka tako i objekta. Zrak je jednostavno definisan svojom polaznom tačkom (odnosno pozicijom kamere) i smerom, dok su objekti najčešće opisani ili kao geometrijski primitivi ili njihovi derivati. Ovakva složena tela najčešće nastaju primenom tehnike konstruktivne stereometrije (eng. *constructive solid geometry*), i tada se presek jednostavno računa ili rešavanjem sistema jednačina zraka i tela, ili vektorskim putem.

1.3 Vokselni prostor

Alternativu opisu trodimenzionalnih objekata zasnovanom na geometrijskim primitivima, odnosno konstruktivnoj stereometriji, predstavlja opis tela na osnovu kolekcije voksela koje to telo zauzima. Vokseli (eng. *voxel*, u prevodu "element zapreme", od eng. *volume* – zapremina, i *element* – elemenat), prema definiciji Džavida u [7], predstavljaju diskrette tačke uzorkovanja na pravilnoj trodimenzionalnoj mreži opisane minimalno pozicijom u prostoru, dok vokselni objekti predstavljaju objekte opisane minimalno kolekcijom voksela koje zauzimaju.

U smislu sredstva za čuvanje informacija o geometrijskim telima, mana voksela se ogleda pre svega u većim zahtevima u pogledu memorijskog prostora nego što je to slučaj sa već razmotrenim alternativama. Ipak, velika prednost je daleko veća raznovrsnost modelovanih objekata, kao i lakoća primene različitih algoritama za proceduralno generisanje velikih ili čak beskonačnih modela (korišćenjem različitih tehnika, poput Perlinovog šuma) koje je takođe moguće jednostavno modifikovati. Upravo lakoća modifikacije vokselnih modela doprinela je njihovom ranom usvajajuju od strane industrije video igara.

Zapremina opisana vokselima najčešće se renderuje indirektno, i to ekstrahovanjem izopovršina poligona koji prate konture zadatih graničnih vrednosti – u tu svrhu se često koristi *marching cubes* algoritam, iako postoje i drugi metodi. Kada se u *ray tracer* programima vokseli renderuju direktno najčešće je reč o kockama jedinične zapremine, te se za prikaz vokselnih scena mogu koristiti uobičajeni metodi za računanje preseka zraka sa kockom. Međutim, veličina kolekcija voksela potrebnih za verno predstavljanje detaljnih objekata čini ovaj pristup neefikasnim, što upućuje na potrebu za boljim algoritmima.

U nastavku rada biće istraženi efikasniji načini renderovanja velikih vokselnih objekata, i to počevši od *brute force* pristupa, preko inkrementalno optimizovanih rešenja, sve do algoritama koji primenjuju *grid marching* tehniku i onih koji koriste oktalna stabla kao način čuvanja informacije o modelima. Sve implementacije biće testirane na različitim vokselnim objektima visoke rezolucije i vremena renderovanja svake od njih biće upoređena jedna sa drugim.

Očekivanje istraživanja je da će predloženi algoritmi višestruko skratiti vreme prikaza trodimenzionalnih vokselnih objekata. Stečeno znanje primenjivo je pre svega u obrazovanju, naročito u slučaju studenata osnovnih i master akademskih studija koji su se pretežno bavili Java programskim jezikom, ali i svih ostalih koji su zainteresovani za računarsku grafiku, pisanje *ray tracing* programa i generisanje i renderovanje objekata opisanim vokselima.

Osnove *ray tracing* algoritma

Ray tracing algoritam funkcioniše tako što izračunava piksele jedan po jedan, gde je osnovni zadatak pronalaženje objekta koji se vidi sa pozicije svakog od piksela na slici. Pravac "posmatranja" različit je za svaki piksel² – objekat koji piksel "vidi" mora se presecati za zrakom posmatranja (eng. *viewing ray*), odnosno linijom od kamere u pravcu u kom piksel "gleda". Prema Maršneru i Širliju u [2, pp. 80], na ovaj način se traži objekat koji prvi preseca zrak posmatranja, odnosno koji je najbliži kameri, budući da on zaklanja pogled na bilo šta što se nalazi iza njega.

Nakon pronalaženja objekta sa kojim se zrak seče, primenjuje se izračunavanje senčenja³ (eng. *shading*) kako bi se izračunala konačna boja, odnosno vrednost piksela. Izračunavanje senčenja u obzir uzima tačku preseka, normalu na površinu u tački preseka, teksturu, kao i potencijalno mnoge druge parametre koji zavise od željenog tipa renderovanja.

Ray tracing program se tako sastoji od tri osnovna dela:

1. generisanja zraka, čime se računa početna tačka i smer zraka posmatranja za svaki od piksela na osnovu geometrije kamere,
2. preseka zraka, čime se pronalazi objekat najbliži kameri koji se nalazi na putanji zraka kroz zadati piksel, i
3. senčenja, kojim se izračunava boja piksela na osnovu rezultata dobijenih izračunavanjem preseka zraka sa telom na sceni.

Ray tracer program koji odgovara opisanoj strukturi može se, prema Safernou u [6, pp. 46], opisati pseudokôdom datom u listingu 1.

```
define objects (with associated materials)
define light sources
define a view plane (a pixel matrix)

for each pixel
    direct a ray from camera towards the center of the pixel
    compute the hit point of the ray with the nearest object
    if the ray lands
        use object's material and lights to compute pixel color
    else
        set pixel color to default
```

Listing 1. Pseudokôd jednostavnog *ray tracer* programa. [6]

Zraci koji se simuliraju *ray tracing* algoritmima razlikuju se od stvarnih svetlosnih zraka, odnosno fotona, po tome što se ispaljuju iz kamere i staju pri prvom susretu sa objektom. Safern u

² Ovo važi kod projekcija koje uzimaju u obzir perspektivu. Kod projekcija koje je zanemaruju (poput ortogonalne) zraci kroz sve piksele su paralelni.

³ Senčenje u računarskoj grafici, prema internet resursu Scratchapixel [8], označava izračunavanje boje trodimenzionalnog objekta na sceni, odnosno simuliranje boje objekta na način na koji se ona vidi sa pozicije kamere.

[6, pp. 46] razlog za odabir ovakvog pristupa ilustruje korišćenjem kamere predstavljene tačkom (eng. *pinhole camera*) – kada bi se zraci modelovali tako da od izvora svetlosti putuju ka kameri, usled problema sa aritmetikom sa pokretnim zarezom gotovo nijedan zrak ne bi nikad završio tačno u kameri. Zbog toga, emitovanje zraka iz kamere predstavlja jedini praktičan način za renderovanje korišćenjem *ray tracing* algoritma.

2.1 Opis zraka i računanje preseka sa geometrijskim telima

U središtu svakog *ray tracer* programa Hejns (*Eric Haines*) et al. [9, pp. 33-34] vide skup metoda za računanje preseka zraka iz kamere sa objektima na sceni. Vrsta povratnih informacija zavisi od namene zraka: zraci koji se koriste za ispitivanje preseka sa tzv. ograničavajućim zapreminama⁴ (eng. *bounding volume*) obično vraćaju samo informaciju o pogotku ili odsustvu istog, dok zraci emitovani iz kamere vraćaju minimalno najbližu tačku preseka i normalu u toj tački, a često i poziciju tačke u referentnom sistemu površine tela sa kojim se zrak seče što je od značaja prilikom mapiranja tekstura.

Zrak je prema Maršneru i Širliju u [2, pp. 83-84] predstavljen početnom tačkom (odnosno pozicijom kamere) i pravcom propagacije zraka, zbog čega se zrak idealno predstavlja trodimenzionalnom parametrijskom linijom. U vektorskom obliku se zraci zapisuju kao

$$R(t) = R_0 + t(S - R_0). \quad (2.1)$$

Zrak se može shvatiti kao pravolinijsko kretanje od tačke R_0 ka tački S , gde je $R(t)$ tačka na toj pravoj koja se nalazi na t -ostrukoj udaljenosti u pravcu $(S - R_0)$, računato od R_0 . Početna tačka zraka je tada R_0 , a njegov pravac $(S - R_0)$, što se često obeležava sa d . Iako tačnost izračunavanja ne zavisi od toga, d je često normalizovan kako vrednost t ne bi zavisila od intenziteta vektora pravca.

Iz (2.1) proizilazi da je $R(0)$ početna tačka zraka⁵, $R(1)$ je tačka na izlaznoj matrici piksela (u slučaju da je S na ravni projekcije), kao i da je za $0 < t_1 < t_2$ tačka $R(t_1)$ bliža kameri od $R(t_2)$, što je na primer značajno prilikom računanja svih mogućih preseka na sceni kod stereometrijskih tela.

U slučaju kada je t negativno tačka preseka se nalazi iza kamere.

2.1.1 Presek zraka sa sferom. Analitičko rešenje

Kao najjednostavnije telo za računanje preseka sa zrakom, sfera je jedan od najčešće korištenih primitiva u trodimenzionalnoj grafici, kako u okviru scene, tako i u vidu pomoćnih struktura poput ograničavajućih zapremina.

Analitički sfera se određuje sledećom jednačinom

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = S_R^2, \quad (2.2)$$

odnosno sfera sa centrom C prečnika S_R predstavlja skup tačaka čije pozicije u pravouglom koordinatnom sistemu zadovoljavaju (2.2).

⁴ Ograničavajuća zapremina se definiše za skup objekata kao zatvoren prostor koji u potpunosti sadrži uniju svih objekata u skupu.

⁵ Kako navode Hejns et al. u [9, pp. 46-47], tačka $R(0)$ ne pripada zraku usled problema sa preciznošću aritmetike sa pokretnim zarezom.

Ukoliko se x , y i z posmatraju kao tačka S , sfera se može izraziti jednačinom

$$|S - S_C|^2 - S_R^2 = 0, \quad (2.3)$$

koja je primer implicitne jednačine (površine) tela. Prema internet resursu *Scratchapixel* [10], implicitno definisana tela u potpunosti su određena svojim implicitnim jednačinama površine, po čemu se razlikuju od daleko poznatijih tela opisanih povezanim trouglovima.

Na implicitan način moguće je definisati ne samo sfere, već i ravni, različite kvadrike (valjci, kupe, i drugi), piramide, i kocke. Primenom konstruktivne stereometrije moguće je od primitiva dobiti i veoma složena trodimenzionalna tela.

Kada dođe do preseka zraka sa površinom sfere jedna od tačaka na zraku zadovoljiće i njenu implicitnu jednačinu, tako da važi

$$|R_0 + td - S_C|^2 - S_R^2 = 0, \quad (2.4)$$

što nakon kvadriranja daje

$$t^2 d^2 + 2 R_0 t d + R_0^2 - S_R^2 = 0, \quad (2.5)$$

što je očigledno kvadratna jednačina oblika $At^2 + Bt + C = 0$. Kada je diskriminanta negativna ova jednačina nema rešenja te zrak ne preseca sferu, kada je diskriminanta jednaka 0 postoji jedno rešenje i tada je zrak tangenta na površinu sfere⁶, a u slučaju pozitivne diskriminante i prisutnosti dva rešenja zrak ulazi u sferu u jednoj i izlazi u drugoj tački na površini.

Kada je poznata vrednost t poznata je i tačka preseka na površini sfere. Normala u tački preseka tada je

$$n = (R(t) - S_C) / \|R(t) - S_C\|. \quad (2.6)$$

Normalizacija je u ovom slučaju neophodna.

2.1.2 Presek zraka sa sferom. Geometrijsko rešenje

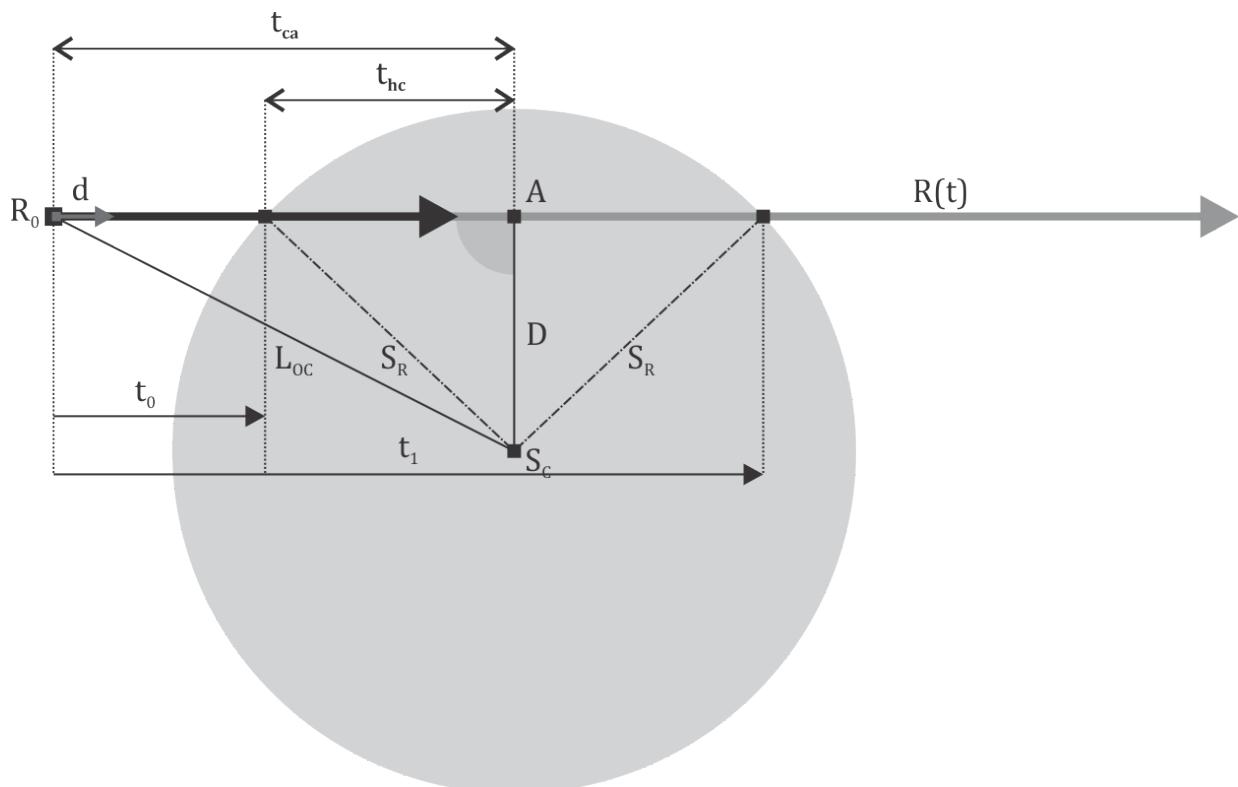
Izračunavanje rešenja na ovaj način je nažalost dosta sporo. Hejns et al. u [9, pp. 41] navode da je operacija kvadratnog korenovanja koja je neizostavni deo rešavanja kvadratne jednačine 15-30 puta zahtevnija od množenja, što takođe, iako u manjoj meri, važi i za deljenje.

Takođe, izračunavanje preseka moguće je obustaviti na određenim mestima ukoliko je očigledno da zrak ne pogađa sferu, što vodi ka daljoj optimizaciji algoritma. Prema slici 2, jasno je da do preseka ne dolazi ukoliko zrak nije usmeren ka sferi. Uvezši ovu i slične činjenice u obzir, moguće je definisati sledeću strategiju pronalaženja preseka zraka sa sferom:

1. proveriti da li se početna tačka zraka nalazi unutar sfere – ako se nalazi, do preseka će jednoznačno doći;

⁶ Detekcija ovakvog preseka zraka sa telom je problematična usled problema sa preciznošću aritmetike sa pokretnim zarezom. U implementaciji koja je deo ovog rada ovakav tangentni preseci se ne razmatraju pri renderovanju prikaza.

2. pronaći rastojanje od početne tačke zraka do tačke na zraku koja je najbliža centru sfere;
3. ukoliko je zrak van sfere, proveriti da li je usmeren ka sferi – ukoliko nije, do preseka jednoznačno ne dolazi;
4. ukoliko je zrak usmeren ka sferi, pronaći kvadrat udaljenosti do najbliže tačke na površini sfere;
5. ukoliko je rezultujuća vrednost negativna, do preseka ne dolazi;
6. pronaći udaljenost do preseka zraka sa površinom, odnosno parametar t ;
7. izračunati poziciju preseka i normalu u presečnoj tački.



Slika 2. Geometrijsko rešenje računanja preseka zraka sa sferom, dato u [9, pp. 42].

Na ovaj način zapravo se kvadratna jednačina (2.5) deli na kraće izraze koji se izračunavaju po potrebi.

Da li je početna tačka zraka unutar sfere jednostavno se proverava računanjem kvadrata dužine vektora od početne tačke zraka do centra sfere, i poređenjem te vrednosti sa kvadratom prečnika. Kvadrati rastojanja čuvaju poredak osnovnih vrednosti, dok istovremeno dozvoljavaju izbegavanje u smislu računarskih resursa zahtevnog korenovanja. Dodatna optimizacija postiže se čuvanjem kvadrata prečnika kao zasebne vrednosti u kôdu sfere.

Sledeći korak je pronalaženje vrednosti t_{CA} , odnosno rastojanje od početne tačke zraka do tačke na zraku koja je najbliža centru sfere, što se jednostavno računa skalarnim proizvodom vektora R_0S_C i d , što predstavlja intenzitet vektora R_0S_C projektovanog na pravac d .

Ukoliko je rezultujuća veličina t_{CA} negativna tada se zrak udaljava od sfere, odnosno ne dolazi do preseka.

Dalje se primenom Pitagorine teoreme računa t_{HC}^2 kao razlika $S_R^2 - D^2$, što je kvadrat polovine dela zraka koji se nalazi u sferi, odnosno tetive sa središtem u tački A. Samo D^2 se izražava kao razlika $L_{OC}^2 - t_{CA}^2$, tako da važi

$$t_{HC}^2 = S_R^2 - L_{OC}^2 + t_{CA}^2. \quad (2.7)$$

Geometrijski smisao (2.7) jasno je prikazan na slici 2. Ukoliko je t_{HC}^2 manje od nule, tada je početna tačka zraka van sfere i zrak ne preseca sferu.

Konačno, parametar t u kom zrak preseca sferu dobija se iz

$$t = t_{CA} - t_{HC}$$

kada je početna tačka zraka van sfere, odnosno

$$t = t_{CA} + t_{HC}$$

kada je početna tačka zraka unutar sfere ili na njoj. Kvadratni koren se računa tek ukoliko do ovog koraka dođe. Normala na površinu sfere u presečnoj tački se izračunava pomoću

$$R_{norm} = (R(t) - S_C) / S_R.$$

2.1.3 Poređenje analitičkog i geometrijskog rešenja

Prema Hejnsu et. al u [9, pp. 44-45] oba rešenja daju tačan rezultat i približni su po broju izvršenih operacija. Prednost geometrijskog rešenja leži u pravovremenu odustajanju od izračunavanja u slučajevima kada je jasno da zrak ne pogda sferu.

Primer toga je prva provera koja se bavi time da li je zrak usmeren ka sferi – u određenim situacijama (kada se sfera koristi kao ograničavajuća zapremina, na primer) nasumično usmeren zrak će u polovini slučajeva promašiti telo, te se time značajno štedi u poređenju sa analitičkim pristupom koji nema ovakvu proveru.

Pažljivim pregledom moguće je primetiti određene paralele između različitih pristupa i tako optimizovati analitičko rešenje do nivoa ekvivalentnog geometrijskom. Ipak, ovo je moguće samo zahvaljujući zaključcima proizašlim iz geometrijskog rešenja problema preseka.

Osnovna pouka iz navedenog je da proučavanje suštine problema može rezultovati optimizacijom algoritma. Prednost analitičkog pristupa je generalizacija – na isti način moguće je izračunati presek zraka sa bilo kojim telom opisanim implicitnom jednačinom površine tela. Međutim, geometrijsko rešenje nalazi dublje u samu prirodu problema, razoktriva njegove osobnosti i tim samim nudi poboljšanja u njegovom rešavanju, što je u računarskoj grafici od velikog značaja.

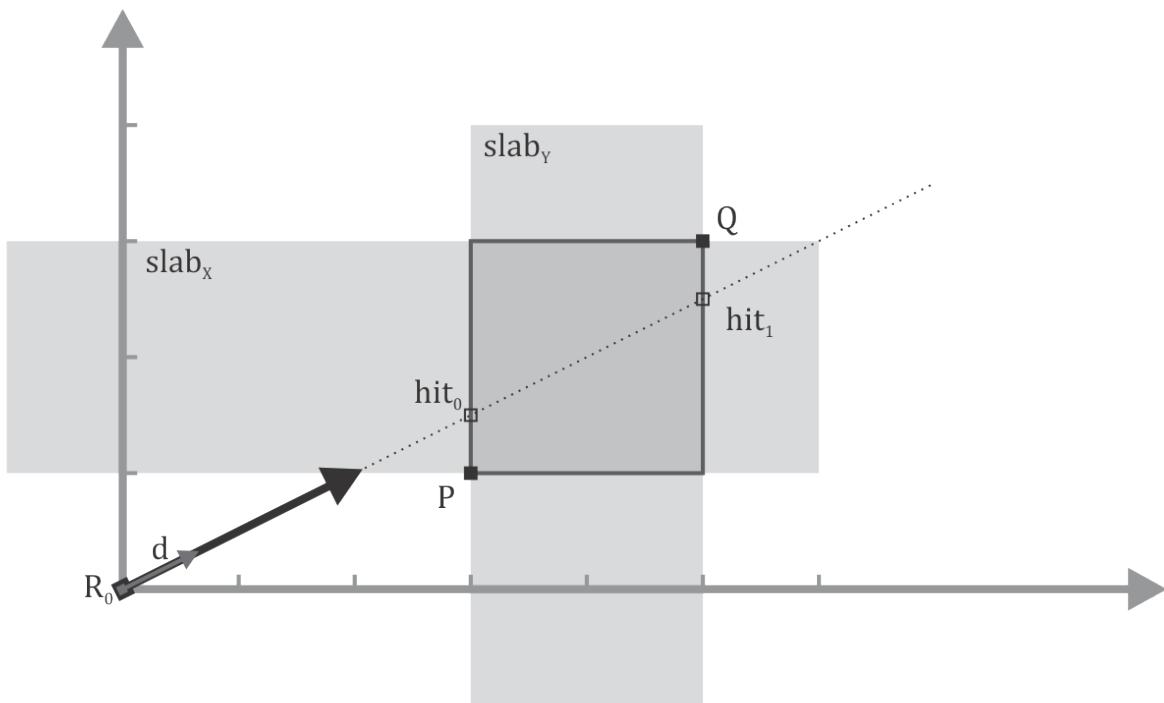
2.2 Presek zraka sa pravougaonom kutijom. Geometrijsko rešenje

Presek zraka sa pravougaonim kutijama (eng. *box*) biće stoga posmatran isključivo kroz prizmu geometrijskog rešenja.

Kej (*Timothy Kay*) i Kadžija (*James Kajiya*) u [11] predlažu efikasan način implementacije ovakvih objekata zasnovanog na pločama (eng. *slabs*), gde je ploča definisana kao prostor između

dve paralelne ravni, dok je kutija telo definisano presekom tri takve ploče poravnatim po X, Y i Z osama.

Presek sa zrakom se tada računa posmatranjem preseka sa svakim parom ravni koje definišu ploču, pazeći pritom na poredak preseka. Kao što se vidi na slici 3, ukoliko je najveća vrednost bližeg preseka veća od najmanje vrednosti daljeg preseka, tada zrak ne pogađa telo.



Slika 3. Geometrijsko rešenje računanja preseka zraka sa kutijom.

Kutija kao presek tri ploče može se definisati pomoću dva suprotna temena P i Q, odnosno kroz minimalno i maksimalno teme. Presek zraka sa svakom od ploča tada se izračunava kao

$$T_P = (P - R_0) / d,$$

$$T_Q = (Q - R_0) / d,$$

gde je prvi presek sa pločom paralelnom sa YZ, ZX, odnosno XY ravnima u komponentama tačke T_P , a drugi presek u analognim komponentama tačke T_Q . Operacija deljenja, odnosno $/$, u ovom slučaju je definisana kao deljenje svake dimenzije vektora odgovarajućom dimenzijom drugog vektora.

Koristeći ove dve tačke kreiraju dva nova vektora t_0 i t_1 , gde prva sadrži minimalne vrednosti X, Y, odnosno Z komponenti $R(t_P)$ i $R(t_Q)$, dok druga sadrži maksimalne vrednosti istih komponenti. Drugim rečima, tačka t_0 predstavlja ulazak u prostor preseka tri ploče kojim je definisana kutija, dok t_1 predstavlja izlazak iz kutije.

Dalje se traži maksimalna komponenta t_0 , odnosno minimalna t_1 , budući da je između ta dva trenutka zrak sigurno unutar kutije. U oba slučaja se čuvaju indeksi pronađenih komponenti kao $iMax_0$ i $iMin_1$. Prema Keju i Kadžiji u [11], ukoliko je max_0 veće od min_1 , tada ne dolazi do preseka zraka sa kutijom.

U slučaju preseka vraća se \max_0 kada je početna tačka zraka van kutije, pa zrak pri prvom susretu sa telom u njega i ulazi, odnosno \min_1 kada je početna tačka unutar kutije i zrak iz nje pri prvom kontaktu izlazi.

Normala u tački preseka računa se uzimanjem komponente jediničnog vektora sa indeksom $i\max_0$ u prvom, odnosno $i\min_1$ u drugom slučaju, nakon čega se množi invertovanim znakom komponente pravca zraka sa istim indeksom u prvom, odnosno znakom komponente sa istim indeksom u drugom slučaju.

Na ovaj način izvedena je i osnovna implementacija preseka zraka i kutije u *ray tracer* programu koji je predmet ovog rada, što se može videti na listingu 2.

```

01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      Vec3 tP = p().sub(ray.p()).div(ray.d());
05      Vec3 tQ = q().sub(ray.p()).div(ray.d());
06
07      Vec3 t0 = Vec3.min(tP, tQ);           Vec3 t1 = Vec3.max(tP, tQ);
08      int iMax0 = t0.maxIndex();           int iMin1 = t1.minIndex();
09      double max0 = t0.get(iMax0);         double min1 = t1.get(iMin1);
10
11      if (max0 < min1) {
12
13          if (max0 > afterTime) return HitData.tnu(
14              max0,
15              Vec3.E[iMax0].mul(-Numeric.sign(ray.d().get(iMax0))),
16              Vector.xy(max0, min1));
17
18          if (min1 > afterTime) return HitData.tnu(
19              min1,
20              Vec3.E[iMin1].mul(Numeric.sign(ray.d().get(iMin1))),
21              Vector.xy(max0, min1));
22      }
23
24      return Hit.POSITIVE_INFINITY;
25  }

```

Listing 2. Presek zraka sa kutijom, implementacija dr Marka Savića. [12]

2.3 Vokseli kao pravougaone kutije jednakih ivica

Vokseli se kao elementi zapremine takođe mogu opisati presekom tri ploče jedinične debljine poravnate po X, Y i Z osama, pa se tako i u prethodnom potpoglavlju opisani algoritam za presek zraka sa kutijom može primeniti i na voksele.

Ovakav naivan pristup svakako nije idealan. Budući da su vokseli uvek jedinične kocke, nije potrebno čuvati informacije o dva suprotna temena – dovoljno je pamtitи poziciju voksla u prostoru da bi se voksel mogao prikazati na sceni.

Trodimenzionalni objekti mogu se, kako je već rečeno, opisati vokselima, u kom slučaju se vokseli ne ponašaju kao zasebna tela već kao sastavni elementi veće celine i tada je moguće implementirati niz optimizacija u algoritmu za detekciju preseka zraka sa ovako opisanim telom, čime se postiže značajno skraćivanje vremena potrebnog za renderovanje scene. Pored toga, ovakav pristup je smisleniji, budući da se čitavo telo posmatra kao celina, a ne kao skup odvojenih zapreminske elemenata.

Konačno, korišćenjem određenih struktura podataka, poput oktalnih stabala, moguće je vokselna tela opisati na takav način da se preseći sa zracima u specifičnim slučajevima isključuju u veoma ranoj fazi i time vreme renderovanja dodatno skraćuje.

VoxelWorld

VoxelWorld predstavlja implementaciju vokselnih tela kao i algoritama za renderovanje istih u okviru većeg *ray tracing* programa *GfxLab-2020-2021* namenjenog demonstracijama i praktičnom radu tokom nastave predmeta Računarska grafika 2 na master studijama, a čiji je autor dr Marko Savić [12]. Sav kôd koji je predmet ovog rada takođe se nalazi u istom repozitorijumu.

Napisan u Java programskom jeziku, reč je o programu nastalom za vreme nastave na predmetu Računarska grafika 2 tokom zimskog semestra 2020/2021 akademske godine. Implementacija dr Savića u konačnoj verziji sadržala je algoritme za presek zraka sa različitim geometrijskim primitivima, anti-aliasing, affine transformacije, spekularne, reflektivne i refraktivne površine, konstruktivnu stereometriju, parametrizaciju površine i teksturisanje, i konačno model globalnog osvetljenja scene.

VoxelWorld se tako oslanja na program *GfxLab-2020-2021*, pre svega u domenu implementacije kamere, afinih transformacija i globalnog osvetljenja, koncentrišući se na generisanje i opis vokselnih tela i efikasno renderovanje istih.

3.1 Opis vokselnih tela

Vokselna tela opisana su klasama u paketu `voxelworld.a`, što ukazuje na činjenicu da su apstraktne i da ih nasleđuju klase koje implementiraju pojedinačne algoritme za renderovanje vokselnih tela koji će detaljno biti opisani u nastavku.

3.1.1 Klasa BaseM

Sva geometrijska tela u *ray tracing* programu *GfxLab-2020-2021* implementiraju interfejs `Solid` koji propisuje implementaciju metoda `hits`. Od ovog metoda se očekuje da vraća sve preseke zraka sa telom na sceni, i na njega se u podrazumevanoj (eng. *default*) implementaciji oslanja češće korišćen metod `firstHit` koji vraća prvi element niza čije je vreme veće od vrednosti `afterTime`, odnosno prvi presek zraka sa telom. Sam metod `firstHit` se u potklasama u svim slučajevima reimplementira tako da koristi geometrijsko rešenje preseka sa zrakom specifično za telo o kojem je reč, čega je cilj optimizacija pronalaženja preseka na načine slične onim koji su opisani za sferu i kutiju.

Apstraktna klasa `BaseM` predstavlja korensku klasu većine algoritama koji su predmet ovog rada, budući da se različiti algoritmi za pronalaženje preseka zraka sa telom zajedno sa strukturama za opis tog tela posmatraju kao različite implementacije `BaseM` klase, odnosno `Solid` interfejsa. Zbog toga `BaseM` metod `hits` ostavlja apstraktnim i prepušta njegovu implementaciju konkretnim potklasama zaduženim za pojedine algoritme.

Vokselna tela koja su instance klase `BaseM` opisana su četverodimenzionalnim nizom `boolean` tipa⁷, trodimenzionalnim nizom `Color` tipa i jednom `Vec3` vrednošću koje sadrži informacije o veličini tela, odnosno dimenzijske njegove ograničavajuće kutije. Klasa nudi konstruktore koji obavezno prihvataju trodimenzionalni, odnosno četvorodimenzionalni niz `boolean` vrednosti koji čuva informacije o prisustvu, odnosno odsustvu pojedinačnih voksela iz trodimenzionalne matrice koja sadrži vokselno telo, a opcionalno i trodimenzionalni niz `Color` vrednosti koji opisuju boju voksela na poziciji određenoj indeksima vrednosti. Klase `ModelData3` i `ModelData4` objedinjuju `boolean` i `Color` matrice i tako nude mogućnost prosleđivanja čitavog modela tela u vidu jedne promenljive.

Instance klase `BaseM` su nepromenljive (eng. *immutable*), te zato ne sadrže metode za modifikovanje vrednosti nakon instanciranja (eng. *set* metode). Što se tiče metoda za preuzimanje vrednosti (eng. *get* metode), postoje različiti načini za pristup dimenzijsama tela, proveru da li je voksel na određenoj poziciji prisutan u modelu, kao i preuzimanje boje voksela na prosleđenoj poziciji. Takođe, kroz odgovarajuće *get* metode omogućen je pristup i čitavoj `boolean`, odnosno `Color` matriци.

Pored opisanih metoda za pristup podacima klasa `BaseM` implementira i metode `getBoundingBoxHits` i `getDiffuseFromTerrainPalette` (listing 3).

```

01  private Color[][][] getDiffuseFromTerrainPalette(TerrainPalette p) {
02
03      Color[][][] output = new Color[lenX()][lenY()][lenZ()];
04
05      for (int i = 0; i < lenX(); i++) {
06          for (int j = 0; j < lenY(); j++) {
07              for (int k = 0; k < lenZ(); k++) {
08
09                  double h = 1.0 * k / lenZ();
10
11                  output[i][j][k] = p.colors()[5];
12
13                  if (h < p.heights()[4]) output[i][j][k] = p.colors()[4];
14                  if (h < p.heights()[3]) output[i][j][k] = p.colors()[3];
15                  if (h < p.heights()[2]) output[i][j][k] = p.colors()[2];
16                  if (h < p.heights()[1]) output[i][j][k] = p.colors()[1];
17                  if (h < p.heights()[0]) output[i][j][k] = p.colors()[0];
18              }
19          }
20      }
21
22      return output;
23  }
```

Listing 3. Metod `getDiffuseFromTerrainPalette` klase `BaseM`.

Metod `getBoundingBoxHits` računa presek zraka sa kutijom čije je minimalno teme u koordinatnom početku, a maksimalno u tački `len`, odnosno dimenzijsama ograničavajuće kutije vokselnog tela, i ovaj metod se koristi u određenim algoritmima za rani prekid izračunavanja ukoliko zrak u potpunosti promašuje telo na sceni.

⁷ Tela opisana `BaseM` klasom su u konačnom prikazu trodimenzionalna tela projektovana na dvodimenzionalni ekran. Četvrta dimenzija `boolean` niza igra ulogu prilikom čuvanja vokselnih tela u vidu oktalnih stabala o čemu će više biti rečeno u odeljku koji će se baviti samim algoritmom na bazi oktalnih stabala.

Metod `getDiffuseFromTerrainPalette` izračunava boju voksla u zavisnosti od njegove normalizovane visine izražene Z koordinatom elementa u matrici i prosleđene palete koja vokselima u određenom delu opsega [0, 1] dodeljuje određenu boju – u pitanju je postupak čija će primena biti demonstrirana kako kod proceduralno generisanog, tako i terena dobijenog čitanjem monohromatskih visinskih karti stvarnog terena.

3.1.2 Klasa BasePF

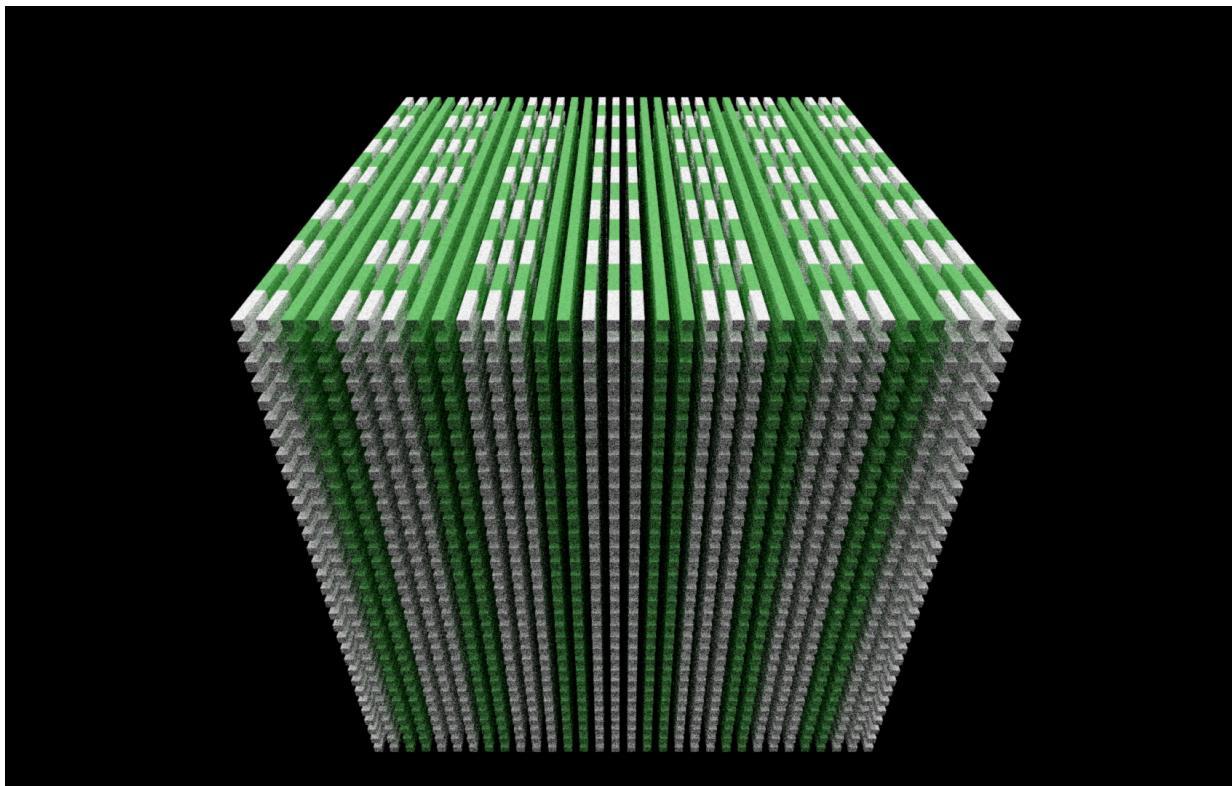
Apstraktna klasa `BasePF` veoma je slična već opisanoj klasi `BaseM` – i u ovom slučaju je reč o implementaciji `Solid` interfejsa koja predstavlja korensku klasu određenog broja algoritama koji su predmet ovog rada. Klasa sadrži konstruktoare i metode pristupa slične onima u `BaseM`, i takođe ostavlja `hits` metod apstraktним.

Glavna razlika tiče se opisa modela vokselnog tela koje u ovom slučaju nije predstavljeno diskretnim matricama `boolean` i `Color` vrednosti koje opisuju pozicije i boje voksla u modelu, već predikatom parametrizovanim `Vec3` tipom koji određuje poziciju, odnosno funkcijom parametrizovanom tipovima `Vec3` i `Color` koja određuje boju voksla.

Primer instanciranja vokselnog tela prosleđivanjem implicitnih definicija oblika i boje prikazano je na listingu 4, dok se grafički rezultat renderovanja takvog tela može videti na slici 4.

```
01  BasePF vo = PFGridMarch1.d
02      dim,
03      v -> v.xInt() % 2 == 0 && v.zInt() % 2 == 0,
04      v -> v.x() % 10 < 5 && v.yInt() % 10 < 5 ? c0 : c1);
```

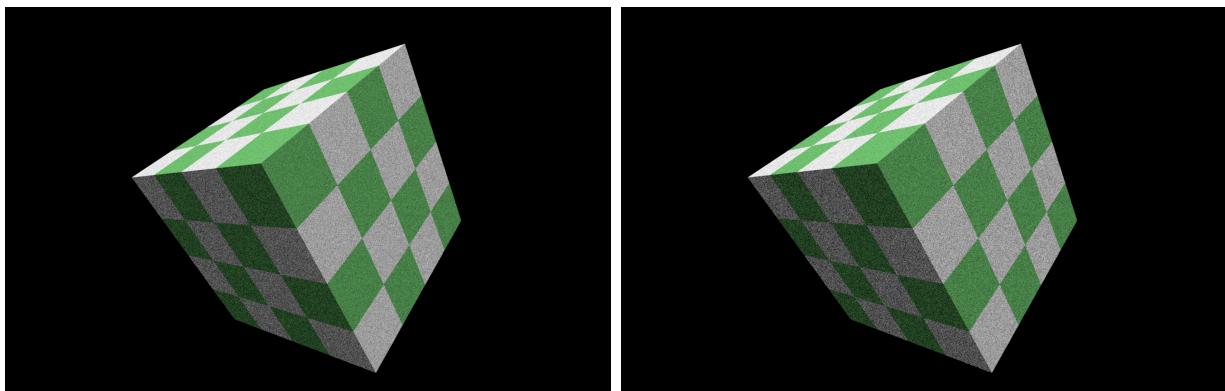
Listing 4. Implicitna definicija oblika i boje vokselnog tela.



Slika 4. Renderovanje implicitno definisanog vokselnog tela (10+ iteracija).

Klasa `PFGridMarch1`, koja je potklasa klase `BasePF` i koja predstavlja implementaciju *grid march* algoritma o kojem će kasnije biti više reči, se na listingu 4 instancira statičkim metodom `d` kojem se prosleđuju dimenzija tela `dim`, predikat koji opisuje sam model i funkcija na osnovu koje se izračunavaju boje voksela. Predikat dat na listingu 4 opisuje telo koje sadrži samo one voksele kojima su X i Z koordinate parne što rezultuje prazninama u modelu po širini i visini, dok funkcija naizmenično boji prisutne voksele u belo (vrednost `c0`) i zeleno (vrednost `c1`) u prugama širine 5 po X i Y dimenzijama tela.

Opis vokselnih tela na ovakav način u određenoj meri liči na parametrizaciju površine tela, odnosno određivanje UV koordinata presečne tačke zraka i tela, što su informacije koje se najčešće koriste za teksturisanje objekata.



Slika 5. Instanca klase `Box` tekstuirana kao vokselni objekat i instanca klase `BasePF` (10+ iteracija).

Iako je na ovaj način moguće postići efekat teksture objekta kao što se vidi na slici 5, reč je o postupku zahtevnijem za izračunavanje, koji u slučaju kada specifičnost oblika ne zahteva upotrebu vokselnih tela nije opravdan.

3.1.3 Klasa `BaseF`

Apstraktna klasa `BaseF` predstavlja kombinaciju pristupa primenjenih u `BaseM` i `BasePF`. Oba konstruktora klase `BaseF` primaju kao ulazne parametre dimenziju tela i funkciju parametrizovanu tipovima `Vec3` i `VoxelData`. Klasa `VoxelData` služi za opis i oblika i boje voksela istom funkcijom koja prima `Vec3` a vraća instancu `VoxelData`.

Na taj način je klasa `BaseF` u mogućnosti da proceduralno generiše vokselni objekat, ali da ga potom ipak sačuva i kasnije mu eksplicitno pristupa, što u određenim scenarijima može dovesti do optimizacije izračunavanja.

3.2 Opis preseka zraka sa vokselom

Presek zraka sa objektom u programu *GfxLab-2020-2021* je u osnovi interfejs koji propisuje metode za vraćanje vremena preseka zraka sa objektom, normalizovanu normalu u tački preseka, kao i UV koordinate presečne tačke na površini objekta.

Interfejs `Hit` dalje implementira unutrašnja apstraktna klasa `HitRayT` čiji konstruktor prima zrak i vreme preseka sa telom, ostavljajući tako mogućnost da svako telo implementira metode za određivanje normale i UV koordinata preseka na optimalan način.

3.2.1 Klasa HitVoxel

Specijalizovana klasa za računanje preseka zraka sa vokselom data je na listingu 5.

```

01  public class HitVoxel extends HitRayT implements Comparable<HitVoxel> {
02
03      private Vec3 n_;
04      private int p, q, r;
05
06      public HitVoxel(Ray ray, double t) {
07          super(ray, t);
08      }
09
10     public HitVoxel(Ray ray, Hit h, Vec3 p) {
11         this(ray, h, p.xInt(), p.yInt(), p.zInt());
12     }
13
14     public HitVoxel(Ray ray, Hit h, int i, int j, int k) {
15         this(ray, h.t());
16         this.n_ = h.n_();
17
18         this.p = i;
19         this.q = j;
20         this.r = k;
21     }
22
23     @Override
24     public Vec3 n_() {
25         return n_;
26     }
27
28     @Override
29     public Vector uv() {
30         return Util.pack(p, q, r);
31     }
32
33     @Override
34     public int compareTo(HitVoxel h) {
35         double d = t() - h.t();
36
37         if      (d > 0) return 1;
38         else if (d < 0) return -1;
39         else           return 0;
40     }
41 }
```

Listing 5. Klasa `HitVoxel`.

Pored zraka i vremena preseka koje nasleđuje od klase `Hit.HitRayT`, klasa `HitVoxel` čuva i normalizovanu normalu u tački preseka `n_` kao i celobrojne koordinate pozicije voksela u matrici tela `p`, `q` i `r`⁸. Takođe, moguće je poređenje različitih instanci klase na osnovu vremena preseka.

⁸ U ovom slučaju koriste se tri celobrojna polja umesto trodimenzionalnog vektora zarad lakoće pristupa vrednostima.

3.2.2 Prenos informacije o boji pogodenog voksa

Od implementacije metoda `uv` se u opštem slučaju očekuje da vrati dvodimenzionalni vektor koji predstavlja poziciju presečne tačke na dvodimenzionalnoj površini tela, što je geometrijski problem koji svako telo rešava na sebi svojstven način. Vokselna tela se od poligonalnih razlikuju time što ne poseduju omotač, već se čitavo telo uniformno opisuje kao skup voksela, pa tako njegovu spoljnu teksturu određuju vokseli koji se nalaze na površini modela. Materijal na vidljivom delu površine je zato vezan ne za čitavo telo u vidu funkcijom definisanog uzorka ili učitane teksture, već je promenljiv na nivou svakog pojedinačnog voksela definisanog pozicijom u prostornoj i matrici boja trodimenzionalnim vektorom.

Ideja rešenja kojim se ove razlike mogu premostiti zasniva se na pakovanju tri celobrojne vrednosti kojima je opisana pozicija voksela u trodimenzionalnoj matrici u dve celobrojne koordinate dvodimenzionalnog vektora, odnosno instance `Vector` klase. Pakovanje se realizuje tako što se svaka od tri koordinate razbija na dva desetobitna komada, a zatim se od delova vrednosti sastavljaju odgovarajući elementi dvodimenzionalnog vektora. Budući da su celobrojne vrednosti 64-bitne, maksimalna veličina vokselnog objekta ograničena je na kocku stranice 2^{20} , odnosno 1.048.576 celija.

Implementacija opisanog postupka data je u vidu metoda `pack` klase `Util` i prikazana je na listingu 6.

```
01  public static Vector pack(int p, int q, int r) {  
02      int[] pp = { p, q, r };  
03      int[] qq = { 0, 0 };  
04  
05      for (int i = 0; i < 6; i++)  
06          qq[i/3] = qq[i/3] + ((  
07              (pp[i/2] >> (i%2) * DEFAULT_CHUNK_SIZE) & DEFAULT_SHIFT) <<  
08                  ((i%3) * DEFAULT_CHUNK_SIZE));  
09  
10      return Vector.xy(qq[0], qq[1]);  
11  }  
12 }
```

Listing 6. Metod `pack` klase `Util`.

Ključni deo metoda je `for` petlja koja kroz šest iteracija obrađuje po dva komada za svaku od tri koordinate ulaznog vektora pozicije voksela. Redosled operacija tokom svake od iteracija je sledeći:

- računanje indeksa $i/3$ koji kroz iteracije prima vrednosti $(0, 0, 0, 1, 1, 1)$ i koji određuje kojem od dva elementa izlaznog vektora se pridružuje trenutni komad,
- računanje indeksa $i/2$ koji kroz iteracije prima vrednosti $(0, 0, 1, 1, 2, 2)$ i koji određuje iz kojeg se elementa ulaznog vektora „izvlači” trenutni komad,
- računanje vrednosti $(i\%2) * \text{DEFAULT_CHUNK_SIZE}$ koja određuje za koliko se bitova pomera 10-bitna vrednost komada koja će biti dodata elementu izlaznog niza,
- maskiranje svega osim deset najmanje značajnih, odnosno deset krajnjih desnih bitova, kroz operaciju $\& \text{DEFAULT_SHIFT}$, gde je vrednost `DEFAULT_SHIFT` jednaka 1023,
- računanje vrednosti $i\%3$ koja određuje poziciju na koju se pomera dobijeni komad, i

- dodavanje pripremljenog komada odgovarajućem elementu izlaznog vektora.

Rezultat metoda je pakovanje trodimenzionalnog vektora `pp` u dvodimenzionalni vektor `qq` tako da `qq[0]` sadrži `pp[0]` i prvu polovinu `pp[1]`, a `qq[1]` drugu polovinu `pp[1]` i ceo element `pp[2]`.

Inverzni metod kojim se na drugom kraju rekonstruiše pozicija voksla u modelu predstavlja `unpack` prikazan na listingu 7.

```

01  public static Vec3 unpack(Vector q) {
02
03      int[] qq = { q.xInt(), q.yInt() };
04      int[] pp = { 0, 0, 0 };
05
06      for (int i = 0; i < 6; i++)
07          pp[i/2] = pp[i/2] + ((
08              (qq[i/3] >> (i%3) * DEFAULT_CHUNK_SIZE) & DEFAULT_SHIFT) <<
09              ((i%2) * DEFAULT_CHUNK_SIZE));
10
11      return Vec3.xyz(pp[0], pp[1], pp[2]);
12 }
```

Listing 7. Metod `unpack` klase `Util`.

3.3 Opis algoritama za renderovanje vokselnih tela

Algoritmi za renderovanje vokselnih tela u paketu *VoxelWorld* oslanjaju se na u najvećoj meri na klasu `BaseM`, dodatno i na `BaseF` i `BasePF`.

Klase koje implementiraju algoritme nasleđuju navedene klase i tako čuvaju informacije o modelu, a razlikuju se po načinu na koji implementiraju metod `firstHit`, odnosno `hits`, interfejsa `Solid` koji se nalazi u osnovi programa *GfxLab-2020-2021*.

Grafički rezultati svih algoritama su naravno identični. Jedina, ali značajna razlika primećuje se u vremenu potrebnom za renderovanje scene, o čemu će više reći biti u nastavku rada.

3.3.1 *Brute force* algoritam i klasa `BruteForce`

Kako je već rečeno, budući da vokseli predstavljaju jedinične elemente zapreme, za svaki element tela opisanog vokselima moguće je računati presek zraka sa kutijom stranica (1, 1, 1) na poziciji voksla i, ukoliko zrak preseca takvu kutiju, računati to kao presek zraka sa vokselnim telom.

Implementacija metoda `firstHit` data je na listingu 8. Na samom početku (linije 3-4) uvode se promenljive `out` tipa `Hit` koja čuva informaciju o potencijalnom prvom preseku zraka i tela i koja se inicijalizuje na `Hit.POSITIVE_INFINITY` (tj. na odsustvo preseka), i `v0` tipa `Vec3` koja čuva informaciju o tačnom vokselu u kom se zrak seče sa telom i koja se inicijalizuje na koordinatni početak.

Brute force pristup podrazumeva jednostavnu ideju, kao i stavljanje akcenta na tačno izvršavanje zadatka, dok optimizovanje procesa primetno stavlja na drugi plan. U skladu sa tim algoritam iterira po svim elementima vokselnog tela počevši od koordinatnog početka (linije 6-8). Ukoliko voksel na trenutnoj poziciji ne postoji iteriranje se nastavlja (linija 10), inače se traže svi preseci zraka sa vokselom, kojih naravno može biti maksimalno dva (linija 12).

```

01  public Hit firstHit(Ray ray, double afterTime) {
02
03      Hit out = Hit.POSITIVE_INFINITY;
04      Vec3 v0 = Vec3.ZERO;
05
06      for (int i = 0; i < lenX(); i++) {
07          for (int j = 0; j < lenY(); j++) {
08              for (int k = 0; k < lenZ(); k++) {
09
10                  if (!isPopulated(i, j, k)) continue;
11
12                  Hit[] h = getHits(Vec3.xyz(i, j, k), ray);
13
14                  if (h.length == 0) continue;
15
16                  if (h[0].t() > afterTime) {
17                      if (h[0].t() < out.t()) {
18                          out = h[0];
19                          v0 = Vec3.xyz(i, j, k);
20
21                  } else if (h.length > 1 && h[1].t() > afterTime) {
22                      if (h[1].t() < out.t()) {
23                          out = h[1];
24                          v0 = Vec3.xyz(i, j, k);
25
26                  }
27
28              }
29
30      }
31
32      return new HitVoxel(ray, out, v0);

```

Listing 8. Metod `firstHit` klase `BruteForce`.

Preseci zraka sa vokselom računaju se na način opisan u potpoglavlju 2.2, a implementirani su kako je prikazano na listingu 2. Ukoliko nema preseka, iteriranje se nastavlja (linija 14).

U slučaju da je vreme preseka prvog elementa niza `h` veće od parametra metoda `afterTime` (linija 16), tada se proverava da li je to vreme ujedno i manje od vremena trenutne vrednosti promenljive `out` (linija 17) i, ako jeste, promenljiva `out` dobija vrednost prvog elementa `h`, dok vrednost `v0` dobija vrednost pozicije trenutnog voksla (linije 18-19).

Kada bilo koji od navedena dva uslova nije zadovoljen proverava se drugi element niza `h`, odnosno proverava se da li uopšte postoji i da li je njegovo vreme veće od parametra `afterTime` (linija 21). Ukoliko su uslovi zadovoljeni i ukoliko je vreme `h[1]` manje od trenutnog minimuma `out`, vrednosti se ažuriraju na način opisan u prethodnom pasusu (linije 22-24).

Na kraju metoda vraća se instanca klase `HitVoxel` koja sadrži upadni zrak, presek sa vokselom `out` i poziciju voksla `v0`.

Klasa `BruteForce` implementira i metod `hits` koji vraća preseke sa svim vokselima tela na putanji zraka sortirane prema vremenu preseka. Ovaj metod je prikazan na listingu 9.

Na početku metoda kreira se privremeni niz `hits` dužine koja odgovara broju voksela u matrici koja sadrži telo, dok se broj „korisnih” elemenata niza prati promenljivom `num` (linije 4-5). Metod dalje iterira kroz sve pozicije u mreži i ignoriše one koje ne sadrže voksele, dok za one koje

```

01  @Override
02  public Hit[] hits(Ray ray) {
03
04      Hit[] hits = new Hit[lenX() * lenY() * lenZ()];
05      int num = 0;
06
07      for (int i = 0; i < lenX(); i++) {
08          for (int j = 0; j < lenY(); j++) {
09              for (int k = 0; k < lenZ(); k++) {
10                  if (isPopulated(i, j, k)) continue;
11
12                  Hit[] h = getHits(Vec3.xyz(i, j, k), ray);
13
14                  if (h.length == 0) continue;
15
16                  if (num > 0) {
17                      if (Math.abs(hits[num - 1].t() - h[0].t()) > 1e-8)
18                          hits[num++] = h[0];
19                      else
20                          num--;
21                  } else {
22                      hits[num++] = h[0];
23                  }
24
25                  hits[num++] = h[1];
26              }
27          }
28      }
29
30      if (num == 0) return Solid.NO_HITS;
31
32      return Arrays.copyOf(hits, num);
33  }

```

Listing 9. Metod `hits` klase `BruteForce`.

sadrže voksele pribavlja niz sa svim presecima zraka i voksela o kom je reč, ignorišući pritom one za kojih nema pogodaka (linije 4-15).

Metod `hits` vraća sve preseke zraka sa vokselima koji leže na njegovoj putanji. Problem nastaje kod susednih voksela, gde se dešava da je izlazni pogodak prethodnog gotovo identičan ulaznom pogotku trenutnog voksela. S obzirom da se ovakva situacija može desiti samo kod ulaznih pogodaka i to ne računajući prvi voksel na putanji zraka, pri pokušaju dodavanja ulaznog preseka za trenutni voksel vrši se provera na osnovu razlike vremena u odnosu na poslednji presek u nizu, i ukoliko je razlika veća od ϵ , što u ovom slučaju iznosi $1e-8$, poslednji element niza se izbacuje i na njegovo mesto dolazi ulazni pogodak trenutnog voksela. Izlazni pogodak se dodaje u svakom slučaju, kao i ulazni kada je niz `hits` prazan (linije 17-26).

Na kraju se proverava izlazni niz i vraća konstanta `Solid.NO_HITS` kada je prazan, odnosno kopija niza kada nije (linije 31-33).

Ovakav pristup pati od dva ozbiljna nedostatka. Prvi, donekle zanemarljiv, je činjenica da se presek zraka sa vokselom praktično izračunava instanciranjem opšte klase `Box` – iako je voksel nesumnjivo kutija, reč je uvek o kutiji stranica $(1, 1, 1)$ zbog čega se računanje preseka može optimizovati, što će i biti istraženo u potonjim algoritmima. Drugi, daleko ozbiljniji problem, je to što ovaj algoritam u svakom slučaju prolazi kroz sve voksele matrice u kojoj se nalazi vokselno telo, odnosno presek tela sa zrakom se nalazi tako što se pronalaze svi vokseli kroz koje zrak prolazi, a tada se iz takvog skupa rezultata izdvaja onaj sa najmanjom vrednošću polja `t`.

Uprkos svojoj jednostavnosti, zbog nabrojanih nedostataka *brute force* algoritam nije naročito značajan u praksi, osim u smislu polazne tačke za istraživanje optimizovanih rešenja i kao referentna vrednost sa kojom će biti poređeni budući rezultati.

3.3.2 *Direction array* algoritam i klase DirArray i DirArray0

Uzimanjem u obzir pravca kretanja zraka moguće je značajno optimizovati *brute force* algoritam te samim tim skratiti vreme renderovanja.

Neophodnost iteriranja kroz sve voksele matrice prepoznata je kao najveća mana ranije opisanog „naivnog” pristupa, pri čemu se iteriranje uvek kreće od koordinatnog početka ka temenu na drugom kraju dijagonale vokselne matrice. Međutim, ukoliko smer iteriranja prati pravac kretanja zraka, ne samo da se može skratiti vreme potrebno za pronalaženje prvog preseka zraka sa telom, već je to i garantovano prvi presek, odnosno onaj sa najmanjom vrednošću polja t , te se u istom koraku iteriranje može i prekinuti.

Metod `getLoopData` klase `Util` služi za izračunavanje smera iteriranja kroz vokselnu matricu i prikazan je na listingu 10.

```

01  public static Vec3[] getLoopData(Vec3 modelSize, Ray ray) {
02
03      int lenX = modelSize.xInt(),
04          lenY = modelSize.yInt(),
05          lenZ = modelSize.zInt();
06
07      int dx = ray.d().x() >= 0 ? +1 : -1,
08          dy = ray.d().y() >= 0 ? +1 : -1,
09          dz = ray.d().z() >= 0 ? +1 : -1;
10
11      int           xs = -1           , xe = -1   , xd =  0;
12      if (dx == 1) { xs =  0           ; xe = lenX ; xd = +1; }
13      else        { xs = lenX - 1; xe = -1   ; xd = -1; }
14      int           ys = -1           , ye = -1   , yd =  0;
15      if (dy == 1) { ys =  0           ; ye = lenY ; yd = +1; }
16      else        { ys = lenY - 1; ye = -1   ; yd = -1; }
17      int           zs = -1           , ze = -1   , zd =  0;
18      if (dz == 1) { zs =  0           ; ze = lenZ ; zd = +1; }
19      else        { zs = lenZ - 1; ze = -1   ; zd = -1; }
20
21      return new Vec3[] {
22          Vec3.xyz(xs, xe, xd),
23          Vec3.xyz(ys, ye, yd),
24          Vec3.xyz(zs, ze, zd),
25      };
26  }

```

Listing 10. Metod `getLoopData` klase `Util` namenjen klasama koje implementiraju *direction array* algoritam.

Nakon smeštanja veličine objekta u zasebne promenljive (linije 3-5), određuju se koraci inkrementiranja za sve tri dimenzije matrice dx , dy i dz , i to na osnovu znaka odgovarajućih elemenata vektora pravca upadnog zraka (linije 7-9). Ovi inkrementi upravo indikuju da li se zrak kreće ka $+\infty$ ili $-\infty$ po zasebnim osama koordinatnog sistema, i kao takvi se koriste i za izračunavanje početne i krajnje tačke svake od tri iteracione petlje.

Dalje se računaju početne tačke petlji xs , ys , odnosno zs , kao i krajnje tačke xe , ye , odnosno ze – kada je inkrement pozitivan početna tačka je 0 a krajnja je maksimalna vrednost odgovarajuće dimenzije modela, dok je kod negativnog inkrementa situacija obrnuta (linije 11-19).

Izlaznu vrednost predstavlja niz tri instance `Vec3` klase koje sadrže početak, kraj i inkrement svake od tri petlje (linije 21-25).

Implementacija metoda `firstHit` klase `DirArray` data je na listingu 11.

```

01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      Vec3[] loopData =
05          Util.getLoopData(Vec3.xyz(lenX(), lenY(), lenZ()), ray);
06
07      int xs = loopData[0].xInt(),
08          xe = loopData[0].yInt(),
09          xd = loopData[0].zInt(),
10
11      int ys = loopData[1].xInt(),
12          ye = loopData[1].yInt(),
13          yd = loopData[1].zInt(),
14
15      int zs = loopData[2].xInt(),
16          ze = loopData[2].yInt(),
17          zd = loopData[2].zInt();
18
19      for (int i = xs; i != xe; i += xd) {
20          for (int j = ys; j != ye; j += yd) {
21              for (int k = zs; k != ze; k += zd) {
22
23                  if (!isPopulated(i, j, k)) continue;
24
25                  Hit[] h = getHits(Vec3.xyz(i, j, k), ray);
26
27                  if (h.length == 0) continue;
28
29                  if (h[0].t() > afterTime)
30                      return new HitVoxel(ray, h[0], i, j, k);
31              }
32          }
33      }
34
35      return Hit.POSITIVE_INFINITY;
}

```

Listing 11. Metod `firstHit` klase `DirArray`.

Pre iteriranja računaju se parametri vezani za smer kretanja petlji metodom `getLoopData` klase `Util` i smeštaju u zasebne promenljive zarad lakšeg korišćenja (linije 4-16), nakon čega se iterira kroz celu matricu modela u pravcu vektora pravca zraka (linije 18-20). Pozicije bez voksele se preskaču, za voksele modela se računaju preseci sa zrakom i, ukoliko postoje, vreme prvog u nizu se poredi sa parametrom metoda `afterTime` – ukoliko je vreme pogotka iznad propisane granice metod odmah vraća instancu `HitVoxel` klase, dok se drugi pogodak kao izlazni uopšte ne uzima u obzir (linije 22-30). Metod iteriranje završava bez prekida samo u slučaju da se zrak ne seče sa vokselima modela, u kom slučaju se vraća konstanta `Hit.POSITIVE_INFINITY`.

Implementacija metoda `hits` slična je metodu `firstHit` i prikazana je na listingu 12. Jednu sušinsku razliku predstavlja otklanjanje identičnih pogodaka između susednih voksela i potonje vraćanje niza umesto jedinstvenog pogotka. Ipak, upravo zbog toga jasno je da u ovom slučaju ne postoji prerani prekid iteriranja, te se tako gubi jedna od najznačajnijih prednosti ovog rešenja. Ovakve i slične mane, u sadejstvu sa odsustvom potrebe za računanjem preseka sa svim

```

01  @Override
02  public Hit[] hits(Ray ray) {
03
04      Hit[] hits = new Hit[lenX() * lenY() * lenZ()];
05      int num = 0;
06
07      Vec3[] loopData =
08          Util.getLoopData(Vec3.xyz(lenX(), lenY(), lenZ()), ray);
09
10     int xs = loopData[0].xInt(),
11         xe = loopData[0].yInt(),
12         xd = loopData[0].zInt(),
13
14     int ys = loopData[1].xInt(),
15         ye = loopData[1].yInt(),
16         yd = loopData[1].zInt(),
17
18     int zs = loopData[2].xInt(),
19         ze = loopData[2].yInt(),
20         zd = loopData[2].zInt();
21
22     for (int i = xs; i != xe; i += xd) {
23         for (int j = ys; j != ye; j += yd) {
24             for (int k = zs; k != ze; k += zd) {
25
26                 if (isPopulated(i, j, k)) continue;
27
28                 Hit[] h = getHits(Vec3.xyz(i, j, k), ray);
29
30                 if (h.length == 0) continue;
31
32                 if (num > 0) {
33                     if (Math.abs(hits[num - 1].t() - h[0].t()) > 1e-8)
34                         hits[num++] = h[0];
35                     else
36                         num--;
37                 } else {
38                     hits[num++] = h[0];
39                 }
40
41                 hits[num++] = h[1];
42             }
43         }
44     }
45
46     if (num == 0) return Solid.NO_HITS;
47
48     return Arrays.copyOf(hits, num);
49 }

```

Listing 12. Metod `hits` klase `DirArray`.

vokselima na putanji zraka, razlog su zašto se prilikom renderovanja u programu *GfxLab-2020-2021* koriste mahom `firstHit` metodi.

Očigledna mana predloženog pristupa vidljiva je u situacijama kada zrak u potpunosti promašuje objekat, budući da se tada ponovo iterira kroz sve voksele i za sve postojeće se računaju preseci sa zrakom do kojih u takvom slučaju ne dolazi. Ovaj nedostatak otklonjen je u implementaciji metoda `firstHit` klase `DirArray0` prikazanoj na listingu 13.

Algoritam je identičan onom koji je implementiran u klasi `DirArray`, s tom razlikom da se sada na početku proverava presek sa ograničavajućom kutijom, odnosno kutijom sa minimalnim

```

01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      Hit[] boundingBoxHits =
05          getHits(Vec3.ZERO, Vec3.xyz(lenX(), lenY(), lenZ()), ray);
06
07      if (boundingBoxHits.length == 0) return Hit.POSITIVE_INFINITY;
08
09      Vec3[] loopData =
10         Util.getLoopData(Vec3.xyz(lenX(), lenY(), lenZ()), ray);
11
12      int xs = loopData[0].xInt(),
13          xe = loopData[0].yInt(),
14          xd = loopData[0].zInt(),
15
16      ys = loopData[1].xInt(),
17      ye = loopData[1].yInt(),
18      yd = loopData[1].zInt(),
19
20      zs = loopData[2].xInt(),
21      ze = loopData[2].yInt(),
22      zd = loopData[2].zInt();
23
24      for (int i = xs; i != xe; i += xd) {
25          for (int j = ys; j != ye; j += yd) {
26              for (int k = zs; k != ze; k += zd) {
27
28                  if (!isPopulated(i, j, k)) continue;
29
30                  Hit[] h = getHits(Vec3.xyz(i, j, k), ray);
31
32                  if (h.length == 0) continue;
33
34                  if (h[0].t() > afterTime)
35                      return new HitVoxel(ray, h[0], i, j, k);
36              }
37          }
38      }
39
40      return Hit.POSITIVE_INFINITY;
41  }

```

Listing 13. Metod `firstHit` klase `DirArray0`.

temenom u koordinatnom početku i maksimalnim u tački određenoj dimenzijama vokselne matrice. Kada zrak u potpunosti promašuje matricu prilikom ove provere neće biti pogodaka, te se izvršavanje metoda može prekinuti vraćanjem konstante `Hit.POSITIVE_INFINITY` (linije 4-7).

Slična provera uvodi se i u metod `hits` klase `DirArray0`, dok je ostatak identičan onom iz klase `DirArray`.

Opisani algoritmi otklanjaju najveći nedostatak *brute force* pristupa kroz iteriranje kroz voksele prateći pravac upadnog zraka i, u slučaju `firstHit` metoda, ranim prekidanjem i vraćanjem prvog pronađenog pogotka voksla sa zrakom. Presek zraka sa vokselom i dalje se izračunava instanciranjem opšte klase `Box` jediničnih dimenzija, iako taj detalj ispoljava daleko manje uticaja na brzinu izvršavanja metoda te tako ne umanjuje značaj implementiranih optimizacija.

Problem međutim i dalje predstavlja pronalaženje ulazne i izlazne tačke zraka iz modela – uprkos optimizaciji kretanja kroz vokselnu matricu i dalje postoji svega osam potencijalnih

početnih tačaka iteracije koje se poklapaju sa temenima ograničavajuće kutije modela. Očigledno je da u pojedinim situacijama, poput slučaja kada su vokseli modela blizu sredine matrice, ovakvo rešenje neće značajno skratiti vreme renderovanja u odnosu na *brute force* algoritam.

3.3.3 Grid march algoritam i klasa GridMarch1

Osnovna ideja *grid march* algoritma proističe iz korišćenja ograničavajuće kutije kako bi se ne samo izračunavanje prekinulo u slučaju kada zrak promašuje matricu, već i doble ulazna i izlazna tačka zraka u preseku sa matricom, kako bi se iskoristile kao polazište i ishodište iteriranja kroz matricu. Smer zraka se ponovo uzima u obzir, ali na nešto drugačiji način koji pruža dodatno skraćivanje vremena potrebnog za izračunavanje preseka tako što se proveravaju samo određeni susedi trenutne pozicije u matrici.

I u ovom slučaju metod `getLoopData` klase `Util` nešto drugačijeg potpisa prikazan na listingu 14 služi za izračunavanje osnovnih vrednosti koje koristi algoritam.

```

01  public static Vec3[] getLoopData(
02      Vec3 size, Ray ray, Hit[] boundingBoxHits) {
03
04      Vec3 vx = Vec3.xyz(
05          ray.d().x() >= 0 ? +1 : -1,
06          ray.d().y() >= 0 ? +1 : -1,
07          ray.d().z() >= 0 ? +1 : -1);
08
09      Vec3 v0 = ray.at(boundingBoxHits[0].t()).floor();
10     Vec3 u0 = Vec3.xyz(
11         v0.xInt() == size.xInt() ? -1 : (v0.xInt() == -1 ? 1 : 0),
12         v0.yInt() == size.yInt() ? -1 : (v0.yInt() == -1 ? 1 : 0),
13         v0.zInt() == size.zInt() ? -1 : (v0.zInt() == -1 ? 1 : 0));
14
15     Vec3 v1 = ray.at(boundingBoxHits[1].t()).floor();
16     Vec3 u1 = Vec3.xyz(
17         (v1.xInt() == size.xInt() ? -1 : (v1.xInt() == -1 ? 1 : 0)),
18         (v1.yInt() == size.yInt() ? -1 : (v1.yInt() == -1 ? 1 : 0)),
19         (v1.zInt() == size.zInt() ? -1 : (v1.zInt() == -1 ? 1 : 0)));
20
21     return new Vec3[] { v0.add(u0), v1.add(u1.add(vx)), vx };
22 }
```

Listing 14. Metod `getLoopData` klase `Util` namenjen klasama koje implementiraju *grid march* algoritam.

Na početku metoda određuje se trodimenzionalni vektor znaka pravca zraka (linije 4-7). Nakon toga pronalazi se ulazni voksel na površini ograničavajuće kutije objekta. Pozicija takvog voksela `v0` računa se nalaženjem tačke na upadnom zraku u trenutku pogotka prvog pogotka u nizu `boundingBoxHits`, što je niz pogodaka nastalih presekom zraka sa ograničavajućom kutijom, a zatim se nad svim elementima rezultujućeg vektora poziva `floor` metod. Drugim rečima, vrednost prvog preseka zraka sa ograničavajućom kutijom se svodi na vektor sa celobrojnim elementima (linija 9).

U slučaju ekstremnih vrednosti, odnosno onih kod kojih je jedan ili više elemenata jednako vrednostima koje prilikom iteriranja izlaze van okvira vokselne matrice, izračunava se „korektivna vrednost” `u0` koja se kasnije dodaje na `v0` čime se sprečavaju greške prilikom iteriranja (linije 10-13).

Analogan postupak primenjuje se na izlazni pogodak, te se dobijaju vrednosti **v1** i **u1** (linije 15-19). Izlaznu veličinu predstavlja element od tri trodimenzionalna vektora sačinjen od pozicije ulaznog voksla, pozicije izlaznog voksla, i vektora znaka pravca zraka (linija 21).

```

01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      Hit[] boundingBoxHits =
05          getHits(Vec3.ZERO, Vec3.xyz(lenX(), lenY(), lenZ()), ray);
06
07      if (boundingBoxHits.length == 0) return Hit.POSITIVE_INFINITY;
08
09      Vec3[] loopData = Util.getLoopData(
10          Vec3.xyz(lenX(), lenY(), lenZ()), ray, boundingBoxHits);
11
12      int xs = loopData[0].xInt(), xe = loopData[1].xInt(),
13          xd = loopData[2].xInt(), ys = loopData[0].yInt(),
14          ye = loopData[1].yInt(), yd = loopData[2].yInt(),
15          zs = loopData[0].zInt(), ze = loopData[1].zInt(),
16          zd = loopData[2].zInt();
17
18      int[] xi = { 0, 0, xd };
19      int[] yi = { 0, yd, 0 };
20      int[] zi = { zd, 0, 0 };
21
22      while (
23          xs != -1 && xs != xe &&
24          ys != -1 && ys != ye &&
25          zs != -1 && zs != ze ) {
26
27          Vec3 p = Vec3.xyz(xs, ys, zs);
28          Hit[] hits = getHits(p, ray);
29
30          if (isPopulated(p) && hits[0].t() > afterTime)
31              return new HitVoxel(ray, hits[0], xs, ys, zs);
32
33          Hit[][][] neighbours = new Hit[3][2];
34
35          int idx = -1;
36          double minTime = Double.MAX_VALUE;
37
38          for (int i = 0; i < neighbours.length; i++) {
39
40              neighbours[i] =
41                  getHits(p.add(Vec3.xyz(xi[i], yi[i], zi[i])), ray);
42
43              if (neighbours[i].length == 0) {
44                  continue;
45              } else if (minTime > neighbours[i][0].t()) {
46                  idx = i;
47                  minTime = neighbours[i][0].t();
48              }
49          }
50
51          xs += xi[idx];
52          ys += yi[idx];
53          zs += zi[idx];
54      }
55
56      return Hit.POSITIVE_INFINITY;
57  }

```

Listing 15. Metod **firstHit** klase **GridMarch1**.

Implementacija metoda `firstHit` klase `GridMarch1` data je na listingu 15. Na početku se računaju preseci zraka sa ograničavajućom kutijom – ukoliko ih nema izračunavanje se prekida, inače se niz rezultujućih pogodaka prosleđuje odgovarajućem metodu `getLoopData` klase `Util`. Konačno, dobijene vrednosti se smeštaju u celobrojne promenljive zarad lakšeg korišćenja (linije 4-16).

Nizovi `xi`, `yi` i `zi` predstavljaju nakon računanja ulaznog i izlaznog voksla drugu značajnu optimizaciju *grid march* algoritma kojoj zapravo duguje i svoj naziv (linije 18-20). Budući da svaki voksel predstavlja jediničnu kocku oko svakog se nalazi šest suseda sa kojima voksel u pitanju deli jednu od svojih stranica.

Ukoliko upadni zrak dospe u ćeliju na određenoj poziciji u vokselnoj mreži, ali ta ćelija ne sadrži voksel (tj. prazna je), zrak će dalje putovati u jedan od šest susednih voksela sa kojima trenutni deli po jednu stranicu⁹. Međutim, ako se u obzir uzme pravac zraka broj mogućih suseda pada na svega tri, budući da nema smisla proveravati voksele koji ne leže na pravcu zraka.

Iteriranje se ovog puta implementira kroz jednu petlju čiji je uslov za prekid izlazak van okvira vokselne matrice po bilo kojoj od tri ose (linije 22-25). Trenutni voksel je uvek onaj na poziciji sa koordinatama `xs`, `ys` i `zs` (linija 27) – na početku je to ulazna tačka zraka u vokselnu matricu, a u kasnijim prolascima kroz petlju je to vrednost ažurirana u prethodnoj iteraciji.

Sledeći korak predstavlja nalaženje pogodaka, pri čemu se u obzir naravno uzima samo prvi – u slučaju da postoji voksel na zadatoj poziciji i da je njegovo vreme veće od parametra `afterTime`, pogodak se vraća i izvršavanje se završava (linije 28-31).

Kada pogodak izostane potrebno je pronaći sledećeg suseda na putanji zraka. Za tri relevantna suseda moguća su maksimalno dva pogotka, te se tri para preseka čuvaju u nizu `neighbours` (linija 33). Samo jedan od tri suseda će imati pogotke, te se u slučaju kada ih nema sused preskače, inače se beleži indeks suseda i vreme njegovog ulaznog pogotka (linije 35-49). Konačno, ažuriraju se vrednosti `xs`, `ys` i `zs` za potrebe sledeće iteracije (linije 51-53).

Implementacija metoda `hits` klase `GridMarch1` data je na listingu 16. Kao i kod prethodnih algoritama metod `hits` se ne razlikuje značajno u odnosu na `firstHit`, s tim da je u ovom slučaju jedino opadanje performansi izazvano neophodnošću pronalaska svih voksela na putanji zraka, za razliku od *directional array* algoritma kada je `hits` zahtevao prolazak kroz celu matricu te se time malo razlikovao od *brute force* pristupa.

Grid march algoritam predstavlja značajno poboljšanje u odnosu na *directional array* budući da se u potpunosti eliminiše „prazan hod” prilikom iteriranja kroz vokselnu matricu – provera se ograničava isključivo na voksele kroz koje zaista prolazi zrak, i s obzirom da se pogodak pronalazi čim zrak postupnim kretanjem kroz matricu nađe na zauzet voksel teško je doći do ideje koja će kardinalno izmeniti, a samim tim i značajno poboljšati ovakav pristup.

Određena čisto računska poboljšanja osnovne ideje *grid march* algoritma ipak su moguća, za šta se očekuje da dovede do osetnog skraćivanja vremena renderovanja scene.

⁹ Teorijski je svakako moguće da zrak iz voksla izade tačno kroz jedno od osam temena ili jednu od dvanaest ivica, čime broj suseda raste na 26. Ipak, zbog nepreciznosti rezultata operacija nad brojevima sa pokretnim zarezom ivice objekata se u *raytracing* programima uopšte smatraju prostorom van objekata, te bi se tako prolazak kroz neku od ivica ili neko od temena uvek tretirao kao prolazak kroz jednu od šest strana vokselne kocke.

```

01  @Override
02  public Hit[] hits(Ray ray) {
03
04      Hit[] boundingBoxHits =
05          getHits(Vec3.ZERO, Vec3.xyz(lenX(), lenY(), lenZ()), ray);
06
07      if (boundingBoxHits.length == 0) return Solid.NO_HITS;
08
09      Vec3[] loopData = Util.getLoopData(
10         Vec3.xyz(lenX(), lenY(), lenZ()), ray, boundingBoxHits);
11
12      int xs = loopData[0].xInt(), xe = loopData[1].xInt(),
13          xd = loopData[2].xInt(), ys = loopData[0].yInt(),
14          ye = loopData[1].yInt(), yd = loopData[2].yInt(),
15          zs = loopData[0].zInt(), ze = loopData[1].zInt(),
16          zd = loopData[2].zInt();
17
18      HitVoxel[] hits = new HitVoxel[lenX() * lenY() * lenZ()]; int num = 0;
19
20      int[] xi = { 0, 0, xd }, yi = { 0, yd, 0 }, zi = { zd, 0, 0 };
21
22      while (
23          xs != -1 && xs != xe && ys != -1 && ys != ye &&
24          zs != -1 && zs != ze ) {
25
26          Vec3 p = Vec3.xyz(xs, ys, zs); Hit[] h = getHits(p, ray);
27
28          if (isPopulated(p)) {
29              if (num > 0) {
30                  if (Math.abs(hits[num - 1].t() - h[0].t()) > 1e-8)
31                      hits[num++] = new HitVoxel(ray, h[0], xs, ys, zs);
32                  else
33                      num--;
34              } else {
35                  hits[num++] = new HitVoxel(ray, h[0], xs, ys, zs);
36              }
37
38              hits[num++] = new HitVoxel(ray, h[1], xs, ys, zs);
39          }
40
41          Hit[][] neighbours = new Hit[3][2];
42          int idx = -1; double minTime = Double.MAX_VALUE;
43
44          for (int i = 0; i < neighbours.length; i++) {
45
46              neighbours[i] =
47                  getHits(p.add(Vec3.xyz(xi[i], yi[i], zi[i])), ray);
48
49              if (neighbours[i].length == 0) {
50                  continue;
51              } else if (minTime > neighbours[i][0].t()) {
52                  idx = i;
53                  minTime = neighbours[i][0].t();
54              }
55          }
56
57          xs += xi[idx]; ys += yi[idx]; zs += zi[idx];
58      }
59
60      if (num == 0) return Solid.NO_HITS;
61
62      return Arrays.copyOf(hits, num);
63  }

```

Listing 16. Metod `hits` klase `GridMarch1`.

3.3.4 Grid march algoritam i klase GridMarch2 i GridMarch20

Rad GridMarch2 algoritma predstavlja optimizaciju GridMarch1 pristupa, i ilustrovan je na primeru dvodimenzionalnog tela opisanog pikselima u dvodimenzionalnoj matrici (slike 6-9). Kao što se vidi na slici 6, zrak ima početnu tačku u koordinatnom početku i normalizovan vektor pravca (0.832, 0.555), a pikselnu matricu preseca u tačkama (4.5, 3) i (7, 4.667).

Piksel koji zrak prvi preseca, odnosno vrednost v_0 , je na poziciji (4, 3), što se dobija zaokruživanjem decimalne vrednosti „naniže”; analogno, piksel kroz koji zrak izlazi iz matrice, odnosno vrednost v_1 , je na poziciji (7, 4). Takođe, sa slike 6 je jasno da je prva pozicija u matrici koju zrak preseca prazna – pogodak se očekuje tek u pikselu na poziciji (5, 3).

Dalje se računaju parametri vezani za pravac zraka, pa je tako vrednost s_0 jednaka (1, 1) budući da su oba elementa smera zraka pozitivna, s_1 je inverzni vektor vektora s_0 , odnosno (-1, -1), a s_2 koji se računa dodavanjem vektora (1, 1) na s_1 i zatim skaliranjem vrednošću 0.5 uzima vrednost (0, 0).

Sledi računanje vremena preseka zraka sa poluravnima koje definišu voksel u susret pravcu kretanja zraka (slike 7-8). Poluravan $x = 4$ definisana je tačkom koja se dobija sabiranjem v_0 i s_2 , što je u ovom slučaju v_0 budući da je s_2 nula vektor, i normalom na poluravan u toj tački što je u ovom slučaju (-1, 0). Sada je moguće izračunati vreme preseka zraka sa poluravni korišćenjem metoda hits klase HalfSpace (listing 18)¹⁰, što je u ovom slučaju približno 4.808 za tačku preseka (4, 2.687).

```
01  @Override
02  public Hit[] hits(Ray ray) {
03
04      double o = n_().dot(ray.d());
05      double t = n_().dot(p().sub(ray.p())) / o;
06
07      if (o < 0) {
08          return new Hit[] {new HitHalfSpace(ray, t), Hit.POSITIVE_INFINITY};
09      }
10
11      if (o > 0) {
12          return new Hit[] {Hit.NEGATIVE_INFINITY, new HitHalfSpace(ray, t)};
13      }
14
15      return Solid.NO_HITS;
16  }
```

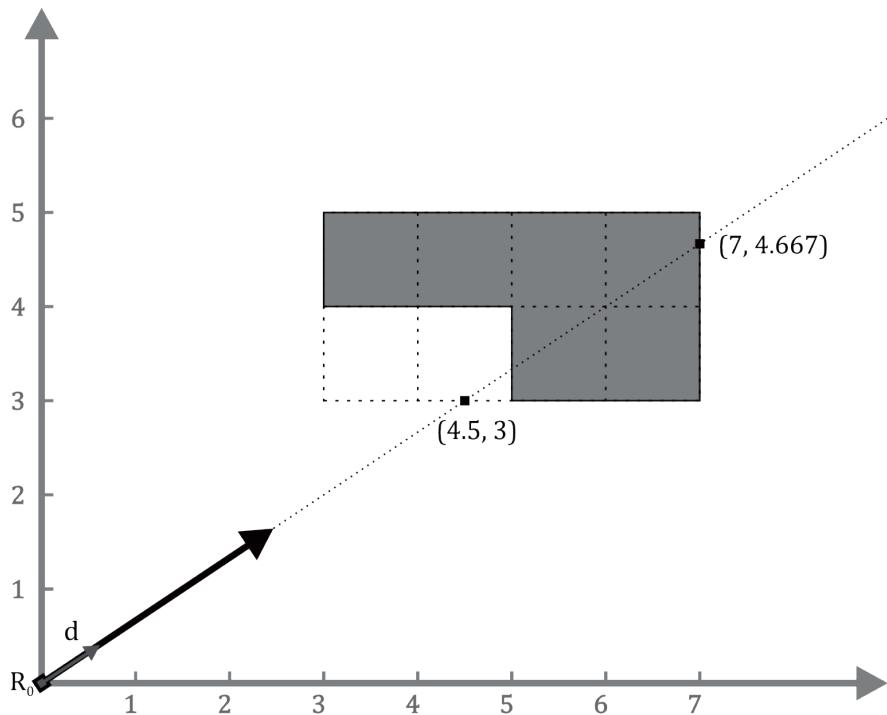
Listing 18. Metod hits klase HalfSpace.

Analogno se računa vreme preseka sa poluravni $y = 3$, što za tačku (4.5, 3) iznosi približno 5.405 (slika 8).

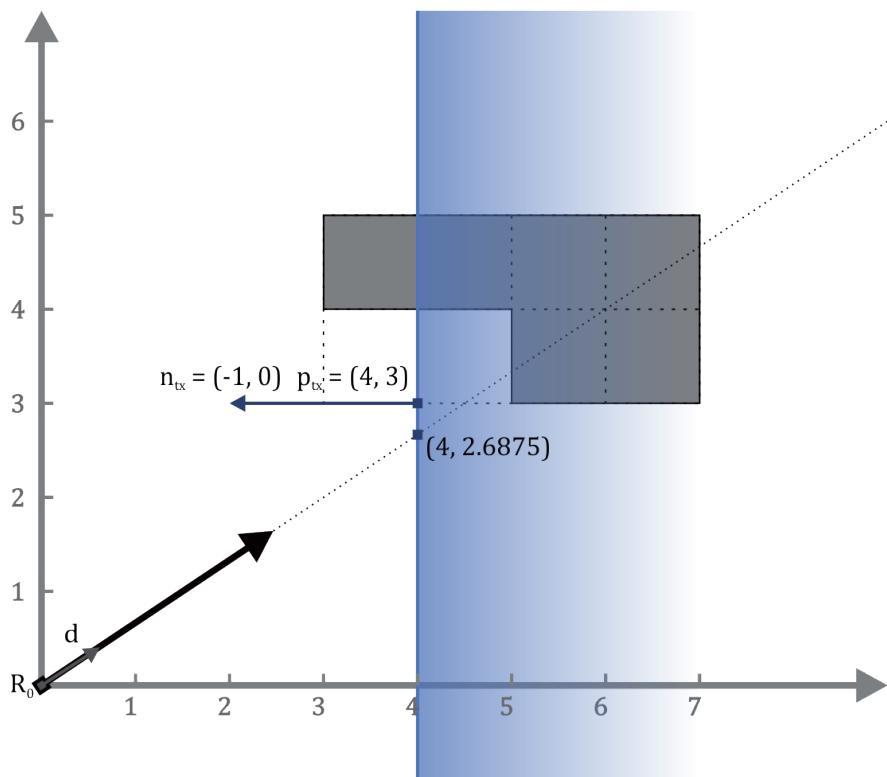
Promenljiva t sa vremenima preseka zraka sa graničnim poluravnima tako uzima vrednost (4.808, 5.405), a promenljiva dt sa vremenima potrebnim da zrak dođe do ivične poluravni sledećeg piksela vrednost (1.2, 1.8).

Sledi iteriranje po pikselima na putanji zraka. Početni piksel je na poziciji (4, 3). Iteriranje se završava kada zrak pogodi bilo koji piksel unutar matrice, ili kada izđe iz ograničavajućeg

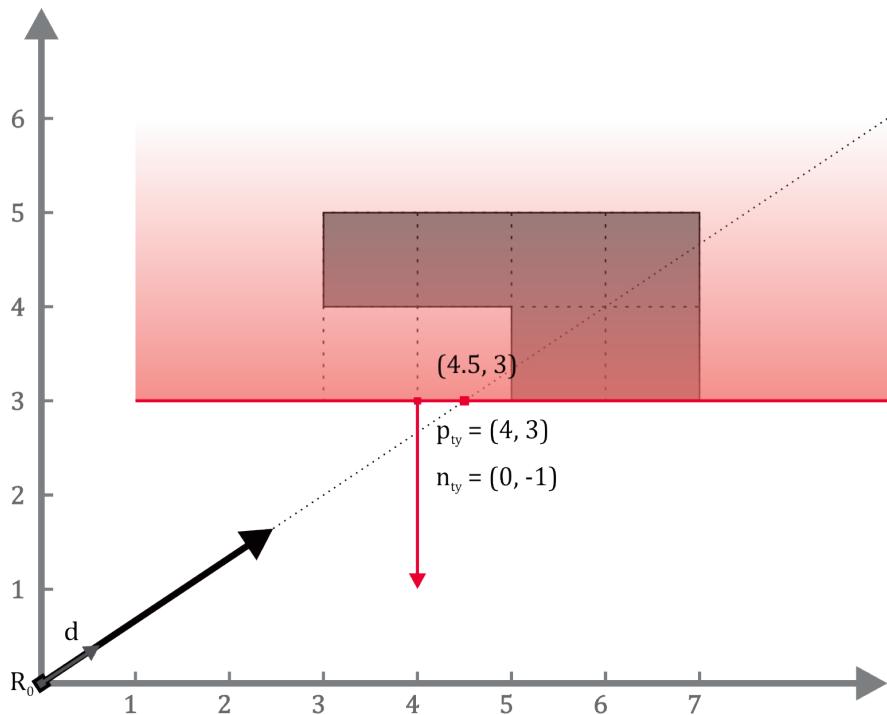
¹⁰ U pitanju je klasa programa GfxLab-2020-2021 koja opisuje poluprostor u trodimenzionalnom sistemu – dvodimenzionalna poluravan funkcioniše na identičan način uz jednu dimenziju manje.



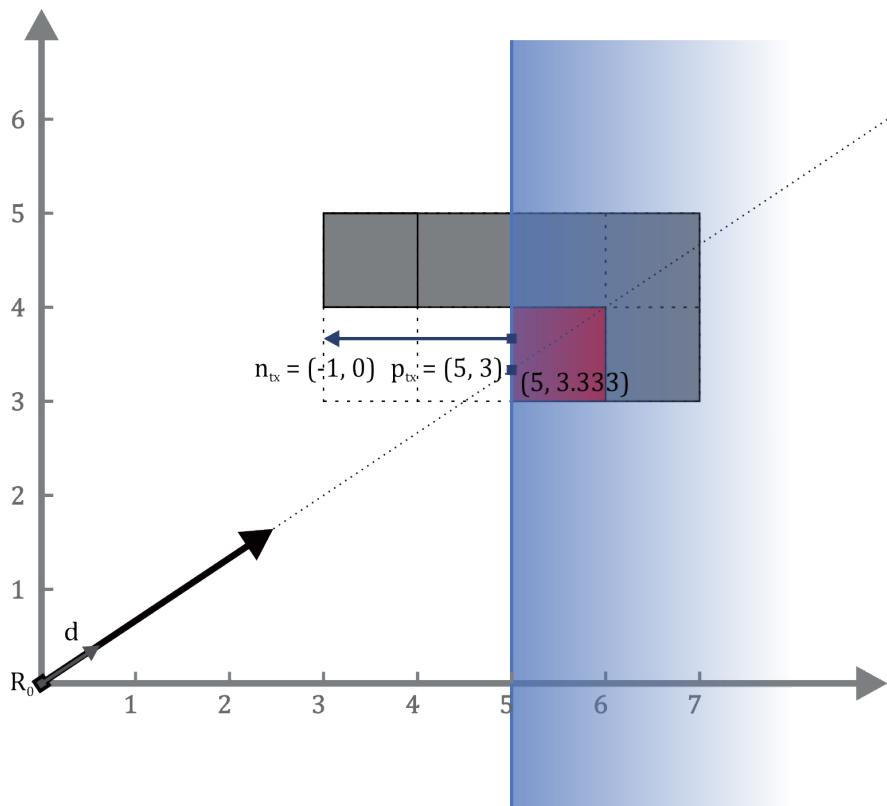
Slika 6. Presek zraka sa dvodimenzionalnom pikselnom matricom.



Slika 7. Presek zraka sa poluravni $x = 4$.



Slika 8. Presek zraka sa poluravni $y = 3$.



Slika 9. Presek zraka sa poluravni $x = 5$ i prvi presek sa pikselnim telom u matrici.

pravougaonika definisanog temenima (0, 0) i (7, 4).

Budući da piksel na poziciji (4, 3) ne postoji, računaju se parametri za sledeću iteraciju petlje. Promenljiva **tNext** se računa kao zbir vektora **t** i **dt**, pa se tako dobija vrednost (6.008, 7.205) što su vremena kada zrak dostiže poluravan $x = 5$, odnosno $y = 4$. Indeks minimalnog elementa ovog vektora **k** je takođe 0, što znači da zrak preseca ravan $x = 5$ pre nego $y = 4$, odnosno da se sledeći piksel na putanji nalazi u pozitivnom smeru trenutne ose u odnosu na trenutni piksel, drugim rečima – desno od njega.

Tako promenljiva **v0** dobija vrednost (5, 3), a vreme preseka sa sledećim vokselom postaje (6.008, 5.405). U sledećeoj iteraciji petlje zrak dolazi do prvog piksela i vraća se pogodak (slika 9).

Na listingu 17 prikazana je implementacija metoda **firstHit** klase **GridMarch2**. Računanje preseka sa ograničavajućom kutijom refaktorisano je i sada izvedeno u pomoćnom metodu nadklase **BaseM**, gde je takođe ponuđen i metod **len** koji vraća dimenzije ograničavajuće kutije (linije 4-10).

```

01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      Hit[] boundingBoxHits = getBoundingBoxHits(ray);
05      if (boundingBoxHits.length == 0) return Hit.POSITIVE_INFINITY;
06
07      Vec3[] loopData = Util.getLoopData(len(), ray, boundingBoxHits);
08
09      Vec3 v0 = loopData[0];
10      Vec3 v1 = loopData[1];
11
12      Vec3 s0 = ray.d().signum();
13      Vec3 s1 = s0.inverse();
14      Vec3 s2 = s1.add(Vec3.EXYZ).mul(0.5);
15
16      Vec3 t = Vec3.xyz(
17          HalfSpace.bn(v0.add(s2), s1.mul(Vec3.EX)).hits(ray)[0].t(),
18          HalfSpace.bn(v0.add(s2), s1.mul(Vec3.EY)).hits(ray)[0].t(),
19          HalfSpace.bn(v0.add(s2), s1.mul(Vec3.EZ)).hits(ray)[0].t());
20
21      Vec3 dt = s0.div(ray.d());
22
23      while (v0.inBoundingBox(v1)) {
24
25          if (isPopulated(v0) && t.max() > afterTime)
26              return new HitVoxel(
27                  ray, HitData.tn(t.max(), s1.mul(Vec3.E[t.maxIndex()])), v0);
28
29          Vec3 tNext = t.add(dt);
30          int k = tNext.minIndex();
31          v0 = v0.add(s0.mul(Vec3.E[k]));
32          t = t.add(dt.mul(Vec3.E[k]));
33      }
34
35      return Hit.POSITIVE_INFINITY;
36  }

```

Listing 17. Metod **firstHit** klase **GridMarch2**.

Promenljiva **s0** predstavlja ništa drugo do kompaktniju verziju izračunavanja vrednosti **loopData[2]**, odnosno vektor znaka pravca zraka koji govori o tome da li se zrak kreće u

pozitivnom ili negativnom smeru duž svake od tri ose. Promenljive $s1$ i $s2$ izvedene su iz vrednosti $s0$ i koriste se u izračunavanjima specifičnim za ovaj algoritam (linije 12-14).

Trodimenzionalni vektor t sadrži vremena u kojima zrak preseca ravnini definisane tačkom $v0.add(s2)$ i normalom $s1.mul(Vec3.E[i])$, gde je i indeks ose za koju se računa presek. Na ovaj način može se utvrditi trenutak u kom je zrak nesumnjivo unutar vokselne matrice, a to je element sa maksimalnom vrednošću vektora t (linije 16-19).

Parametar p prilikom definisanja ravnini računa se kao zbir vektora $v0$ i $s2$, gde je $v0$ ulazna tačka preseka zraka sa matricom, a $s2$ „korektivni parametar” koji služi konstrukciji ravnini sa odgovarajuće strane voksela kada se zrak kreće u negativnom smeru trenutne ose. Na primer, ukoliko bi $s0$, odnosno vektor znaka pravca zraka, bio jednak $(-1, 1, 1)$, tada bi $s1$ bio $(1, -1, -1)$, a $s2$ $(1, 0, 0)$ pa bi tako prilikom posmatranja kretanja zraka po X osi ravnini bile konstruisane za jedan voksel bliže pozitivnom delu ose što osigurava tačno izračunavanje. Parametar n , odnosno normala ravnini, računa se jednostavno odbacivanjem svih osim koordinate relevantne za posmatranu osu.

Trodimenzionalni vektor dt sadrži vremena potrebna da zrak po X, Y, odnosno Z osi pređe iz jednog voksela u drugi (linija 21). Na primer, ukoliko zrak prolazi kroz tačku $(1, 0.5)$ tada je normalizovani pravac vektora $(0.894, 0.447)^{11}$, $s0$ vrednost za takav vektor je $(1, 1)$, a dt je $(1.118, 2.23)$.

Slično algoritmu u klasi `GridMarch1` i ovde se za iteriranje kroz voksele na putanji zraka koristi samo jedna petlja, ali se provera sada svodi na utvrđivanje da li je promenljiva $v0$ - koja se ažurira pri svakoj iteraciji `while` petlje - izašla iz ograničavajuće kutije definisane koordinatnim početkom i pozicijom voksela kroz koji zrak izlazi iz matrice $v1$ (linija 23).

Pozicija trenutno aktuelnog voksela nalazi se u promenljivoj $v0$, dok se vreme za koje sa sigurnošću može tvrditi da se zrak nalazi unutar tog voksela izračunava kao maksimalni element vektora t . Ukoliko je voksel zauzet i ukoliko je vreme preseka veće od parametra `afterTime`, vraća se pogodak na do sada uobičajeni način (linije 25-27).

Ukoliko to nije slučaj potrebno je ažurirati promenljive $v0$ i t za potrebe sledeće iteracije petlje. Trodimenzionalni vektor $tNext$ sadrži vremena u kojima zrak preseca ravan na sledećem podeoku u pravcu zraka za svaku od ose (linija 29). Drugim rečima, u trenutku $tNext[0]$ zrak će preseći ravan za jedan voksel u pozitivnom ili negativnom smeru X ose u zavisnosti od smera zraka, u $tNext[1]$ ravan za jedan voksel u pozitivnom ili negativnom smeru Y ose, analogno za $tNext[2]$ i Z osu.

Dalje se traži indeks minimalnog elementa vektora $tNext$. Budući da je uslov petlje taj da tačka $v0$ bude unutar ograničavajuće kutije definisane koordinatnim početkom i tačkom $v1$, može se sa sigurnošću tvrditi da će zrak u momentu koji uzima vrednost bilo kog elementa $tNext$ biti unutar vokselne matrice; međutim, osnovna ideja algoritma je pronalaženje tačno onog suseda u koji dalje prelazi zrak, a to je upravo po onoj osi koja ima najmanju vrednost vremena sledećeg preseka (linija 30).

Pozicija sledećeg voksela sada se određuje inkrementiranjem koordinate sa indeksom k , odnosno elementa vektora $v0$ koji odgovara minimalnom elementu vektora $tNext$. Vreme preseka

¹¹ Navedene decimalne vrednosti u ovom poglavlju su približne usled nepreciznosti operacija sa brojevima sa pokretnim zarezom.

sa sledećim vokselom ažurira se dodavanjem samo odgovarajućeg elementa vektora dt vektoru t (linije 31-32).

Konačno, klasa `GridMarch20` sadrži optimizovanu implementaciju predstavljenog algoritma prikazanu na listingu 19. U ovoj verziji odsustvuje računanje izlaznog voksla metodom `getLoopData` budući da je nepotrebno, već se direktno računa ulazni voksel $v\theta$ (linije 7-12). Izračunavanje promenljivih $s0$, $s1$ i $s2$ je nepromenjeno (linije 14-16).

```

01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      Hit[] boundingBoxHits = getBoundingBoxHits(ray);
05      if (boundingBoxHits.length == 0) return Hit.POSITIVE_INFINITY;
06
07      Vec3 vθ = ray.at(boundingBoxHits[0].t()).floor();
08      Vec3 uθ = Vec3.xyz(
09          vθ.xInt() == lenX() ? -1 : (vθ.xInt() == -1 ? 1 : 0),
10         vθ.yInt() == lenY() ? -1 : (vθ.yInt() == -1 ? 1 : 0),
11         vθ.zInt() == lenZ() ? -1 : (vθ.zInt() == -1 ? 1 : 0));
12      vθ = vθ.add(uθ);
13
14      Vec3 s0 = ray.d().signum();
15      Vec3 s1 = s0.inverse();
16      Vec3 s2 = s1.add(Vec3.EXYZ).mul(0.5);
17
18      Vec3 t = vθ.add(s2).sub(ray.p()).div(ray.d());
19      Vec3 dt = s0.div(ray.d());
20
21      Vec3[] s0A = new Vec3[]
22          { s0.mul(Vec3.E[0]), s0.mul(Vec3.E[1]), s0.mul(Vec3.E[2]) };
23      Vec3[] dtA = new Vec3[]
24          { dt.mul(Vec3.E[0]), dt.mul(Vec3.E[1]), dt.mul(Vec3.E[2]) };
25
26      while (true) {
27
28          int k = t.maxIndex();
29          double tMax = t.get(k);
30
31          if (isPopulated(vθ) && tMax > afterTime)
32              return new HitVoxel(ray, HitData.tn(tMax, s1.mul(Vec3.E[k])), vθ);
33
34          Vec3 tNext = t.add(dt);
35          k = tNext.minIndex();
36          vθ = vθ.add(s0A[k]);
37
38          if (vθ.get(k) == -1 || vθ.get(k) == lenA()[k]) break;
39
40          t = t.add(dtA[k]);
41      }
42
43      return Hit.POSITIVE_INFINITY;
44  }

```

Listing 19. Metod `firstHit` klase `GridMarch20`.

Računanje promenljive t u ovoj implementaciji značajno je drugačije (linija 18). Na vektor $v\theta$ se prvo dodaje $s2$, čime se teme voksla na poziciji $v\theta$ pomera za pola jedinice duž smera zraka, čime se izračunavanje centrirano na stranicu voksla sa kojom je najverovatnije da će se zrak prvo susresti. Oduzimanjem početne tačke zraka od $v\theta.add(s2)$ dobija se vektor od početne tačke zraka do centrirane tačke na granici voksla, što predstavlja put potreban zraku da od početne tačke

dode do granice voksele. Konačno, vreme preseka računa se skaliranjem dobijenog vektora vektorom pravca zraka. Promenljiva dt se računa na isti način kao i ranije (linija 19).

Niz $s0A$ sadrži potencijalne susede trenutnog voksela u koje zrak može da pređe u sledećem koraku prolaska, i računaju se na osnovu vektora znaka pravca zraka (linija 21), dok vektor $\text{dt}A$ sadrži vremena preseka sa tim susedima (linija 22).

Za vreme jednakog vrednosti maksimalnog elementa vektora t sa sigurnošću se tvrdi da je zrak unutar voksela, pa se tako ukoliko je pozicija u matrici zauzeta vraća pogodak kao i u prethodnoj implementaciji (linije 28-32), inače je potrebno izračunati elemente koje će koristiti sledeća iteracija. Promenljiva $tNext$ predstavlja vremena preseka sa sva tri moguća suseda trenutnog voksela, k predstavlja indeks najmanjeg elementa vektora $tNext$, odnosno suseda sa kojim će se zrak najpre susrest, i konačno $v0$ uzima vrednost sledećeg voksela koji zrak preseca (linije 34-36).

U slučaju izlaska van okvira matrice petlja se prekida (linija 38) i vraća se konstanta `Hit.POSITIVE_INFINITY`, inače se vektoru t dodaje dt i pokreće se nova iteracija (linija 40).

Ovako definisana realizacija *grid march* algoritma ostavlja malo prostora za poboljšanja. Zaista, u nastavku rada izloženi rezultati praktične primene algoritama potvrđiće ovu tvrdnju. Ipak, problemu optimizacije renderovanja vokselnih objekata moguće je prići i iz drugog ugla i tako ostvariti bolje rezultate u specifičnim slučajevima. O tome će biti reči u narednom potpoglavlju.

3.3.5 Octree brute force algoritam i klasa OctreeBF

Oktalna stabla su strukture podataka u kojima svaki čvor ima osam podčvorova, i često se koriste za particionisanje trodimenzionalnog prostora rekursivnim deljenjem istog na osam oktanata.

Vokselni objekti mogu se predstaviti ovakvim strukturama čime se otvara mogućnost za reprezentaciju modela u različitim rezolucijama – svaki nivo stabla aproksimira trodimenzionalne podatke prethodnog nivoa, ali redukovanih dimenzija. Što je viši nivo, manji je broj voksele dimenzija većih od 1, od kojih svaki sumira grupu od osam voksele nižeg nivoa. Ukoliko je voksel višeg nivoa prisutan, najmanje jedan voksel od osam mogućih nižeg nivoa je takođe popunjeno, inače se sa sigurnošću može tvrditi da na nižim nivoima ne postoje vokseli u modelu.

Klasa `Octree` prikazana na listingu 20 bavi se kreiranjem oktalnog stabla od prosleđenog modela kakav su koristili do sada opisani algoritmi. Samo stablo je tehnički realizovano kao tip podataka `boolean[][][][],` gde je prva dimenzija, odnosno dubina niza jednaka $\log_2 \text{dim} + 1$, gde je dim širina ograničavajuće kocke koja obuhvata vokselnu matricu modela, dok su preostale tri dimenzije jednake 2^{depth} , gde je depth trenutna vrednost dubine niza.

Na primer, ukoliko se učitava model koji se može smestiti u ograničavajuću kocku stranica 16, tada će dubina niza biti $\log_2 16$, odnosno 4, dok će po dubinama biti kreirani nizovi sledećih dimenzija: `[16][16][16]` na dubini 0 (prvi nivo sadrži sam model), `[8][8][8]` na dubini 1, `[4][4][4]` na dubini 2, `[2][2][2]` na dubini 3, i `[1][1][1]` na dubini 4 (poslednji nivo sadrži ograničavajuću kocku).

Opisani postupak konverzije vokselnog modela u oktalno stablo implementiran je u metodu `fromModel` klase `Octree`, prikazanom na listingu 20. Pomoćni metod `treeDepthFromModel` povećava dubinu niza za jedan za slučaj da prosleđeni model nije veličine jednake stepenu broja 2

```

01  public class Octree {
02
03      public static final int MAX_NODES = 8;
04
05      private final boolean[][][][] data;
06      private final int treeDepth;
07
08      private Octree(int treeDepth) {
09
10          this.treeDepth = treeDepth;
11
12          data = new boolean[treeDepth][][][];
13
14          for (int i = 0; i < treeDepth; i++) {
15              int dim = (int) Math.pow(2, treeDepth - i - 1);
16              data[i] = new boolean[dim][dim][dim];
17          }
18      }
19
20      private static int treeDepthFromModel(boolean[][][] model) {
21
22          double d = Math.log(model.length) / Math.log(2) + 1;
23          return (int) d + (d == (int) d ? 0 : 1);
24      }
25
26      public static Octree fromModel(boolean[][][] model) {
27
28          Octree o = new Octree(treeDepthFromModel(model));
29
30          o.copyModel(model);
31
32          for (int l = 0; l < o.treeDepth - 1; l++) {
33              for (int i = 0; i < o.data[l].length; i++) {
34                  for (int j = 0; j < o.data[l].length; j++) {
35                      for (int k = 0; k < o.data[l].length; k++) {
36                          o.data[l + 1][i / 2][j / 2][k / 2] |= o.data[l][i][j][k];
37
38          return o;
39      }
40
41      private void copyModel(boolean[][][] model) {
42
43          for (int i = 0; i < model.length; i++) {
44              for (int j = 0; j < model[0].length; j++) {
45                  for (int k = 0; k < model[0][0].length; k++) {
46                      data[0][i][j][k] = model[i][j][k];
47      }
48
49      public boolean[][][][] data() { return data; }
50      public int treeDepth() { return treeDepth; }
51  }

```

Listing 20. Klasa Octree.

(linija 23). Koristeći četiri petlje oktalno stablo se iterativno popunjava na osnovu informacija u trenutnoj dubini stabla (linije 32-36).

Sam algoritam ponovo je implementiran kao metod `firstHit` klase `OctreeBF` i prikazan je na listingu 21. Osnovna ideja je iteracija kroz različite nivoje stabla počevši od ograničavajuće kocke, sve do pojedinačnih voksela na najnižem nivou. Kako je već rečeno, ukoliko ne postoji presek sa elementu na višem nivou iteriranje za dati element i njegove podčvorove se prekida.

```

01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      boolean[][][] b = model();
05
06      List<Vec3> l0 = new ArrayList<>();
07      List<Vec3> l1 = new ArrayList<>();
08
09      l0.add(Vec3.ZERO);
10
11      for (int l = b.length - 1; l > 0; l--) {
12
13          int unit = 1 << (l - 1);
14          Vec3 unitVec3 = Vec3.EXYZ.mul(unit);
15
16          for (Vec3 v : l0) {
17              for (int i = 0; i < 8; i++) {
18
19                  Vec3 idx = getVec3FromIndex(i);
20                  Vec3 pos = v.mul(2);
21                  Vec3 currPos = pos.add(idx);
22
23                  Vec3 p = currPos.mul(unit);
24
25                  Hit[] hits = Box.$.pd(p, unitVec3).hits(ray);
26
27                  if (isPopulated(l - 1, currPos) &&
28                      hits.length > 0 && hits[0].t() > afterTime)
29                      l1.add(currPos);
30
31          }
32
33          l0.clear(); l0.addAll(l1);
34          l1.clear();
35      }
36
37      List<Vec3Hit> vec3HitList = new ArrayList<>();
38
39      for (Vec3 v : l0) {
40
41          Hit hit = Box.$.pd(v, Vec3.EXYZ).firstHit(ray, afterTime);
42
43          if (hit != Hit.POSITIVE_INFINITY && hit.t() > afterTime)
44              vec3HitList.add(new Vec3Hit(v, hit));
45      }
46
47      if (!vec3HitList.isEmpty()) {
48          vec3HitList.sort(Comparator.comparingDouble(Vec3Hit::t));
49          return new HitVoxel(
50              ray, vec3HitList.get(0).h(), vec3HitList.get(0).v());
51      }
52
53      return Hit.POSITIVE_INFINITY;
54  }
55
56  private Vec3 getVec3FromIndex(int idx) {
57
58      int f = idx & 1;
59      int g = (idx >> 1) & 1;
60      int h = (idx >> 2) & 1;
61
62      return Vec3.xyz(f, g, h);
63  }

```

Listing 21. Metod `firstHit` klase `OctreeBF`.

Na početku metoda inicijalizuju se elementi poput oktalog stabla u promenljivoj `b` i liste koja sadrži pogotke na prethodnom nivou `10`, odnosno analogne liste za trenutni nivo `11`. Lista `10` inicijalno sadrži koren stabla, odnosno nula vektor koji predstavlja ograničavajuću kocku (linije 4-9).

Iteriranje kroz stablo počinje od najvišeg nivoa koji sadrži samo koren stabla (linija 11), i na početku svake iteracije izračunava se promenljiva `unit` koja sadrži veličinu čvora na trenutnom nivou, kao i vektorska varijanta iste informacije u vidu promenljive `unitVec3` (linije 13-14). Zatim se za svaki čvor u listi `10` proverava da li se upadni zrak seče sa ma kojim od njegovih osam podčvorova, što zahteva konstrukciju kutije stranica `unit` na odgovarajućoj poziciji.

Promenljiva `idx` predstavlja vektorskog poziciju oktanta u odnosu na roditelja, odnosno predstavlja njegovu relativnu poziciju, dok promenljiva `pos` sadrži poziciju roditelja na trenutnom nivou skaliranu sa `2`, tako da kada se ove dve vrednosti sabiju se `currPos`, odnosno apsolutna pozicija oktanta u jedinicama, koja se zatim skalira promenljivom `unit` kako bi se dobila pozicija kutije na sceni. Preseci sa zrakom se tada lako računaju na uobičajen način i ukoliko do istih dođe, pozicija oktanta se dodaje u listu `11` (linije 19-29).

Nakon prolaska kroz svih osam oktanata, lista `10` se prazni i zatim puni sadržajem `11`, dok se `11` nakon toga takođe prazni kako bi se pripremili za sledeću iteraciju (linije 33-34). Nakon iteracije kroz sve nivoe stabla lista `10` sadrži voksel koji se zaista preseca sa zrakom.

Jasno je da ovaj algoritam nije zamišljen kao optimizacija prethodno opisanog *grid march* pristupa. Kao i u slučaju *brute force* rešenja opisanog u potpoglavlju 3.3.1, ova implementacija služiće kao referentna tačka za vrednovanje optimizacija koje će biti uvedene u poslednjem algoritmu kojim se ovaj rad bavi.

3.3.6 Octree brute force algoritam i klase OctreeRec i OctreeRecO

Klasa `OctreeRec` sadrži implementaciju `firstHit` metoda koji preseke zraka sa vokselima pronalazi rekurzivnim DFS prolaskom kroz stablo (listing 22). Metod `firstHit` oslanja se na metod `octreeDFS` koji predstavlja rekurzivni poziv kojim se zapravo izvršava pretraga (listing 23).

```

01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      List<Vec3Hit> container = new ArrayList<>();
05
06      octreeDFS(container, Vec3.ZERO, model().length - 1, ray, afterTime);
07
08      if (container.isEmpty()) return Hit.POSITIVE_INFINITY;
09
10      if (container.get(0) != Hit.POSITIVE_INFINITY &&
11          container.get(0).t() > afterTime)
12          return new HitVoxel(
13              ray, container.get(0).h(), container.get(0).v());
14
15  } return Hit.POSITIVE_INFINITY;
16 }
```

Listing 22. Metod `firstHit` klase `OctreeRec`.

```

01 private void octreeDFS(List<Vec3Hit> container, Vec3 curr,
02     int lvl, Ray ray, double afterTime) {
03
04     int unit = 1 << (lvl - 1);
05     Vec3 unitVec3 = Vec3.EXYZ.mul(unit);
06
07     int xs, xe, xd, ys, ye, yd, zs, ze, zd;
08
09     if (ray.d().x() >= 0) {
10         xs = 0; xe = 2; xd = +1;
11     } else {
12         xs = 1; xe = -1; xd = -1;
13     }
14
15     if (ray.d().y() >= 0) {
16         ys = 0; ye = 2; yd = +1;
17     } else {
18         ys = 1; ye = -1; yd = -1;
19     }
20
21     if (ray.d().z() >= 0) {
22         zs = 0; ze = 2; zd = +1;
23     } else {
24         zs = 1; ze = -1; zd = -1;
25     }
26
27     for (int i = xs; i != xe; i += xd) {
28         for (int j = ys; j != ye; j += yd) {
29             for (int k = zs; k != ze; k += zd) {
30
31                 Vec3 idx = Vec3.xyz(i, j, k);
32                 Vec3 pos = curr.mul(2);
33                 Vec3 currPos = pos.add(idx);
34
35                 Vec3 p = currPos.mul(unit);
36
37                 if (isPopulated(lvl - 1, currPos)) {
38
39                     Hit hit = Box.$.pd(p, unitVec3).firstHit(ray, afterTime);
40
41                     if (hit != Hit.POSITIVE_INFINITY)
42                         if (lvl > 1)
43                             octreeDFS(container, currPos, lvl - 1, ray, afterTime);
44                         else
45                             container.add(new Vec3Hit(currPos, hit));
46
47                 }
48             }
49         }
50     }

```

Listing 23. Metod `octreeDFS` klase `OctreeRec`.

Metod `firstHit` u ovom slučaju je trivijalan i u potpunosti se oslanja na rezultate koje vraća poziv metoda `octreeDFS`. Na početku metoda `octreeDFS` inicijalizuju se veličina oktanta (linije 4-5), početne i krajnje tačke petlji, kao i inkrement za svaku od tri ose u zavisnosti od smera prosleđenog zraka (linije 7-25) i zatim se pokreću petlje (linije 27-29). Presek zraka sa oktantom računa se na isti način kao i u `octreeBF` implementaciji (linije 31-39). U slučaju da je registrovan pogodak, rekurzivno se poziva `octreeDFS` nad trenutnim oktantom, odnosno voksel se dodaje u izlaznu listu ako je trenutni nivo najniži nivo modela (linije 41-45).

```

01  private void octreeDFS(Vec3 curr,
02      int lvl, Ray ray, double afterTime) {
03
04      int unit = 1 << (lvl - 1);
05      Vec3 unitVec3 = Vec3.EXYZ.mul(unit);
06
07      int xs, xe, xd, ys, ye, yd, zs, ze, zd;
08
09      if (ray.d().x() >= 0) {
10          xs = 0; xe = 2; xd = +1;
11      } else {
12          xs = 1; xe = -1; xd = -1;
13      }
14
15      if (ray.d().y() >= 0) {
16          ys = 0; ye = 2; yd = +1;
17      } else {
18          ys = 1; ye = -1; yd = -1;
19      }
20
21      if (ray.d().z() >= 0) {
22          zs = 0; ze = 2; zd = +1;
23      } else {
24          zs = 1; ze = -1; zd = -1;
25      }
26
27      for (int i = xs; i != xe; i += xd) {
28          for (int j = ys; j != ye; j += yd) {
29              for (int k = zs; k != ze; k += zd) {
30
31                  Vec3 idx = Vec3.xyz(i, j, k);
32                  Vec3 pos = curr.mul(2);
33                  Vec3 currPos = pos.add(idx);
34
35                  Vec3 p = currPos.mul(unit);
36
37                  if (isPopulated(lvl - 1, currPos)) {
38
39                      Hit hit = Box.$.pd(p, unitVec3).firstHit(ray, afterTime);
40
41                      if (hit != Hit.POSITIVE_INFINITY) {
42                          if (lvl > 1) {
43                              Vec3Hit pp = octreeDFS(currPos, lvl - 1, ray, afterTime);
44                              if (pp.h() != Hit.POSITIVE_INFINITY) return pp;
45                          } else {
46                              return new Vec3Hit(currPos, hit);
47                          }
48                      }
49                  }
50              }
51          }
52      }
53  }

```

Listing 23. Metod `octreeDFS` klase `OctreeRecO`.

Klasa `OctreeRecO` implementira metod `octreeDFS` na identičan način (listing 23), s tim da prekida pretragu čim je prvi presek zraka sa vokselnim telom pronađen, odnosno nakon što se utvrđi da do pogotka ne dolazi (linije 41-49).

Kako je rečeno, od predstavljenih algoritama na osnovi oktalnih stabala se ne očekuje da u svakoj situaciji budu bolje rešenje od *grid march* algoritama. Pretpostavka je da će ovakvi algoritmi pokazati bolje performanse kod modela kod kojih je većina voksela prazno, budući da se

rekurzivnom pretragom stabla redukuje broj izračunavanja preseka zraka sa vokselima, što će u takvom slučaju uglavnom vraćati promašaje. Ove prepostavke biće testirane u nastavku rada.

Rezultati testiranja

Opisani algoritmi za renderovanje vokselnih tela biće testirani na modelima različite veličine, gustine i složenosti. Rezultati će biti dati kao vremena renderovanja prvih dvadeset iteracija, biće prikazani u tabelarnom vidu i međusobno upoređeni.

Osnovna pretpostavka rada je da će vreme potrebno za renderovanje jedne iteracije opadati u nizu od `BruteForce`, preko `DirArray0` do `GridMarch20` implementacije, takođe i od `OctreeBF` do `OctreeRec0`. Pored toga, očekuje se da će `OctreeRec0` pokazati bolje rezultate od `GridMarch20` implementacije kod objekata sa velikim matricama i malo voksela.

Sva merenja izvršena su na Apple M1 MacBook Pro računaru iz 2021. godine sa 16GB RAM-a.

4.1 Nasumično generisana vokselna kocka (6.25% popunjeno)

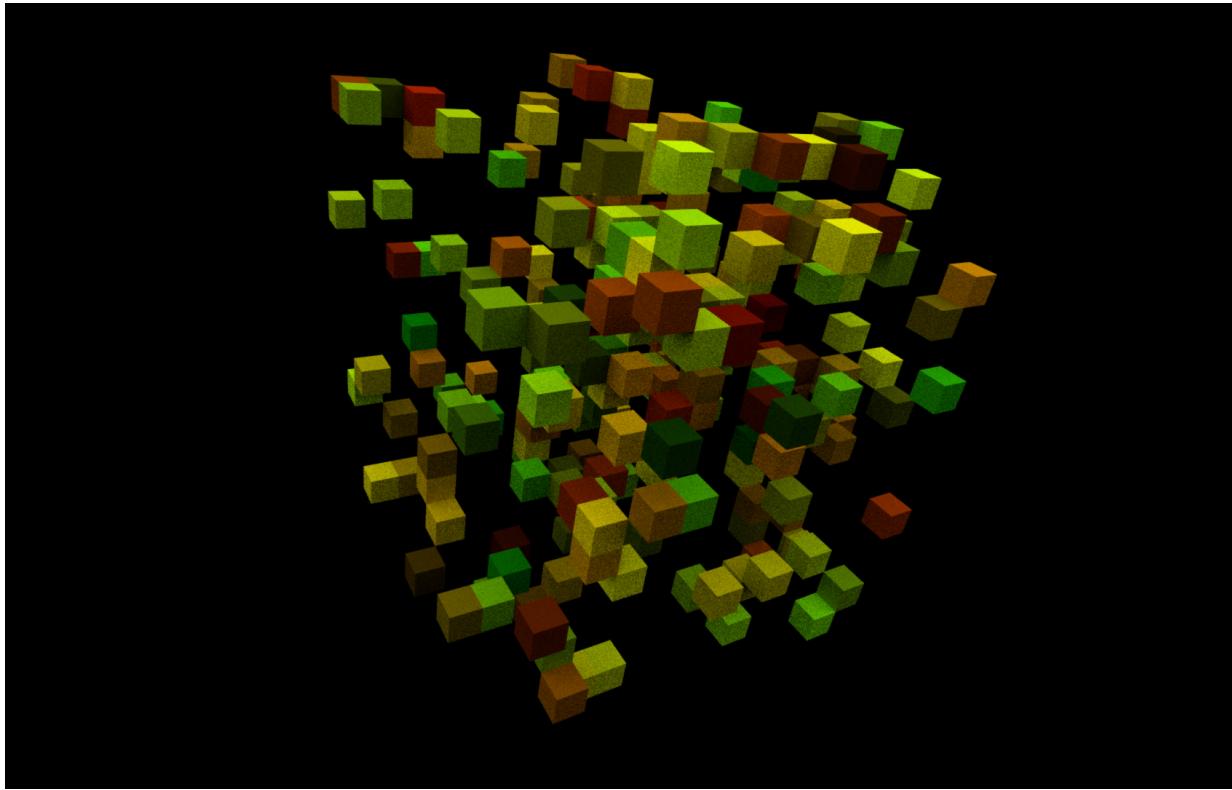
Prvi test model predstavlja vokselna kocka nasumično generisana kodom prikazanim na listingu 24.

Dužina stranice kocke određena je stepenom broja 2 datog promenljivom `lvl` (linija 3). Prolaskom kroz tri petlje vokseli se generišu sa verovatnoćom 6.25% i nasumičnom bojom biranom iz definisanog opsega (linije 15-18).

```

01 Random rng = new Random(seed);
02 int lvl = 5;
03 Vec3 dim = Vec3.xyz(
04     Math.pow(2, lvl), Math.pow(2, lvl), Math.pow(2, lvl));
05 boolean[][][] arr0 =
06     new boolean[dim.xInt()][dim.yInt()][dim.zInt()];
07 Color[][][] arr1 =
08     new Color[dim.xInt()][dim.yInt()][dim.zInt()];
09
10 for (int i = 0; i < dim.xInt(); i++) {
11     for (int j = 0; j < dim.yInt(); j++) {
12         for (int k = 0; k < dim.zInt(); k++) {
13             if (rng.nextDouble() < 0.0625) {
14                 arr0[i][j][k] = true;
15                 arr1[i][j][k] = Color.rgb(
16                     rng.nextDouble(), rng.nextDouble(), 0.0);
17             }
18         }
19     }
20 }
```

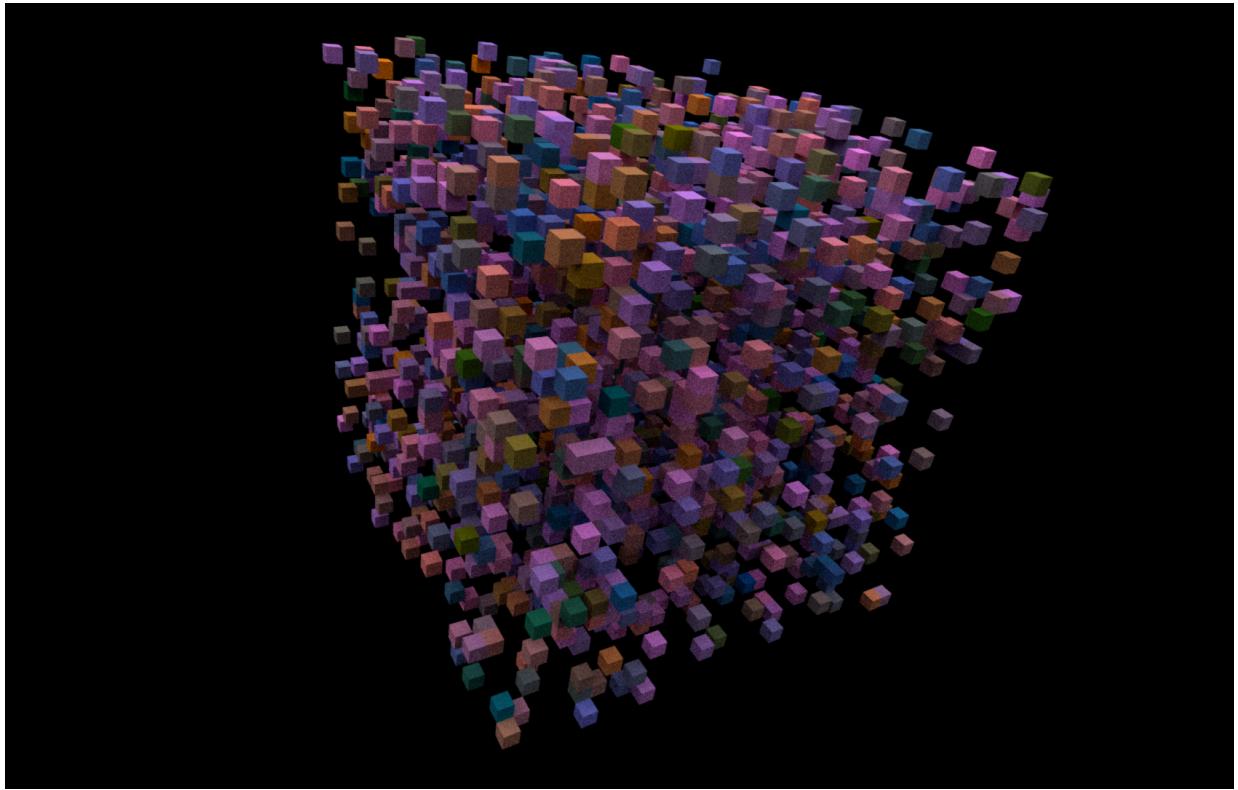
Listing 24. Generisanje vokselne kocke stranice dužine `Math.pow(2, lvl)`.



Slika 10. Nasumično generisana vokselna kocka dužine ivica 16, sa verovatnoćom pojave voksla 6.25% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GMI</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	156,015	48,972	8,140	39,224	5,705	6,690	27,737	25,220	22,820
2	156,701	47,941	7,889	38,967	5,585	6,737	27,661	25,427	20,282
3	157,185	47,431	7,835	38,684	5,484	6,784	27,729	26,045	18,331
4	156,967	47,308	8,516	38,796	5,490	6,805	26,839	26,459	17,358
5	158,141	47,044	9,091	38,866	5,483	6,771	26,218	26,683	17,491
6	159,314	46,978	9,470	38,997	5,631	6,739	25,816	26,828	17,593
7	159,521	46,820	9,759	38,971	5,860	6,705	26,081	26,954	17,685
8	159,029	46,753	9,989	38,984	6,047	6,685	26,491	27,096	17,791
9	158,717	46,758	10,156	38,958	6,187	6,654	26,172	27,188	17,867
10	158,532	46,750	10,291	38,976	6,103	6,637	25,902	27,281	17,924
11	158,876	46,746	10,370	38,995	6,157	6,623	25,672	27,363	17,984
12	159,245	46,734	10,459	39,011	6,196	6,605	25,442	27,443	18,036
13	160,102	46,715	10,558	39,028	6,223	6,587	25,294	27,517	18,093
14	160,325	46,698	10,621	39,041	6,241	6,572	25,102	27,589	18,153
15	160,785	46,681	10,695	39,055	6,253	6,556	24,875	27,657	18,211
16	160,921	46,664	10,742	39,069	6,275	6,541	24,742	27,721	18,264
17	161,149	46,648	10,796	39,083	6,287	6,528	24,620	27,783	18,315
18	161,491	46,633	10,825	39,096	6,301	6,517	24,511	27,841	18,365
19	161,758	46,619	10,872	39,108	6,319	6,505	24,423	27,898	18,412
20	161,982	46,605	10,901	39,120	6,335	6,494	24,357	27,952	18,458
avg	159,338	46,975	9,899	39,001	6,008	6,637	25,784	27,097	18,372
min	156,015	46,605	7,835	38,684	5,483	6,494	24,357	25,220	17,358
max	161,982	48,972	10,901	39,224	6,335	6,805	27,737	27,952	22,820

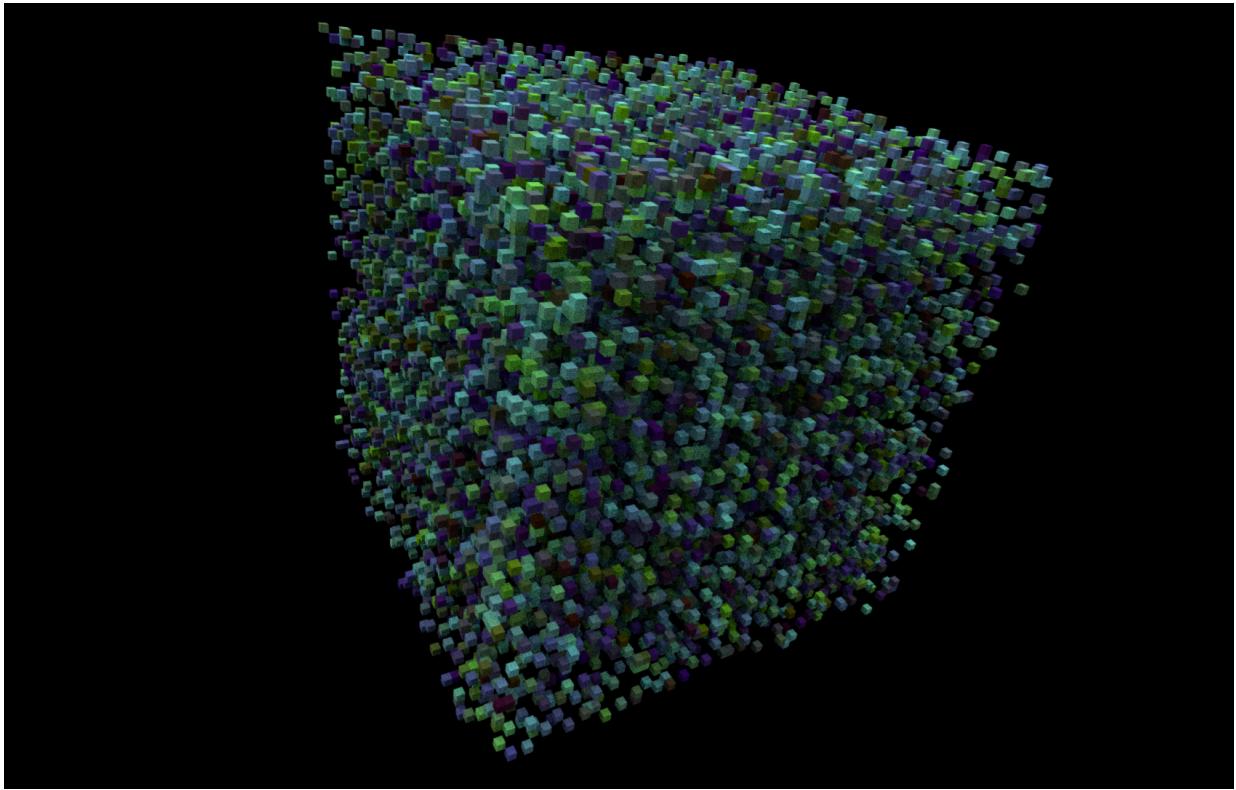
Tabela 1. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 16, sa verovatnoćom pojave voksla 6.25% i vrednošću *seed* parametra 129832191.



Slika 11. Nasumično generisana vokselna kocka dužine ivica 32, sa verovatnoćom pojave voksla 6.25% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GMI</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	1.686,15	1.365,55	50,393	93,455	7,083	4,183	59,063	58,871	46,630
2	1.689,89	1.289,80	51,112	93,153	6,855	4,240	57,670	60,410	46,768
3	1.680,16	1.239,69	51,566	94,175	7,045	4,177	57,726	61,002	47,214
4	1.692,25	1.228,22	52,542	90,621	6,986	4,157	58,054	61,525	43,820
5	1.688,24	1.212,54	53,032	88,193	6,929	4,348	58,053	59,954	41,958
6	1.684,00	1.202,08	53,233	84,788	6,894	4,302	58,479	59,188	40,653
7	1.684,90	1.194,00	53,364	82,121	6,910	4,273	58,557	58,022	39,548
8	1.684,90	1.196,75	53,555	80,789	6,915	4,235	58,646	58,082	38,921
9	1.686,67	1.198,67	53,745	79,862	6,871	4,240	58,765	60,419	38,485
10	1.685,19	1.202,93	53,854	79,111	6,844	4,203	58,966	61,392	38,262
11	1.690,11	1.204,62	54,001	78,987	6,876	4,211	59,243	61,543	38,145
12	1.687,23	1.199,35	54,122	78,543	6,899	4,233	59,567	61,789	38,034
13	1.689,88	1.201,88	54,245	78,234	6,932	4,245	59,890	61,987	37,967
14	1.688,45	1.203,43	54,366	77,956	6,965	4,254	60,022	62,112	37,854
15	1.691,22	1.206,00	54,500	77,689	6,984	4,265	60,145	62,230	37,756
16	1.685,34	1.210,34	54,643	77,345	6,943	4,298	60,234	62,354	37,689
17	1.684,57	1.211,03	54,789	76,987	6,899	4,312	60,312	62,411	37,643
18	1.686,99	1.213,88	54,865	76,854	6,877	4,332	60,445	62,523	37,554
19	1.690,45	1.209,46	54,912	76,643	6,889	4,355	60,543	62,689	37,432
20	1.692,11	1.212,79	55,034	76,412	6,866	4,365	60,678	62,743	37,365
avg	1.687,43	1.220,15	53,594	82,096	6,923	4,261	59,253	61,062	39,985
min	1.680,16	1.194,00	50,393	76,412	6,844	4,157	57,670	58,022	37,365
max	1.692,25	1.365,55	55,034	94,175	7,083	4,365	60,678	62,743	47,214

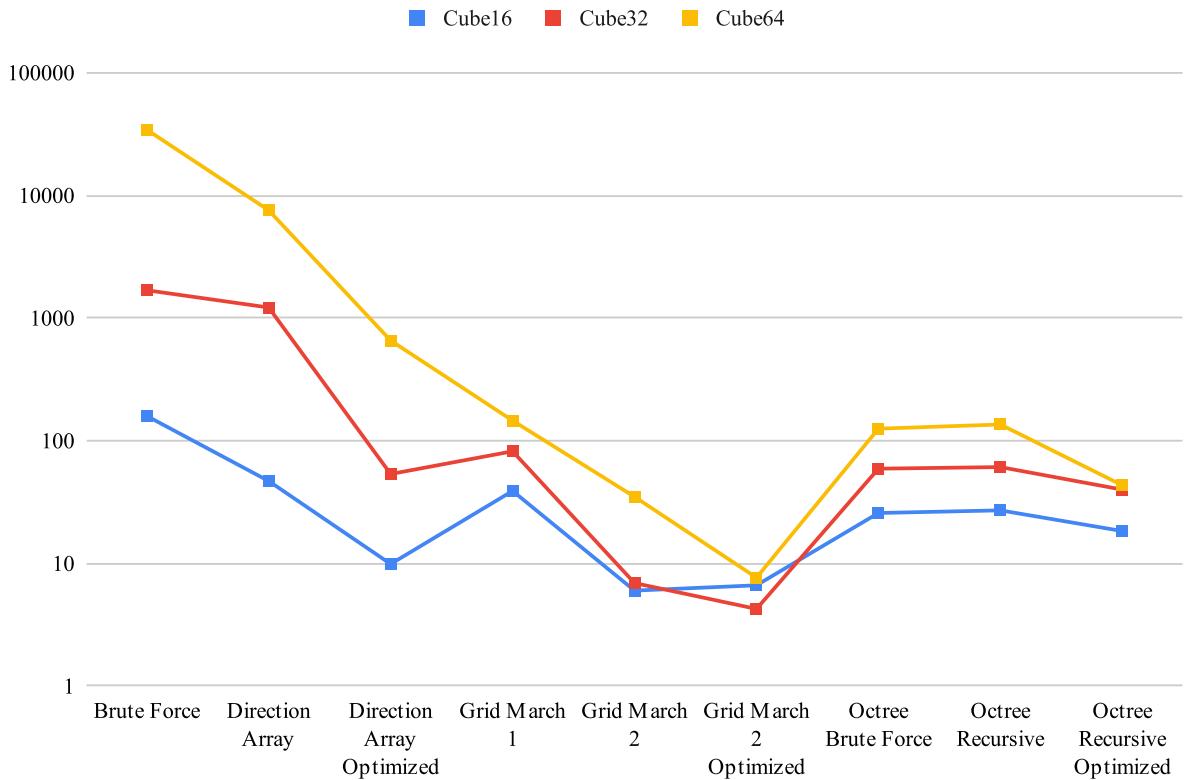
Tabela 2. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 32, sa verovatnoćom pojave voksla 6.25% i vrednošću *seed* parametra 129832191.



Slika 12. Nasumično generisana vokselna kocka dužine ivica 64, sa verovatnoćom pojave voksla 6.25% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GMI</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	34.342,37	7.543,24	634,341	142,688	36,548	8,226	125,931	122,106	41,155
2	34.343,35	7.544,22	627,841	142,563	37,013	7,472	122,203	130,524	41,115
3	34.347,65	7.548,52	634,867	143,239	36,413	7,387	121,030	132,818	41,799
4	34.349,71	7.550,58	666,029	147,314	37,252	7,963	122,378	135,181	42,268
5	34.350,61	7.551,48	682,189	150,225	38,305	7,779	121,734	135,741	42,330
6	34.349,08	7.549,95	680,686	148,685	37,807	7,727	122,591	135,870	42,534
7	34.352,00	7.552,87	674,602	147,852	36,485	7,820	123,177	135,821	42,753
8	34.350,75	7.551,62	672,081	147,737	35,604	7,887	123,797	135,904	43,245
9	34.358,58	7.559,46	668,250	147,787	34,976	7,799	124,413	136,152	43,539
10	34.367,86	7.568,73	663,631	147,460	34,500	7,785	125,117	136,408	43,547
11	34.369,14	7.570,32	661,543	146,987	34,320	7,743	125,654	136,754	43,654
12	34.370,60	7.572,65	659,230	146,543	34,145	7,689	126,243	137,021	43,798
13	34.372,25	7.575,14	657,980	146,234	33,987	7,643	126,897	137,432	43,987
14	34.375,38	7.577,90	655,431	145,875	33,756	7,598	127,354	137,789	44,145
15	34.378,69	7.579,35	653,761	145,432	33,543	7,532	127,789	137,932	44,243
16	34.380,90	7.580,89	652,120	145,298	33,321	7,489	128,123	138,145	44,367
17	34.381,57	7.582,43	650,489	144,954	33,089	7,456	128,543	138,354	44,543
18	34.384,76	7.583,97	648,320	144,765	32,876	7,422	128,965	138,543	44,632
19	34.388,34	7.586,32	645,981	144,678	32,743	7,398	129,210	138,765	44,798
20	34.391,46	7.588,67	644,298	144,543	32,610	7,365	129,543	138,987	44,932
avg	34.365,25	7.565,92	656,684	146,043	34,965	7,659	125,535	135,812	43,369
min	34.342,37	7.543,24	627,841	142,563	32,610	7,365	121,030	122,106	41,115
max	34.391,46	7.588,67	682,189	150,225	38,305	8,226	129,543	138,987	44,932

Tabela 3. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 64, sa verovatnoćom pojave voksla 6.25% i vrednošću *seed* parametra 129832191.



Grafik 1. Srednje vrednosti vremena renderovanja nasumično generisane vokselne kocke stranica dužine 16, 32 i 64 sa verovatnoćom pojave voksla 6.25%.

Rezultati merenja za vokselne kocke dužine ivica 16 prikazani su na slici 10 i tabeli 1, za kocku dužine ivica 32 na slici 11 i tabeli 2, i za kocku dužine ivica 64 na slici 12 i tabeli 3. Grafički prikaz srednje vrednosti vremena renderovanja kocaka različitih dužina ivica različitim algoritmima na logaritamskoj skali dat je na grafiku 1.

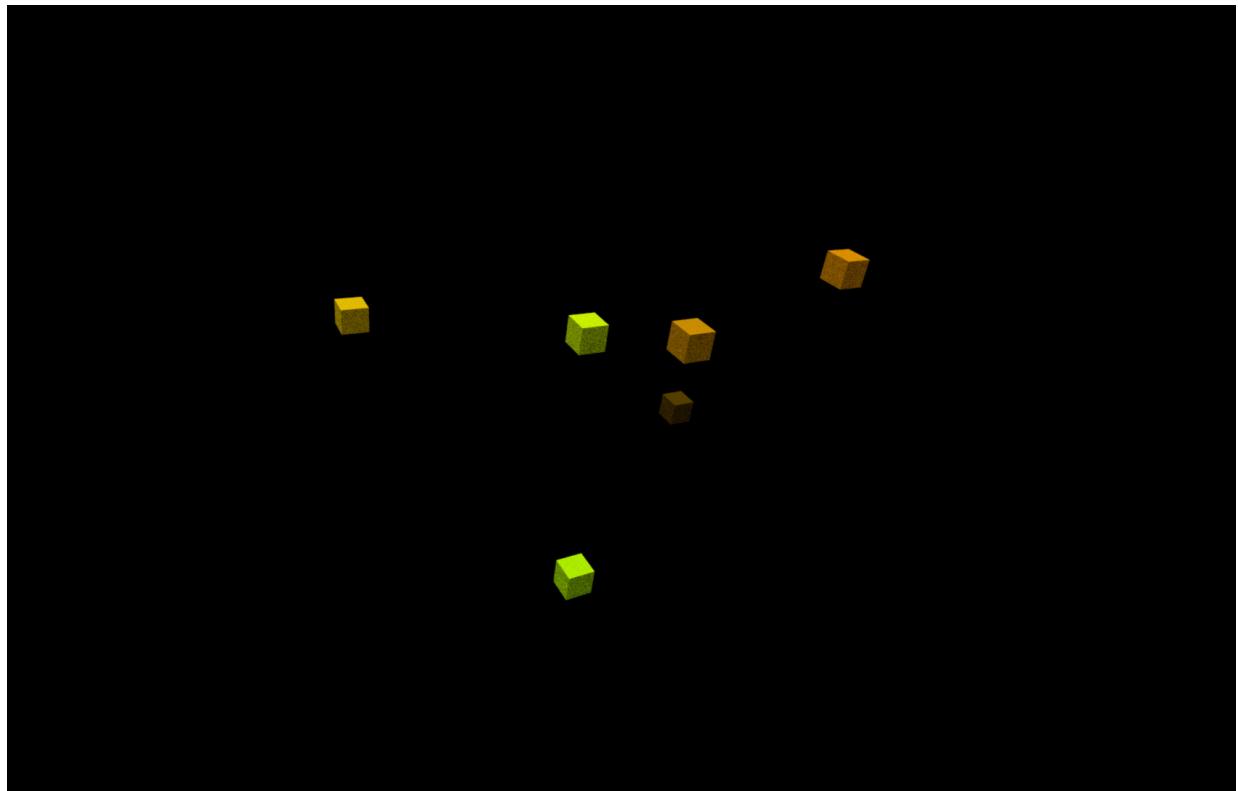
Korišćeni algoritmi mogu se podeliti u četiri grupe: *brute force* kao referentni, *direction array* algoritmi, *grid march* algoritmi, i algoritmi na osnovi oktalnih stabala. U svakoj od grupa sa više od jednog algoritma očekivano se primećuje optimizacija kod kasnijih implementacija.

Kod manjih modela implementacija *grid march 1* algoritma zaostaje u performansama za optimizovanom *direction array* implementacijom, ali kod većih modela ta razlika nestaje, tako da se kod kocke dimenzija 64 primećuje gotovo linearni pad vremena renderovanja u zavisnosti od "količine optimizacije" počevši od *brute force* rešenja do optimizovanog *grid march 2* algoritma.

Algoritmi na osnovi oktalnih stabala pokazuju rezultate slične *grid march 1* algoritmu za neoptimizovane varijante, odnosno *grid march 2* algoritmu za optimizovanu varijantu *octree recursive* rešenja. Varijante koje iterativno, odnosno rekurzivno, prolaze kroz stablo očekivano pokazuju gotovo identične rezultate. Poboljšanje koje se primećuje kod optimizovane *octree recursive* varijante rezultat je ranog prekida izvršavanja **firstHit** metoda u slučaju kada se pronađe prvi pogodak.

Grafik takođe pruža uvid u činjenicu da performanse *brute force* i *direction array* algoritama opadaju sa porastom veličine modela, dok se isto ne primećuje kod *grid march* ili algoritama na osnovi oktalnih stabala. Naročito su interesantni rezultati optimizovane *grid march 2* implementacije koji naizgled nimalo ne zavise od veličine modela.

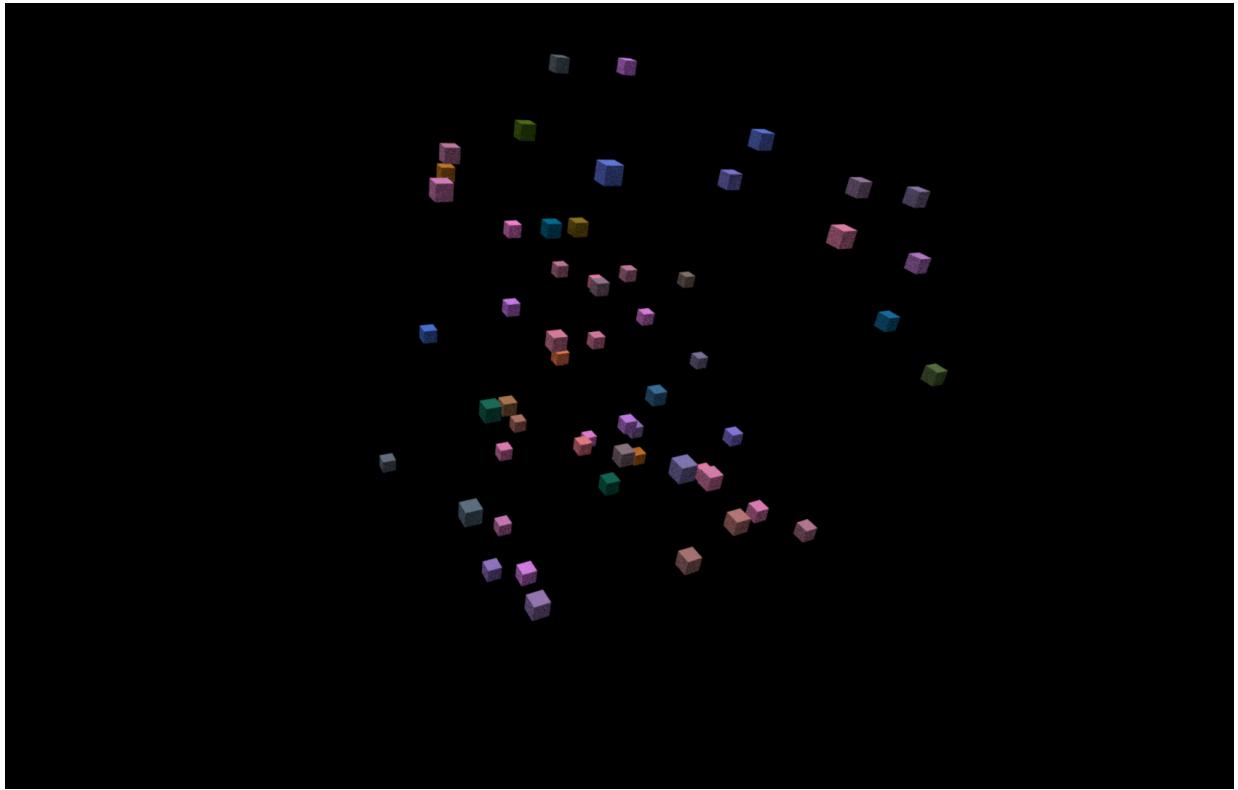
4.2 Nasumično generisana vokselna kocka (0.19% popunjenošću)



Slika 13. Nasumično generisana vokselna kocka dužine ivica 16, sa verovatnoćom pojave voksela 0.19% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GMI</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	54,93	11,40	2,707	14,240	7,403	3,093	15,722	5,835	5,645
2	56,20	11,25	2,446	13,962	7,265	2,736	15,360	5,591	5,455
3	55,39	10,93	2,321	13,989	7,303	2,602	15,382	5,518	5,465
4	53,47	10,95	2,284	13,617	7,621	2,639	15,494	5,507	5,527
5	50,26	10,83	2,312	13,880	7,262	2,799	15,811	5,519	5,517
6	48,17	10,84	2,266	14,025	6,956	2,904	15,891	5,508	5,494
7	46,65	10,81	2,251	14,087	6,712	2,971	16,014	5,501	5,468
8	45,65	10,81	2,218	14,146	6,508	3,002	15,758	5,592	5,468
9	44,74	10,80	2,215	14,215	6,362	3,050	15,641	5,698	5,469
10	44,08	10,82	2,199	14,222	6,254	3,080	15,590	5,794	5,464
11	43,48	10,80	2,204	14,248	6,173	3,108	15,558	5,833	5,455
12	43,01	10,81	2,181	14,255	6,100	3,144	15,513	5,890	5,449
13	42,63	10,89	2,198	14,242	6,038	3,165	15,512	5,932	5,443
14	42,30	10,91	2,190	14,262	5,991	3,186	15,501	5,967	5,445
15	42,01	10,89	2,190	14,221	5,932	3,200	15,483	6,010	5,448
16	41,73	10,89	2,196	14,229	5,913	3,218	15,462	6,031	5,447
17	41,46	10,93	2,195	14,237	5,877	3,229	15,440	6,051	5,458
18	41,27	10,97	2,189	14,257	5,846	3,240	15,418	6,075	5,456
19	41,12	11,03	2,182	14,292	5,813	3,250	15,393	6,099	5,452
20	41,02	11,06	2,176	14,314	5,795	3,259	15,375	6,110	5,450
avg	45,98	10,93	2,256	14,147	6,456	3,044	15,566	5,803	5,474
min	41,02	10,80	2,176	13,617	5,795	2,602	15,360	5,501	5,443
max	56,20	11,40	2,707	14,314	7,621	3,259	16,014	6,110	5,645

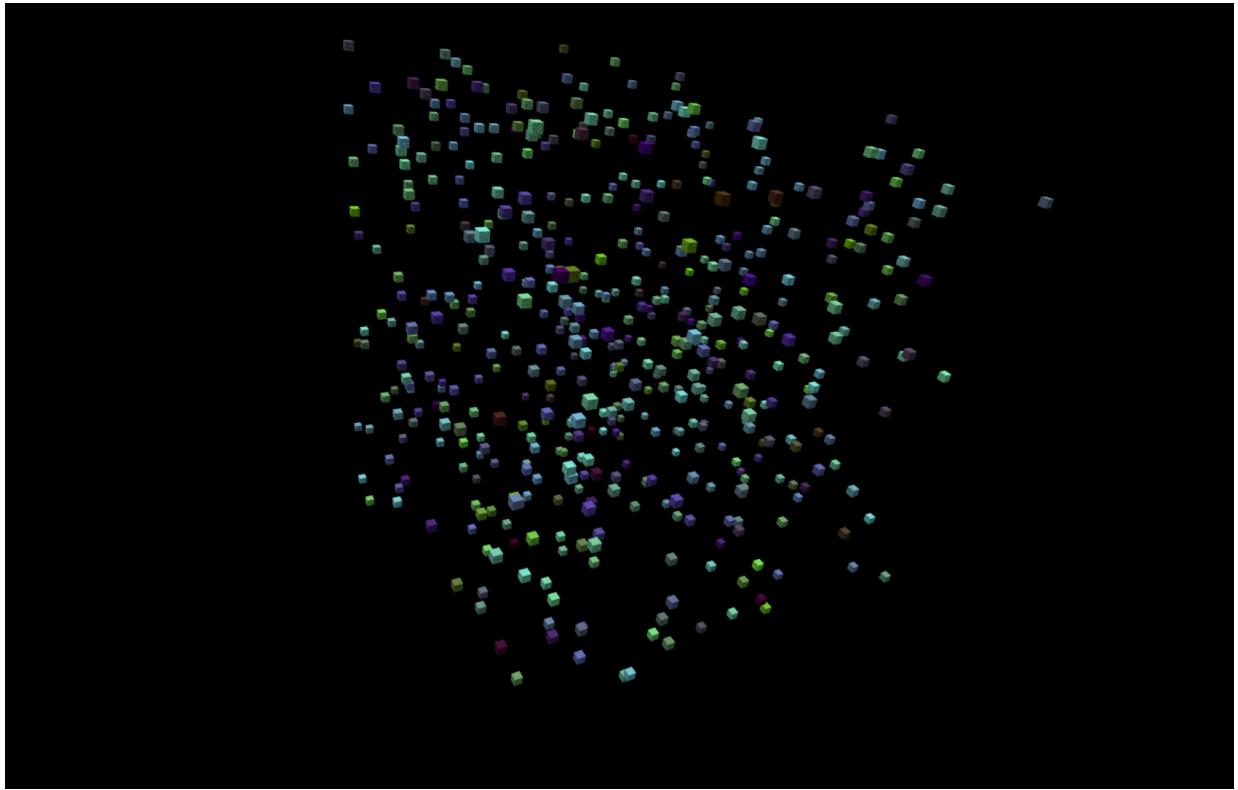
Tabela 4. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 16, sa verovatnoćom pojave voksela 0.19% i vrednošću *seed* parametra 129832191.



Slika 14. Nasumično generisana vokselna kocka dužine ivica 32, sa verovatnoćom pojave voksla 0.19% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GM1</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	299,88	70,47	4,612	37,460	7,686	10,820	15,631	11,236	17,948
2	296,64	71,14	4,652	37,363	8,496	9,106	15,948	11,126	16,892
3	231,94	71,19	4,842	37,101	9,217	8,355	16,747	12,682	16,527
4	199,37	71,45	4,885	37,287	9,666	8,069	17,090	13,590	16,402
5	179,57	71,19	4,941	37,327	9,825	7,831	17,297	14,145	16,338
6	166,32	70,99	4,962	37,249	9,978	7,695	17,367	14,479	16,280
7	157,02	70,82	4,978	37,200	10,104	7,560	17,482	14,796	16,259
8	149,79	70,69	4,987	37,182	10,219	7,478	17,503	14,938	16,259
9	144,33	70,67	4,992	37,144	10,299	7,412	17,577	15,110	16,221
10	140,34	71,01	4,991	37,202	10,352	7,347	17,656	15,157	16,196
11	137,24	70,80	5,017	37,242	10,413	7,317	17,727	15,252	16,188
12	134,41	70,63	5,015	37,103	10,443	7,294	17,713	15,293	16,186
13	132,26	70,64	5,010	36,981	10,491	7,272	17,719	15,350	16,186
14	130,10	70,54	5,001	36,977	10,506	7,244	17,714	15,428	16,179
15	128,41	70,45	5,015	37,023	10,549	7,233	17,790	15,465	16,185
16	126,91	70,36	5,021	37,017	10,561	7,222	17,837	15,540	16,166
17	125,55	70,34	5,024	36,985	10,547	7,217	17,896	15,586	16,150
18	124,28	70,36	5,031	36,975	10,575	7,208	17,896	15,624	16,135
19	123,20	70,40	5,024	36,948	10,580	7,201	17,936	15,657	16,130
20	122,21	70,32	5,020	36,964	10,600	7,194	17,955	15,712	16,123
avg	162,49	70,72	4,951	37,137	10,055	7,704	17,424	14,608	16,348
min	122,21	70,32	4,612	36,948	7,686	7,194	15,631	11,126	16,123
max	299,88	71,45	5,031	37,460	10,600	10,820	17,955	15,712	17,948

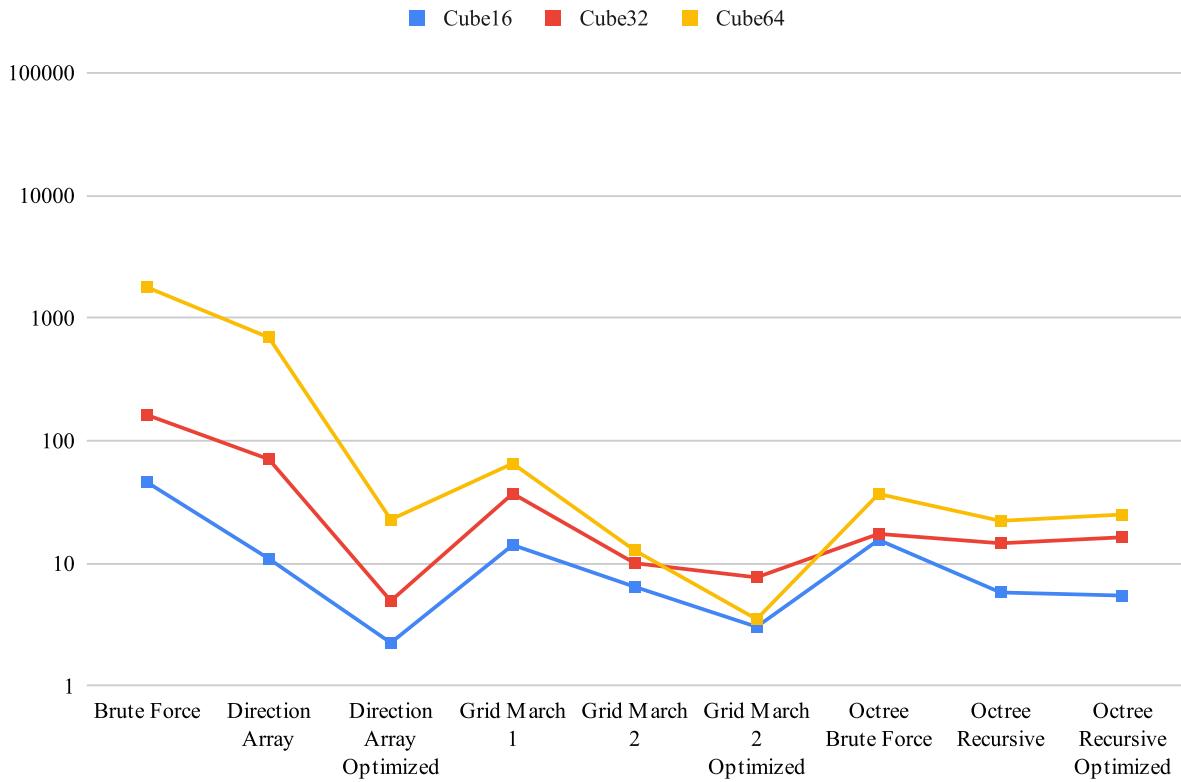
Tabela 5. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 32, sa verovatnoćom pojave voksla 0.19% i vrednošću *seed* parametra 129832191.



Slika 15. Nasumično generisana vokselna kocka dužine ivica 64, sa verovatnoćom pojave voksla 0.19% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GMI</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	1.264,523	585,716	23,041	66,276	12,237	3,710	33,260	22,374	31,415
2	3.209,33	621,833	22,435	65,286	12,446	3,587	31,373	23,291	30,720
3	3.482,58	645,308	22,527	64,553	12,805	3,686	32,946	22,471	30,655
4	2.762,62	648,899	22,699	65,028	12,729	3,583	34,841	22,169	28,949
5	2.378,55	651,411	22,867	64,701	13,216	3,536	35,864	21,966	27,325
6	2.117,35	687,522	22,779	64,931	13,218	3,539	36,544	21,831	26,297
7	1.934,94	691,124	22,734	65,092	13,288	3,525	36,527	21,955	25,562
8	1.798,08	694,887	22,649	64,977	13,134	3,504	37,129	21,982	24,972
9	1.691,40	698,761	22,641	64,897	13,009	3,497	37,445	21,991	24,408
10	1.608,67	702,843	22,892	65,007	12,918	3,497	37,377	21,983	24,033
11	1.537,66	707,031	22,906	65,149	12,799	3,477	37,648	22,002	23,729
12	1.475,96	711,320	22,790	65,187	12,774	3,473	37,702	22,079	23,412
13	1.421,63	715,715	22,790	65,283	12,680	3,474	38,048	22,138	23,161
14	1.375,06	720,234	22,803	65,388	12,711	3,466	38,213	22,231	23,020
15	1.338,88	724,869	22,789	65,525	12,647	3,454	38,332	22,307	22,768
16	1.309,37	729,625	22,771	65,559	12,623	3,470	38,557	22,374	22,550
17	1.281,30	734,498	22,835	65,631	12,581	3,463	38,743	22,429	22,337
18	1.255,67	739,493	22,976	65,673	12,540	3,459	38,697	22,442	22,155
19	1.231,54	744,613	23,103	65,761	12,526	3,450	38,858	22,452	22,007
20	1.210,19	749,864	23,150	65,901	12,528	3,444	39,211	22,480	21,912
avg	1.784,27	695,278	22,809	65,290	12,770	3,515	36,866	22,247	25,069
min	1.210,19	585,716	22,435	64,553	12,237	3,444	31,373	21,831	21,912
max	3.482,58	749,864	23,150	66,276	13,288	3,710	39,211	23,291	31,415

Tabela 6. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 64, sa verovatnoćom pojave voksla 0.19% i vrednošću *seed* parametra 129832191.



Grafik 2. Srednje vrednosti vremena renderovanja nasumično generisane vokselne kocke stranica dužine 16, 32 i 64 sa verovatnoćom pojave voksla 0.19%.

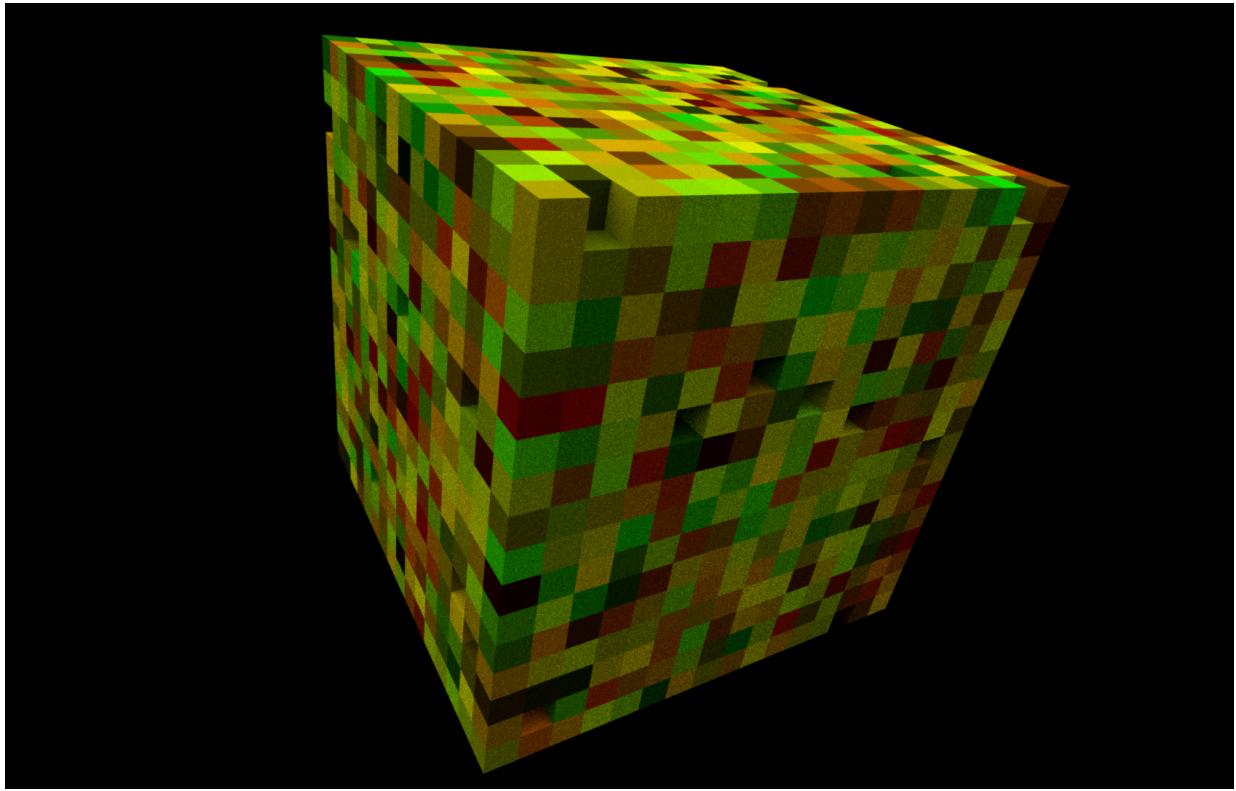
Rezultati merenja za vokselne kocke dužine ivica 16 prikazani su na slici 13 i tabeli 4, za kocku dužine ivica 32 na slici 14 i tabeli 5, i za kocku dužine ivica 64 na slici 15 i tabeli 6. Grafički prikaz srednje vrednosti vremena renderovanja kocaka različitih dužina ivica različitim algoritmima na logaritamskoj skali dat je na grafiku 2.

Slično prethodnom modelu, ponovo se primećuje skraćivanje vremena renderovanja unutar četiri grupe algoritama. Razlika se primećuje kod optimizovane implementacije *direction array* pristupa, budući da ovaj put dolazi do znatnog slabijeg opadanje performansi sa povećanjem veličine vokselnog objekta.

Grid march 1 algoritam ovog puta zaostaje za optimizovanom *direction array* implementacijom kod svakog od tri modela, što je najverovatnije izazvano velikim brojem promašaja, odnosno preseka zraka sa praznim celijama matrice, te nepotrebnog računanja preseka zraka sa velikim brojem suseda trenutnog voksla. Optimizovana varijanta *grid march 2* algoritma ni ovog puta ne prelazi granicu od 10 sekundi po iteraciji, ali je zanimljivo da se ovim algoritmom ponovo dešava situacija u kojoj renderovanje manjeg modela traje duže nego većeg.

Algoritmi na osnovi oktalnih stabala ovog puta pokazuju bolje rezultate, i to slične *grid march 1* implementaciji za *brute force* rešenje, odnosno *grid march 2* za algoritme sa rekurzivnim prolaskom kroz stablo. Poboljšanje performansi verovatno je uzrokovano modelima sa manje voksela, gde struktura oktalnog stabla može pomoći u ranom prekidu izvršavanja kada se zrak nalazi u „manje naseljenim” oblastima. Ipak, osim u slučaju kocke dužine ivica 16, *octree* algoritmi i dalje daju slabije rezultate od optimizovanog *grid march 2* algoritma. Takođe, i ovog puta je očigledno da performanse *grid march* i *octree* algoritama donekle zavise od veličine modela.

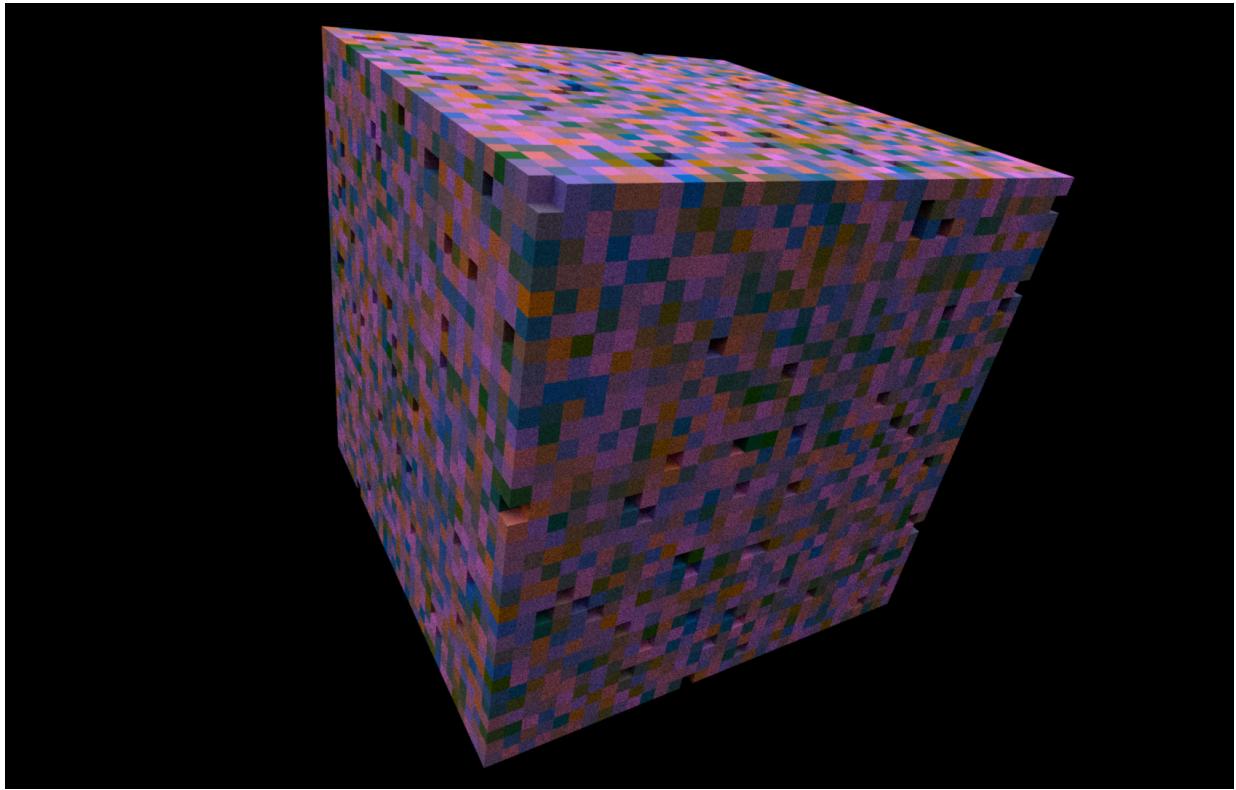
4.3 Nasumično generisana vokselna kocka (96.87% popunjenošći)



Slika 16. Nasumično generisana vokselna kocka dužine ivica 16, sa verovatnoćom pojave voksela 96.87% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GMI</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	2.193,58	1.975,24	48,121	15,530	8,848	3,649	30,635	29,438	15,628
2	2.187,84	2.100,67	47,247	14,510	8,328	3,421	30,412	28,867	15,866
3	2.195,23	2.775,58	47,223	14,036	8,050	3,375	30,489	28,759	16,130
4	2.310,94	3.037,75	47,861	13,616	7,575	3,341	30,683	29,047	16,390
5	2.742,59	2.902,74	47,873	13,432	7,249	3,331	31,405	29,220	16,479
6	2.635,47	2.752,85	47,416	13,362	7,144	3,383	32,232	29,262	16,493
7	2.780,19	2.834,93	47,264	13,240	6,980	3,410	32,145	29,386	16,650
8	2.687,52	2.917,02	46,883	13,153	6,874	3,420	32,092	29,492	16,734
9	2.901,35	2.999,10	46,492	13,138	6,812	3,448	31,978	29,639	16,893
10	2.815,92	3.081,19	46,146	13,067	6,735	3,418	31,867	29,758	16,922
11	2.954,74	3.163,26	45,897	12,996	6,689	3,408	31,942	29,849	17,077
12	2.872,19	3.245,35	45,615	12,983	6,637	3,392	31,905	29,872	17,117
13	2.998,36	3.327,43	45,259	12,958	6,618	3,412	31,912	29,900	17,179
14	2.910,48	3.409,50	44,903	12,941	6,592	3,398	31,896	29,944	17,238
15	2.987,80	3.491,60	44,663	12,951	6,559	3,385	31,958	29,973	17,264
16	2.854,62	3.573,68	44,480	12,912	6,542	3,372	31,980	30,014	17,265
17	2.992,44	3.655,75	44,318	12,905	6,517	3,362	31,994	30,050	17,324
18	2.917,86	3.737,83	44,208	12,910	6,493	3,355	31,984	30,023	17,373
19	2.968,50	3.819,92	43,973	12,905	6,493	3,362	31,998	30,041	17,456
20	2.895,28	3.902,00	43,786	12,902	6,512	3,365	32,014	30,035	17,477
avg	2.740,14	3.135,17	45,981	13,322	7,012	3,400	31,676	29,628	16,848
min	2.187,84	1.975,24	43,786	12,902	6,493	3,331	30,412	28,759	15,628
max	2.998,36	3.902,00	48,121	15,530	8,848	3,649	32,232	30,050	17,477

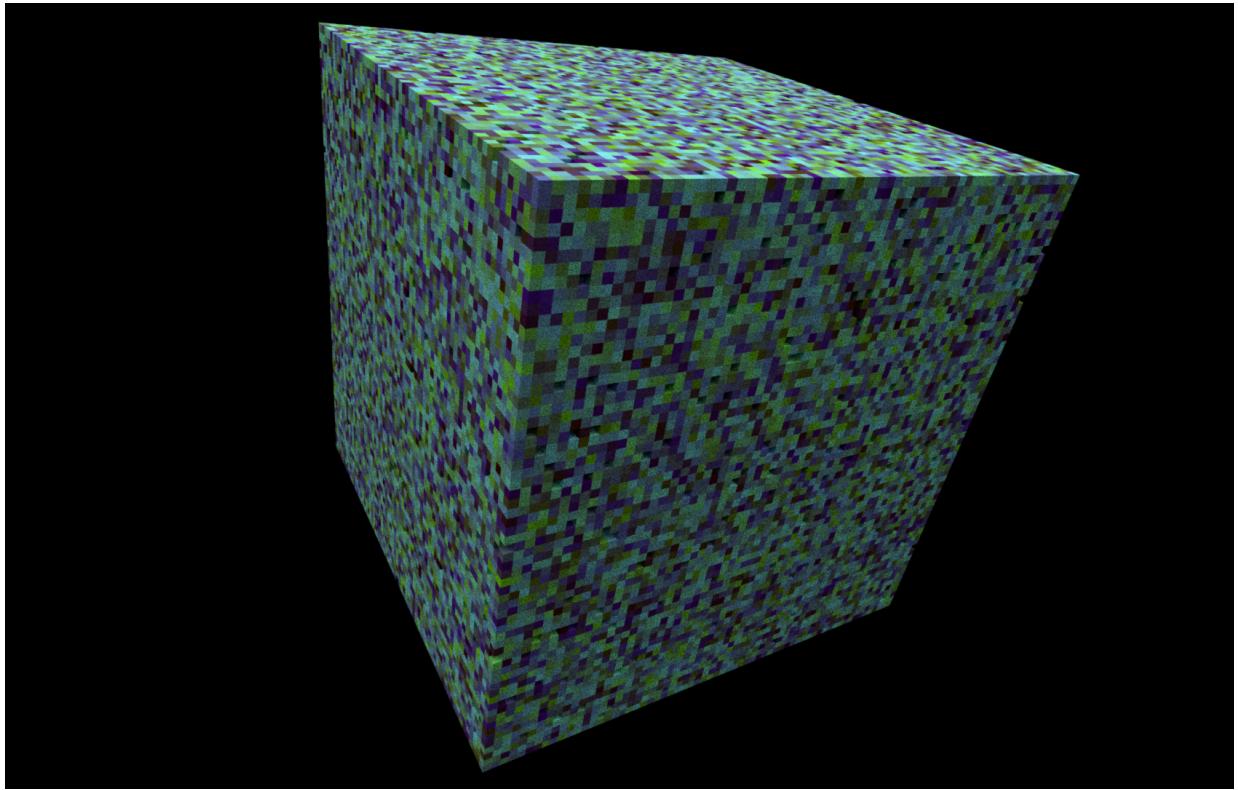
Tabela 7. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 16, sa verovatnoćom pojave voksela 96.87% i vrednošću *seed* parametra 129832191.



Slika 17. Nasumično generisana vokselna kocka dužine ivica 32, sa verovatnoćom pojave voksla 96.87% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GMI</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	16.580,69	11.447,30	208,848	25,648	9,154	3,646	83,545	103,360	10,103
2	16.617,62	11.447,07	208,642	25,960	8,454	3,404	84,087	105,548	9,255
3	16.654,55	11.446,84	206,914	26,676	8,209	3,374	84,120	105,676	8,957
4	16.691,48	11.446,61	220,847	27,211	8,377	3,326	88,331	110,268	8,952
5	16.728,42	11.446,38	227,158	27,431	8,198	3,309	90,492	110,706	8,985
6	16.765,35	11.446,16	231,192	27,722	8,179	3,350	94,069	108,502	8,987
7	16.802,28	11.445,93	235,962	27,669	8,118	3,354	96,710	109,278	8,988
8	16.839,21	11.445,70	241,799	28,327	8,083	3,367	99,283	110,042	8,995
9	16.876,14	11.445,47	245,817	28,194	8,017	3,489	101,359	110,696	9,017
10	16.913,07	11.445,24	248,261	28,243	8,010	3,570	102,781	111,168	8,974
11	16.950,00	11.445,02	249,196	28,322	7,957	3,639	104,101	112,292	8,958
12	16.986,94	11.444,79	249,333	28,355	7,963	3,710	105,189	112,496	8,968
13	17.023,87	11.444,56	249,482	28,323	8,007	3,785	105,565	112,621	8,983
14	17.060,80	11.444,33	249,229	28,273	8,026	3,865	106,035	112,633	9,014
15	17.097,73	11.444,10	249,666	28,324	8,061	3,926	106,402	112,933	9,027
16	17.134,66	11.443,88	249,905	28,314	8,026	3,961	106,794	113,102	9,051
17	17.171,59	11.443,65	254,034	28,345	8,004	3,999	107,241	113,279	9,059
18	17.208,52	11.443,42	255,762	28,317	7,979	4,061	107,769	113,508	9,069
19	17.245,46	11.443,19	258,569	28,317	7,967	4,071	108,337	113,701	9,080
20	17.282,39	11.442,96	260,350	28,331	7,948	4,097	108,835	113,825	9,096
avg	16.931,54	11.445,13	240,048	27,815	8,137	3,665	99,552	110,782	9,076
min	16.580,69	11.442,96	206,914	25,648	7,948	3,309	83,545	103,360	8,952
max	17.282,39	11.447,30	260,350	28,355	9,154	4,097	108,835	113,825	10,103

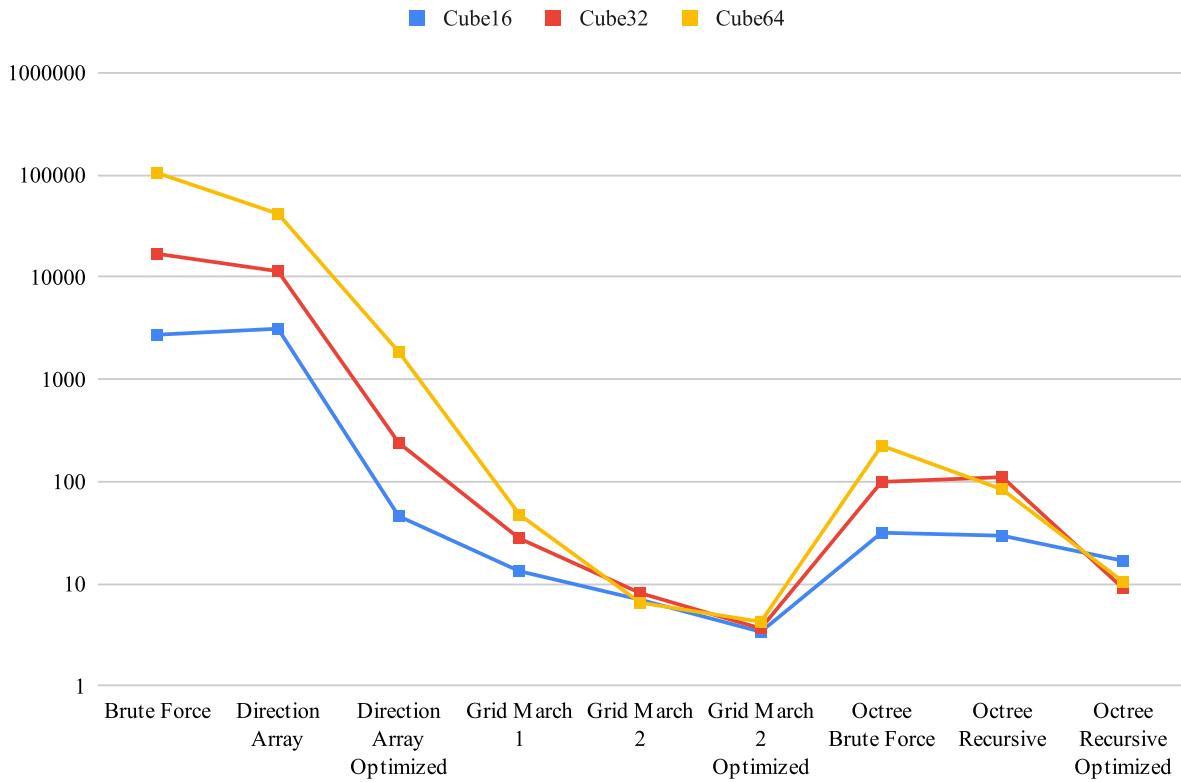
Tabela 8. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 32, sa verovatnoćom pojave voksla 96.87% i vrednošću *seed* parametra 129832191.



Slika 18. Nasumično generisana vokselna kocka dužine ivica 64, sa verovatnoćom pojave voksela 96.87% i vrednošću *seed* parametra 129832191 (50+ iteracija).

Iteracija	<i>BF</i>	<i>DirArr</i>	<i>DirArrO</i>	<i>GMI</i>	<i>GM2</i>	<i>GM2O</i>	<i>OctBF</i>	<i>OctRec</i>	<i>OctRecO</i>
1	102.453,2	41.789,07	1838,735	60,119	7,852	4,851	219,750	82,274	11,054
2	102.681,4	41.788,24	1841,446	53,947	7,199	4,565	211,451	82,940	10,508
3	102.909,6	41.787,41	1844,156	51,674	6,812	4,389	212,830	83,524	10,417
4	103.137,8	41.786,58	1846,867	50,397	6,960	4,326	207,295	84,027	10,435
5	103.366,0	41.785,74	1849,578	49,204	6,724	4,308	202,747	84,447	10,836
6	103.594,2	41.784,91	1852,288	48,123	6,627	4,290	209,133	84,785	11,287
7	103.822,4	41.784,08	1854,999	47,313	6,565	4,300	215,713	85,042	10,978
8	104.050,6	41.783,24	1857,709	46,789	6,406	4,233	218,420	85,216	10,803
9	104.278,8	41.782,41	1860,420	46,393	6,360	4,246	220,985	85,308	10,761
10	104.507,0	41.781,58	1863,131	46,070	6,393	4,198	222,445	85,319	10,618
11	104.735,2	41.780,74	1865,841	45,718	6,353	4,182	223,846	85,247	10,543
12	104.963,4	41.779,91	1868,552	45,438	6,331	4,185	226,375	85,094	10,488
13	105.191,6	41.779,08	1871,263	45,253	6,296	4,209	229,196	84,858	10,402
14	105.419,8	41.778,25	1873,973	45,278	6,273	4,181	232,232	84,540	10,332
15	105.648,0	41.777,41	1876,684	45,300	6,282	4,146	235,089	84,141	10,265
16	105.876,2	41.776,58	1879,394	45,198	6,296	4,136	237,946	83,659	10,239
17	106.104,4	41.775,75	1882,105	45,200	6,371	4,118	240,803	83,096	10,197
18	106.332,6	41.774,91	1884,816	45,139	6,351	4,128	243,660	82,451	10,169
19	106.560,8	41.774,08	1887,526	45,232	6,347	4,125	246,517	81,723	10,143
20	106.789,0	41.773,25	1890,237	45,268	6,340	4,109	249,374	80,914	10,104
avg	104.621,1	41.781,16	1864,486	47,653	6,557	4,261	225,290	83,930	10,529
min	102.453,2	41.773,25	1838,735	45,139	6,273	4,109	202,747	80,914	10,104
max	106.789,0	41.789,07	1890,237	60,119	7,852	4,851	249,374	85,319	11,287

Tabela 9. Vreme renderovanja prvih 10 iteracija nasumično generisane vokselne kocke dužine ivica 64, sa verovatnoćom pojave voksela 96.87% i vrednošću *seed* parametra 129832191.



Grafik 3. Srednje vrednosti vremena renderovanja nasumično generisane vokselne kocke stranica dužine 16, 32 i 64 sa verovatnoćom pojave voksela 96.87%.

Rezultati merenja za vokselne kocke dužine ivica 16 prikazani su na slici 16 i tabeli 7, za kocku dužine ivica 32 na slici 17 i tabeli 8, i za kocku dužine ivica 64 na slici 18 i tabeli 9. Grafički prikaz srednje vrednosti vremena renderovanja kocaka različitih dužina ivica različitim algoritmima na logaritamskoj skali dat je na grafiku 3.

Vreme renderovanja za *brute force* i *direction array* algoritme i ovde se uvećava približno za red veličina kao kod prethodna dva modela. Međutim, u ovom slučaju srednje vreme renderovanja *brute force* algoritmom već za najmanji model iznosi oko 45 minuta, tako da su te vrednosti za srednji model oko 4 sata i 40 minuta, odnosno za veliki čak 29 sati. Uvezši u obzir da je za kvalitetan prikaz potrebno oko 50 iteracija, *brute force* i *direction array* pristupi neće biti uzeti u obzir u nastavku testiranja, budući da je njihova neefikasnost očigledna.

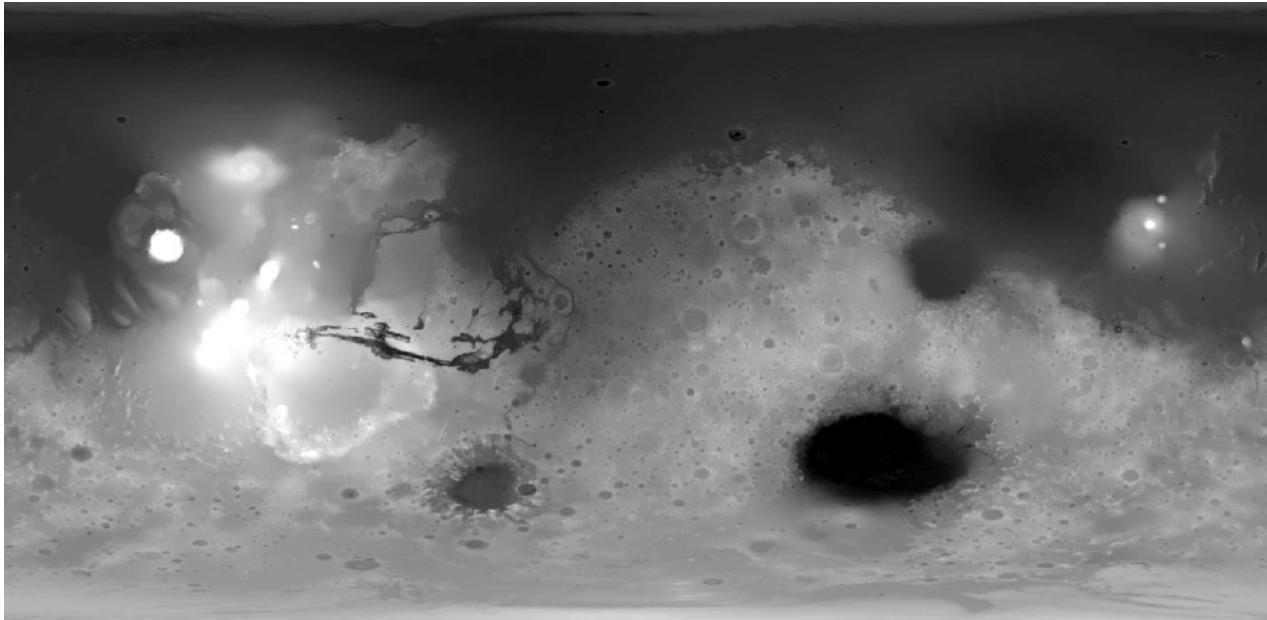
Grid march 1 algoritam ovaj put beleži značajno bolje rezultate od *direction array* pristupa, čemu verovatno doprinosi činjenica da je *direction array* implementacija morala da proveri presecanje zraka sa svakim popunjениm mestom u vokselnoj matrici, a takvih je ovde većina, dok je *grid march 1* smesta nalazio voksel sa kojim se zrak seče i tako vraćao rezultat značajno ranije u odnosu na neoptimizovanija rešenja.

Kao i u prethodna dva slučaja, *grid march 2* algoritam nalazi se značajno ispod vremena od 10 sekundi po iteraciji, te se time ističe kao rešenje na koje najmanje utiče veličina modela.

Algoritmi na bazi oktalnih stabala beleže rezultate slične modelima iz prethodnog potpoglavlja, čime se nameće zaključak da najlošije performanse pokazuju kod modela „srednje“ popunjenošći. Ipak, u najboljem slučaju renderovanje ovim algoritmima traje 2 puta duže nego *grid march 2* algoritmom, a u najgorem slučaju i do 20 puta duže.

4.4 Mars MGS MOLA DEM, izometrijski detalj doline Ares

Drugi test model predstavlja vokselni model detalja površine Marsa generisan na osnovu digitalnog modela elevacije Marsa (eng. *Digital Elevation Model* – DEM) iz 2001. godine, sastavljenog na osnovu podataka Marsovog orbitalnog laserskog altimetra (eng. *Mars Orbital Laser Altimeter* – MOLA) koji je bio deo programa NASA-e pod nazivom Marsov globalni geometar (eng. *Mars Global Surveyor* - MGS) [13].



Slika 19. Digitalni model elevacije Marsa.

Kreiranje vokselnog modela od detalja digitalnog modela elevacije realizovano je u dva koraka. U prvom koraku, koji je implementiran metodom `map` klase `Loaders` (listing 24), izdvojeni detalj mape elevacije se čita piksel po piksel, pri čemu se za pročitani piksel na poziciji (u, v) na mapi kreira "stub" voksela u modelu na poziciji (x, y) , gde je $u = x$ i $v = y$, dok je visina stuba određena osvetljenošću piksela tako da se potpuno beli piksel tumači kao maksimalna, a crni kao minimalna visina.

Tokom generisanja samog modela uzima se u obzir i paleta terena u vidu `DEFAULT_PALETTE` konstante (linije 39-58). Naime, razlikuju se palete sa vodenim površinama i one bez njih, što na teren utiče tako što se vokseli koji pripadaju vodenim površinama renderuju na istoj visini kako bi se simulirala ujednačena površina vode. Na ovaj način boja ulaznih piksela ne utiče ne sam model, ali i dalje utiče na boju voksela budući da se pročitana crno-bela vrednost piksela meša sa bojom dobijenom iz palete, na koji način se boji piksel modela.

Nakon kreiranja modela određuje se boja voksela u modelu, što je implementirano različitim metodama klase `TerrainPalette` (listing 25). U suštini, palete su definisane kao niz boja i njima odgovarajućih visina, pri čemu se boja voksela utvrđuje linearnom interpolacijom boja između čijih se visina voksel nalazi. Na taj način dobija se postupan prelaz od jedne do druge boje renderovanog vokselnog modela terena. Opciono, moguće je onemogućiti linearnu interpolaciju boja i tada se vokseli boje u zavisnosti od toga u opsegu koje boje se njihova visina nalazi.

Prvi model predstavlja detalj doline Ares, oblasti na severnoj polulopti Marsa za koje se smatra da predstavlja rezultat vodene erozije tla.

```

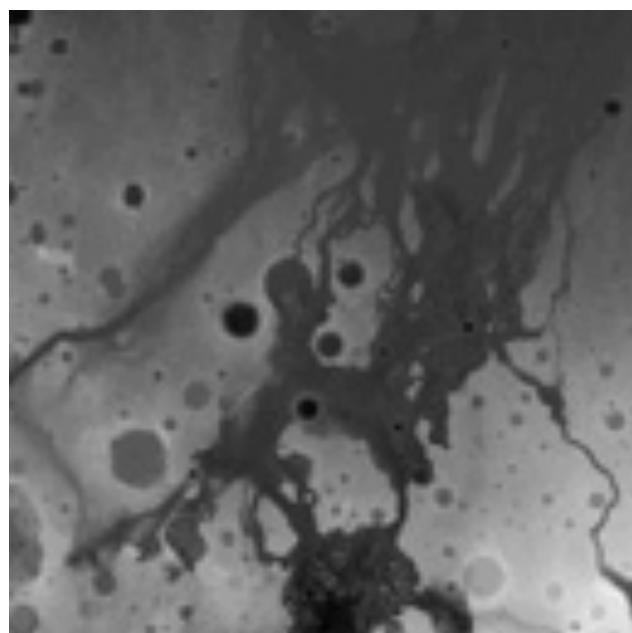
01 public static ModelData3 map(String path) throws IOException {
02
03     Image image = null;
04
05     try {
06         image = new Image(new FileInputStream(path));
07     } catch (FileNotFoundException e) {
08         System.err.println(e.getMessage());
09     }
10
11     int x = (int) image.getWidth();
12     int y = (int) image.getHeight();
13     int z = DEFAULT_MAP_HEIGHT;
14
15     boolean[][][] arr0 = new boolean[x][y][z];
16     Color[][][] arr1 = new Color[x][y][z];
17
18     PixelReader pr = image.getPixelReader();
19
20     for (int j = 0; j < y; j++) {
21         for (int i = 0; i < x; i++) {
22
23             Color imageValue = pr.getColor(i, j);
24             int currZ = (int) (imageValue.getBrightness() * z);
25
26             for (int k = 0; k < currZ; k++) {
27
28                 arr0[i][j][k] = true;
29
30                 arr1[i][j][k] = lerpMapColors ?
31                     DEFAULT_PALETTE
32                         .lerpedColorAtHeightMixedWithImageValue(
33                             imageValue.getBrightness(), 1.0, imageValue) :
34                     DEFAULT_PALETTE
35                         .colorAtHeightMixedWithImageValue(
36                             imageValue.getBrightness(), 1.0, imageValue);
37             }
38
39             if (!DEFAULT_PALETTE.isArid()) {
40
41                 int seaLevelHeight =
42                     (int) (DEFAULT_PALETTE.heightNormalized(1) * z);
43
44                 if (currZ <= seaLevelHeight) {
45                     for (int k = 0; k < seaLevelHeight; k++) {
46
47                         arr0[i][j][k] = true;
48
49                         arr1[i][j][k] = lerpMapColors ?
50                             DEFAULT_PALETTE
51                                 .lerpedColorAtHeightMixedWithImageValue(
52                                     imageValue.getBrightness(), 1.0, imageValue) :
53                             DEFAULT_PALETTE
54                                 .colorAtHeightMixedWithImageValue(
55                                     imageValue.getBrightness(), 1.0, imageValue);
56                 }
57             }
58         }
59     }
60 }
61
62     return ModelData3.arr(arr0, arr1);
63 }
```

Listing 24. Metod `map` klase `Loaders` namenjen generisanju vokselnih modela od crno-belih 2D slika.

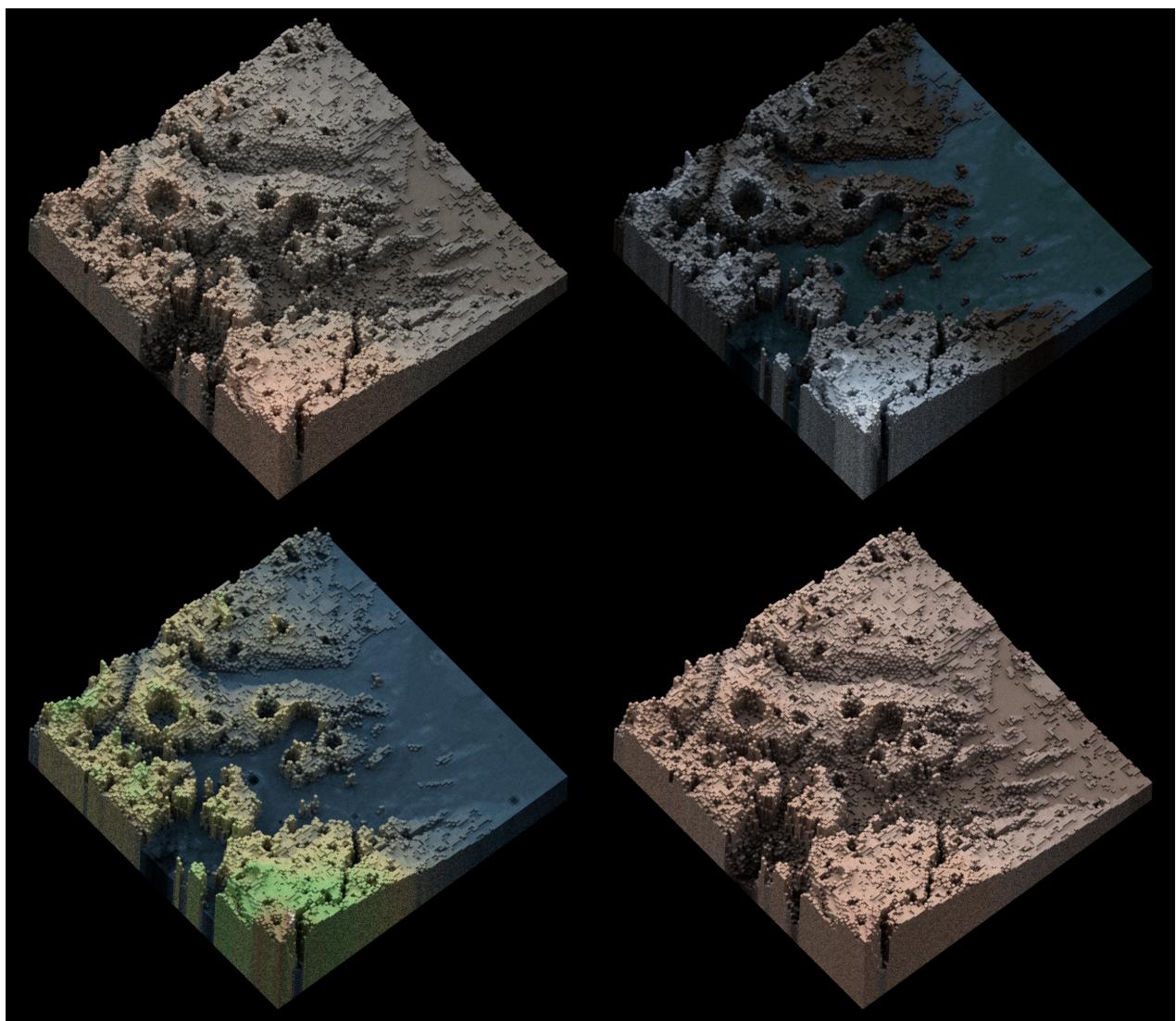
```

01 public Color colorAtHeight(double value, double min, double max) {
02
03     double valueScaled = 2.0 * (value - min) / (max - min) - 1.0;
04     Color output = c[5];
05
06     if (valueScaled < h[4]) output = c[4];
07     if (valueScaled < h[3]) output = c[3];
08     if (valueScaled < h[2]) output = c[2];
09     if (valueScaled < h[1]) output = c[1];
10     if (valueScaled < h[0]) output = c[0];
11
12     return output;
13 }
14
15 public Color colorAtHeight(double value, double max) {
16     return colorAtHeight(value, 0, max);
17 }
18
19 public Color colorAtHeightMixedWithImageValue(
20     double value, double max, Color imageValue) {
21     return Color.rgb(imageValue.getRed(), imageValue.getGreen(),
22         imageValue.getBlue()).mul(colorAtHeight(value, max));
23 }
24
25 public Color lerpedColorAtHeightMixedWithImageValue(
26     double value, double max, Color imageValue) {
27     return Color.rgb(imageValue.getRed(), imageValue.getGreen(),
28         imageValue.getBlue()).mul(getLerpedColor(value));
29 }
30
31 public double heightNormalized(int idx) {
32     return (heights()[idx] + 1.0) * 0.5;
33 }
34
35 public Color getLerpedColor(double value) {
36
37     value = value * 2.0 - 1.0;
38     Color c0 = colors()[0];           double h0 = -1.0;
39     Color c1 = colors()[1];           double h1 = heights()[0];
40
41     for (int i = 0; i < heights().length; i++) {
42         if (value > heights()[i]) {
43             c0 = colors()[i];           h0 = heights()[i];
44             c1 = colors()[i + 1];
45             h1 = i + 1 < heights().length ? heights()[i + 1] : 1.0;
46         } else {
47             break;
48         }
49     }
50
51     h1 -= h0;
52
53     value = (value - h0) / h1;
54
55     double r0 = c0.r(), g0 = c0.g(), b0 = c0.b();
56     double r1 = c1.r(), g1 = c1.g(), b1 = c1.b();
57
58     double r = r0 < r1 ? r0 + (r1 - r0) * value : r0 - (r0 - r1) * value;
59     double g = g0 < g1 ? g0 + (g1 - g0) * value : g0 - (g0 - g1) * value;
60     double b = b0 < b1 ? b0 + (b1 - b0) * value : b0 - (b0 - b1) * value;
61
62     return Color.rgb(r, g, b);
63 }
```

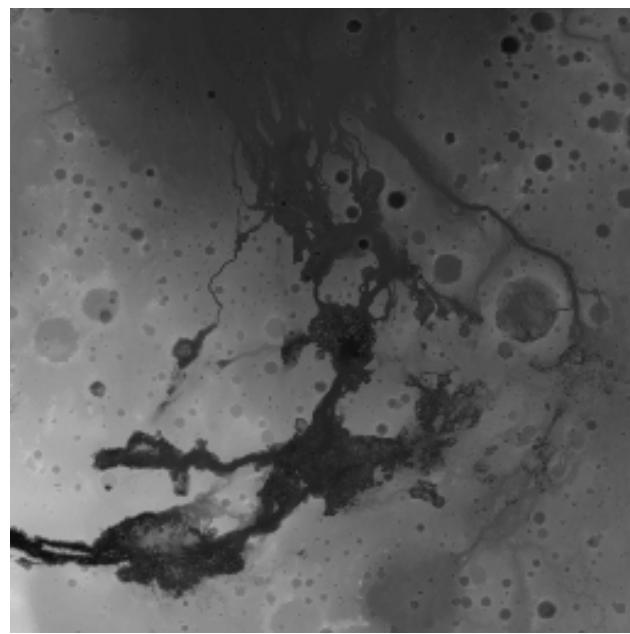
Listing 25. Metodi klase `TerrainPalette` namenjeni generisanju boje vokselnih modela na osnovu visine voksela.



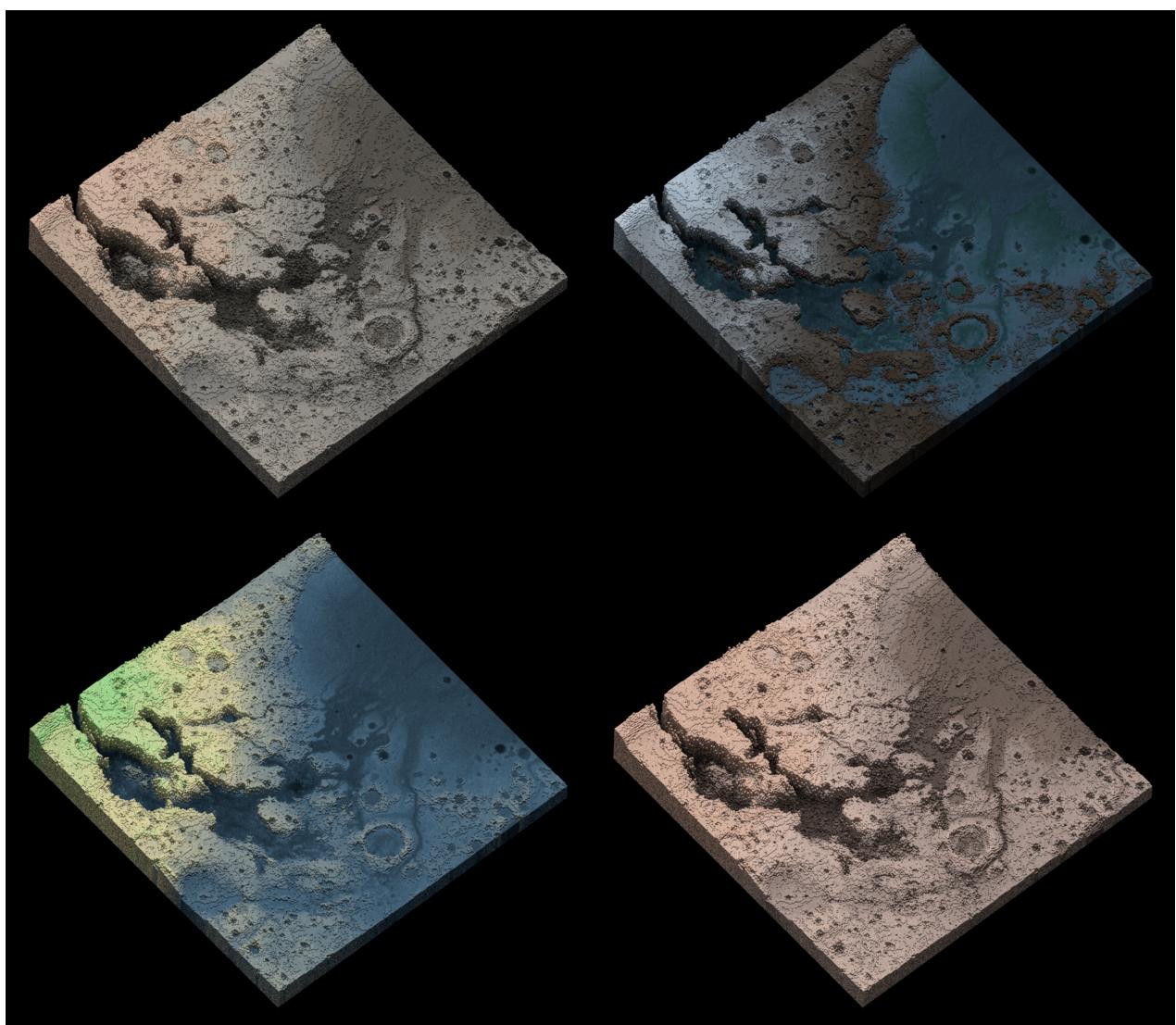
Slika 20. Digitalni model elevacije Marsa, detalj doline Ares dimenzija 128x128.



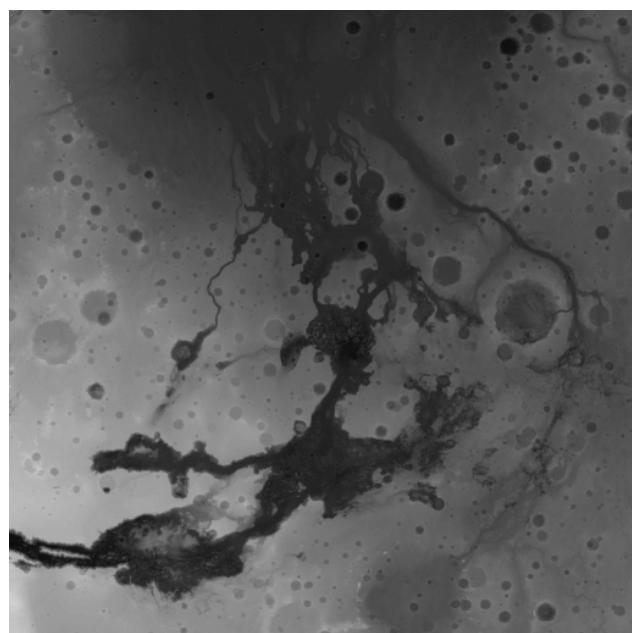
Slika 21. Detalj doline Ares dimenzija osnove 128x128 piksela, visine 50 voksela (20+ iteracija).



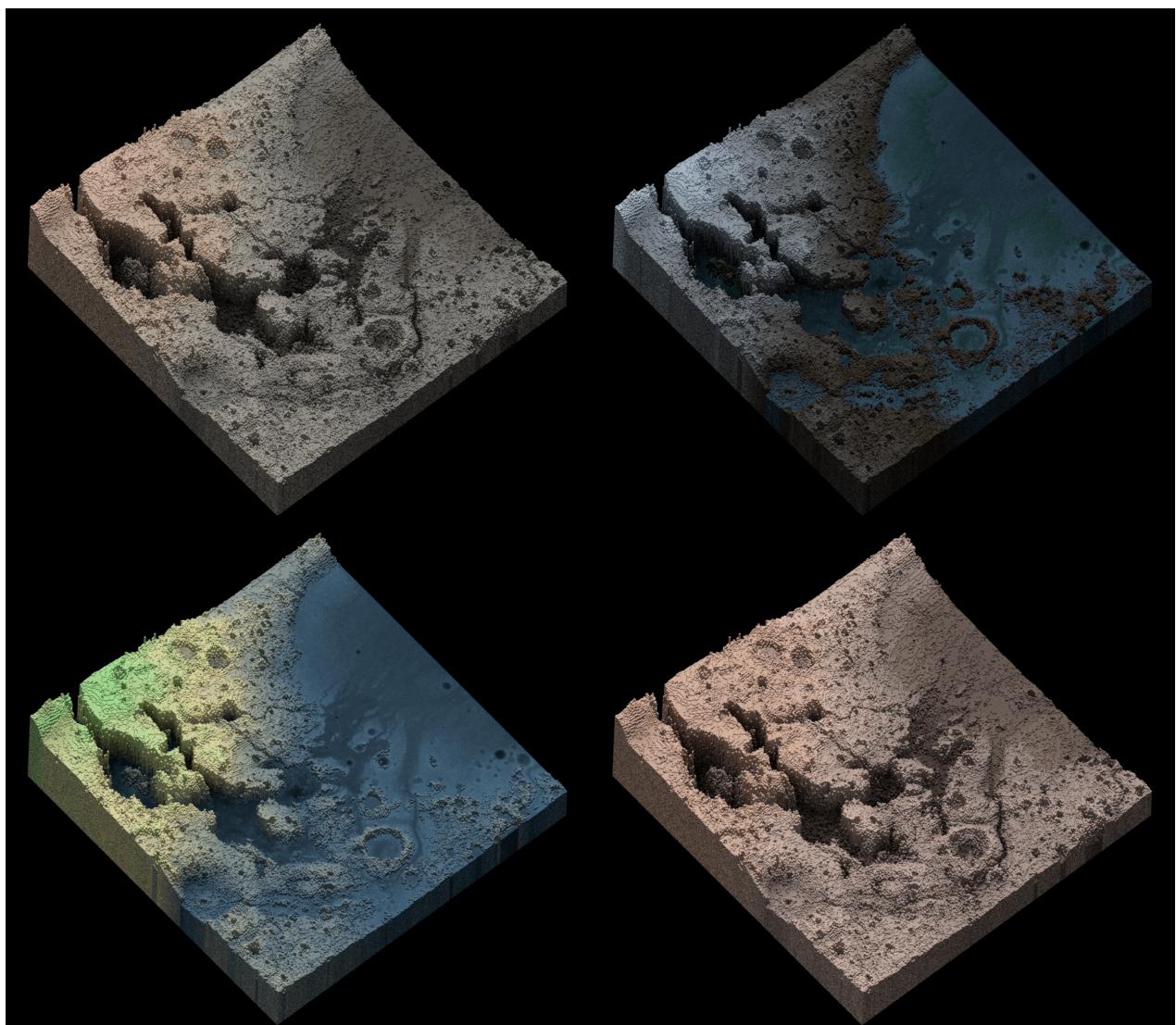
Slika 22. Digitalni model elevacije Marsa, detalj doline Ares dimenzija 256x256.



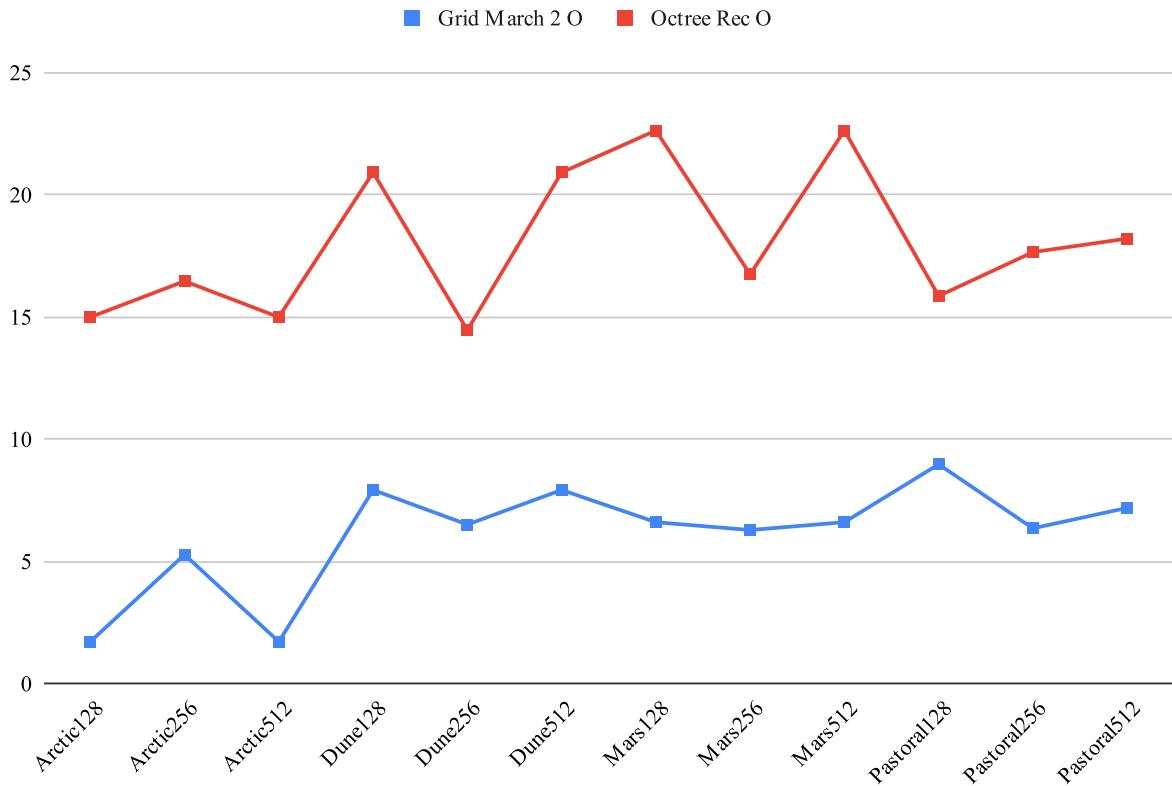
Slika 23. Detalj doline Ares dimenzija osnove 256x256 piksela, visine 50 voksela (20+ iteracija).



Slika 24. Digitalni model elevacije Marsa, detalj doline Ares dimenzija 512x512.



Slika 25. Detalj doline Ares dimenzija osnove 512x512 piksela, visine 100 voksela (20+ iteracija).



Grafik 4. Srednje vrednosti vremena renderovanja detalja doline Ares dimenzija kvadratne osnove 128, 256 i 512 koristeći palete *Dune*, *Arctic*, *Mars* i *Pastoral*.

U poređenju sa nasumično generisanim vokselnom kockom ivica 64 čija matrica sadrži 262,144 voksele, modeli renderovani u ovom poglavlju su mnogo veći i sastoje se od 812,200 voksele kod modela osnove 128px, 3,276,800 voksele za model osnove 256px, odnosno 26,214,400 voksele u slučaju modela osnove 512px.

Ukoliko se rezultati merenja vremena renderinga *brute force* algoritmom iz tabela 1, 2 i 3 kao najgoreg rešenja uporede sa navedenim veličinama modela, procene trajanja renderinga zasnovane na kvadratnom trendu izmerenih vrednosti su oko 3 dana za model osnove 128px, odnosno 44.5 dana za model osnove 256px, i oko 2,765 dana za model osnove 512px.

S druge strane, optimizovane varijante *grid march 2* i algoritma zasnovanog na rekurzivnom prolasku kroz oktalno stablo pokazuju neuporedivo bolje i stabilne rezultate koji ne zavise od veličine modela. Implementacija optimizovanog *grid march 2* algoritma ni u jednom slučaju nije dala vreme renderovanja duže od 10 sekundi, dok se optimizovani rekurzivni prolazak kroz stablo stabilno kretao između 15 i 25 sekundi po iteraciji.

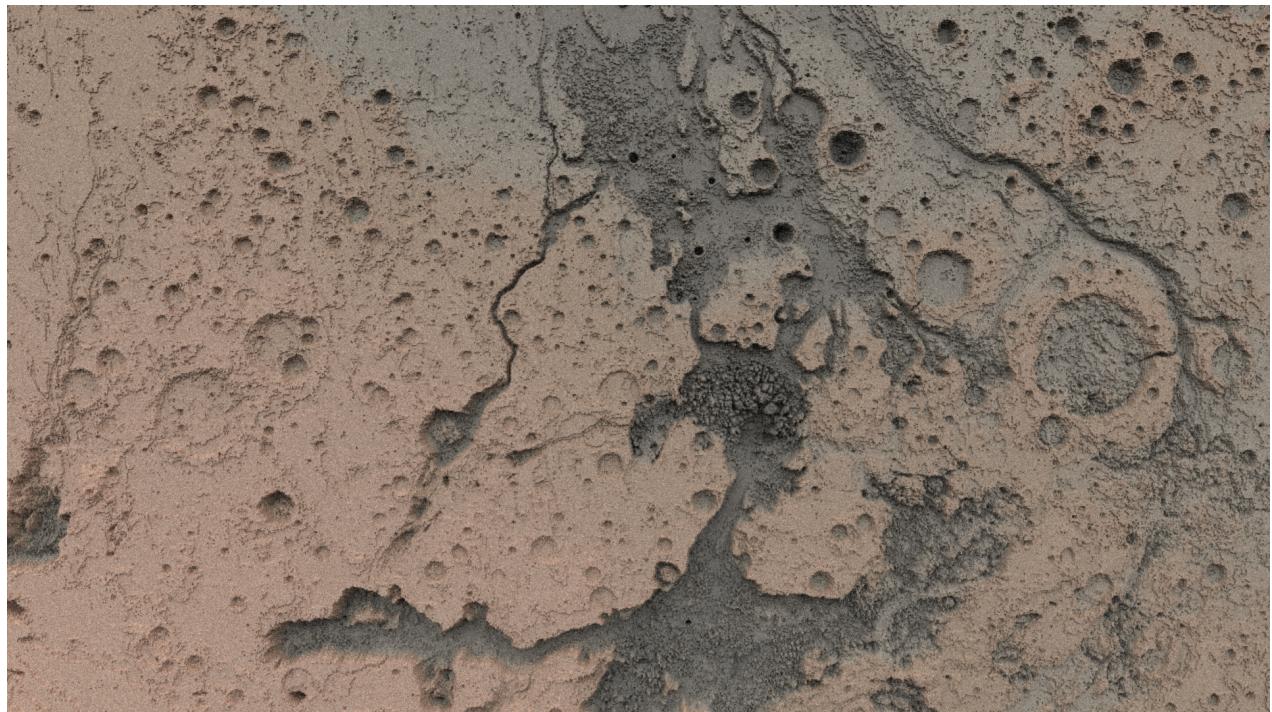
Nemogućnost algoritma na osnovi oktalnog stabla da nadmaši performanse optimizovanog *grid march 2* rešenja može biti uzrokovano činjenicom da, kada postoji voksel na putanji zraka, dolazi do većeg broja računanja preseka zraka sa kutijom, odnosno kockom, dok zrak ne dođe do nivoa konkretnih voksele modela.

Grid march 2 s druge strane počinje od putanju kroz vokselnu matricu koja sadrži telo upravo od vokselne pozicije sa kojom se prvom seče, i ukoliko voksel već nije na površini, najoptimalnijim načinom dolazi do prvog voksela tela sa kojim se preseca.

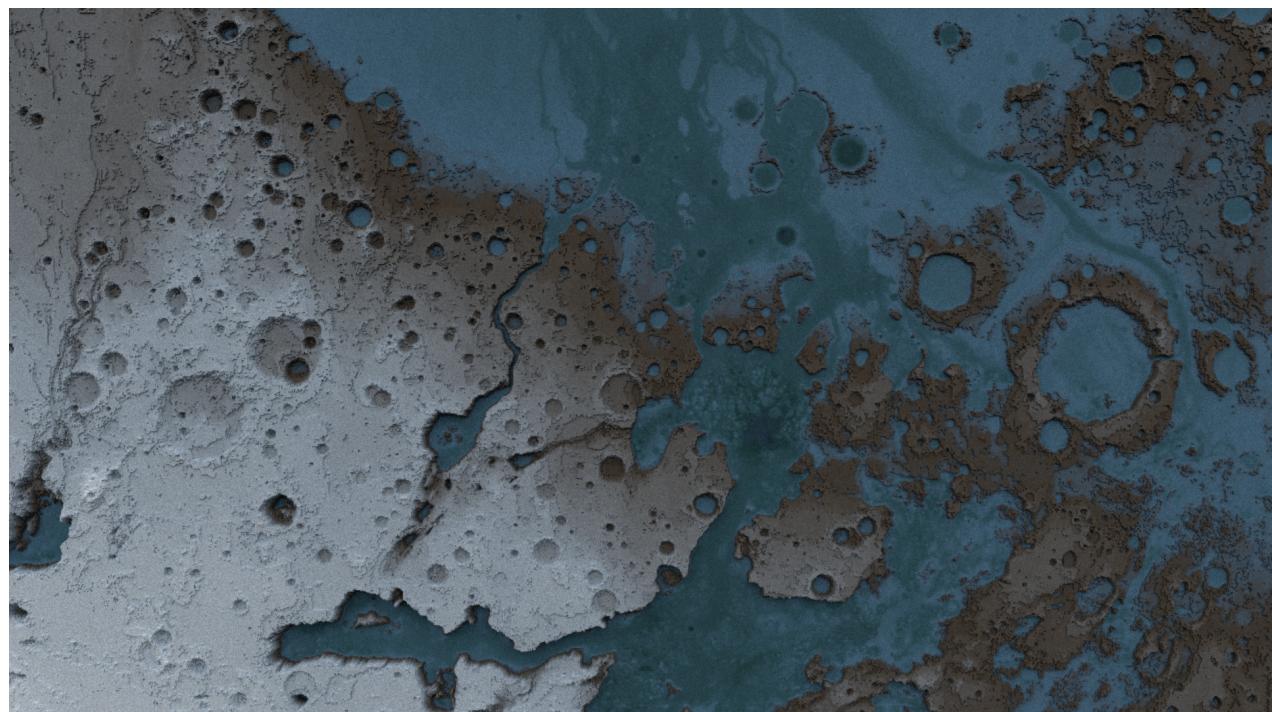
Izbor palete terena, kako je i očekivano, nije uticao na vreme renderovanja.

4.5 Mars MGS MOLA DEM, zapadni deo četvorougla *Oxia Palus*

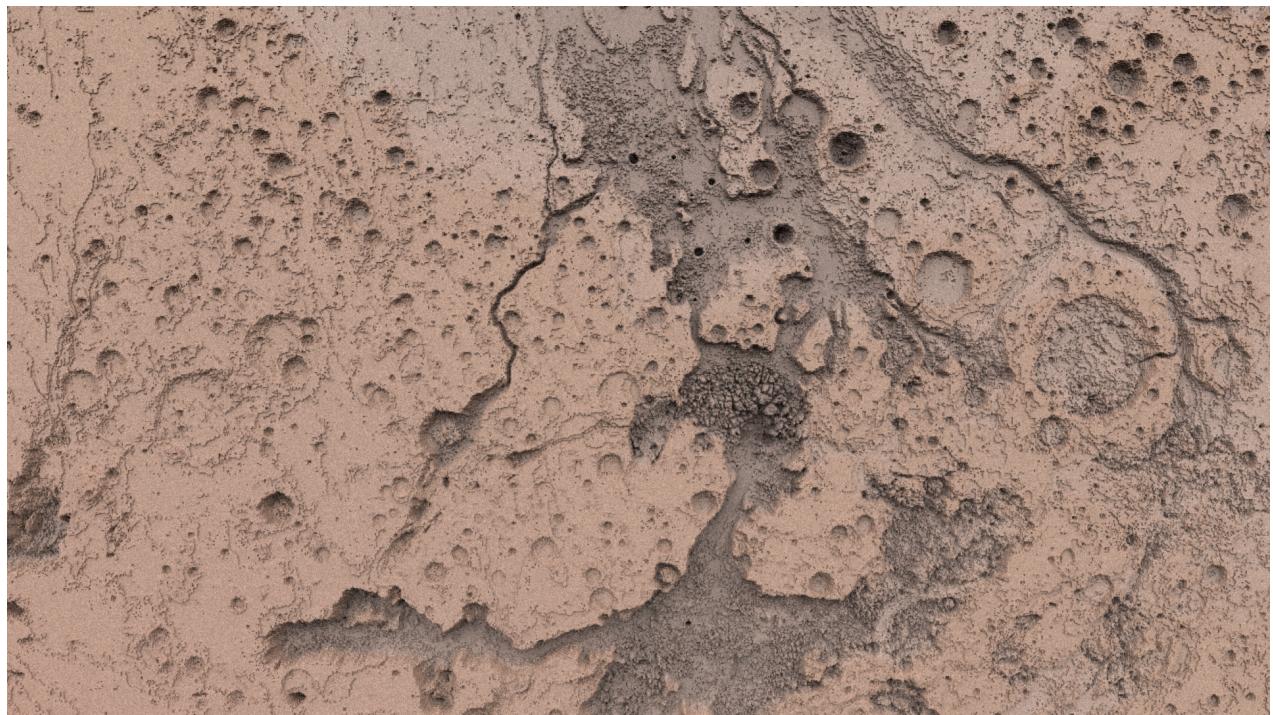
Poslednji analizirani model predstavlja četvorougao *Oxia Palus* na severnoj polulopti Marsa koji je predstavljen isečkom MGS MOLA DEM skupa podataka širine 1536px i visine 1024px, pri čemu je visina generisanog vokselnog tela 50 voksela. Vokselna matrica ovog modela sadrži 78,643,200 voksela.



Slika 26. Region *Oxia Palus* dimenzija osnove 1536x1024 piksela, visine 50 voksela, paleta *Dune* (20+ iteracija).



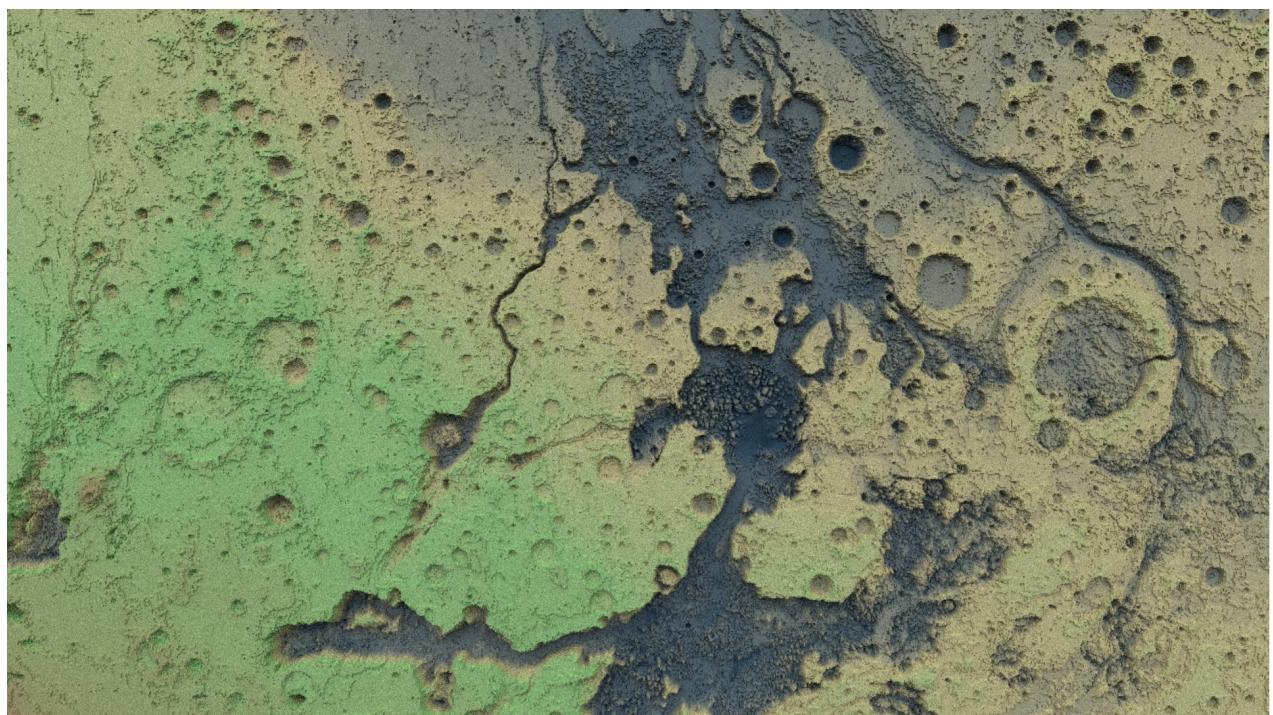
Slika 27. Region *Oxia Palus* dimenzija osnove 1536x1024 piksela, visine 50 voksela, paleta *Arctic* (20+ iteracija).



Slika 28. Region *Oxia Palus* dimenzija osnove 1536x1024 piksela, visine 50 voksela, paleta *Mars* (20+ iteracija).

Renderovanje ovakvog modela pokazalo je određene razlike u vremenu renderovanja u zavisnosti od korišćene palete, pa je tako renderovanje sa paletom *Dune* trajalo u proseku oko 110 sekundi, sa paletom *Arctic* oko 55 sekundi, sa paletom *Mars* oko 20 sekundi, i sa paletom *Pastoral* oko 65 sekundi po iteraciji.

Na ovaj način optimizovana implementacija algoritma *grid march 2* pokazala se ne samo kao najefikasnija, već i kao jedina u stanju da renderuje vokselne modele visoke rezolucije.



Slika 29. Region *Oxia Palus* dimenzija osnove 1536x1024 piksela, visine 50 voksela, paleta *Pastoral* (20+ iteracija).

Poglavlje 5

Zaključak

U ovom radu predstavljene su osnovne *raytracing* tehnike za renderovanje trodimenzionalnih tela na dvodimenzionalnu površinu ekrana, sa naročitim osvrtom na renderovanje pravougaonih kutija, odnosno kocaka jedinične zapremine kao reprezentacije voksela. Progresivno su uočene tačke u kojima je moguće optimizovati opšti algoritam detekcije preseka zraka i trenutnog voksela, pa je tako dat niz rešenja od naivnog brute force pristupa, preko naprednjeg algoritma zasnovanog na iteriranju kroz vokselnu matricu u pravcu vektora pravca upadnog zraka, do daleko optimizovanijih grid march i algoritama na osnovi oktalnih stabala.

Implementacije svih algoritama detaljno su objašnjene i testirane na nasumično generisanim vokselnim modelima, njihove performanse su upoređene i rezultati merenja vremena renderovanja objašnjeni u zavisnosti od specifičnosti modela i samih algoritama. Dalje su optimizovani grid march 2 i algoritam na osnovi oktalnih stabala testirani na značajno većim modelima terena generisanim na osnovu rezultata Mars Global Surveyor MOLA DEM misije. Konačno, optimizovani grid march 2 algoritam testiran je na modelu terena veličine gotovo 80 miliona voksela.

Rezultati izloženi u radu predstavljaju pokušaj da se da doprinos proučavanju *raytracing* metoda u računarskoj grafici na nivou osnovnih i master akademskih studija. Numerički rezultati istraživanja zajedno sa grafičkim artefaktima predstavljaju ilustraciju značaja optimizacije u domenu grafike, kao i motivaciju za dalja istraživanja u oblasti.

Pravac potencijalnog budućeg istraživanja mogla bi biti kako dalja optimizacija, tako i optimizovani način indirektnog renderovanja vokselnih tela, odnosno konverzija vokselnog tela u trodimenzionalno telo zasnovano na trouglovima u momentu renderovanja, čime bi se postigla ne samo dalja optimizacija već i donekle drugačiji vizuelni rezultat.

Literatura

- [1] "What is Computer Graphics?," graphics.cornell.edu. <https://www.graphics.cornell.edu/online/tutorial/> (pristupljen 15. oktobra 2024.)
- [2] S. Marschner, P. Shirley, *Fundamentals of Computer Graphics*, 5th ed. Natick, MA: A K Peters/CRC Press, 2021
- [3] W. Schroeder, K. Martin, B. Lorensen, *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics*, 3rd ed. Clifton Park, NY: Kitware, 2002
- [4] T. Whitted, "An Improved Illumination Model for Shaded Display," Communications of the ACM, vol. 23, no. 6, pp. 343-349, 1980.
- [5] "Overview of the Ray-Tracing Rendering Technique," scratchapixel.com. <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/light-transport-ray-tracing-whitted.html> (pristupljen 15. oktobra 2024.)
- [6] K. Suffern, *Ray Tracing from the Ground Up*, 1st ed. Natick, MA: A K Peters/CRC Press, 2007
- [7] S. F. Javid, "Three-Dimensional Image Processing Using Voxels," Ph.D. dissertation, University College London, University of London, London, 1991
- [8] "Introduction to Shading," scratchapixel.com. <https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/what-is-shading-light-matter-interaction.html> (pristupljen 20. oktobra 2024.)
- [9] E. Haines et al., *An Introduction to Ray Tracing*, 1st ed. Cambridge, MA: Academic Press, 1991
- [10] "A Minimal Ray-Tracer," scratchapixel.com. <https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection.html> (pristupljen 26. oktobra 2024.)
- [11] T. Kay, J. Kajiya, "Ray Tracing Complex Scenes," ACM SIGGRAPH Computer Graphics, vol. 20, no. 4, pp. 269-278, 1986.
- [12] M. Savić (2020) RayTracer source code (Version 1.0), sa dodatkom VoxelWorld kandidata N. Vetnića [Source code]. https://github.com/NikolaVetnic/RG2_GI
- [13] MOLA Team, "Mars MGS MOLA DEM 463m." astrogeology.usgs.gov. https://astrogeology.usgs.gov/search/map/mars_mgs_mola_dem_463m (accessed Nov. 30, 2024)

Spisak slika i grafika

Slika 1. Vitedov originalni rendering kao rezultat algoritma opisanog u radu iz 1980. godine, prikazan u [5].

Slika 2. Geometrijsko rešenje računanja preseka zraka sa sferom, dato u [9, pp. 42].

Slika 3. Geometrijsko rešenje računanja preseka zraka sa kutijom.

Slika 4. Renderovanje implicitno definisanog vokselnog tela (10+ iteracija).

Slika 5. Instanca klase **Box** teksturirana kao vokselni objekat i instanca klase **BasePF** (10+ iteracija).

Slika 6. Presek zraka sa dvodimenzionalnom pikselnom matricom.

Slika 7. Presek zraka sa poluravni $x = 4$.

Slika 8. Presek zraka sa poluravni $y = 3$.

Slika 9. Presek zraka sa poluravni $x = 5$ i prvi presek sa pikselnim telom u matrici.

Slika 10. Nasumično generisana vokselna kocka dužine ivica 16, sa verovatnoćom pojave voksela 6.25% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 11. Nasumično generisana vokselna kocka dužine ivica 32, sa verovatnoćom pojave voksela 6.25% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 12. Nasumično generisana vokselna kocka dužine ivica 64, sa verovatnoćom pojave voksela 6.25% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 13. Nasumično generisana vokselna kocka dužine ivica 16, sa verovatnoćom pojave voksela 0.19% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 14. Nasumično generisana vokselna kocka dužine ivica 32, sa verovatnoćom pojave voksela 0.19% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 15. Nasumično generisana vokselna kocka dužine ivica 64, sa verovatnoćom pojave voksela 0.19% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 16. Nasumično generisana vokselna kocka dužine ivica 16, sa verovatnoćom pojave voksela 96.87% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 17. Nasumično generisana vokselna kocka dužine ivica 32, sa verovatnoćom pojave voksela 96.87% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 18. Nasumično generisana vokselna kocka dužine ivica 64, sa verovatnoćom pojave voksela 96.87% i vrednošću seed parametra 129832191 (50+ iteracija).

Slika 19. Digitalni model elevacije Marsa.

Slika 20. Digitalni model elevacije Marsa, detalj doline Ares dimenzija 128x128.

Slika 21. Detalj doline *Ares* dimenzija osnove 128x128 piksela, visine 50 voksela (20+ iteracija).

Slika 22. Digitalni model elevacije Marsa, detalj doline *Ares* dimenzija 256x256.

Slika 23. Detalj doline *Ares* dimenzija osnove 256x256 piksela, visine 50 voksela (20+ iteracija).

Slika 24. Digitalni model elevacije Marsa, detalj doline *Ares* dimenzija 512x512.

Slika 25. Detalj doline *Ares* dimenzija osnove 512x512 piksela, visine 100 voksela (20+ iteracija).

Slika 26. Region *Oxia Palus* dimenzija osnove 1536x1024 piksela, visine 50 voksela, paleta *Dune* (20+ iteracija).

Slika 27. Region *Oxia Palus* dimenzija osnove 1536x1024 piksela, visine 50 voksela, paleta *Arctic* (20+ iteracija).

Slika 28. Region *Oxia Palus* dimenzija osnove 1536x1024 piksela, visine 50 voksela, paleta *Mars* (20+ iteracija).

Slika 29. Region *Oxia Palus* dimenzija osnove 1536x1024 piksela, visine 50 voksela, paleta *Pastoral* (20+ iteracija).

Grafik 1. Srednje vrednosti vremena renderovanja nasumično generisane vokselne kocke stranica dužine 16, 32 i 64 sa verovatnoćom pojave voksela 6.25%.

Grafik 2. Srednje vrednosti vremena renderovanja nasumično generisane vokselne kocke stranica dužine 16, 32 i 64 sa verovatnoćom pojave voksela 0.19%.

Grafik 3. Srednje vrednosti vremena renderovanja nasumično generisane vokselne kocke stranica dužine 16, 32 i 64 sa verovatnoćom pojave voksela 96.87%.

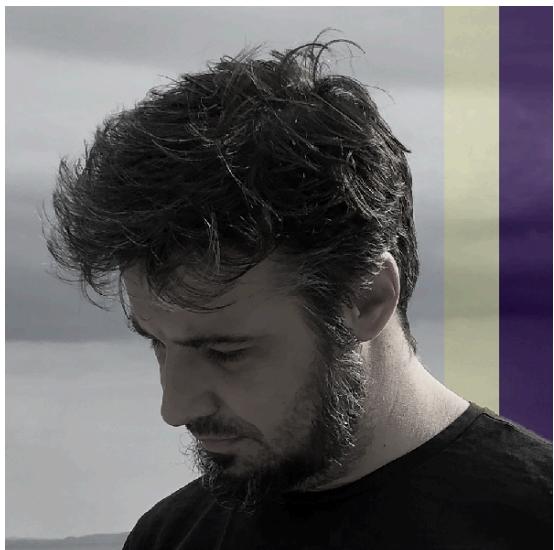
Grafik 4. Srednje vrednosti vremena renderovanja detalja doline *Ares* dimenzija kvadratne osnove 128, 256 i 512 koristeći palete *Dune*, *Arctic*, *Mars* i *Pastoral*.

Biografija

Nikola Vetnić (1984) završio je osnovne i master studije kompozicije na Akademiji umetnosti Novi Sad u klasi prof. Milana Mihajlovića, doktorske studije kompozicije na Fakultetu muzičke umetnosti u Beogradu u klasi prof. Zorana Erića, i osnovne studije smera Informacione tehnologije na Prirodno-matematičkom fakultetu u Novom Sadu.

Autor je velikog broja kompozicija i muzičkih izdanja u domenu umetničke, popularne i primenjene muzike. Govori engleski, ruski, norveški i nemački jezik. Bavi se razvojem *web* aplikacija, razvojem video igara i računarskom grafikom.

Oženjen je i ima dva energična sina.



Ključna dokumentacijska informacija

UNIVERZITET U NOVOM SADU
PRIRODNO MATEMATIČKI FAKULTET
KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj:

RBR

Identifikacioni broj:

IBR

Tip dokumentacije: Monografska dokumentacija

TZ

Tip zapisa: Tekstualni štampani materijal

TZ

Vrsta rada: Master rad

VR

Autor: dr Nikola Vetnić

AU

Mentor: dr Marko Savić

ME

Naslov rada: Implementacija i poređenje *raytracing* algoritama za renderovanje vokselnih objekata

NR

Jezik publikacije: Srpski

JP

Jezik izvoda: Srpski / Engleski

JI

Zemlja publikovanja: Republika Srbija

ZP

Uže geografsko područje: Vojvodina

UGP

Godina: 2025.

GO

Izdavač: Autorski reprint

IZ

Mesto i adresa: Prirodno-matematički fakultet, Trg Dositeja Obradovića 4, Novi Sad
MA

Fizički opis rada: (5 / 81 / 21 / 9 / 29 / 4 / 0) (broj poglavlja / broj strana / broj citata / broj tabela / broj slika / broj grafika / broj priloga)

FO

Naučna oblast: Računarske nauke
NO

Naučna disciplina: Računarska grafika
ND

Predmetna odrednica / ključne reči: *raytracing*, renderovanje, vokseli, optimizacija, *grid march*, *octree*

PO, UDK

Čuva se: U biblioteci Departmana za matematiku i informatiku, Prirodno-matematički fakultet, Univerzitet u Novom Sadu

ČU

Važna napomena:
VN

Izvod: Rad se bavi osnovnim tehnikama *raytracing* renderovanja i njihovom primenom na kreiranje prikaza vokselnih tela u trodimenzionalnom prostoru. Počevši od jednostavne brute force implementacije daje se niz alternativnih algoritama za pronalaženje preseka zraka i vokselnog tela, čime se postiže značajno skraćivanje vremena renderovanja, a što je prikazano u praksi renderovanjem različitih vokselnih modela. Rezultati izloženi u radu predstavljaju pokušaj da se da doprinos proučavanju raytracing metoda u računarskoj grafici na nivou osnovnih i master akademskih studija.

IZ

Datum prihvatanja teme od NN veća:
NN

Datum odbrane:
DO

Članovi komisije:
ČK

Predsednik: dr Dragan Mašulović, redovni profesor Prirodno-matematičkog fakulteta u Novom Sadu

Član: dr Miloš Radovanović, redovni profesor Prirodno-matematičkog fakulteta u Novom Sadu

Mentor: dr Marko Savić, docent na Prirodno-matematičkom fakultetu u Novom Sadu

UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCES
KEY WORD DOCUMENTATION

Accession number:

ANO

Identification number:

INO

Document type: Monograph type

DT

Type of record: Printed text

TR

Content code: Master thesis

CC

Author: dr Nikola Vetnić

AU

Mentor: dr Marko Savić

MN

Title: Implementation and comparison of raytracing algorithms for rendering voxel objects

TI

Language of text: English

LT

Language of abstract: Serbian / English

LA

Country of publication: Republic of Serbia

CP

Locality of publication: Vojvodina

LP

Publication year: 2025.

PY

Publisher: Author's reprint

PU

Publication place: Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4

PP

Physical description: (5 / 81 / 21 / 9 / 29 / 4 / 0) (chapters / pages / literature / tables / pictures / graphics / appendices)

PD

Scientific field: Computer science

SF

Scientific discipline: Computer graphics

SD

Subject / Key words: raytracing, rendering, voxel, optimization, grid march, octree

SKW

Holding data: The Library of the Department of Mathematics and Informatics, Faculty of Science and Mathematics, University of Novi Sad

HD

Note:

N

Abstract: The thesis deals with basic ray tracing rendering techniques and their application to the visualization of voxel objects in three-dimensional space. Starting from a simple brute-force implementation, a series of alternative algorithms for detecting intersections between rays and voxel objects is presented, significantly reducing rendering time, which is demonstrated in practice through the rendering of various voxel models. The results presented in the work represent an attempt to contribute to the study of ray tracing methods in computer graphics at the undergraduate and master's academic levels.

AB

Accepted by the Scientific Board:

ASB

Defended on:

DE

Thesis defend board:

DB

President: Dragan Mašulović PhD, full professor at Faculty of Science in Novi Sad

Member: Miloš Radovanović PhD, full professor at Faculty of Science in Novi Sad

Mentor: Marko Savić PhD, assistant professor at Faculty of Science in Novi Sad