

# Implementacija i poređenje *raytracing* algoritama za renderovanje vokselnih objekata

Master rad

mentor  
dr Marko Savić

student  
dr Nikola Vetnić, 82m/23

# 1 Uvod

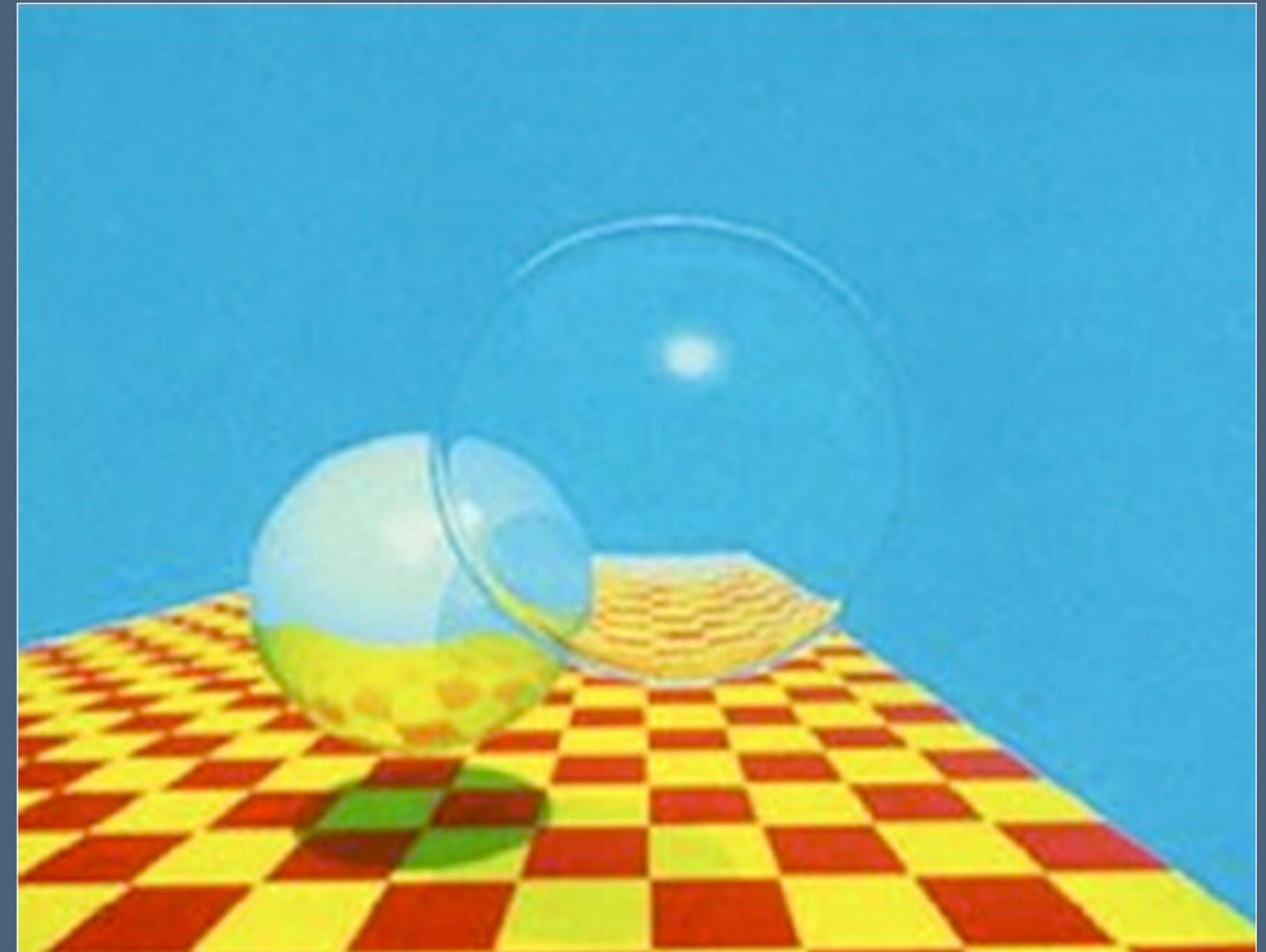
## *Object Order i Image Order* algoritmi

- Računarska grafika – “bilo šta na računaru što nije tekst ili zvuk”.
- *Rendering* – generisanje 2D prikaza 3D scene putem fizičkog i matematičkog modelovanja.
- *Object Order* renderovanje – elementi se pojavljuju od najdaljeg ka najbližem, pri čemu bliži objekti po potrebi prepisuju vrednosti već izrendanih piksela.
- *Image Order* algoritmi – iteriranje po pikselima izlaza i računanje uticaja svakog od objekata na sceni na trenutni piksel.
- Turner Whitted (Turner Whitted) 1980. objavljuje rad na temu alternativnog pristupa renderovanja 3D grafike – početak *raytracing-a*.

# 1 Uvod

## *Raytracing algoritam*

- Zasnovan na konstruisanju stablu zraka od posmatrača do površine prvog objekta na sceni sa kojim se sudara.
- Uzima se u obzir globalno osvetljenje i postiže verna simulacija refleksije, senke i refrakcije.
- *Raytracing* ne zahteva namenski hardver, lak je za implementaciju i prilagodljiv – brzo prihvatanje i velika popularnost.



# 1 Uvod

## *Raytracing algoritam i vokselni prostor*

- Koraci:
  1. Iteriranje kroz izlaznu matricu piksela;
  2. Pronalaženje prvog objekta koji zrak preseca;
  3. Izračunavanje nijanse boje tela.
- Zrak je obično opisan početnom tačkom i smerom, a tela geometrijskim primitivima ili derivatima.
- Alternativni opis tela jeste kolekcija “zapreminskih piksela” – voksela – koje telo zauzima.
- Vokselna tela renderuju se indirektno (ekstrahovanjem poligona) ili direktno.

# 2 Osnove *raytracing* algoritma

## Opis zraka i računanje preseka

- *Raytracing* program sastoји се од три дела:
  1. Generisanja zraka – računanja почетка и смера зрaka за сваки пиксел излаза,
  2. Preseka zraka – налађења објекта најближег камери на путањи зрaka кроз пикSEL, i
  3. Senčenja – računanja боје пиксела на основу резултата пресека зрaka са телом.
- Зрак је представљен почетном тачком и правцем:

$$R(t) = R_0 + t(S - R_0)$$

- Зрак је тако праволиниско кретање од  $R_0$  ка  $S$ , где је  $R(t)$  тачка на праву на  $t$ -острукотој удаљености у правцу  $(S - R_0)$  почеvши од  $R_0$ .

# 2 Osnove raytracing algoritma

## Presek zraka sa sferom – analitičko rešenje

- Uzevši da je  $S_R$  prečnik,  $S_C$  centar i  $S$  tačka na površini, sfera se izražava jednačinom:

$$|S - S_C|^2 - S_R^2 = 0$$

- Preseci zraka  $S = R_0 + td$  sa sferom predstavljaju rešenja kvadratne jednačine po  $t$ :

$$t^2d^2 + 2 R_0 t d + R_0^2 - S_R^2 = 0$$

- Kada je diskriminanta negativna zrak ne preseca sferu, kada je nula zrak je tangenta na površinu sfere, inače zrak preseca sferu u dve tačke na površini.
- Analitičko rešenje preseka nažalost je dosta sporo usled operacija korenovanja i deljenja.

# 2 Osnove *raytracing* algoritma

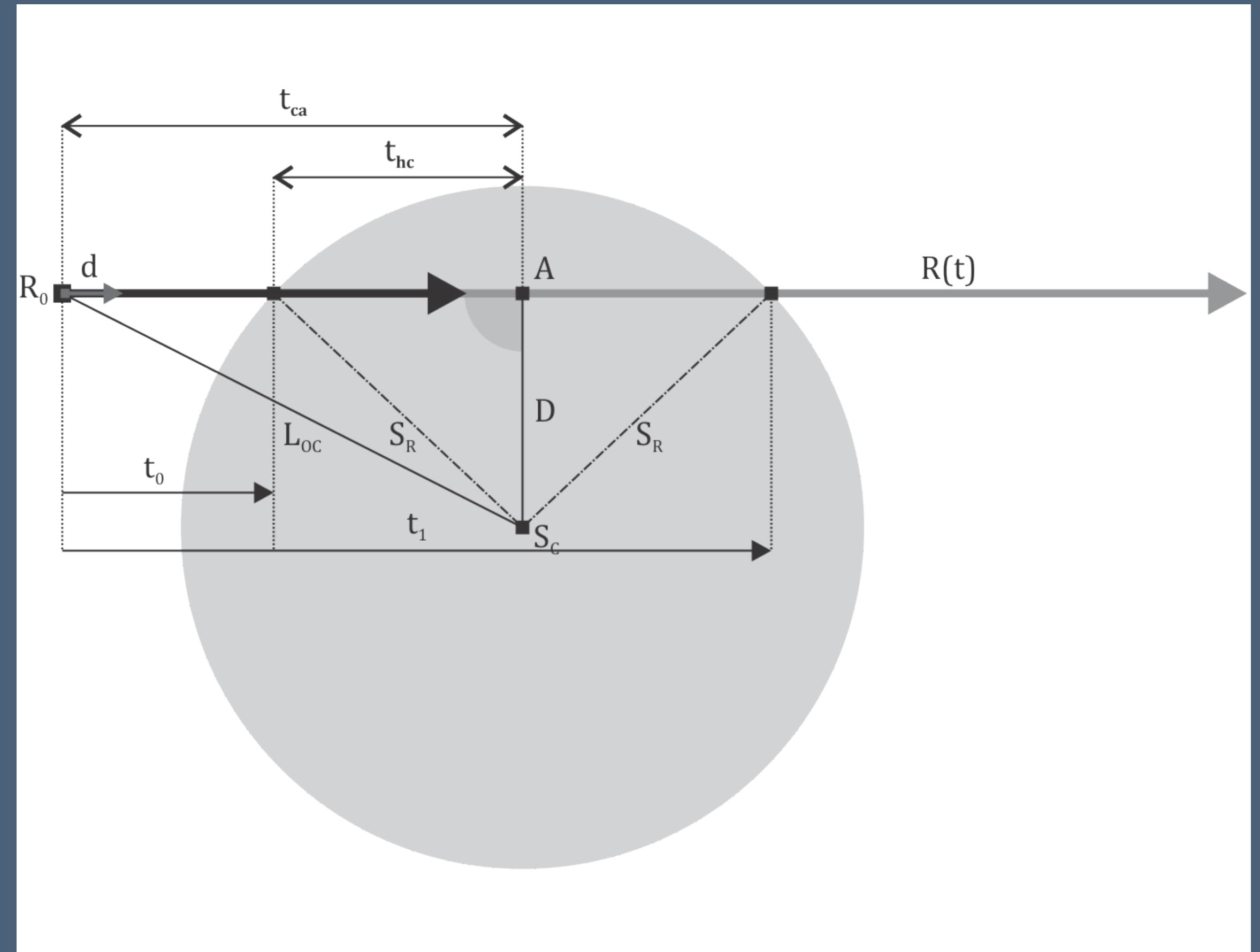
## Presek zraka sa sferom – geometrijsko rešenje

- Alternativna strategija:
  1. Proveriti da li je početna tačka unutar sfere – ako jeste, postoji presek;
  2. Pronaći rastojanje od početka zraka do tačke na zraku najbliže centru sfere  $S_c$ ;
  3. Proveriti da li je zrak usmeren ka sferi – ako nije, ne postoji presek;
  4. Pronaći kvadrat udaljenosti do najbliže tačke na površini sfere – ako je negativna, ne postoji presek;
  5. Pronaći udaljenost do preseka zraka sa površinom, odnosno vreme  $t$ ;
  6. Izračunati poziciju preseka i normalu u presečnoj tački.
- Ovako se kvadratna jednačina deli na kraće izraze koji se izračunavaju po potrebi.

# 2 Osnove raytracing algoritma

## Presek zraka sa sferom – geometrijsko rešenje

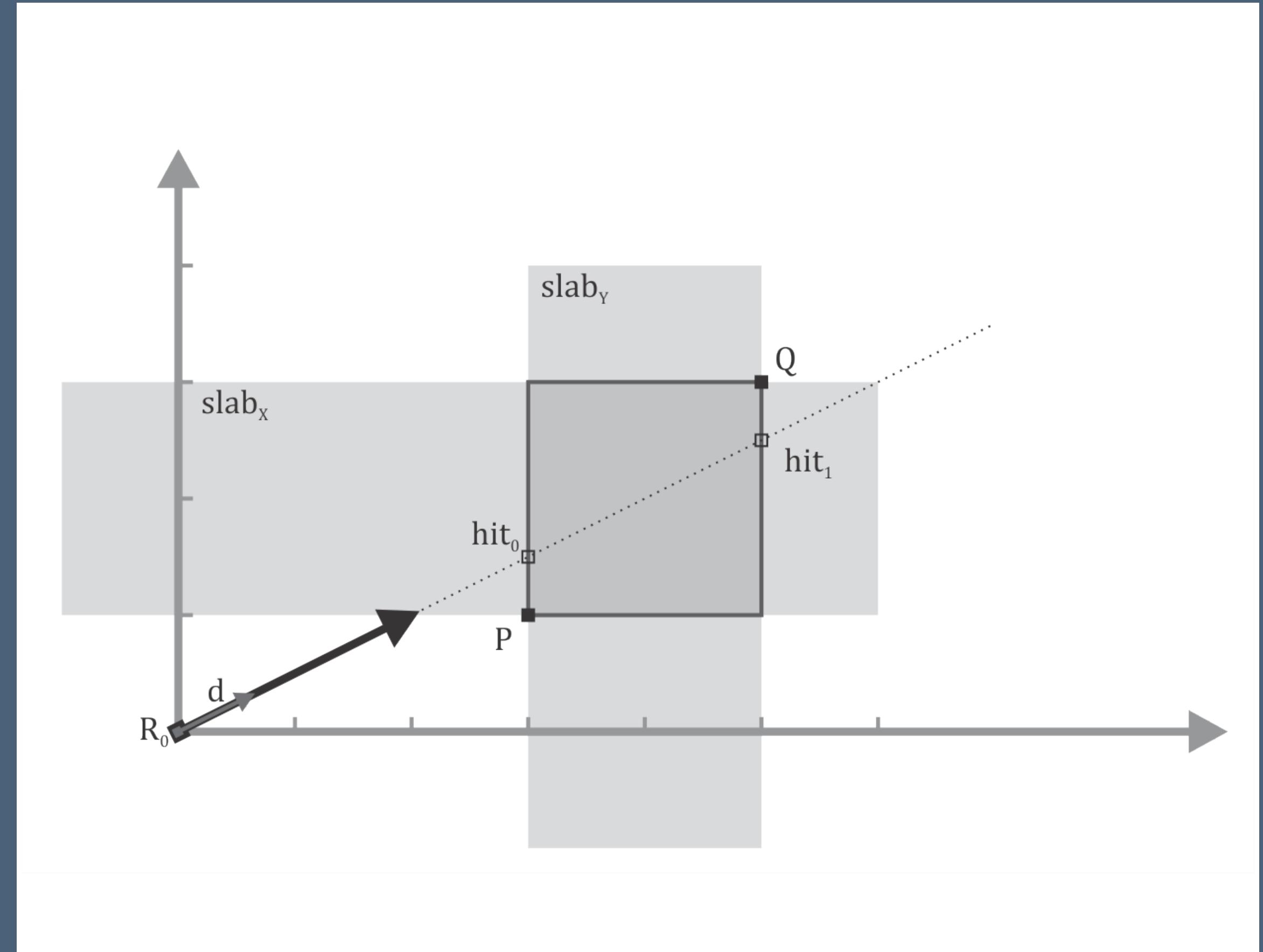
- Da li je početna tačka zraka unutar sfere – poređenje kvadrata rastojanja do centra sa kvadratom prečnika.
- Računanje rastojanja  $t_{CA}$  – skalarni proizvod vektora  $R_0S_c$  i  $d$ , što je intenzitet vektora  $R_0S_c$  projektovanog na pravac  $d$ .
- Vrednost  $t_{HC}^2$  računa se kao razlika  $S_R^2$  i  $D^2$ , koje se računa kao razlika  $L_{OC}^2$  i  $t_{CA}^2$ .
- Vreme  $t$  računa se kao  $t_{CA} \mp t_{HC}$ .



# 2 Osnove raytracing algoritma

Presek zraka sa pravougaonim kutijom - geometrijsko rešenje

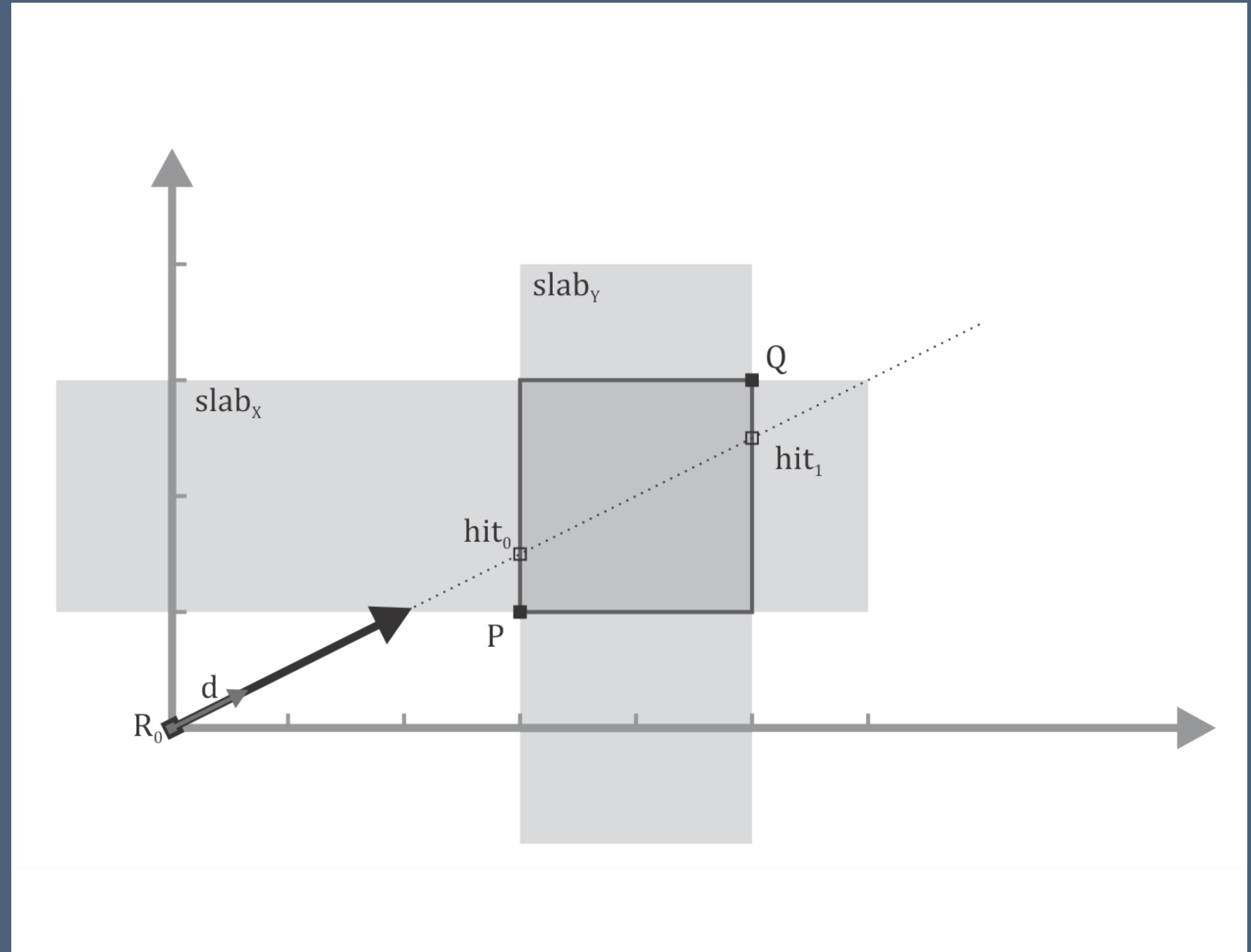
- Pravougaone kutije kao presek tri ploče poravnate po X, Y i Z osama, gde su ploče prostor između dve paralelne ravni. Kutija je definisana temenima P i Q.
- Izračunavanje tačaka  $T_P$  i  $T_Q$  čije komponente sadrže prvi, odnosno poslednji, presek sa svakom od tri ploče.
- Vektori  $t_0$  i  $t_1$  sadrže minimalne, odnosno maksimalne komponente tačaka  $T_P$  i  $T_Q$ .



# 2 Osnove raytracing algoritma

Presek zraka sa pravougaonim kutijom - geometrijsko rešenje

- Traži se maksimalna komponenta  $t_0$ , odnosno minimalna  $t_1$  – pored vrednosti  $\max_0$  i  $\min_1$  čuvaju se i indeksi  $iMax_0$  i  $iMin_1$ .
- Ako je  $\max_0$  veće od  $\min_1$  – do preseka ne dolazi.
- Kada je početna tačka zraka van kutije vreme preseka je  $\max_0$ , inače  $\min_1$ .
- Normala u tački preseka se računa korišćenjem indeksa  $iMax_0$ , odnosno  $iMin_1$ .



# 3 VoxelWorld

## Opis vokselnih tela

- *VoxelWorld* – implementacija vokselnih tela i *rendering* algoritama u okviru većeg *raytracing* programa *GfxLab-2020-2021* prof. dr Marka Savića.
- Apstraktna klasa **BaseM** – korenska klasa većine predstavljenih algoritama:
  - Vokselna tela opisana su nizovima **boolean[][][]** i **Color[][][]**, kao i **Vec3** veličinom ograničavajuće kutije;
  - Instance klase su nepromenljive;
  - Metodi **getBoundingBoxHits** i **getDiffuseFromTerrainPalette**.
- Klasa **BasePF** (opis oblika tela i boje voksela posebnim predikatima) i **BaseF** (opis i oblika tela i boje voksela istim predikatom).

# 3 VoxelWorld

Presek zraka sa vokselom – klasa **HitVoxel**

- Presek – interfejs **Hit** koji vraća vreme preseka zraka s objektom, normalizovanu normalu i UV koordinate presečne tačke.
- Apstraktna klasa **HitRayT** – implementira interfejs **Hit** i čuva zrak i vreme preseka.
- Presek zraka sa vokselom – klasa **HitVoxel** koja nasleđuje **HitRayT** i čuva poziciju voksela.
- Boja voksela enkodirana je u vidu 2D vektora kako bi odgovarala propisanom izlazu metoda **uv** interfejsa **Hit**.
- Veličina modela koji **VoxelWorld** time je ograničena na kocku veličine  $2^{20}$ .

# 3 VoxelWorld

*Brute force algoritam i klasa BruteForce*

```
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) {
```

# 3 VoxelWorld

Brute force algoritam i klasa **BruteForce**

```
17         if (h[0].t() < out.t()) {
18             out = h[0];
19             vθ = 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             vθ = Vec3.xyz(i, j, k);
25         }
26     }
27 }
28 }
29 }
30
31     return new HitVoxel(ray, out, vθ);
32 }
```

# 3 VoxelWorld

## *Direction array* algoritam

- Uzimanjem u obzir pravca kretanja zraka moguće je značajno optimizovati *brute force* algoritam.
- Najveća mana *brute force* pristupa – iteriranje kroz sve voksele matrice.
- Ideja – kada smer iteriranja prati pravac kretanja zraka moguće je:
  1. Pronaći presek sa telom ranije, ali i
  2. Sa sigurnošću tvrditi da je to zaista prvi presek sa telom.
- Podaci o iteriranju izvode se pomoćnom metodom `getLoopData`.
- Sama implementacija gotovo je identična *brute force* rešenju.

# 3 VoxelWorld

Direction array algoritam i metod **getLoopData**

```
01     public static Vec3[] getLoopData(Vec3 modelSize, Ray ray) {  
02  
03         int lenX = modelSize.xInt(),  
...  
07         int dx = ray.d().x() >= 0 ? +1 : -1,  
...  
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; }  
...  
21         return new Vec3[] {  
22             Vec3.xyz(xs, xe, xd),  
...  
25         };  
26     }
```

# 3 VoxelWorld

Direction array algoritam i klasa **DirArray**

```
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(),
...
18      for (int i = xs; i != xe; i += xd) {
19          for (int j = ys; j != ye; j += yd) {
20              for (int k = zs; k != ze; k += zd) {
21
22                  if (!isPopulated(i, j, k)) continue;
```

# 3 VoxelWorld

*Direction array algoritam i klasa DirArray*

```
23
24         Hit[] h = getHits(Vec3.xyz(i, j, k), ray);
25
26         if (h.length == 0) continue;
27
28         if (h[0].t() > afterTime)
29             return new HitVoxel(ray, h[0], i, j, k);
30     }
31 }
32 }
33
34     return Hit.POSITIVE_INFINITY;
35 }
```

# 3 VoxelWorld

## *Direction array algoritam i klasa DirArray0*

- Očigledna manja predloženog pristupa vidljiva je u situacijama kada zrak u potpunosti promašuje objekat.
- Jednostavno rešenje predstavlja provera preseka zraka sa ograničavajućom kutijom objekta, i vraćanje `Hit.POSITIVE_INFINITY` vrednosti ukoliko takav presek izostaje.

```
04     Hit[] boundingBoxHits =
05         getHits(Vec3.ZERO, Vec3.xyz(lenX(), ...), ray);
06
07     if (boundingBoxHits.length == 0) return Hit.POSITIVE_INF;
```

- Problem je i dalje (efikasno) nalaženje ulazne i izlazne tačke zraka u i iz modela, kao i modeli kod kojih je većina voksela skoncentrisana u sredini modela.

# 3 VoxelWorld

## *Grid march* algoritam

- Ideja – korišćenje ograničavajuće kutije kako bi se:
  1. Prekinulo izračunavanje kada zrak promašuje matricu,
  2. Dobile ulazna i izlazna tačka zraka u preseku sa matricom, i
  3. Izračunali polazište i ishodište iteriranje uzevši u obzir ulaz i pravac kretanja zraka.
- *Grid march* pristup je u osnovi pokušaj da se matrica voksla ograniči na kvadar čija su temena ulazna i izlazna tačka zraka, odnosno čija je dijagonala presek zraka i ograničavajuće kutije.
- Početne vrednosti iteriranja računaju se drugačije nego što je to bio slučaj kod *direction array* pristupa.

# 3 VoxelWorld

Grid march algoritam i metod `getLoopData`

```
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         ...
07         ray.d().y() >= 0 ? +1 : -1,
08         ray.d().z() >= 0 ? +1 : -1);
09
10    Vec3 v0 = ray.at(boundingBoxHits[0].t()).floor();
11    Vec3 u0 = Vec3.xyz(
12        v0.xInt() == size.xInt() ? -1 : (v0.xInt() == -1 ? 1 : 0),
13        ...
14        v0.yInt() == size.yInt() ? -1 : (v0.yInt() == -1 ? 1 : 0),
15        v0.zInt() == size.zInt() ? -1 : (v0.zInt() == -1 ? 1 : 0));
16
17    Vec3 v1 = ray.at(boundingBoxHits[1].t()).floor();
18    Vec3 u1 = Vec3.xyz(
19        (v1.xInt() == size.xInt() ? -1 : (v1.xInt() == -1 ? 1 : 0)),
20        ...
21        (v1.yInt() == size.yInt() ? -1 : (v1.yInt() == -1 ? 1 : 0)),
22        (v1.zInt() == size.zInt() ? -1 : (v1.zInt() == -1 ? 1 : 0)));
23
24    return new Vec3[] { v0.add(u0), v1.add(u1.add(vx)), vx };
25 }
```

# 3 VoxelWorld

## *Grid march* algoritam i klasa **GridMarch1**

- Nakon provere da li zrak uopšte seče vokselnu matricu, izračunavaju ulazna tačka zraka  $S = (x_s, y_s, z_s)$ , izlazna tačka zraka  $E = (x_e, y_e, z_e)$  i inkrement  $D = (x_d, y_d, z_d)$ .
- Iteriranje se ovog puta ne vrši po svim ćelijama dela matrice – ukoliko se u trenutnoj ćeliji ne nalazi voksel traže se preseci zraka sa susedima ćelije.
- Svaka ćelija ima ukupno šest suseda, ali se uzevši u obzir smer zraka broj mogućih sledećih ćelija svodi na tri.
- Sledeća posmatrana ćelija je onaj sused čije je vreme preseka sa zrakom najmanje. Ova informacija čuva se u promenljivoj **idx** pomoću koje se ažuriraju koordinate **xs**, **ys** i **zs**. U slučaju izlaska iz matrice vraća se **Hit.POSITIVE\_INFINITY**.

# 3 VoxelWorld

Grid march algoritam i klasa **GridMarch1**

```
01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04      ... // bounding box hit check
05
06
07
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(),
14
15      ...
16
17      int[] xi = { 0, 0, xd };
18      int[] yi = { 0, yd, 0 };
19      int[] zi = { zd, 0, 0 };
20
21
22      while (
```

# 3 VoxelWorld

Grid march algoritam i klasa **GridMarch1**

```
23         xs != -1 && xs != xe && ... ) {  
24  
25             Vec3 p = Vec3.xyz(xs, ys, zs);  
26             Hit[] hits = getHits(p, ray);  
27  
28             if (isPopulated(p) && hits[0].t() > afterTime)  
29                 return new HitVoxel(ray, hits[0], xs, ys, zs);  
30  
31             Hit[][] neighbours = new Hit[3][2];  
32  
33             int idx = -1;  
34             double minTime = Double.MAX_VALUE;  
35  
36             for (int i = 0; i < neighbours.length; i++) {  
37                 neighbours[i] =  
38             }
```

# 3 VoxelWorld

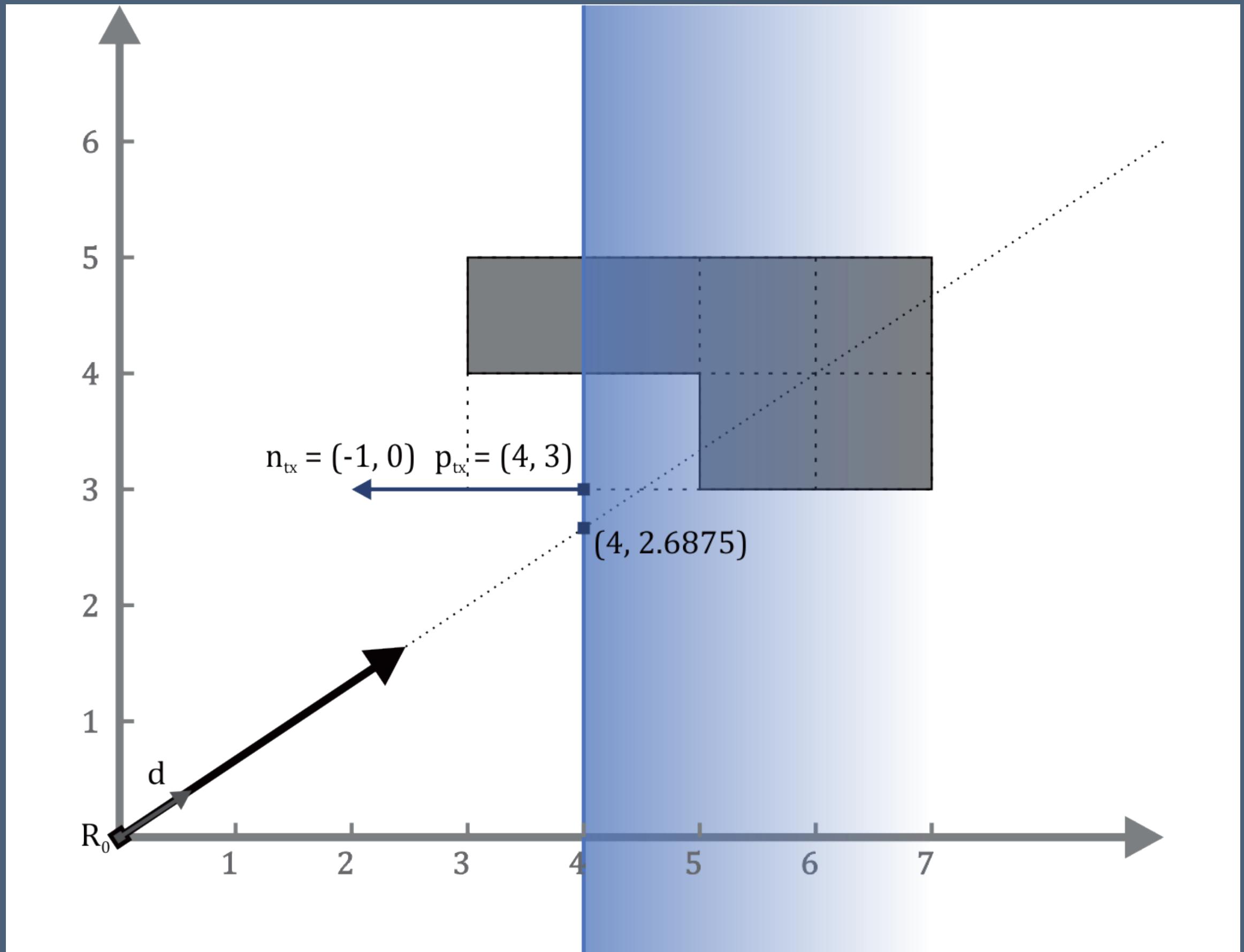
Grid march algoritam i klasa **GridMarch1**

```
39         getHits(p.add(Vec3.xyz(xi[i], yi[i], zi[i])), ray);
40
41         if (neighbours[i].length == 0) {
42             continue;
43         } else if (minTime > neighbours[i][0].t()) {
44             idx = i;
45             minTime = neighbours[i][0].t();
46         }
47     }
48
49     xs += xi[idx];
...
52     }
53
54     return Hit.POSITIVE_INFINITY;
55 }
```

# 3 VoxelWorld

Grid march algoritam i klasa **GridMarch2**

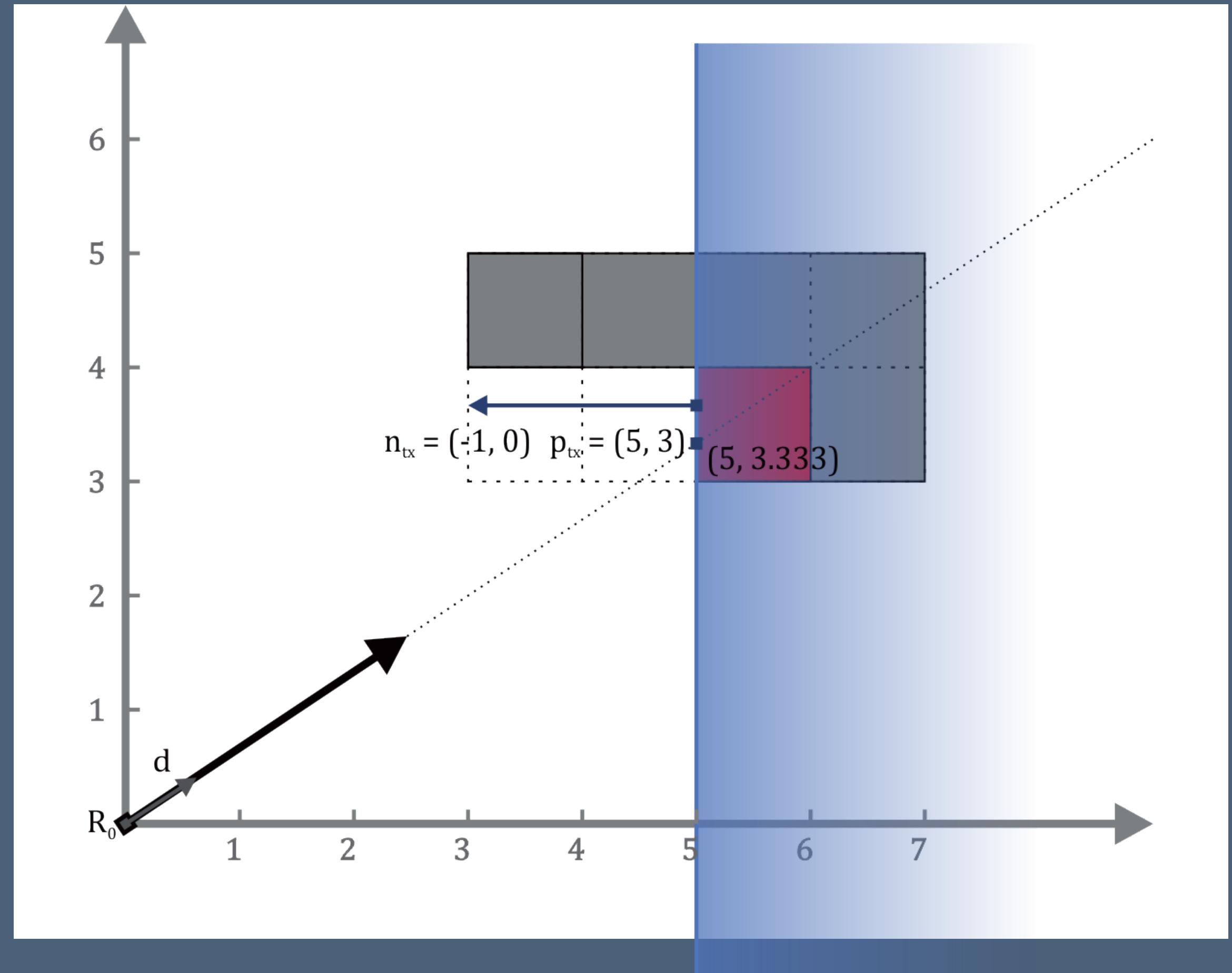
- Prva ćelija matrice koju preseca zrak je na poziciji  $(4, 3)$ .
- Ćelija u kojoj zrak izlazi iz matrice je na poziciji  $(7, 4)$ .
- Izračunava se vreme preseka sa poluravnima  $x = 4$  i  $y = 3$ , čime se dobijaju tačke  $(4, 2.6875)$  i  $(4.5, 3)$ .
- Vremena preseka su  $(4.808, 5.405)$ , a  $dt$  iznosi  $(1.2, 1.8)$ .



# 3 VoxelWorld

## *Grid march* algoritam i klasa **GridMarch2**

- Počevši od ćelije (4, 3) sledeća se, ukoliko je ona prazna računa na sledeći način:
  1. Na vremena preseka  $t$  dodaje se  $dt$  - tako (4.808, 5.405) postaje (6.008, 7.205) - i traži se minimum, što je u ovom slučaju x koordinata.
  2. Vreme preseka sa sledećom ćelijom tako postaje (6.008, 5.405).
- U sledećem ponavljanju iteracije dolazi do preseka sa popunjenoj ćelijom.



# 3 VoxelWorld

Grid march algoritam i klasa **GridMarch2**

```
01  @Override
02  public Hit firstHit(Ray ray, double afterTime) {
03
04    ...
05    // bounding box hit check
06    Vec3[] loopData = Util.getLoopData(len(), ray, boundingBoxHits);
07
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.pn(v0.add(s2), s1.mul(Vec3.EX)).hits(ray)[0].t(),
18
19    ...
20  }
```

# 3 VoxelWorld

*Grid march algoritam i klasa **GridMarch2***

```
21     Vec3 dt = s0.div(ray.d());  
22  
23     while (v0.inBoundingBox(v1)) {  
24         if (isPopulated(v0) && t.max() > afterTime)  
25             return new HitVoxel(  
26                 ray, HitData.tn(t.max(), s1.mul(Vec3.E[t.maxIndex()])), v0);  
27  
28         Vec3 tNext = t.add(dt);  
29         int k = tNext.minIndex();  
30         v0 = v0.add(s0.mul(Vec3.E[k]));  
31         t = t.add(dt.mul(Vec3.E[k]));  
32     }  
33  
34     return Hit.POSITIVE_INFINITY;  
35 }  
36 }
```

# 3 VoxelWorld

## Grid march algoritam i klasa **GridMarch20**

- Dalja optimizacija *grid march* pristupa kroz izbegavanje nepotrebnih i efikasnije izvršavanje neophodnih izračunavanja.

```
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 v0 = ray.at(boundingBoxHits[0].t()).floor();
08      Vec3 u0 = Vec3.xyz(
09          v0.xInt() == lenX() ? -1 : (v0.xInt() == -1 ? 1 : 0),
10          v0.yInt() == lenY() ? -1 : (v0.yInt() == -1 ? 1 : 0),
11          v0.zInt() == lenZ() ? -1 : (v0.zInt() == -1 ? 1 : 0));
12      v0 = v0.add(u0);
```

# 3 VoxelWorld

*Grid march algoritam i klasa **GridMarch20***

```
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  = v0.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);
```

# 3 VoxelWorld

*Grid march algoritam i klasa **GridMarch20***

```
30
31     if (isPopulated(v0) && tMax > afterTime)
32         return new HitVoxel(
33             ray, HitData.tn(tMax, s1.mul(Vec3.E[k])), v0);
34
35     Vec3 tNext = t.add(dt);
36     k = tNext.minIndex();
37     v0 = v0.add(s0A[k]);
38
39     if (v0.get(k) == -1 || v0.get(k) == lenA()[k]) break;
40
41     t = t.add(dtA[k]);
42 }
43
44 return Hit.POSITIVE_INFINITY;
45 }
```

# 3 VoxelWorld

## Oktalna stabla

- Oktalna stabla su strukture podataka u kojima svaki čvor ima osam podčvorova.
- Koriste za particonisanje trodimenzionalnog prostora rekurzivnim deljenjem istog na osam oktanata.
- Vokselni objekti se putem oktalnog stabla mogu prikazati u različitim rezolucijama.
- Kada je ćelija višeg nivoa prazna, tada je moguće tvrditi da na nižem nivou takođe nema voksela.
- Oktablna stabla opisana su 4D nizovima, gde prva dimenzija predstavlja nivo stabla, a preostale tri položaj određene ćelije.

# 3 VoxelWorld

## Octree brute force algoritam i klasa **OctreeBF**

- Algoritam počinje proveru preseka zraka sa modelom od korena stabla koji predstavlja celu scenu.
- Za svako dete trenutnog čvora proverava se presek sa zrakom i, ukoliko do dolazi do preseka, dete se smešta u listu čvorova čija će deca biti proverena u sledećoj iteraciji.
- Pozicija svakog deteta enkodirana je u indeksu u opsegu [0.. 7].
- Najniži nivo stabla predstavljaju sami vokseli koji čine model. Ukoliko se zrak preseca sa nekim od njih ti preseci se smeštaju u listu koja se sortira.
- Prvi presek zraka sa vokselnim telom je prvi element sortirane liste preseka.

# 3 VoxelWorld

Octree brute force algoritam i klasa `OctreeBF`

```
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) {
```

# 3 VoxelWorld

Octree brute force algoritam i klasa `OctreeBF`

```
17         for (int i = 0; i < 8; i++) {  
18             Vec3 idx = getVec3FromIndex(i);  
19             Vec3 pos = v.mul(2);  
20             Vec3 currPos = pos.add(idx);  
21  
22             Vec3 p = currPos.mul(unit);  
23  
24             Hit[] hits = Box.$.pd(p, unitVec3).hits(ray);  
25  
26             if (isPopulated(l - 1, currPos) &&  
27                 hits.length > 0 && hits[0].t() > afterTime)  
28                 l1.add(currPos);  
29         }  
30     }  
31 }
```

# 3 VoxelWorld

Octree brute force algoritam i klasa `OctreeBF`

```
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));
```

# 3 VoxelWorld

Octree brute force algoritam i klasa OctreeBF

```
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 }
```

# 3 VoxelWorld

Octree brute force algoritam i klasa OctreeRec

```
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  }
```

# 3 VoxelWorld

Octree brute force algoritam i metod **octreeDFS** 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         if (ray.d().y() >= 0) {
15             ys = 0; ye = 2; yd = +1;
16         } else {
```

# 3 VoxelWorld

Octree brute force algoritam i metod **octreeDFS** klase **OctreeRec**

```
17         ys = 1; ye = -1; yd = -1;
18     }
19     if (ray.d().z() >= 0) {
20         zs = 0; ze = 2; zd = +1;
21     } else {
22         zs = 1; ze = -1; zd = -1;
23     }
24
25     for (int i = xs; i != xe; i += xd) {
26         for (int j = ys; j != ye; j += yd) {
27             for (int k = zs; k != ze; k += zd) {
28
29                 Vec3 idx = Vec3.xyz(i, j, k);
30                 Vec3 pos = curr.mul(2);
31                 Vec3 currPos = pos.add(idx);
32             }
33         }
34     }
35 }
```

# 3 VoxelWorld

Octree brute force algoritam i metod **octreeDFS** klase **OctreeRec**

```
33         Vec3 p = currPos.mul(unit);
34
35         if (isPopulated(lvl - 1, currPos)) {
36
37             Hit hit = Box.$.pd(p, unitVec3).firstHit(ray, afterTime);
38
39             if (hit != Hit.POSITIVE_INFINITY)
40                 if (lvl > 1)
41                     octreeDFS(container, currPos, lvl - 1, ray, afterTime);
42                 else
43                     container.add(new Vec3Hit(currPos, hit));
44
45     }
46
47 }
48 }
```

# 3 VoxelWorld

Octree brute force algoritam i metod **octreeDFS** klase **OctreeRec0**

```
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;
```

# 3 VoxelWorld

Octree brute force algoritam i metod **octreeDFS** klase **OctreeRec0**

```
17      } else {
18          ys = 1; ye = -1; yd = -1;
19      }
...
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)) {
```

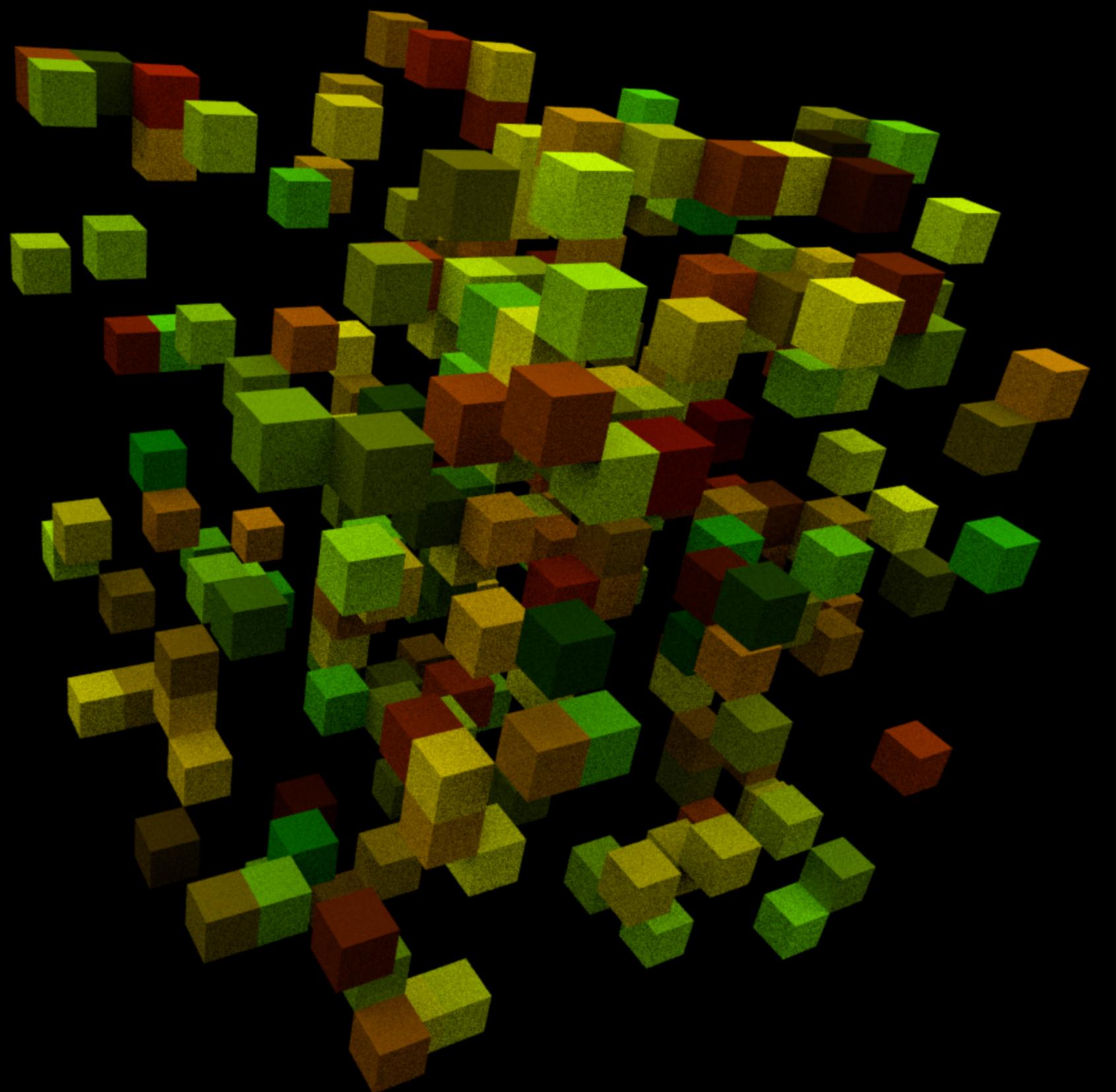
# 3 VoxelWorld

Octree brute force algoritam i metod **octreeDFS** klase **OctreeRec0**

```
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 }
```

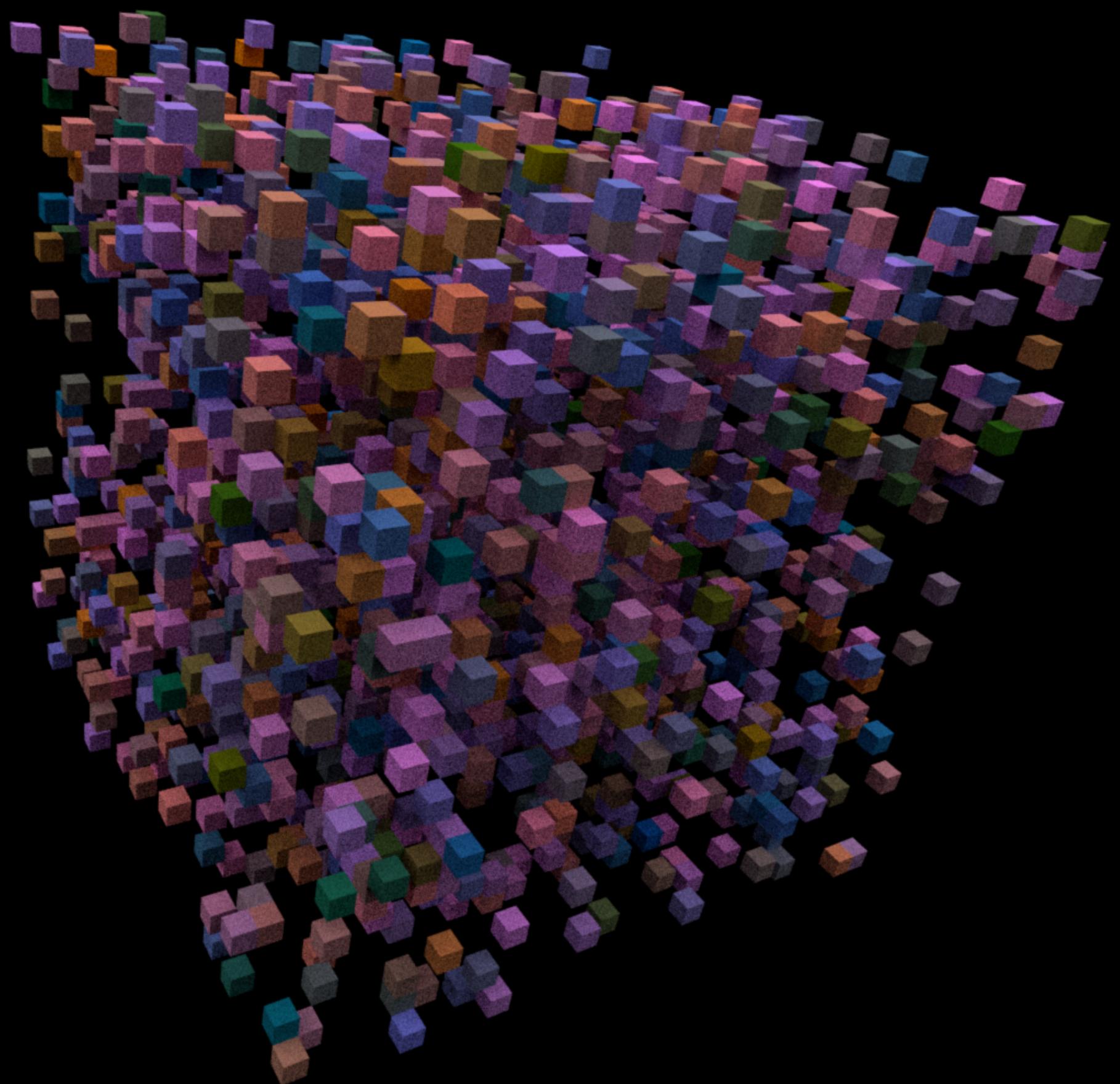
# 4 Rezultati testiranja

Nasumično generisana kocka. 6,25% popunjenošti, veličina 16



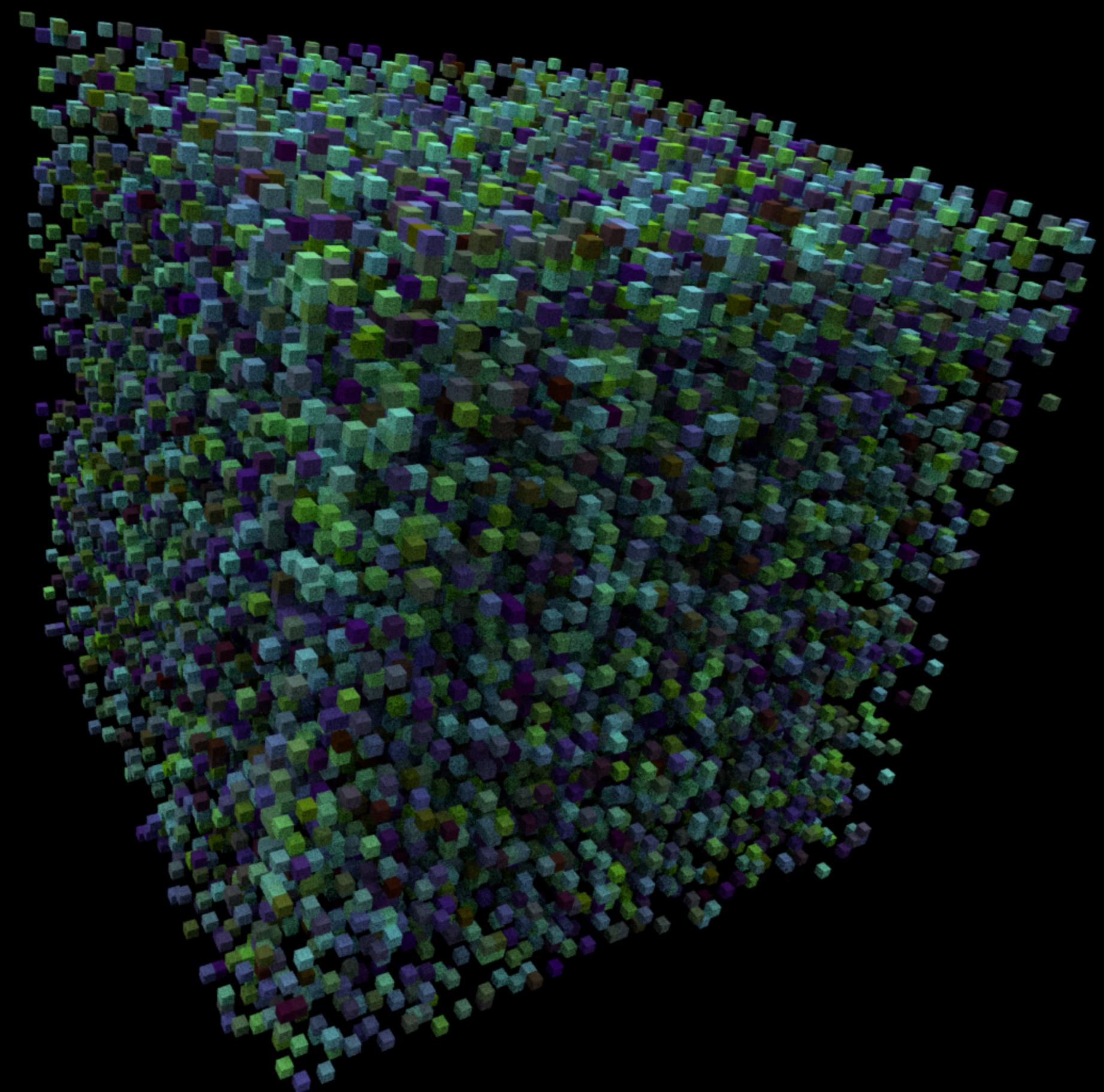
# 4 Rezultati testiranja

Nasumično generisana kocka. 6,25% popunjenošti, veličina 32



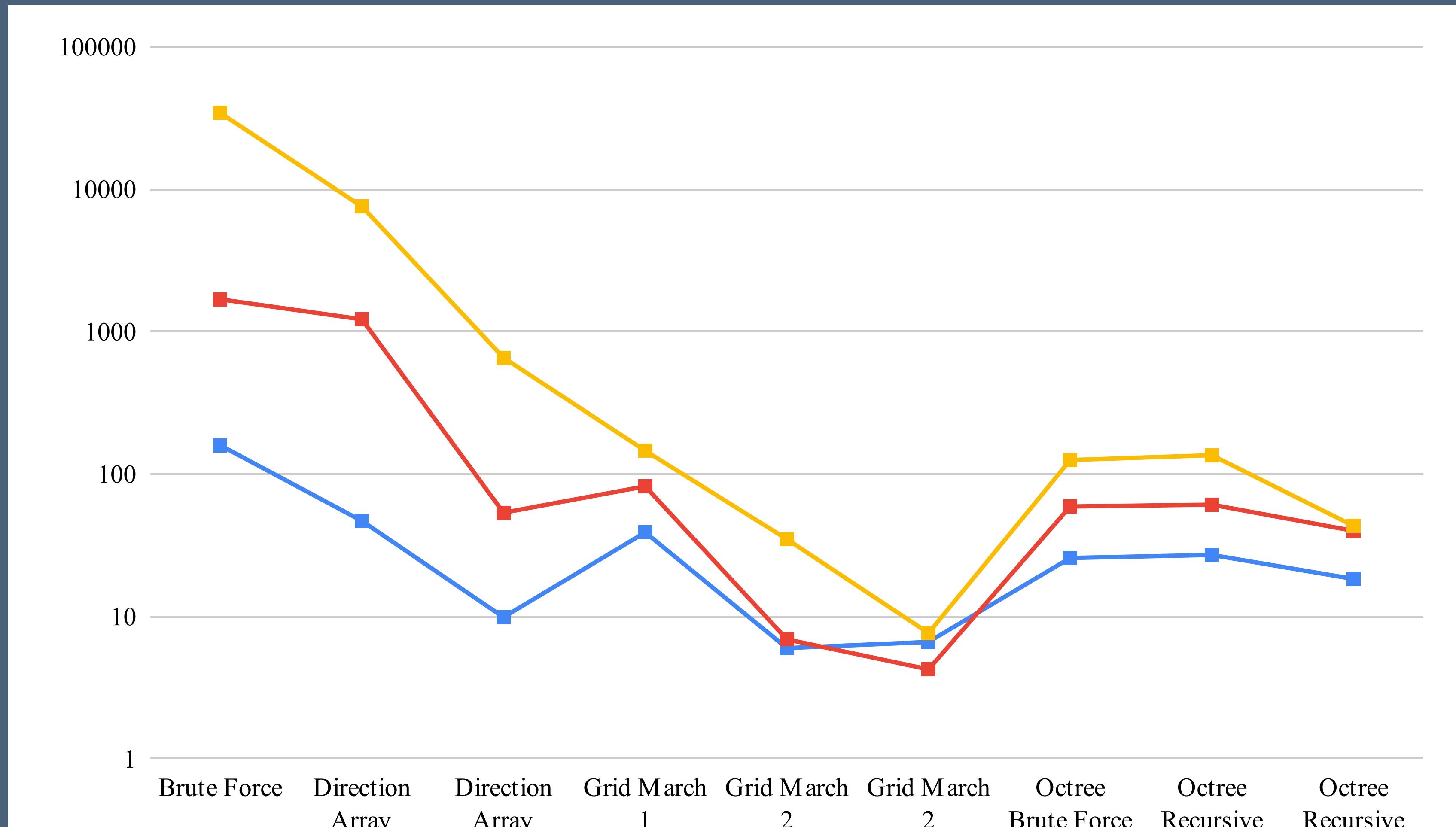
# 4 Rezultati testiranja

Nasumično generisana kocka. 6,25% popunjenošti, veličina 64



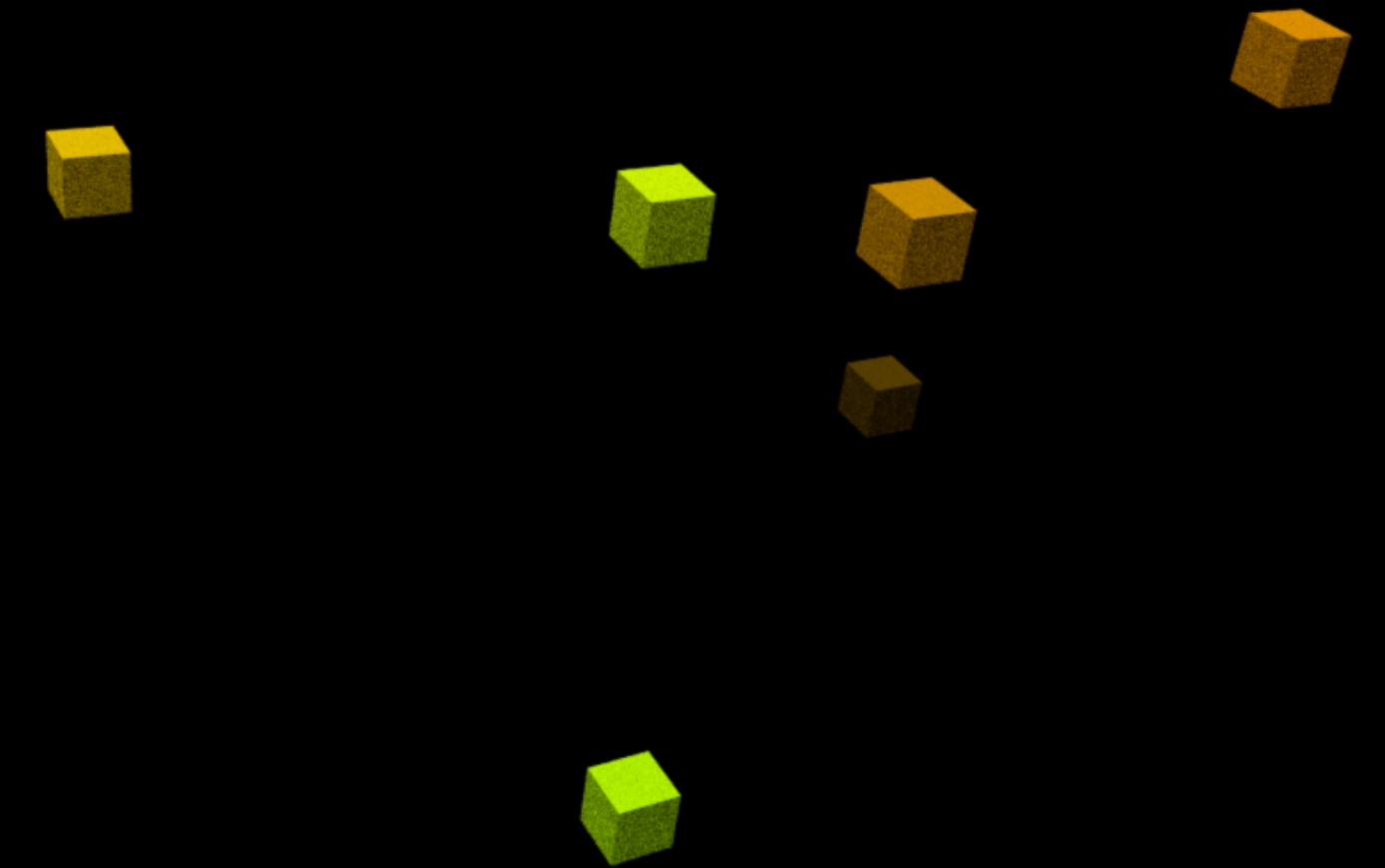
# 4 Rezultati testiranja

Nasumično generisana kocka. 6,25% popunjenošti, rezultati



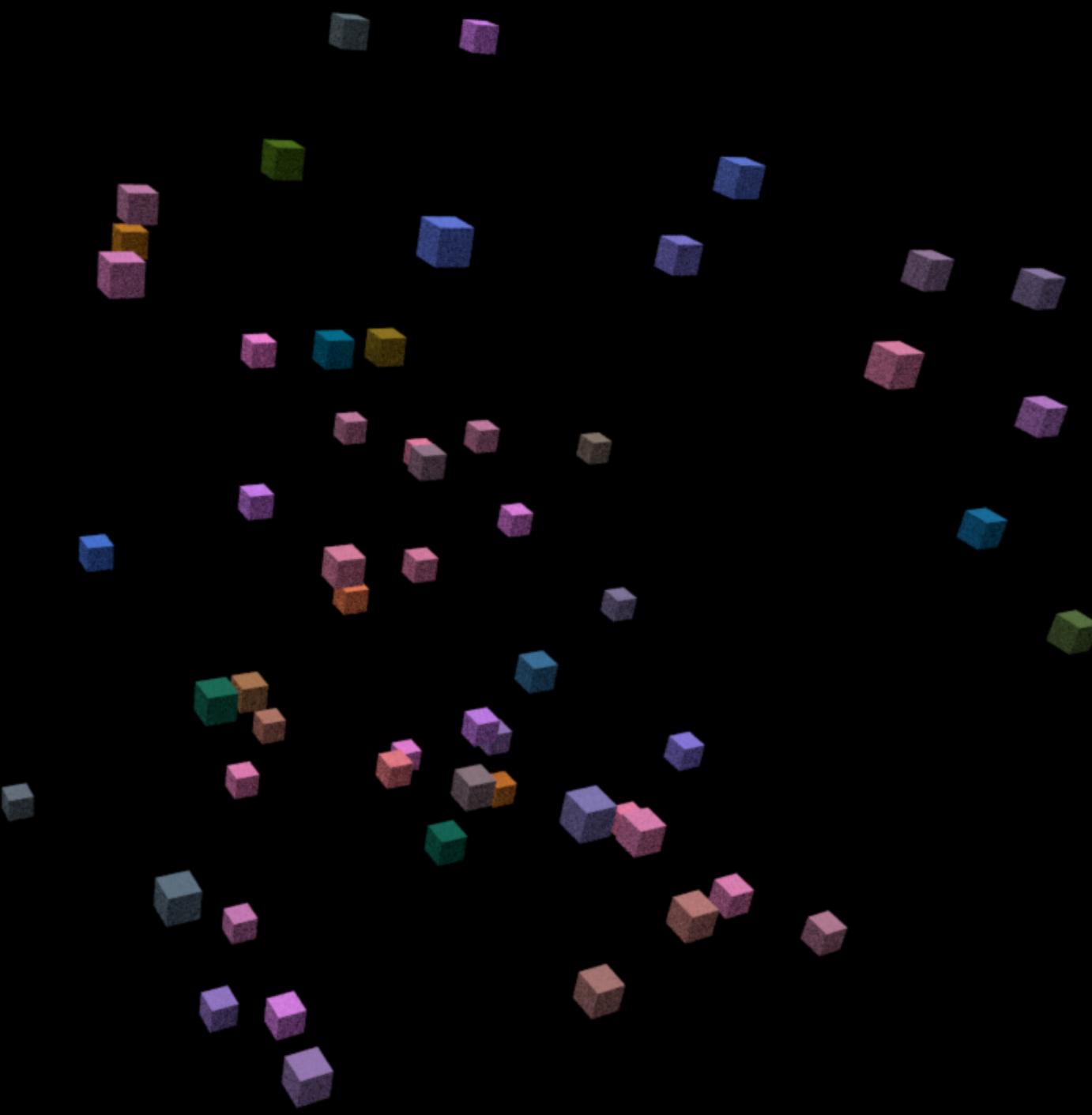
# 4 Rezultati testiranja

Nasumično generisana kocka. 0,19% popunjenošti, veličina 16



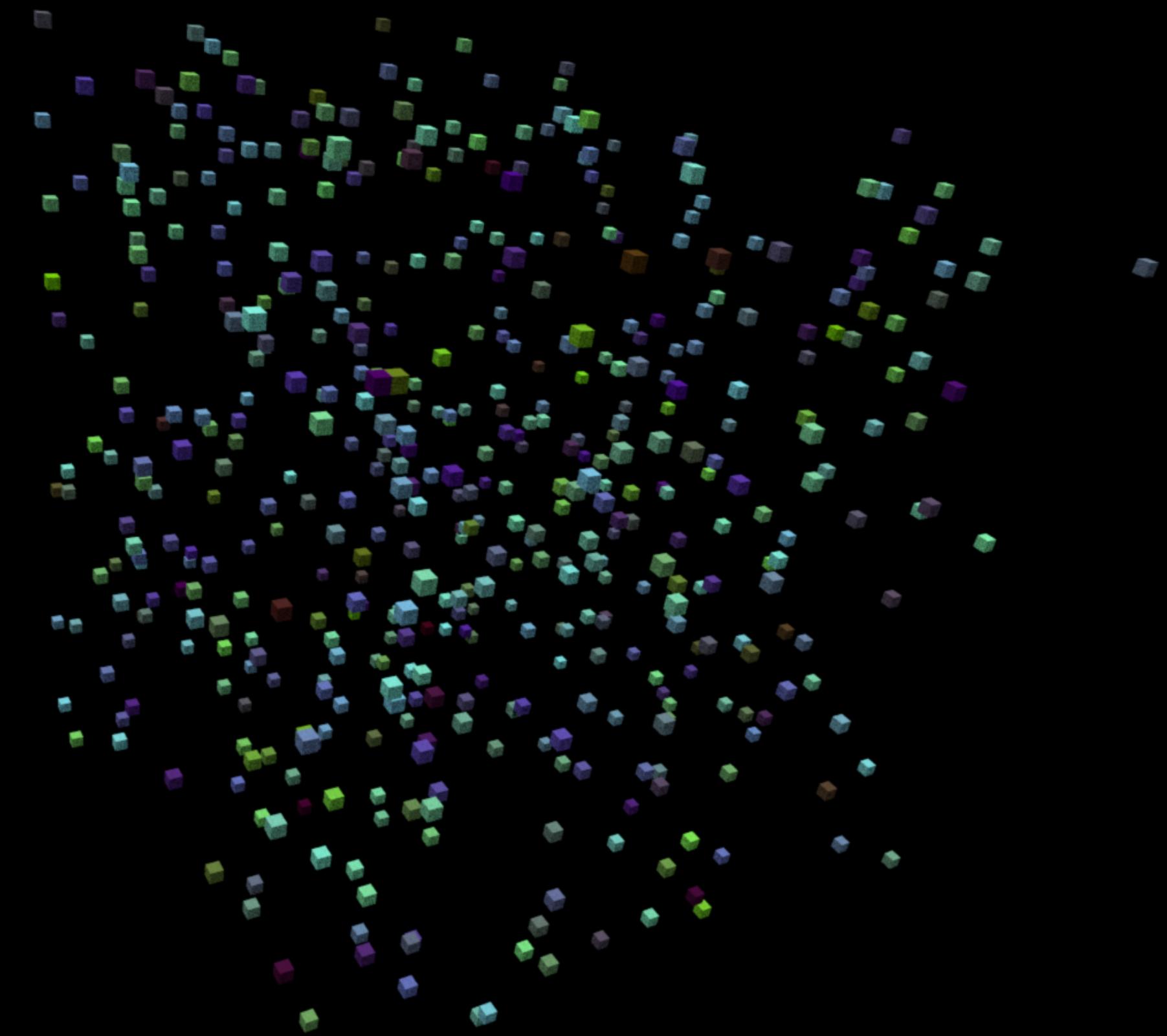
# 4 Rezultati testiranja

Nasumično generisana kocka. 0,19% popunjenošti, veličina 32



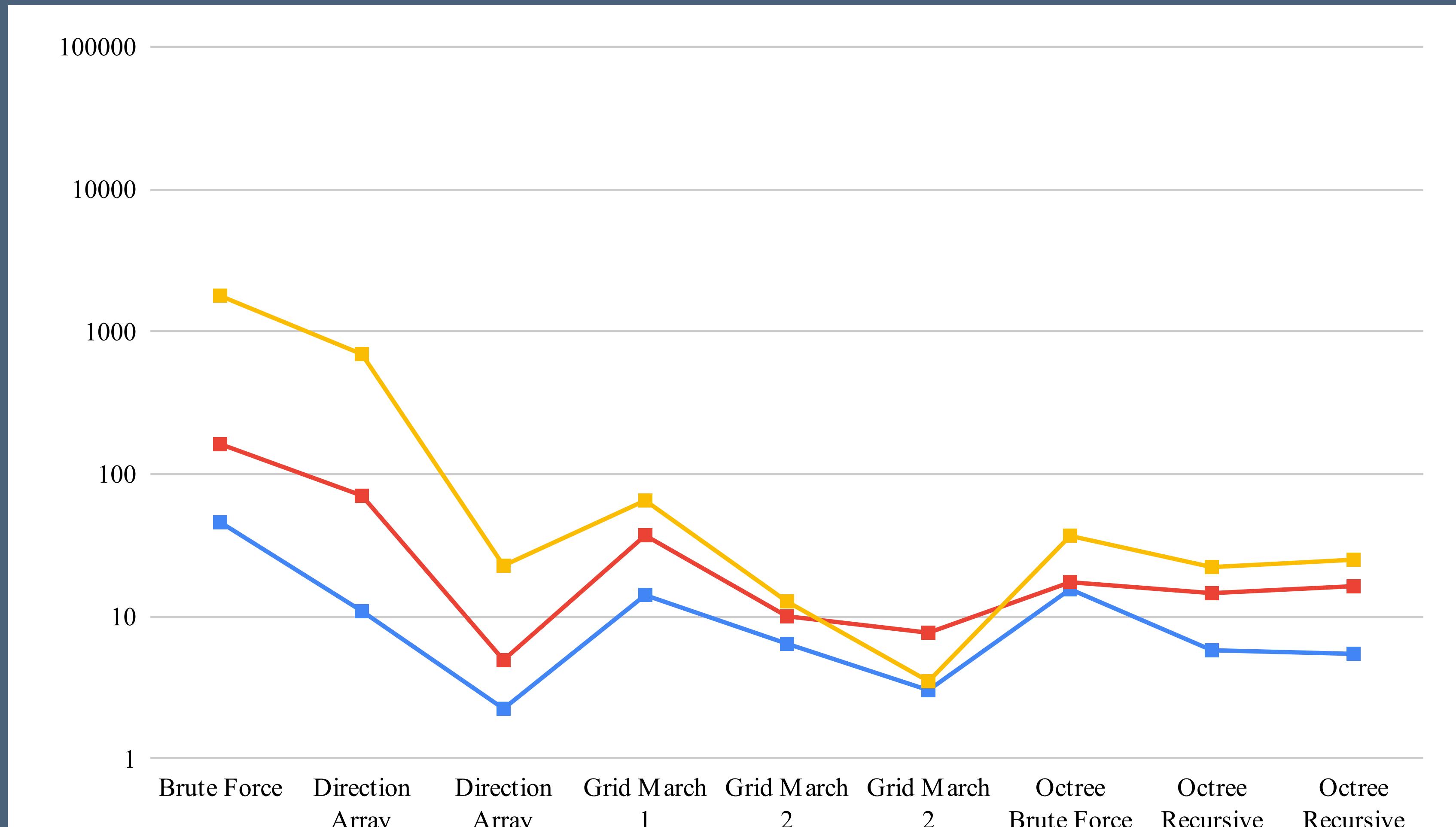
# 4 Rezultati testiranja

Nasumično generisana kocka. 0,19% popunjenošti, veličina 64



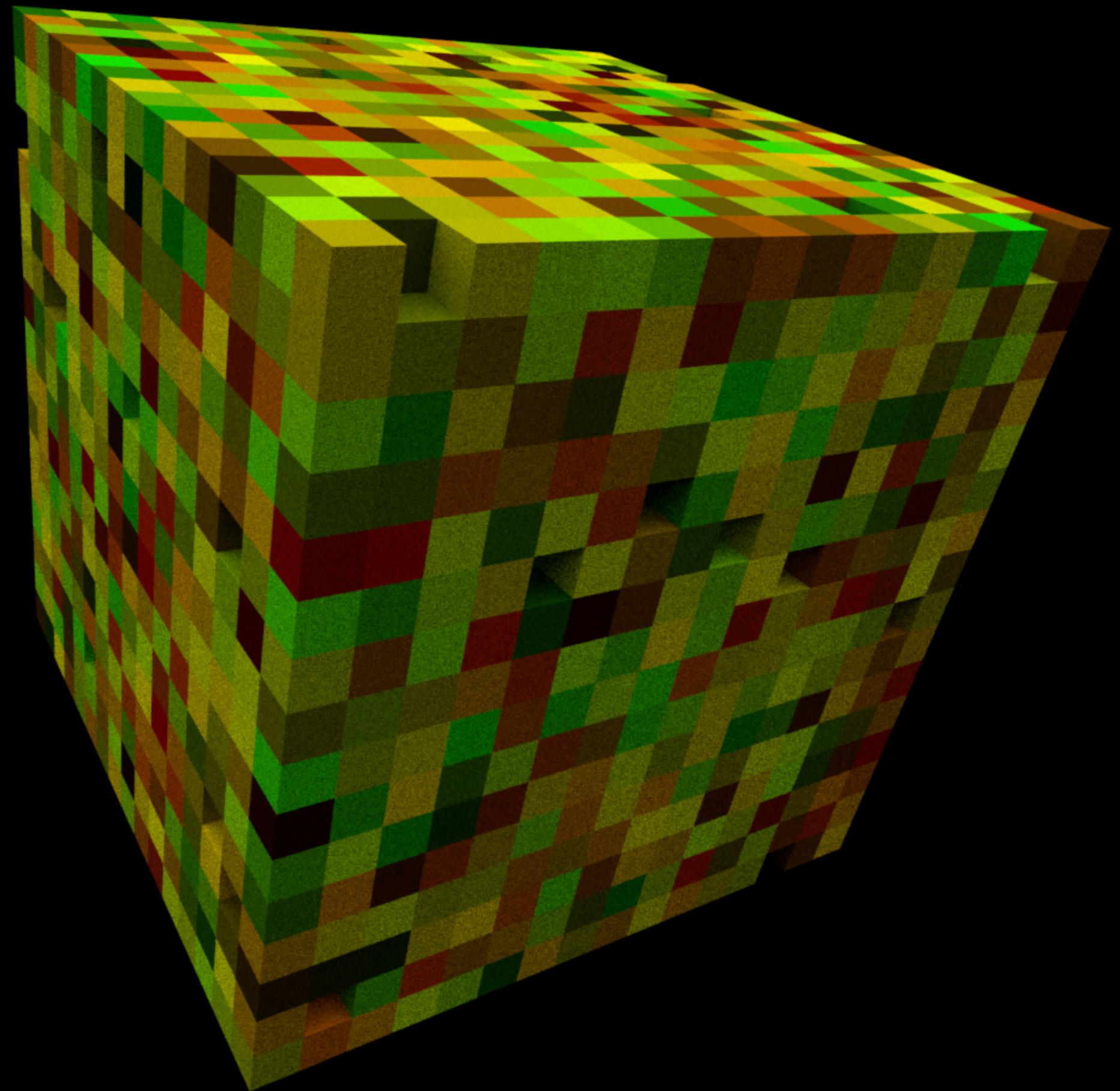
# 4 Rezultati testiranja

Nasumično generisana kocka. 0,19% popunjenošti, rezultati



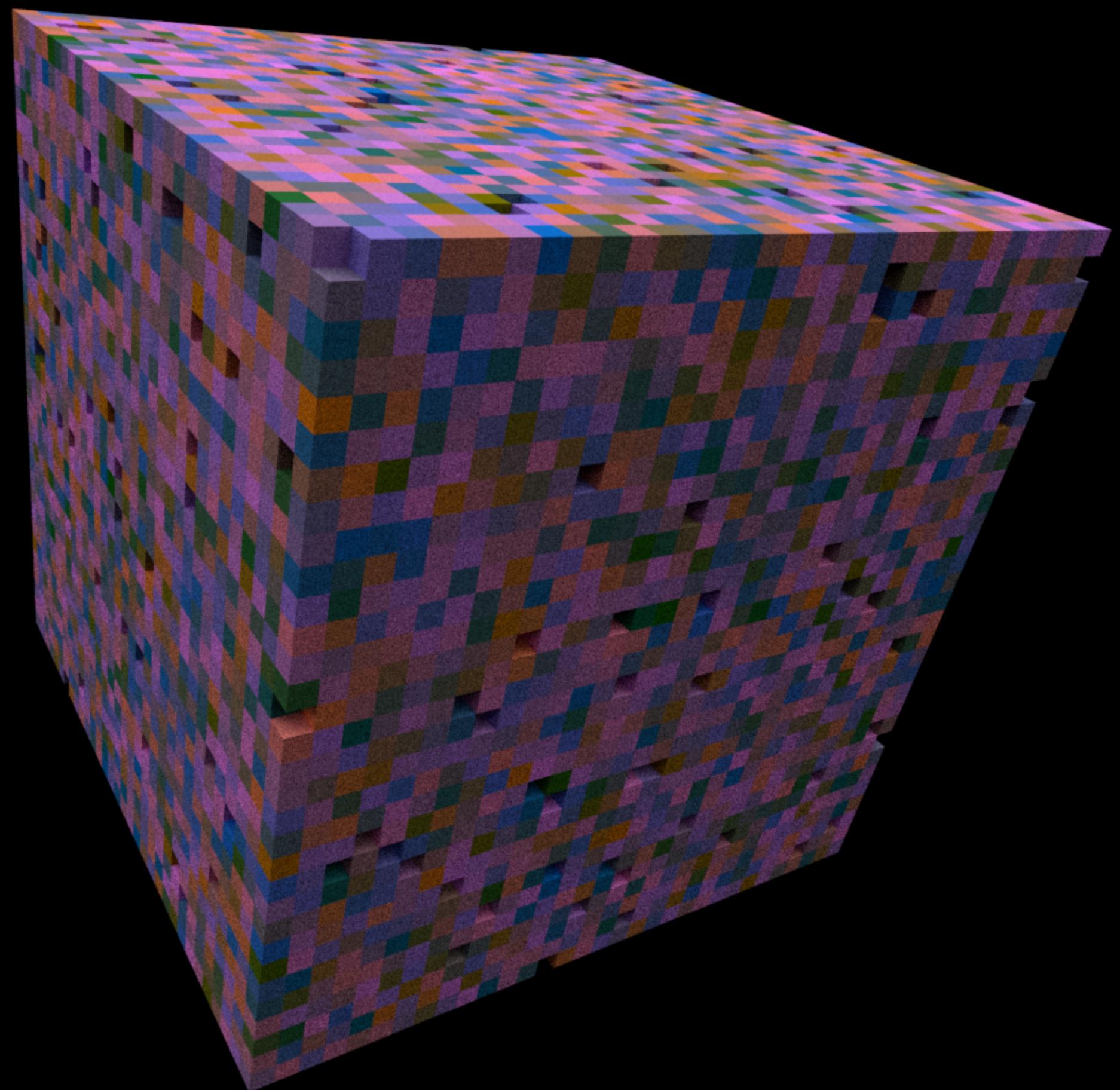
# 4 Rezultati testiranja

Nasumično generisana kocka. 96,87% popunjenošti, veličina 16



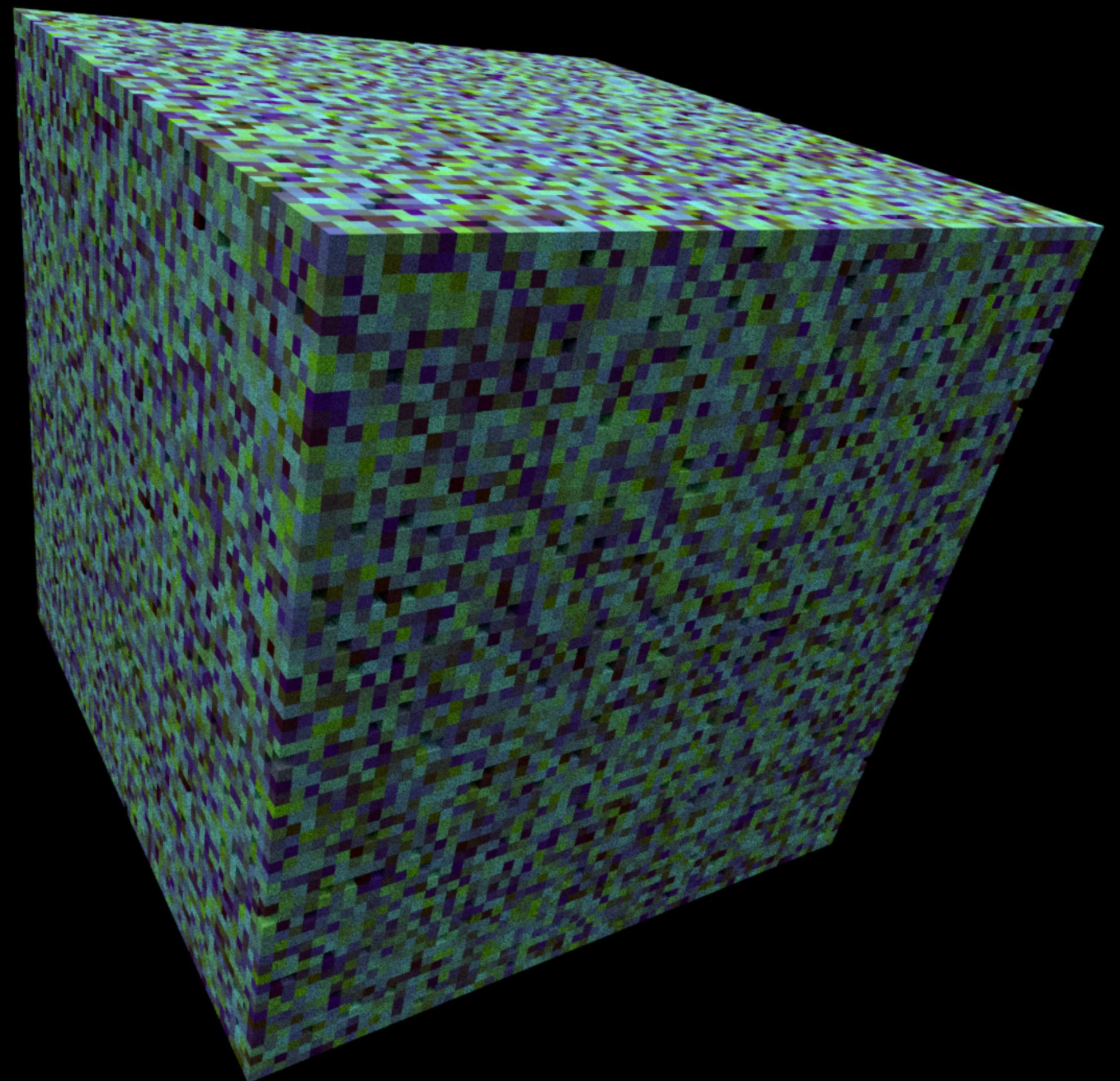
# 4 Rezultati testiranja

Nasumično generisana kocka. 96,87% popunjenošti, veličina 32



# 4 Rezultati testiranja

Nasumično generisana kocka. 96,87% popunjenošti, veličina 64



# 4 Rezultati testiranja

Nasumično generisana kocka. 0,19% popunjenošti, rezultati



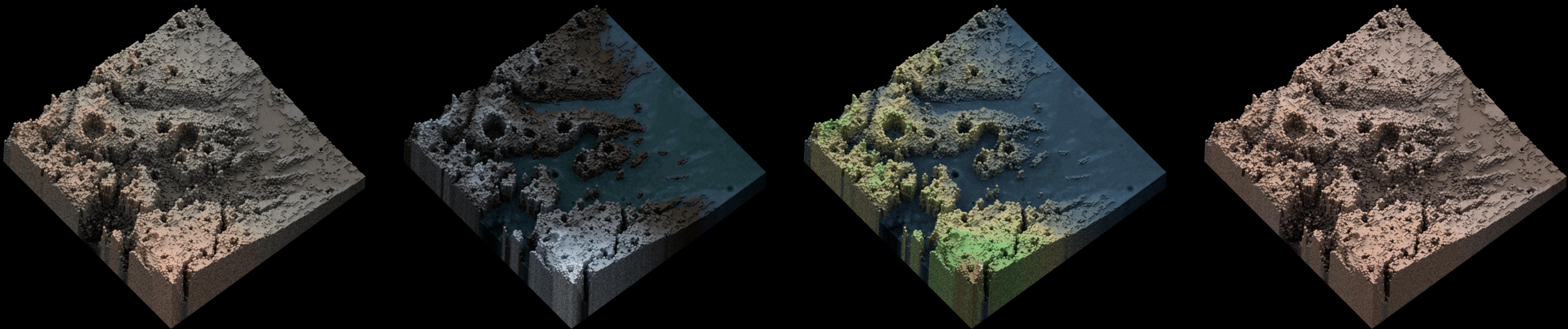
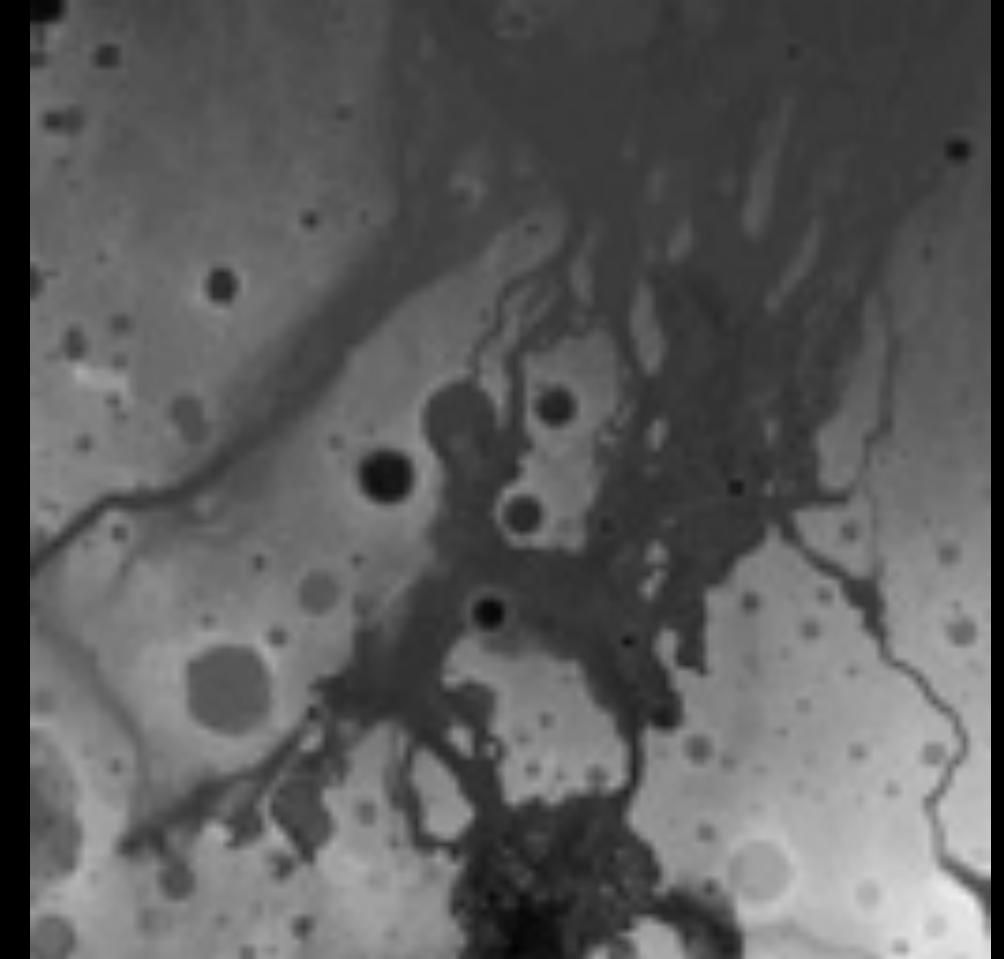
# 4 Rezultati testiranja

## Mars MGS MOLA DEM

- Drugi test model predstavlja vokselni model detalja površine Marsa generisan na osnovu digitalnog modela elevacije Marsa.
- Kreiranje vokselnog modela realizovano je u dva koraka:
  1. Za svaki piksel mape kreira se "stub" voksela u modelu visine koja je određena osvetljeniču piksela, odnosno visinom terena;
  2. Nakon kreiranja modela određuje se boja voksela u modelu prema jednoj od predefinisanih paleta terena koja se zatim meša sa vrednošću sive sa mape.
- Određene palete simuliraju prisustvo vodenih površina tako što delove terena pod vodom renderuju na uniformnoj visini.

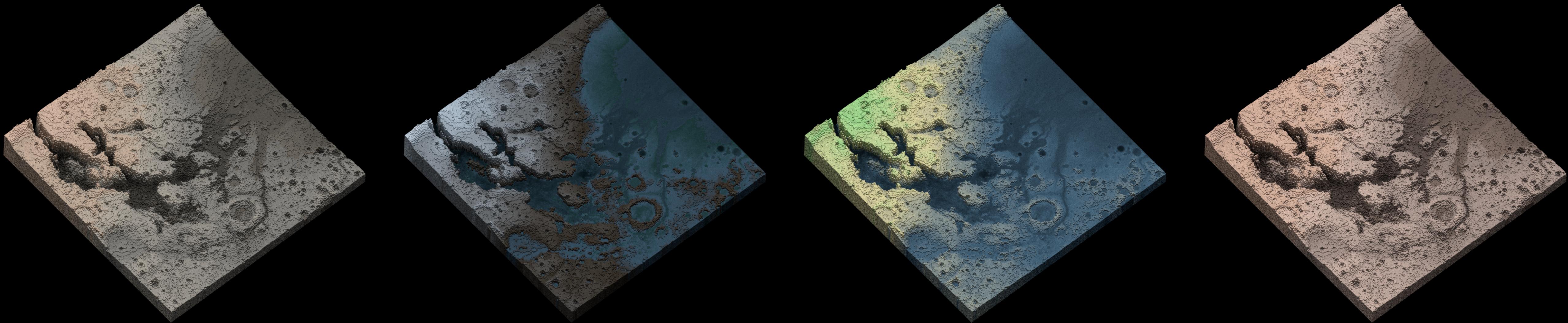
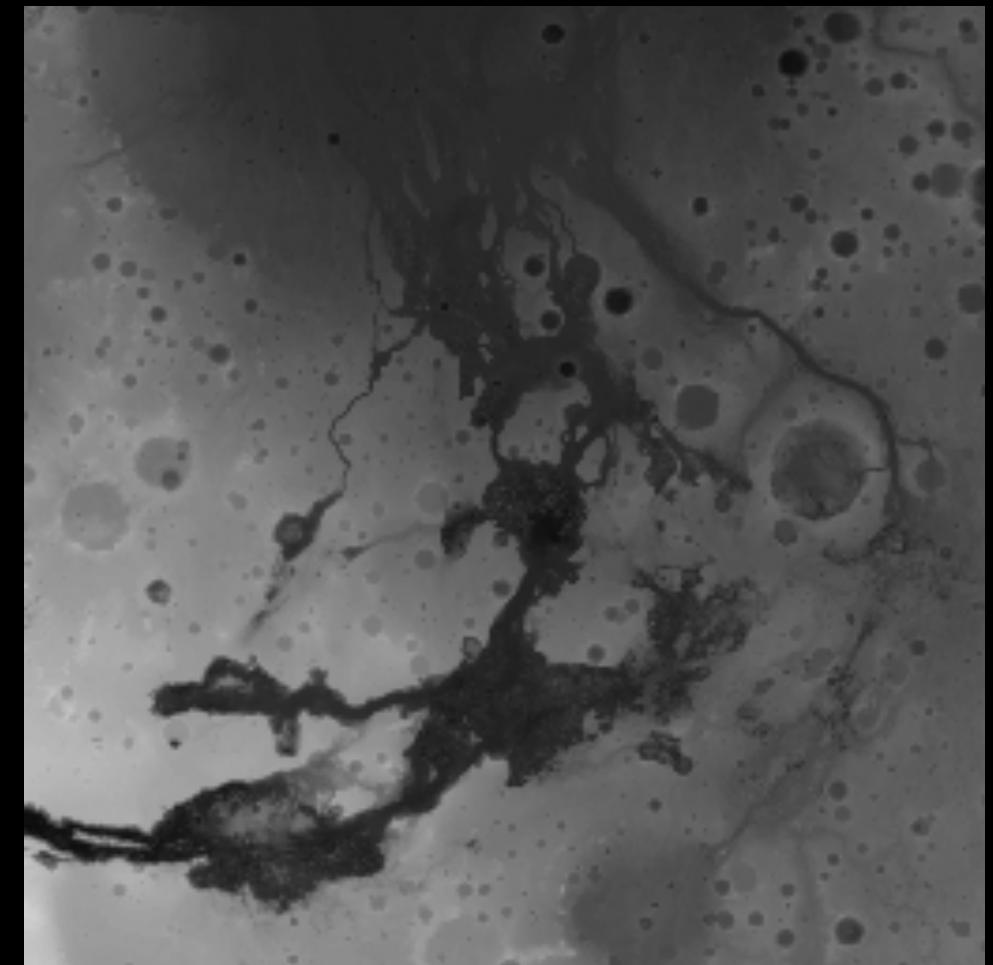
# 4 Rezultati testiranja

Mars MGS MOLA DEM, detalj doline Ares 128x128



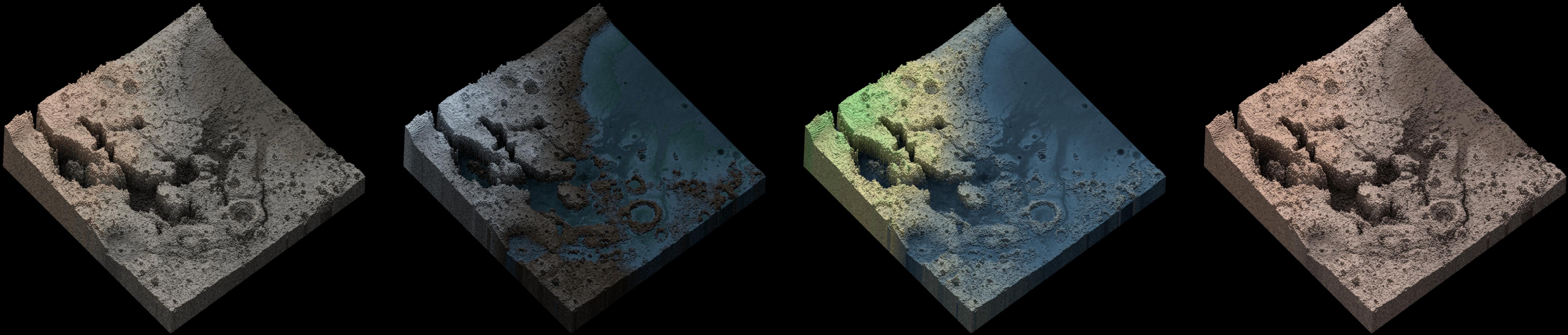
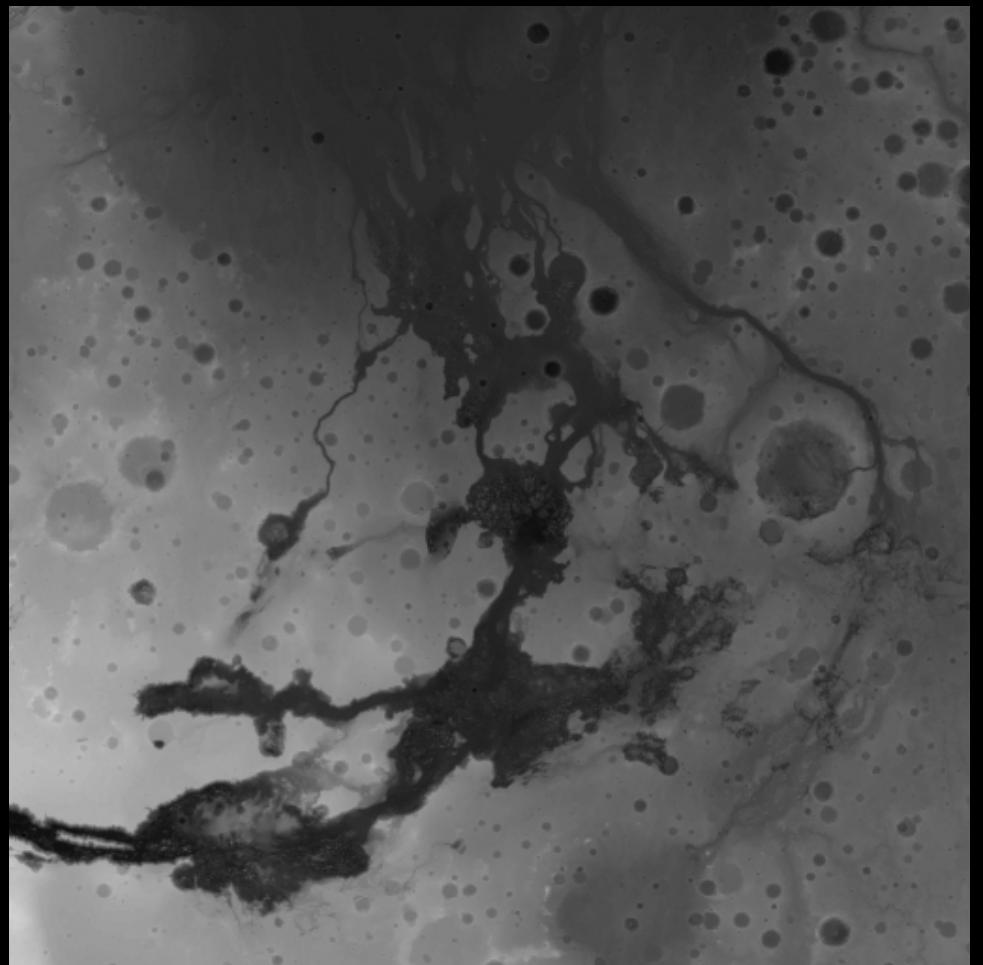
# 4 Rezultati testiranja

Mars MGS MOLA DEM, detalj doline Ares 256x256



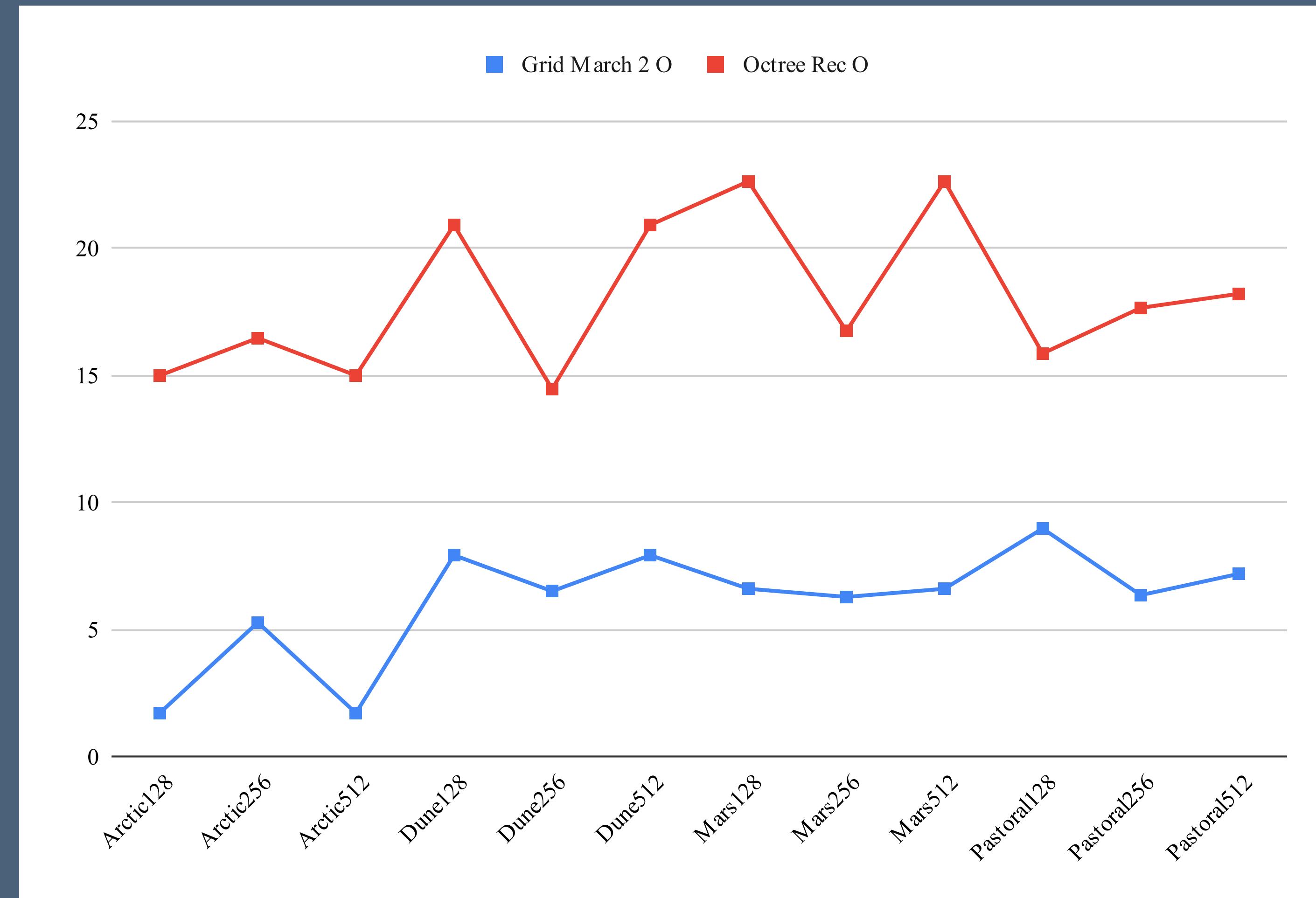
# 4 Rezultati testiranja

Mars MGS MOLA DEM, detalj doline Ares 512x512



# 4 Rezultati testiranja

Mars MGS MOLA DEM, rezultati



# 4 Rezultati testiranja

Oxia Palus – Nadmoć Grid march 2 algoritma

# 4 Rezultati testiranja

Oxia Palus – Nadmoć Grid march 2 algoritma

# 4 Rezultati testiranja

Oxia Palus – Nadmoć Grid march 2 algoritma

# 4 Rezultati testiranja

Oxia Palus – Nadmoć Grid march 2 algoritma

# 5 Zaključak

- Rad se bavi osnovnim tehnikama *raytracing* renderovanja vokselnih tela. Razmatra se više metoda za detekciju preseka zraka sa vokselima: *brute force*, *direction array*, *grid march* i pristup zasnovan na oktalnim stablima.
- Svi algoritmi su implementirani, objašnjeni i testirani na vokselnim modelima različite veličine i složenosti. Kao najefikasniji pokazao se *grid march 2* algoritam kojim su zatim renderovani detalji površine Marsa visoke rezolucije veličine skoro 80 miliona voksela.
- Rad ima za cilj da doprinese nastavi i istraživanju u oblasti raytracing-a na nivou osnovnih i master studija. Numerički i vizuelni rezultati pokazuju važnost optimizacije u renderovanju vokselnih scena.
- Prostor za buduća istraživanja prepoznaće se u daljoj optimizaciji postojećih algoritama, kao i primeni indirektnog renderovanja.

