SVEUČILIŠTE U ZAGREBU

**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

DIPLOMSKI RAD br. **1920**

# Tehnike određivanja skrivenih linija i površina

**Mirko Fabris**

Zagreb, rujan 2012.

**Sadržaj**

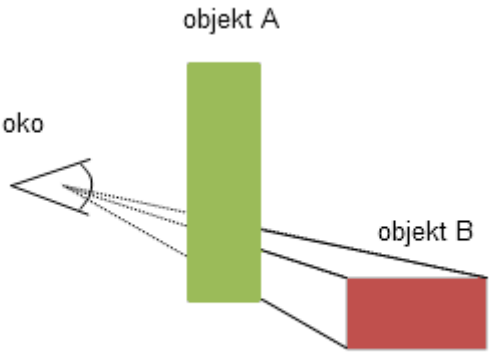
**Popis tablica**

**Popis slika**

# 1. Određivanje vidljivosti u virtualnoj sceni

## 1.1. Vidljivost

Vidljivost u računalnoj grafici je u osnovi simuliranje ljudskog vida. Ljudi vide one zrake svijetlosti koje se odbiju od nekog objekta i upadnu u oko, a pošto svijetlost ne ogiba oko predmeta[1], ljudi ne vide objekte koji su skriveni iza drugih objekata.




**Sl. 1** Objekt A je vidljiv, objekt B je skriven

Za čovjeka je, dakle, određivanje vidljivosti trivijalno, odnosno svijetlost to radi umjesto njega, međutim u računalnoj grafici to nikako nije trivijalan problem i uvijek je predstavljalo kompromis između točnosti rezultata i raspoloživih računalnih resursa pa je zbog toga razvijeno mnogo različitih tehnika i algoritama za određivanje vidljivosti. U ovome radu ćemo proučiti neke od tih tehnika i algoritma.

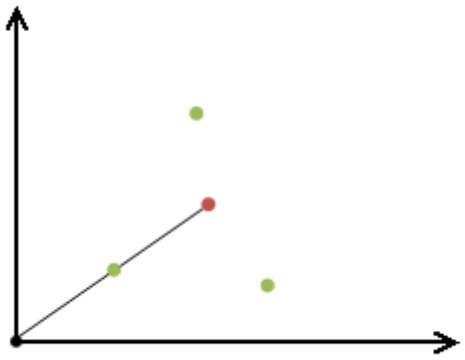
[1] Na razini ljudskog vida ogibanje svijetlosti je neprimjetno

## 1.2. Definicija problema

Promotrimo problem u jednoj dimenziji. Ako prođemo kroz sve točke i uspoređujemo im vrijednost možemo lako naći točku je najbliža točki gledišta i nju označiti kao vidljivu. Ako sortiramo točke po vrijednosti prilikom dodavanja u scenu onda je pretraga još jednostavnija i brža.

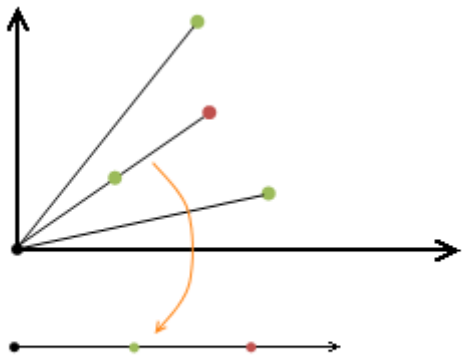

**Sl. 2** Vidljivost u jednoj dimenziji je trivijalna

Dodamo li još jednu dimenziju problem postaje puno kompleksniji. Ne možemo više pregledavati samo blizinu neke točke, jer neke točke mogu biti udaljenije od gledišta a biti vidljive.



**Sl. 3** Vidljive točke u 2d

Najjednostavnije rješenje ovog problema je za svaku točku povući zraku od gledišta i svesti problem na jednu dimenziju.



**Sl. 4** Pretvaranje 2d problema u 1d

Na ovaj način možemo riješiti vidljivost točaka, međutim u dvodimenzionalnom prostoru točke su vrlo rijetka pojava. U dvodimenzionalnom prostoru uglavnom se nalaze linije i poligoni (koji se mogu rastaviti na linije), a svaka linija se sastoji od beskonačno mnogo točaka tako da je ova metoda beskorisna za njih.

**Sl. 5** Kako odrediti vidljive linije

Vidimo da određivanje vidljivosti u dvodimenzionalnom prostoru nikako nije trivijalno i da su potrebne posebne tehnike kako bi se vidljivost izračunala. Pomicanjem točke gledišta iz ishodišta problem postaje još zahtjevniji za izračunati, jer sada za svaku liniju, i točku, trebamo prvo izračunati relativan pomak u odnosu na točku gledišta, što su dvije operacije oduzimanja $(x2 - x1,$ $y2 - y1)$ više što dodatno usporava izračun.

Proširenjem u trodimenzionalni prostor problem postaje još kompleksniji i većina metodi koje rade za dvodimenzionalno područje nije primjenjiva za trodimenzionalno područje.

## 1.3. Osnove vidljivosti

Za dvije točke kažemo da su međusobno vidljive ako između njih možemo povući liniju koja ne prolazi kroz neki objekt koji blokira vidljivost. Određivanje vidljivosti u nekoj virtualnoj sceni uvelike ovisi o načinu zapisa i strukturi te scene, ali se uglavnom svodi na rješavanje slijedećih problema:

- Vidljivost duž linije
- Vidljivost od točke
- Vidljivost od segmenta linije
- Vidljivost od poligona
- Vidljivost od regije

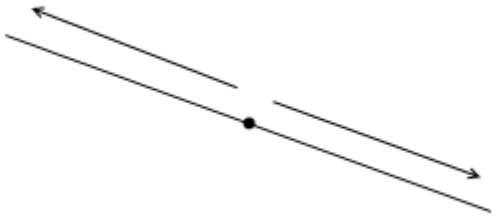
### 1.3.1. Vidljivost duž linije

Rješava se računanjem sjecišta linije s objektima u sceni. Koristi se kada se neki višedimenzionalni problem svede na samo jednu dimenziju. Algoritmi koji se koriste su:

- Računanje zrake kolizije (engl. Ray shooting)
    - povlačimo zraku od izvorišne točke u zadanom smjeru i računamo sjecište s prvim objektom u sceni
- Vidljivost između dvije točke
    - računamo zraku kolizije između dvije točke i ako zraka ne siječe nijedan objekt točke su međusobno vidljive
    - najčešće se koristi za računanje osvijetljenosti neke točke na način da se računa zraka kolizije od te točke prema izvoru svijetlosti

Primjeri korištenja:

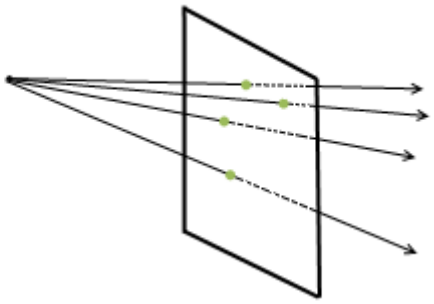
- Računanje osvjetljenja neke točke
- Vidljivost između dvije točke
- Maksimalni broj vidljivih segmenata – odredi se zraka i duljina segmenta te se računa koliki je maksimalni broj vidljivih segmenata u nekom smjeru



**Sl. 6** Vidljivost duž linije

### 1.3.2. Vidljivost od točke

Odabere se nekoliko zraka u 3D (2D) prostoru koje sve imaju istu točku kao izvorište te se računa njihova kolizija s ravninom (pravcem).



**Sl. 7** Vidljivost od točke

Ova metoda se najviše koristi prilikom određivanja vidljivosti u sceni pa će algoritmi i tehnike koje se koriste za ovo biti detaljno proučeni kasnije.

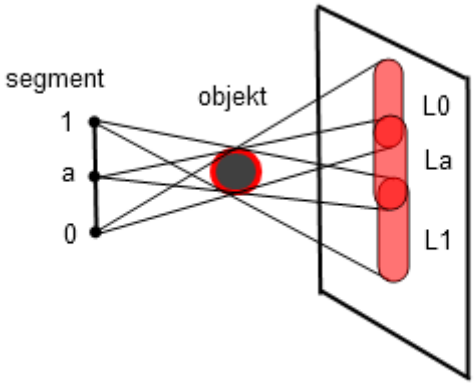
Primjeri korištenja:

- Računanje vidljivih površina
- Konstrukcija mape vidljivosti

### 1.3.3. Vidljivost od segmenta linije

Računamo vidljivost od jednog segmenta linije do neke ravnine tako da računamo zrake koje prolaze preko rubova nekog objekta za točku na početku segmenta, na kraju segmenta te proizvoljan broj točaka između.

Primjer korištenja je računanje sjena kod linearnih izvora svijetlosti.




**Sl. 8** Vidljivost od segmenta linije

### 1.3.4. Vidljivost od poligona

Računa se analogno kao i vidljivost od segmenta linije ali se zrake računaju iz vrhova poligona te proizvoljnog broja točaka iz unutrašnjosti ili s rubova poligona. Primjeri korištenja su za mekane sjene (engl. soft shadow).

### 1.3.5. Vidljivost od regije

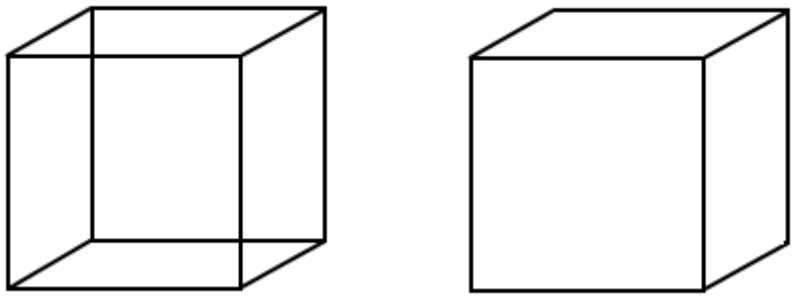
Računa se analogno kao i vidljivost od poligona, ali se računaju zrake koje se protežu od točaka koje pripadaju jednoj regiji. Primjeri korištenja su provjeravanje jesu li dvije regije prostora međusobno vidljive, parcijalno vidljive ili se uopće ne vide.

# 2. Uklanjanje skrivenih linija i ploha

Kada bi se za svaki objekt iscrtala svaka njegova linija, odnosno ploha to bi dovelo do nekoliko problema:

**Predugo vrijeme iscrtavanja**

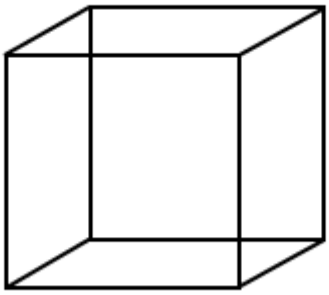
Za svaku liniju i svaku plohu koju iscrtamo na ekran potrebno je određeno vrijeme. Ako uklonimo one linije i plohe koje su skrivene povećat ćemo brzinu iscrtavanja nekog objekta, što znači da u istom vremenskom periodu možemo iscrtati veći broj objekata, odnosno kompleksniju scenu.



**Sl. 9** Cijeli objekt se sastoji 12 linija (lijevo), objekt bez skrivenih linija možemo iscrtati sa samo 9 linija (desno)

**Kriva slika**

Ako ne uklonimo skrivene linije i plohe prilikom iscrtavanja objekta, rezultat nam može izgledati krivo, ovisno o redoslijedu kojim smo iscrtali linije, odnosno plohe, na objektu.

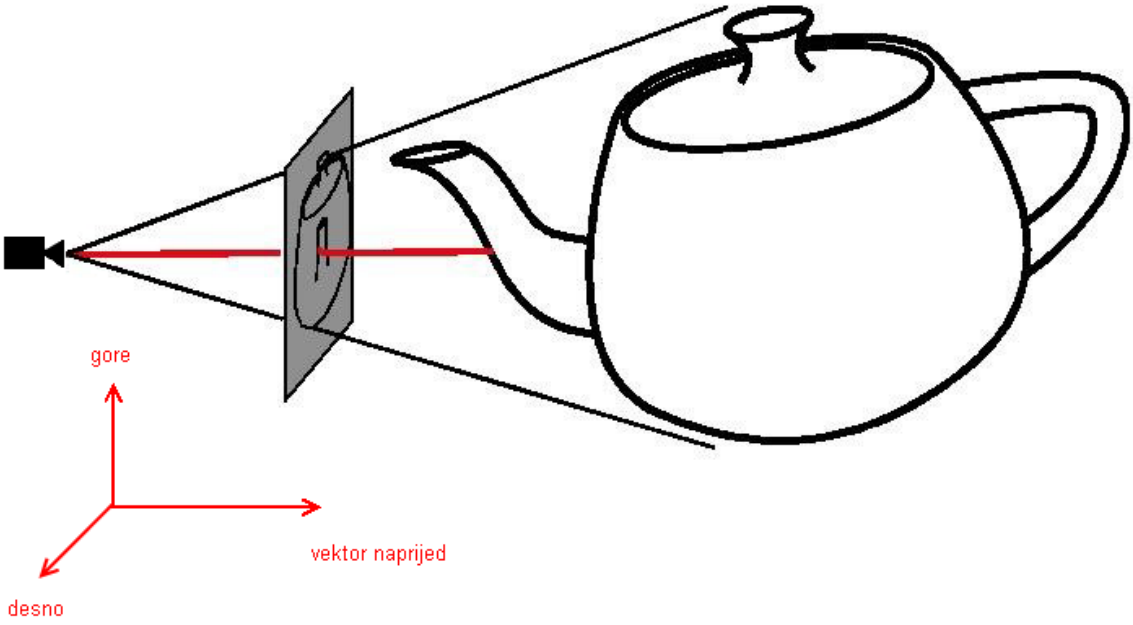


**Sl. 10** Bez uklonjenih nevidljivih linija na temelju ove slike nemoguće je odrediti koja stranica kocke je ispred a koja iza

Neki od algoritama i tehnika koje se koriste za određivanje vidljivosti se koriste i za uklanjanje skrivenih linija i ploha.

## 2.1. Prostor slike i prostor scene

Pošto ekrani prikazuju dvodimenzionalnu sliku, prikaz trodimenzionalne scene je potrebno napraviti preko virtualne kamere. Virtualna kamera se sastoji od točke gledanja i tri vektora koji određuju smjer kamere: naprijed, desno i gore. Na temelju tih podataka možemo napraviti transformaciju koja svaku točku, odnosno poligon, transformira iz prostora scene u prostor kamere. Kada imamo tako transformiran prostor možemo iscrtati sliku scene tako da napravimo projekciju svih objekata na ravninu koja je okomita na smjer gledanja kamere.




**Sl. 11** Objekt i njegova projekcija na ravninu kamere

Algoritmi se mogu podijeliti u dvije skupine, oni koji rade u prostoru slike (projekcije objekata na ravninu) i oni koji rade u prostoru scene.

Algoritmi koji rade u prostoru slike:

- Z spremnik
- Algoritam iscrtavanja slikara
- Warnockov algoritam

Algoritmi koji rade u prostoru scene:

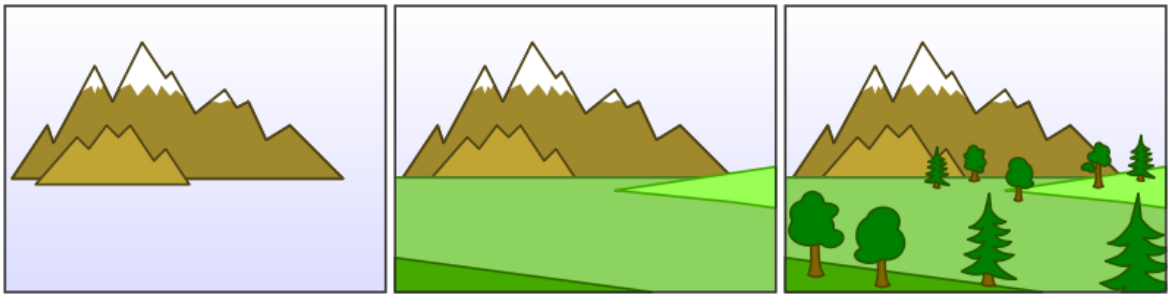
- Binarna podjela prostora
- Uklanjanje poligona
- Metoda ćelija i portala

## 2.2. Z spremnik

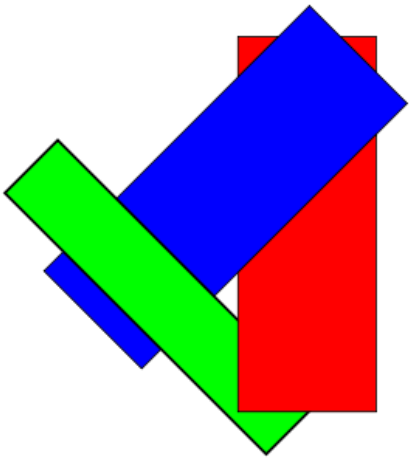
Z spremnik (engl. z-buffer) je algoritam koji se koristi prilikom iscrtavanja 3D scene na ekran. Služi tome da se sakriju skrivene površine na način da se zapamti dubina (udaljenost objekta od kamere) za svaki piksel koji se iscrtao pa kada se računa dubina za piksel koji već ima neku vrijednost u z spremniku, ako je njegova dubina veća taj piksel ignoriramo. U osnovi svodimo problem na jednu dimenziju. U današnje vrijeme ovo je standardan način uklanjanja skrivenih površina i skoro sve grafičke kartice imaju već implementiran ovaj algoritam sklopovski.

## 2.3. Algoritam iscrtavanja slikara

Algoritam iscrtavanja slikara (engl. Painter algorithm) sortira sve objekte u sceni od najdaljeg prema najbližem u odnosu na kameru i po tom redoslijedu ih iscrta. Udaljenost objekta od kamere se računa pomoću geometrijske sredine tog objekta, tako da algoritam ne radi za sve moguće kombinacije objekata. Problem također može nastati kad se neki poligoni preklapaju kao u primjeru na slici Sl. 13.



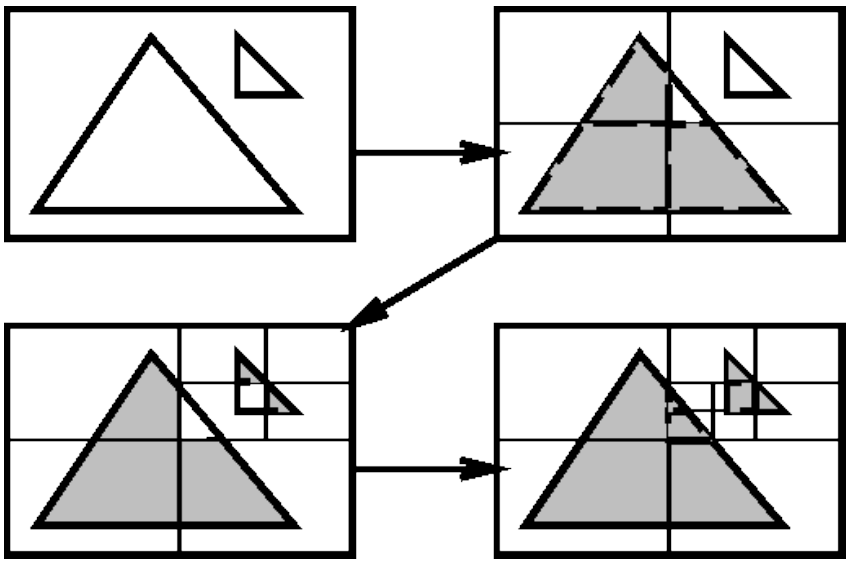
**Sl. 12** Prvo se iscrtaju udaljene planine pa ravnica i na kraju najbliži objekti - drva.



**Sl. 13** Problem kod poligona koji se preklapaju

## 2.4. Warnockov algoritam

Warnockov algoritam radi na principu da rekurzivno dijeli scenu na manje segmente za koje je izračun vidljivosti trivijalan. U prvom koraku provjeravamo cijelu scenu i ako ona sadrži kompleksne objekte dijelimo ju na četiri segmenta. Nakon toga svaki od segmenata posebno provjeravamo i ako nađemo koji segment koji ima kompleksne objekte u sebi njega opet dijelimo i tako redom dok ne dobijemo samo segmente koje možemo lako izračunati. Najmanja jedinica do koje dijelimo je jedan piksel.

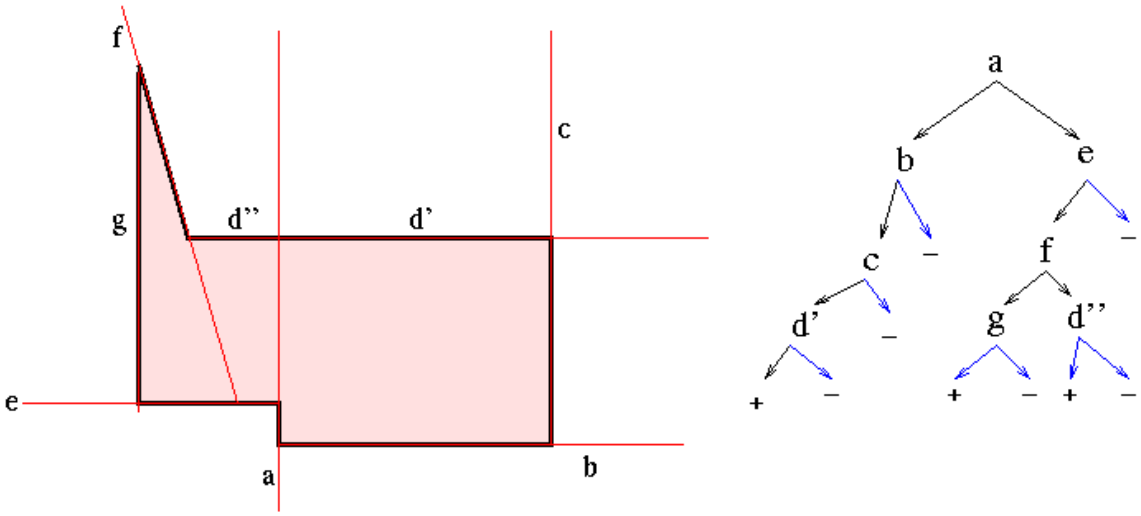


**Sl. 14** Početnu scenu rekurzivno dijelimo dok ne dobijemo segmente koji su trivijalni za izračunati

## 2.5. Binarna podjela prostora

Binarna podjela prostora (engl. Binary space partitioning, BSP) je metoda kojom rekurzivno dijelimo prostor na dvije cjeline za svaku plohu svakog objekta u sceni. Na taj način dobijemo strukturu koja se zove BSP stablo. Pomoću BSP stabla možemo brzo pretraživati prostor i naći u kojem segmentu prostora se nalazimo i na temelju toga odrediti udaljenost objekata od kamere.

Ovaj način podjele prostora rješava probleme algoritma iscrtavanja slikara ali ima i svojih nedostataka. Nedostatak ove metode je što samo računanje BSP stabla traje dosta dugo pa se treba izračunati unaprijed ako se želi koristiti za iscrtavanje u realnom vremenu. To znači da je ova metoda dobra ako se koristi za statične scene, međutim svako dodavanje novih objekata ili micanje dinamičkih objekata po sceni bi rezultiralo novim računanjem BSP stabla.

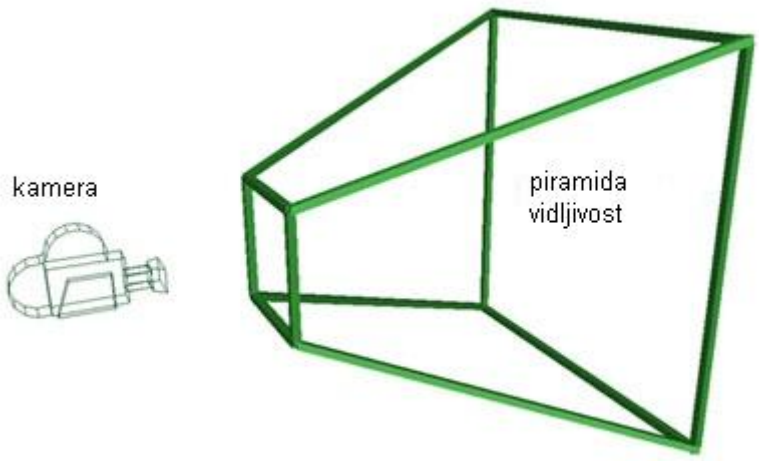



**Sl. 15** Generiranje BSP stabla, linija *a* dijeli liniju *d* na dvije linije *d'* i *d''*

## 2.6. Uklanjanje poligona

Na temelju podataka o kameri možemo izračunati piramidu vidljivosti (engl. view frustum) koja određuje vidljivi prostor scene. Sve objekte koji nisu u toj piramidi možemo ignorirati prilikom iscrtavanja, a objektima koji su djelomično u piramidi odsiječemo (engl. clipping) onaj dio koji nije u piramidi i njega ignoriramo.

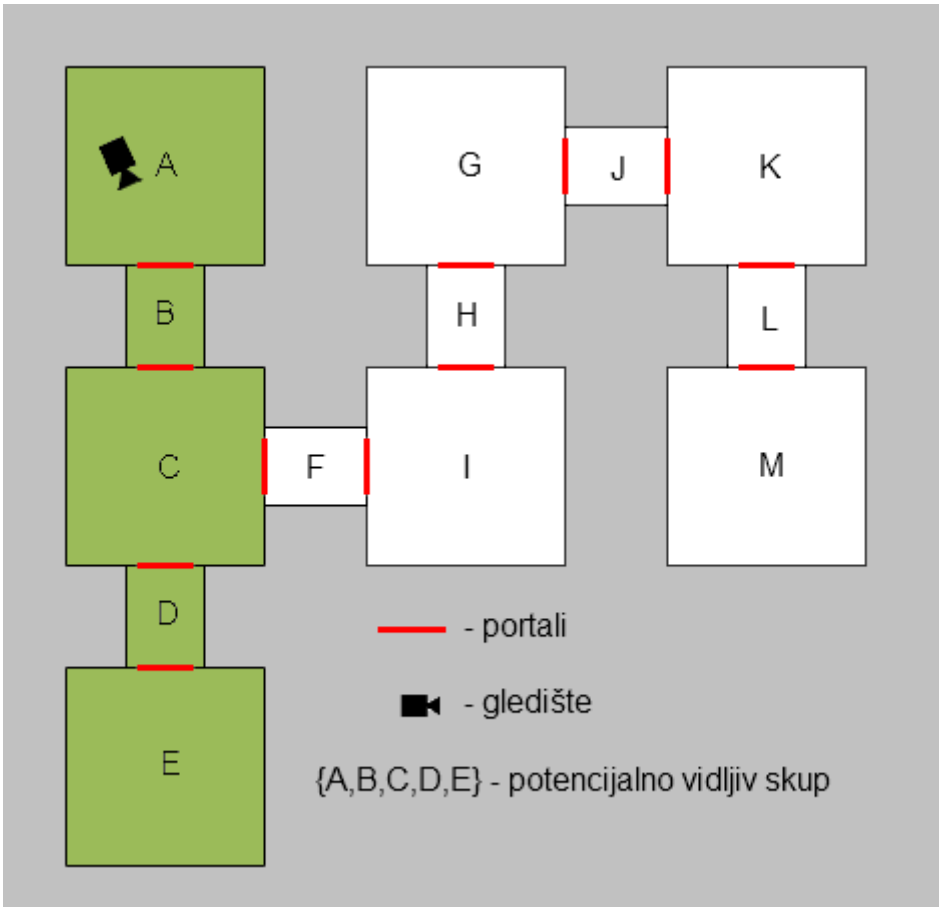
Ako svaki poligon u sceni ima normalu koja određuje prednju stranu poligona, a stražnja strana je prozirna, izračunamo kut kojeg normala poligona zatvara sa vektorom prema točki gledanja. Svaki poligon koji ima taj kut veći od 90° možemo ignorirati. Ako želimo napraviti da su obje strane poligona vidljive definiramo dvije međusobno suprotne normale i obje ih pridružimo istom poligonu.




**Sl. 16** Piramida vidljivosti

## 2.7. Metoda ćelija i portala

Metoda ćelija i portala se zasniva na ideji da scenu podijelimo na ćelije koje su međusobno povezane portalima i na taj način odredimo koje dijelove scene potencijalno vidimo, a koje sigurno ne vidimo.

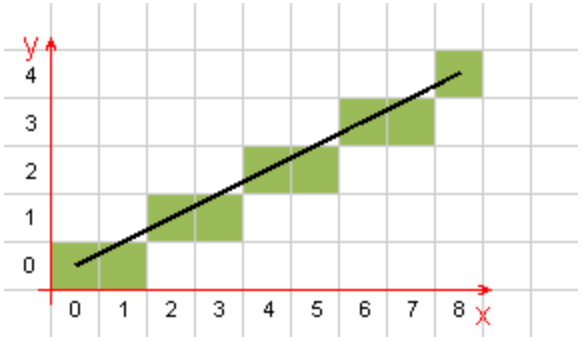



**Sl. 17** Potencijalno vidljiv skup za gledište u ćeliji A

Kada imamo tako podijeljenu scenu možemo za svaku ćeliju izračunati potencijalno vidljiv skup koji određuje ćelije koje potencijalno možemo vidjeti iz zadane ćelije. Sve ostale ćelije možemo ignorirati. Ova metoda je idealna za korištenje prilikom određivanja vidljivosti u interijerima zgrada jer su one već podijeljene na taj način.

# 3. Bresenhamov algoritam

Bresenhamov algoritam za iscrtavanje linija je jedan od najstarijih algoritama u području računalne grafike. Razvio ga je Jack Elton Bresenham 1962. godine i po njemu nosi ime. U svojoj osnovnoj verziji to je algoritam koji određuje koje točke se trebaju označiti u 2-dimenzionalnom prostoru ako se želi aproksimirati ravna linija između dvije točke u tom prostoru. Njegova osnovna prednost je to što koristi samo operacije bitovnog posmaka te zbrajanja i oduzimanja nad cjelobrojnim podatcima, a to su sve procesorski jeftine operacije i jednostavne su za implementirati u sklopovima. U ovom radu će biti razmatrana verzija Bresenhamovog algoritma koji koristi i aritmetiku s pomičnim zarezom (engl. *float*) odnosno taj tip podataka jer će biti oni potrebni za računanje nekih stvari koje nisu potrebne ako se algoritam koristi samo za iscrtavanje linija.

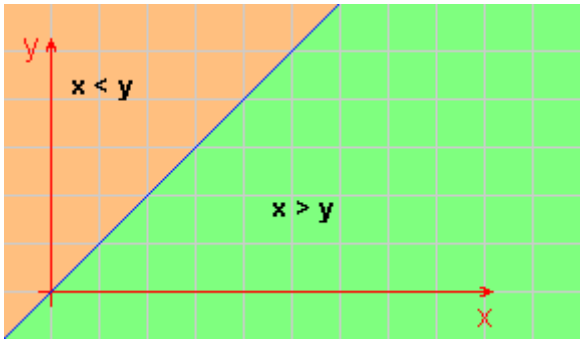



**Sl. 18** Primjer aproksimacije linije Bresenhamovim algoritmom od točke (0,0) do točke (8,4)

## 3.1. Osnove algoritma

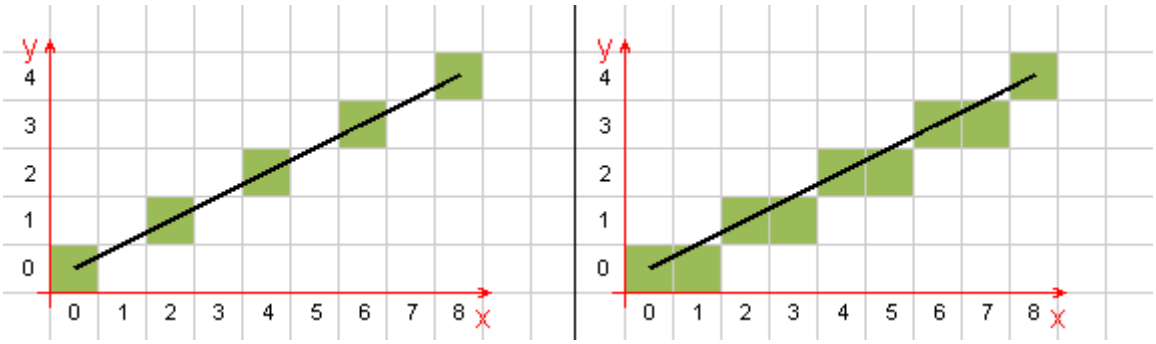
Bresenhamov algoritam služi za iscrtavanje linije u 2-dimenzionalnom prostoru, što znači da su nam potrebne točke $T1 = (x1, y1)$ i $T2 = (x2, y2)$. Radi jednostavnosti osnovni način rada algoritma će biti prikazan u prvom kvadrantu koordinatnog sustava, pa će kasnije biti proširen na cijelo područje x-y ravnine. To znači da će nam, zasad, prva točka uvijek biti $(0,0)$ tako da ćemo imati samo jednu točku $T = (x, y)$ čije će koordinatne vrijednosti uvijek biti veće ili jednake nuli.

Osnovna ideja je da krenemo od $(0,0)$ i izaberemo glavnu koordinatnu os po kojoj ćemo se micati za jedan svaki korak, a za drugu koordinatu računamo trebamo li se pomaknuti po njoj ili ne. Prva stvar koju moramo naći je nagib linije prema koordinatnim osima, odnosno koja koordinata je veća $x$ ili $y$. U slučaju da su obje koordinate isto velike svejedno je koju os izaberemo.




**Sl. 19** Podjela prostora u ovisnosti o nagibu prema koordinatnim osima

Nagib nam je potreban jer ako krivo izaberemo dobit ćemo isprekidanu liniju i neke točke neće biti označene, kao što je prikazano na Slici 20.




**Sl. 20** Krivo odabrana glavna koordinata (lijevo), ispravno odabrana koordinata (desno)

Nakon toga računamo kut koji linija zatvara s pozitivnom stranom osi po kojoj smo se odlučili kretati svaki korak. Na temelju tog kuta, odnosno tangensa tog kuta, ćemo znati trebamo li se u nekom koraku pomaknuti po drugoj koordinatnoj osi ili ne. Radi bržeg izračuna se koristi svojstvo da je tangens kuta približno jednak vrijednosti kuta, ako je taj kut jako mali, što je u našem slučaju uvijek istina

tako da ne trebamo računati tangens tog kuta već je dovoljno izračunati samo $y/x$, ako se mičemo po x osi, odnosno $x/y$ ako se mičemo po y osi.

Pomak po drugoj koordinati se računa tako da se u jednoj varijabli pamti kut koji se u svakoj iteraciji uveća opet za vrijednost kuta, i ako ta vrijednost postane pozitivna onda se ta varijabla umanji za jedan a druga koordinata se poveća za jedan.

Kada imamo izračunatu glavnu os i nagib linije prema toj osi možemo započeti sa petljom u kojoj ćemo iterirati po toj osi od početne točke do krajnje točke i u svakoj iteraciji petlje računati trebamo li se pomaknuti po drugoj koordinati ili ne.

Pseudokod algoritma koji vraća listu polja koje treba označiti za razmatrani slučaj, i to samo za slučaj kada je x > y:

```
funkcija linija_od_nule_x_veci_y( x1, y1 )
    float y_x := y1 / x1
    float D := y_x – 0.5
    y := 0
    za x od 0 do x1:
        ako je D>0:
            y := y + 1
            D:= D - 1
        D := D + y_x
        lista_točaka.dodaj( (x,y) )
    lista_točaka.dodaj( (x1,y1) )
    vrati lista_točaka
```

**Tablica 1** Vrijednosti varijabli na kraju prolaska kroz petlju za primjer t=(8,4)

| x | y | D |
|---|---|---|
| 0 | 0 | 0.5 |
| 1 | 0 | 0 |
| 2 | 1 | 0.5 |
| 3 | 1 | 0 |
| 4 | 2 | 0.5 |
| 5 | 2 | 0 |
| 6 | 3 | 0.5 |
| 7 | 3 | 0 |

Funkcija *linija_od_nule_x_veci_y()* je najjednostavniji i najspecifičniji primjer, ali prikazuje osnovnu ideju algoritma. Za praktičnu primjenu trebamo proširiti ovaj osnovni algoritam na cijelu x-y ravninu.

## 3.2. Proširenje na cijelu x-y ravninu

Gore navedena funkcija *linija_od_nule_x_veci_y()* nam je u praksi beskorisna i trebamo ju proširiti da pokriva cijelu x-y ravninu kako bi dobili algoritam koji će imati praktičnu primjernu.

Prvo proširenje treba napravit tako da nam funkcija pokriva i slučajeve kada je $x > y$. To ćemo napraviti tako da prvo ispitamo je li $x > y$ i ako da dodamo analogan dio koda samo sa drugim varijablama na slijedeći način:

```
funkcija linija_od_nule( x1, y1 ):
    ako je x1 > y1:
        float y_x := y1 / x1
        float D := y_x – 0.5
        y := 0
        za x od 0 do x1:
                ako je D>0:
                        y := y + 1
                        D:= D - 1
                D := D + y_x
            lista_točaka.dodaj( (x,y) )
    inače:
        float x_y := x1 / y1
        float D := x_y – 0.5
        x := 0
        za y od 0 do y1:
                ako je D>0:
                        x := x + 1
                        D:= D - 1
                D := D + x_y
            lista_točaka.dodaj( (x,y) )


    lista_točaka.dodaj( (x1,y1) )
    vrati lista_točaka
```

Sada imamo pokriveni cijeli prvi kvadrant, međutim to nam još nije dosta nego trebamo proširiti algoritam da može obrađivati slučajeve iz sva 4 kvadranta i nakon toga da početna točka ne mora uvijek biti $(0,0)$. Pošto smo vidjeli na gornjem

primjeru da su proširenja za svaki korak prilično jednostavna navest ćemo još samo finalnu verziju algoritma. Potrebno je napomenuti da ova funkcija koristi dvije funkcije koje nisu navede ovdje, *abs( num )* koja vraća apsolutnu vrijednost od *num*, i *sgn( num )* koja vraća jedan ako je *num* pozitivan ili nula, ili minus jedan ako je *num* negativan.

```
funkcija linija( x1, y1, x2, y2 ):
        int absx0 := abs(x2 – x1)
        int absy0 := abs(y2 – y1)
int sgnx0 := sgn(x2 – x1)
int sgny0 := sgn(y2 – y1)
x := x1
y := y1
lista_točaka.dodaj( (x,y) )
        ako je absx0 > absy0:
float y_x := absy0 / absx0
float D := y_x – 0.5
za i od 0 do absx0:
                ako je D>0:
                        ako je sgny0 = -1:
                                y := y – 1
inače:
                                y := y + 1
                        D:= D - 1
                ako je sgnx0 = 1:
                        x := x + 1
                inače:
                        x := x – 1
                D := D + y_x
lista_točaka.dodaj( (x,y) )

        inače:
float x_y := absx0 / absy0
float D := x_y – 0.5
za i od 0 do absy0:
                ako je D>0:
                        ako je sgnx0 = -1:
                                x := x – 1
inače:
                                x := x + 1
                        D:= D - 1
                ako je sgny0 = 1:
                        y := y + 1
                inače:
                        y := y – 1
                D := D + x_y
lista_točaka.dodaj( (x,y) )
vrati lista_točaka
```

Iz ove finalne funkcije se vidi da je princip isti kao i u iscrtavanju linije u prvom kvadrantu smo što je potrebno provjeriti je li $x0 > x1$ i ovisno o tome korigirati smjer u kojem ćemo iterirati po x osi, i analogno za y.

# 4. Upotreba Bresenhamovog algoritma za izračun vidljivosti

Bresenhamov algoritam u svojoj osnovi služi za aproksimaciju kontinuirane linije diskretnim prikazom i samim time je pogodan za računanje vidljivosti u nekom diskretnom prostoru. Iako je pogodan za računanje vidljivosti ima nekih nedostataka tako da ga treba modificirati kako bio bolje služio za taj zadatak. U ovom poglavlju će se izložiti njegovi nedostaci i modifikacije koje rješavaju te probleme.

## 4.1. Osnovni pojmovi

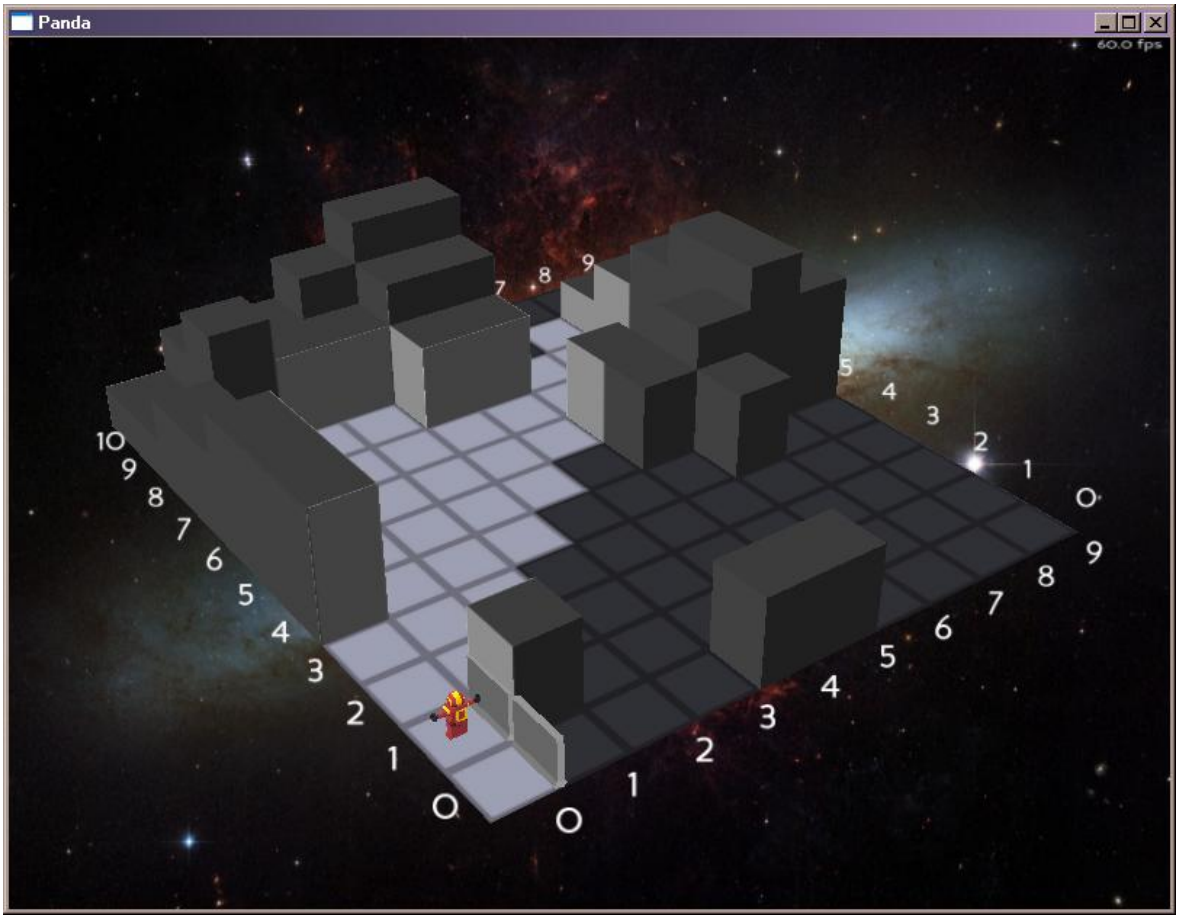
Da bi uopće mogli isprobavati algoritam u smislu vidljivosti potreban nam je neki virutalni prostor na kojem ćemo raditi ispitivanja. Prostor će biti trodimenzionalni svijet koji je podijeljen na diskretne dijelove koji su svi iste veličine. Svaki taj dio neka se zove **kocka**. Svaka kocka može biti prazna, u tom slučaju se kroz nju može vidjeti, ili može biti popunjena, u tom slučaju je vidljivost blokirana. Između svake dvije susjedne kocke postoji prostor u kojem može biti **zid** u tom slučaju on može blokirati vidljivost, ili ne mora, ovisno o tipu zida.

Pošto radimo aproksimaciju kontinuiranog prostora diskretnim svijetom vidljivost definiramo ako iz centra početne kocke možemo povući liniju koja ne prolazi kroz nikakve prepreke do centra ciljne kocke.

**Tablica 2** Vrste zidova i vidljivost kroz njih

| Vrsta zida | Blokira vidljivost |
|---|---|
| Obični zid | Da |
| Polu-zid | Ne |
| Otvorena vrata | Ne |
| Zatvorena vrata | Da |
| Lasersko polje | Ne |

Pošto tražimo vidljivost u virtualnom svijetu, potreban nam je i barem jedan agent za kojeg ćemo računati vidljivost, njega ćemo nazvati **jedinica**. Program će prikazivati dijelove svijeta koji su vidljivi kao osvijetljene dok će dijelovi koji nisu vidljivi biti zatamnjeni.

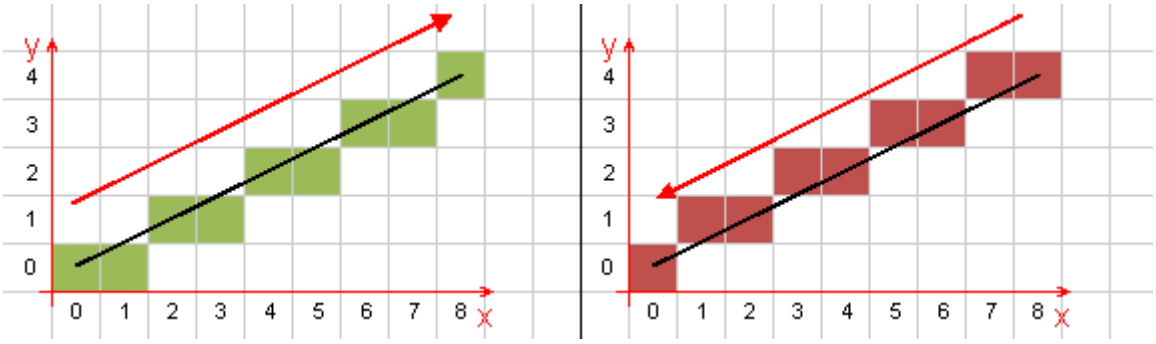



**Sl. 21** Primjer testnog programa sa jedinicom na kocki (0,1)

## 4.2. Nedostaci Bresenhamovog algoritma

### 4.2.1. Redoslijed odabira točaka

Prvi nedostatak Bresenhamovog algoritama je da, nekada, neće dati isti rezultat za isti par točaka ako se zadanim točkama promijeni redoslijed.




**Sl. 22** Različiti rezultati u ovisnosti o smjeru pretraživanja

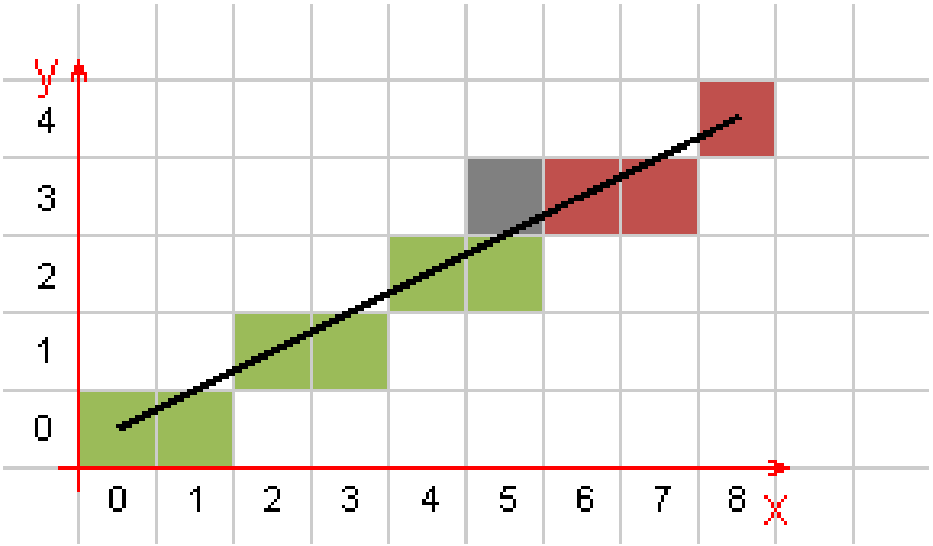
**Tablica 3** Tablica sa vrijednostima x i y za primjer sa slike **Sl. 22**

| Redoslijed: (0,0), (8,4) | | Redoslijed: (8,4), (0,0) (rezultat invertiran) | |
|---|---|---|---|
| x | y | x | y |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 2 | 1 | 2 | 1 |
| 3 | 1 | 3 | 2 |
| 4 | 2 | 4 | 2 |
| 5 | 2 | 5 | 3 |
| 6 | 3 | 6 | 3 |
| 7 | 3 | 7 | 4 |
| 8 | 4 | 8 | 4 |

To nije problem ako se algoritam koristi za svoju osnovnu namjenu, jer obadvije verzije daju grafički ispravan rezultat, međutim ako ga upotrebljavamo u svrhu određivanja vidljivosti onda se dobiju krivi rezultati. Uzmimo slučaj sa slike Sl. 22 i pretpostavimo da se na kocki (3,2) nalazi prepreka koja blokira vidljivost a ostala polja su sva prazna. U tom bi slučaju s kocke (0,0) vidjeli kocku (8,4) jer dobivena linija ne prolazi kroz (3,2), međutim s kocke (8,4) ne bi vidjeli kocku (0,0) jer dobivena linija u ovom slučaju prolazi kroz polje (3,2).

### 4.2.2. Ignoriranje parcijalnih prolaza

Drugi nedostatak je taj da Bresenhamov algoritam ignorira neke kocke kroz koje linija prolazi samo parcijalno. Isto kao i kod preciznosti to nije problem u grafičkom smislu ali kada se algoritam koristi za određivanje vidljivosti onda tu nastaje problem jer dobivamo rezultate koji nisu realni, odnosno algoritam će odrediti da je neka kocka vidljiva iako ona to nije.

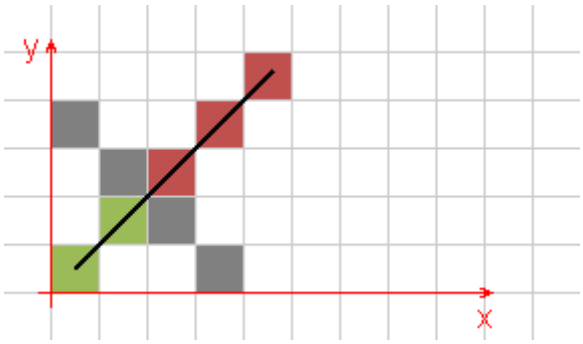



**Sl. 23** Linija vidljivosti parcijalno prolazi kroz (5,3)

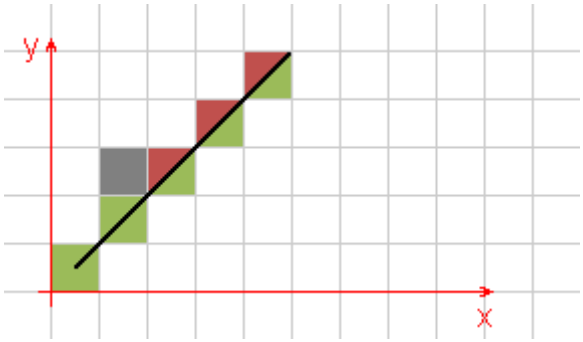
Ako promotrimo primjer na slici Sl. 23 primijetit ćemo da linija vidljivosti prolazi parcijalno kroz kocku (5,3), koja blokira vidljivost, što bi trebalo značiti da se s kocke (0,0) ne vidi kocka (8,4) međutim osnovni algoritam će ignorirati kocku (5,3) te dati krivi rezultat.

### 4.2.3. Slučaj kada je linija pod 45°

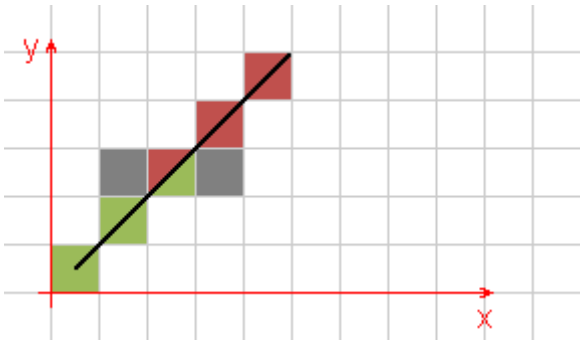
Posebno treba promotriti i slučaj kada je linija vidljivosti pod kutom od 45° na osi koordinatnog sustava. U tom slučaju osnovni algoritam u svakoj iteraciji petlje povećava i x i y koordinatu te ignorira sve okolne kocke, zbog čega dolazi do krivih rezultata u slijedećim slučajevima.




**Sl. 24** Vidljivost je blokirana sa lijeve i sa desne strane




**Sl. 25** Crveno-zelene kocke su parcijalno vidljive jer im je vidljivost blokirana s lijeve strane




**Sl. 26** Crveno zelena kocka je parcijalno vidljiva, međutim nakon prepreke s desne strane više ne vidimo po zadanoj liniji

## 4.3. Rješenja nedostataka

### 4.3.1. Redoslijed odabira točaka

Nepreciznost rješavamo tako da prije ulaska u petlju namjestimo redoslijed koordinata da bude uvijek isti bez obzira na njihov zadani redoslijed, na primjer da će se uvijek ići od manje prema većoj vrijednosti. Vidjeli smo da će rezultat uvijek biti ispravan bez obzira na redoslijed, ali ako odredimo uvijek isti smjer dobivat ćemo konzistentne rezultate.

Ovo rješenje se implementira tako da prije početka petlje prvo provjerimo koja će nam biti glavna os, i ako je vrijednost te koordinate veća u drugoj točki nego u prvoj onda interno zamijenimo vrijednosti prve i druge točke. Još je potrebno da u jednoj varijabli zapamtimo da smo napravili promjenu tako da kada vraćamo listu označenih točaka da znamo da ju moramo preokrenuti.

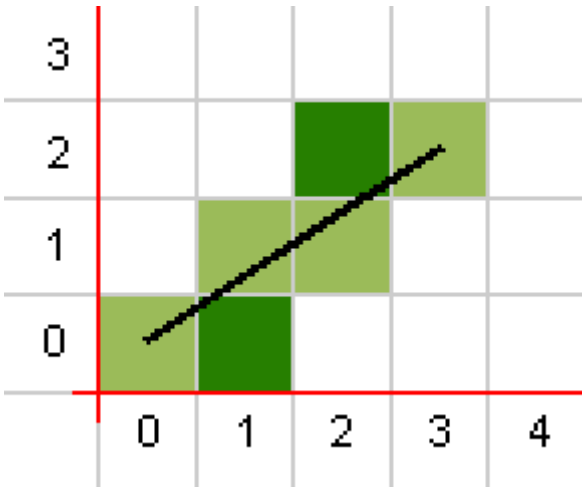
```
ako je absx0 > absy0:
        ako je x2 > x1:
                zamijeni t1 i t2
                reverse := True
inače:
        ako je y2 > y1:
                zamijeni t1 i t2
                reverse := True
… glavna petlja …
ako je reverse:
        preokreni listu_točaka
vrati listu_točaka
```

### 4.3.2. Ignoriranje parcijalnih prolaza

Ignoriranje parcijalnih prolaza rješavamo tako da svaki put kada se moramo pomaknuti po drugoj koordinati provjeravamo kroz koju kocku parcijalno prolazi linija vidljivosti i provjeravamo da li ta kocka blokira vidljivost.

To računanje se obavlja tako da se prije ulaska u glavnu petlju, kada se računa nagib linije (y/x ako nam je glavna os x, odnosno x/y ako nam je glavna os y), izračuna i polovica tog nagiba. Kada se dođe do dijela u glavnoj petlji u kojem se provjerava je li pogreška D veća od 0, što znači da se moramo pomaknuti po

drugoj koordinati, usporedimo D s polovicom izračunatog nagiba i ako je D veći od te vrijednosti onda znači da prelazimo preko "gornje" kocke, a ako je manji prelazimo preko "donje" kocke.



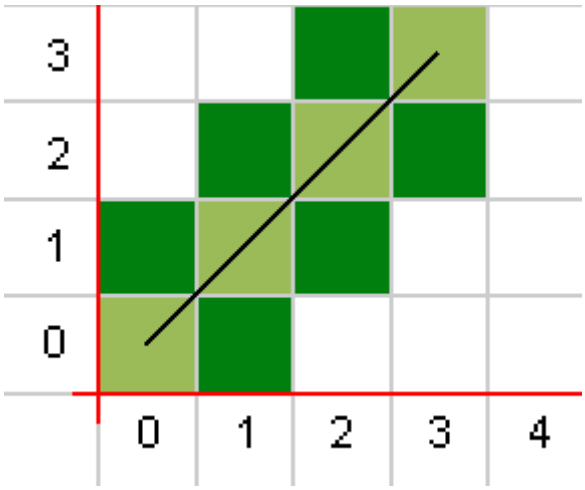

**Sl. 27** Prvo parcijalno prelazimo preko donje kocke (1,0), a kasnije preko gornje (2,2)

**Tablica 4** Vrijednost varijable D u svakom koraku i njena usporedba sa polovicom y/x za slučaj sa slike **Sl. 27**

| y/x = 0.666<br>(y/x)/2 = 0.333 | | | |
|---|---|---|---|
| **x** | **y** | **D** | **Odnos D sa (y/x)/2** |
| 0 | 0 | 0.1666 | Manji |
| 1 | 1 | -0.1666 | - |
| 2 | 1 | 0.5 | Veći |

### 4.3.3. Slučaj kuta od 45°

Za slučaj kada je linija vidljivosti pod kutom od 45° trebamo posebno provjeriti sve kocke koje dotiču svaku kocku, osim početne i krajnje, kroz koju linija prolazi.




**Sl. 28** Tamno zeleno su označene sve kocke koje moramo provjeriti

Razmotrimo primjer kada su sve kocke na liniji prazne. U svakoj iteraciji glavne petlje provjerimo lijevu stranu pa desnu stranu i zapamtimo u varijable ako je koja strana blokirana, zato što u ovom slučaju mogu biti tri različita rezultata:

- **Potpuna vidljivost** – kocke s obje strane su prazne
- **Parcijalna vidljivost** – sve kocke s jedne strane su prazne, ali na drugoj strani postoji jedna ili više kocaka koje blokiraju vidljivost, Sl. 25
- **Blokirana vidljivost** – postoji barem jedna kocka sa svake strane koja blokira vidljivost, Sl. 24 i Sl. 26

## 4.3.4. Pseudokod

### Pseudokod konačne verzije algoritma

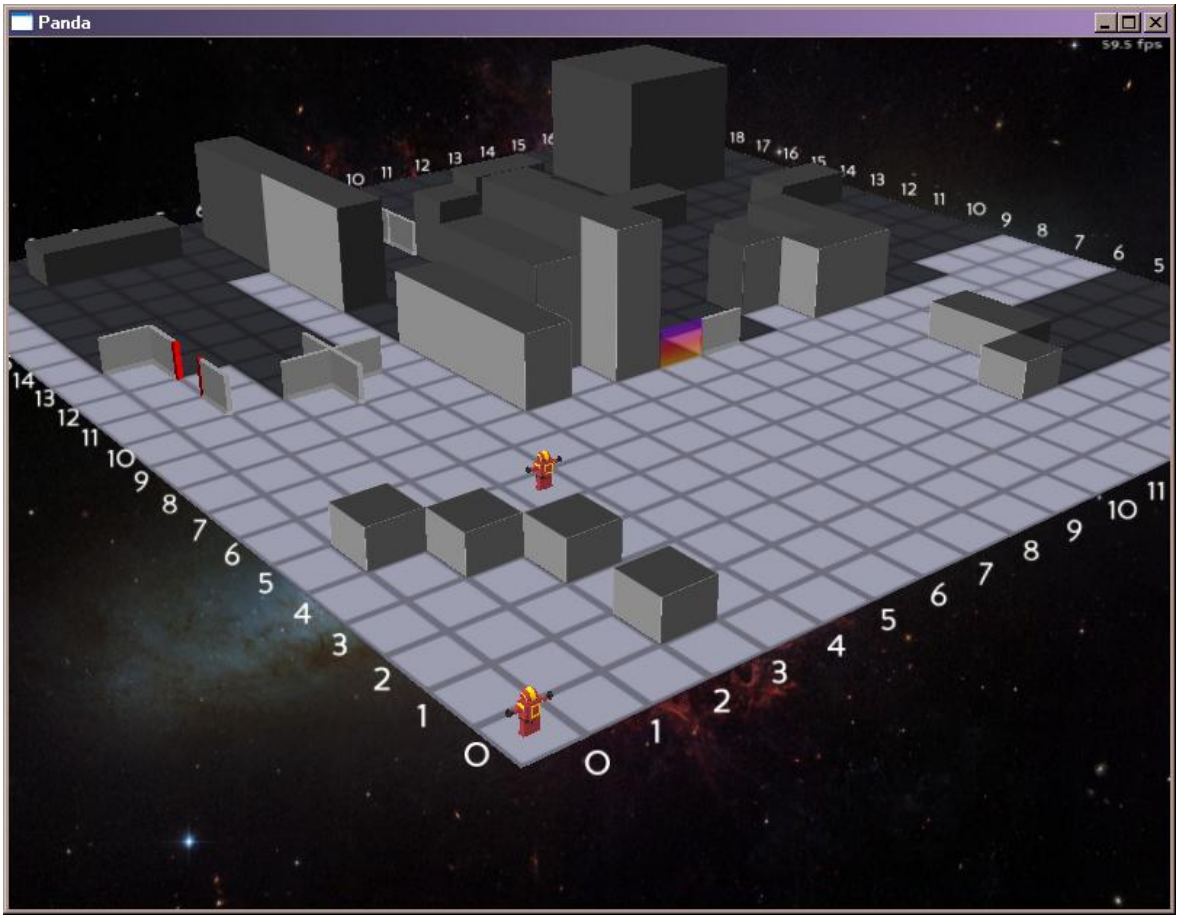
```
Funkcija provjeri( x1, y1, x2, y2 ):
absx0 := abs( x2 – x1 )
absy0 := abs( y2 – y1 )
lista_točaka.dodaj( (x1,y1) )
reverse := False
ako je absx0 > absy0:
        ako je x2 < x1:
                zamijeni x1 i x2 te y1 i y2
                reverse := True
inače:
        ako je y2 < y1:
                zamijeni x1 i x2 te y1 i y2
                reverse := True
x := x1
y := y1
ako je absx0 > absy0:
        sgny0 := sig( y2 – y1 )
        y_x := absy0 / absx0
        y_x_2 := y_x / 2
        D := y_x – 0.5
za i od 0 do absx0:
        ako je D > 0:
                provjeri(x+1, y+sgny0)
                ako je D < y_x_2:
                        provjeri(x+1, y)
                        lista_točaka.dodaj( (x+1,y) )
                inače:
                        provjeri(x, y+sgny0)
                        lista_točaka.dodaj( (x,y+sgny0) )
                y := y + sgny0
                D := D – 1
        inače:
                provjeri(x+1, y)
        x := x + 1
        D := D + y_x
        Lista_točaka.dodaj( (x,y) )
inače:
        sgnx0 := sig( x2 – x1 )
        x_y := absx0/absy0
        x_y_2 := x_y/2
        D := x_y – 0.5
        left_blocked := False
        right_blocked := False
        za i od 0 do absy0:
                ako je D > 0:
                        provjeri(x+sgnx0, y+1)
                        ako je x_y = 1:
                                ako nije right_blocked:
                                        provjeri_desno
                                ako nije left_blocked:
                                        provjeri_lijevo
                                ako je left_blocked i right_blocked:
                                        vrati False
                        inače:
                                ako je D < x_y_2:
                                        provjeri(x, y+1)
```

```
                                    inače:
                                                provjeri(x+sgnx0, y)
                            x := sgnx0
                            D := D – 1
                    inače:
                            provjeri(x, y+1)
                    y := y + 1
                    D := D + x_y
                    lista_točaka.dodaj( (x,y) )
ako je reverse:
        preokreni listu točaka
lista_točaka.dodaj( (x2,y2) )
vrati lista_točaka
```

## 4.4. Pretraživanje prostora

Diskretna aproksimacija kontinuiranog prostora ima neke nedostatke, kao osnovno to što je aproksimacija pa se određena količina informacije gubi, ali baš zbog toga ima neke prednosti jer kada svedemo beskonačno informacija na dobro uređenu malu količinu možemo lako baratati s njima. Najveća prednost je što možemo lako i brzo pretraživati cijeli prostor ako imamo dovoljno dobar alat, a modificirani Bresenhamov algoritam je baš napravljen da obavlja taj zadatak.

Praktični dio ovoga rada je predstavljen s testnim programom koji, nad već učitanim virtualnim prostorom, određuje vidljivost za zadane jedinice tako da pretražuje cijeli prostor i za svaku kocku odredi njenu vidljivost u odnosu na zadane jedinice. Rezultat takvog pretraživanja je lista kocaka, odnosno dijelova zadanog prostora, koje su vidljive jednoj ili više jedinica i program ih grafički prikaže tako da osvijetli te kocke.




**Sl. 29** Primjer pretraživanja otvorenog prostora u testnom programu

Na primjeru na Sl. 29 možemo vidjeti najveću prednost ovakvoga zapisa prostora, jer računanje vidljivosti cijelog prostora u kontinuiranom zapisu bi bilo praktički nemoguće.

### 4.4.1. Optimiziranje

Ako se želi izračunati vidljivost u prostoru koji ima $x \cdot y$ kocaka, za $n$ jedinica bit će nam potrebno $N=x \cdot y \cdot n$ poziva, što znači da $N$ vrlo brzo raste s povećanjem bilo kojeg faktora i potrebna je optimizacija kako bi se broj poziva smanjio, i samim time brzina izvođenja povećala. Pošto se broj poziva $x \cdot y$ ne može smanjiti, jer u svakom slučaju je potrebno pretražiti sva polja barem jednom, optimizacija će se svesti na smanjenje utjecaja broja jedinica na ukupnu brzinu.

Sva testiranja načinjena su u istom prostoru, veličine 20x21 polje, te je kao rezultat uzeto prosječno vrijeme poziva funkcije koja vraća sva vidljiva polja. Prosječno vrijeme je izračunato kao prosjek između 10,000 poziva funkcije.

### Pomoćno polje

Napravimo pomoćno polje u memoriji koje je veličine $x \cdot y$ i onda provjerimo sve kocke za jednu jedinicu, ako je koja kocka vidljiva zapišemo to u pomoćno polje, ako nije ne zapisujemo ništa jer će neka druga jedinica možda vidjeti tu kocku. Nakon toga krećemo provjeravat sve kocke za drugu jedinicu ali ne provjeravamo za one kocke za koje je već označeno da su vidljive, i tako redom dok ne prođemo sve jedinice.

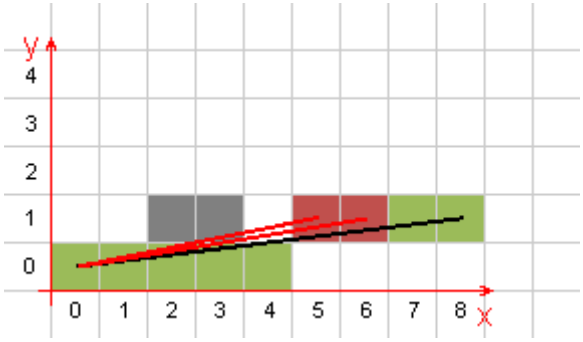
**Tablica 5** Prosječno vrijeme trajanje jednog poziva funkcije koja vraća sva vidljiva polja s pomoćnim poljem

| Broj jedinica | Bez optimiziranja (ms) | Pomoćno polje (ms) | Ubrzanje (%) |
|---|---|---|---|
| 1 | 10.89 | 10.58 | 3.8 |
| 2 | 18.80 | 14.81 | 21.2 |
| 3 | 27.77 | 17.75 | 37.1 |
| 4 | 34.54 | 19.52 | 43.5 |
| 5 | 40.32 | 21.03 | 47.8 |
| 6 | 50.27 | 23.97 | 52.3 |
| 7 | 59.67 | 25.97 | 56.5 |

Očekivano, vidimo kako metoda pomoćnog polja ubrzava izvođenje, i što je veći broj jedinica ubrzanje je veće.

### Lista vidljivih polja

Kada bi nam algoritam vratio listu polja koja su vidljiva, mogli bi napraviti još bolju optimizaciju na način da sva ta polja označimo u pomoćnom polju. Na taj način bi mogli s jednim pozivom izračuna linije označiti nekoliko polja u pomoćnom polju te na taj način smanjiti broj poziva algoritma. Međutim lista kocaka koju nam modificirani Bresenhamov algoritam vrati nije dobra za tu namjenu jer smo vidljivost definirali ako se linija može povući od centra početne kocke do centra ciljne kocke, a algoritam vraća kocke između za koje ne možemo biti sigurni da li ih vidimo parcijalno ili u potpunosti.




**Sl. 30** Kocke (5,1) i (6,1) su parcijalno vidljive i kroz njih vidimo kocku (8,1), međutim one same nisu vidljive iz kocke (0,0)

Jedini slučaj kada možemo za kocke u listi, osim početne i krajnje, biti sigurni da su vidljive je kada provjeravamo liniju pod 45°, pa je zbog toga napravljena još jedna optimizacija u samom algoritmu a ta je da algoritam vraća listu sigurno vidljivih polja, ili ništa ako je vidljivost blokirana. U toj listi je uvijek ciljno polje, a ako je linija pod 45° onda su i sva polja između početnog i ciljnog.

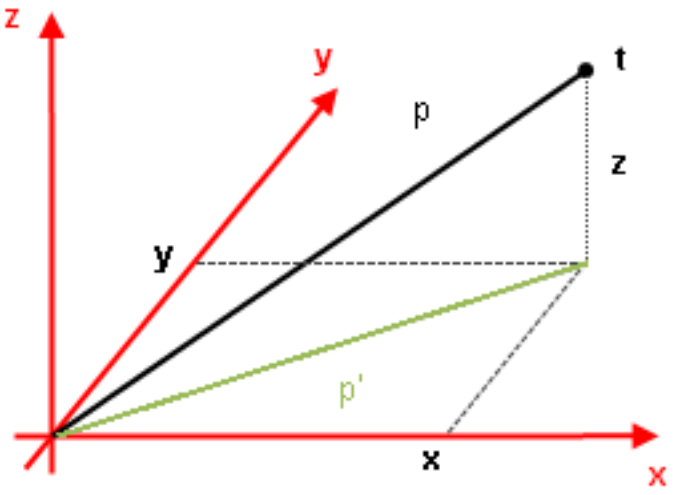
**Tablica 6** Prosječno vrijeme trajanje jednog poziva funkcije koja vraća sva vidljiva polja s listom vidljivih polja

| Broj jedinica | Bez liste vidljivih polja (ms) | S listom vidljivih polja (ms) | Ubrzanje (%) |
|---|---|---|---|
| 1 | 10.58 | 9.06 | 14.3 |
| 2 | 14.81 | 12.60 | 14.9 |
| 3 | 17.75 | 15.25 | 14.1 |
| 4 | 19.52 | 17.12 | 12.3 |
| 5 | 21.03 | 18.19 | 13.5 |
| 6 | 23.97 | 20.96 | 12.5 |
| 7 | 25.97 | 22.63 | 12.9 |

Kada smo maknuli bespotrebno dodavanje točaka u listu i u nekim slučajevima preokretanje same liste, te napravili da se za 45° dodaju točke i time jednim pozivom eliminiraju nekoliko slijedećih poziva algoritma dobili smo ubrzanje oko 13%.

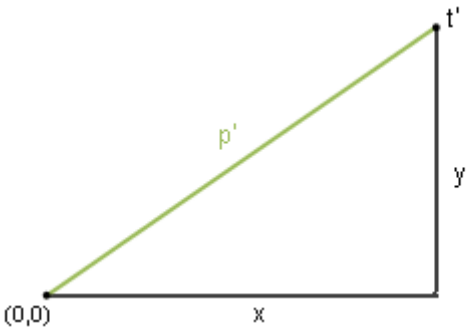
# 5. Proširenje na tri dimenzije

Vidjeli smo kako se Bresenhamov algoritam može modificirati za određivanje vidljivosti u dvodimenzionalnom prostoru a u ovom poglavlju će biti objašnjeno kako ga modificirati za pretraživanje trodimenzionalnog prostora.

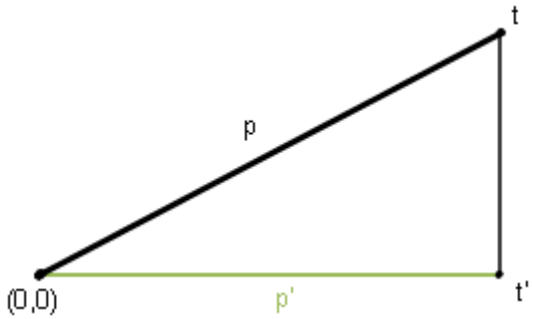



**Sl. 31** Projekcija linije *p* na x-y ravninu

Pošto Bresenhamov algoritam radi u dvije dimenzije napravimo projekciju linije *p* u x-y ravninu i dobijemo liniju *p'* koju možemo izračunati s osnovnom verzijom algoritma koristeći *x* kao jednu koordinatu i *y* kao drugu. Kada imamo *p'* možemo izračunati liniju *p*, koristeći osnovni algoritam, tako da koristimo *p'* kao jednu koordinatu i *z* kao drugu. Na taj način svedemo izračun linije u trodimenzionalnom prostoru na dva izračuna u dvodimenzionalnom prostoru za koja već imamo napravljen algoritam.




**Sl. 32** U x-y ravnini dobijemo trokut x-y-p'




**Sl. 33** U ravnini koja je okomita na x-y ravninu i prolazi kroz *p'* dobivamo trokut p'-z-p

## 5.1. Optimizacija

Radi jednostavnosti i lakše vizualizacije promatrat ćemo slučajeve u okruženju od kocke (0,0,0) i to u pozitivnom smjeru svake osi, znači do kocke (1,1,1), ostali slučajevi se rješavaju analogno ovome.

### 5.1.1. Brzina

Pošto sada kada dodamo treću dimenziju povećavamo broj poziva algoritma na $N=x \cdot y \cdot z \cdot n$ potrebno je algoritam još ubrzati. To radimo tako da modificiramo glavnu petlju pa umjesto da imamo 2 prolaza, jedan da nađemo *p'* a drugi da nađemo *p*, imamo samo jedan prolaz u kojem odmah povećavamo z koordinatu. Zbog tog rješenja gubimo mogućnost pronalaska kocki kroz koje linija parcijalno prolazi tako da u ovoj verziji algoritma nećemo dobiti tako točne rezultate u nekim rubnim područjima.

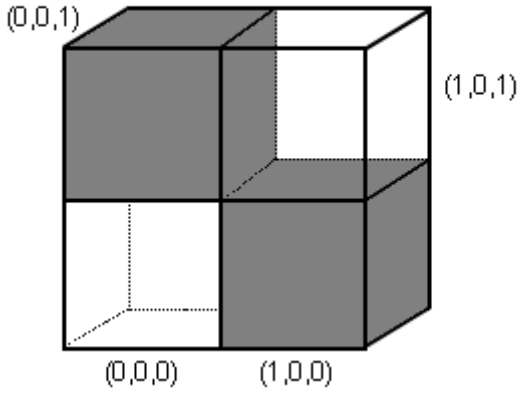
Zbog navedenih modifikacija nam je potreban drugačiji način provjere vidljivosti pa to radimo na slijedeći način:

- Pomak samo po jednoj koordinati je trivijalan za provjeru
- Pomak po dvije koordinate se svodi na provjeru dviju susjednih kocaka po tim koordinatama, na primjer za pomak sa (0,0,0) na (1,0,1) provjeravamo kocke (1,0,0) i (0,0,1) i ako je bar jedna prazna, onda vidljivost nije blokirana
- Pomak po sve tri koordinate ima samo dva slučaja kada je vidljivost blokirana:
    - Kada su blokirane slijedeće 4 kocke, kao na slici Sl. 35
        - (x+1,y,z)

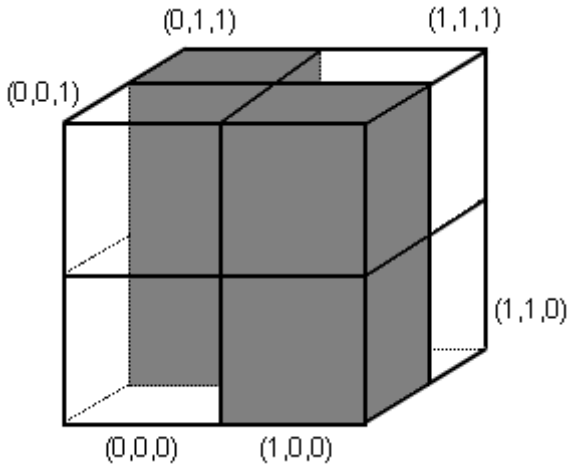
- (x+1,y,z+1)
- (x,y+1,z)
- (x,y+1,z+1)

○ Kada su blokirane slijedeće 3 kocke, kao na slici Sl. 36:

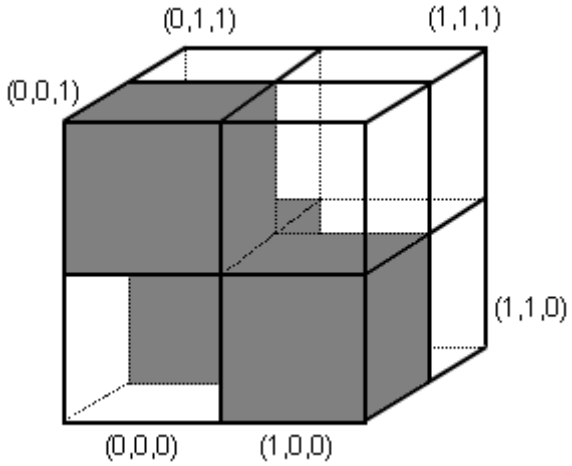
- (x+1,y,z)
- (x,y,z+1)
- (x,y+1,z)




**Sl. 34** Ako se pomičemo po samo po dvije koordinate (u ovom slučaju x i z), treba provjeriti slučaj da obje susjedne kocke ne blokiraju vidljivost




**Sl. 35** Ako se pomičemo po sve tri koordinate, ovaj slučaj blokira vidljivost




**Sl. 36** Ako se pomičemo po sve tri koordinate i ovaj slučaj blokira vidljivost

### 5.1.2. Točnost

Točnost izračuna vidljivosti možemo povećati na način da koristimo i algoritam za dvodimenzionalnu provjeru i modificirani za trodimenzionalnu u istoj funkciji. Treba ispitati slučaj kada obje točke imaju vrijednost z koordinate jednaku nula, onda algoritam pozove funkciju koja obavlja točniju dvodimenzionalnu provjeru i vrati njezinu vrijednost.

Trodimenzionalna verzija algoritma ne uzima u obzir zidove, jer se zidovi nalaze uvijek i samo na kockama koje imaju vrijednost z koordinate jednake 0. Međutim kada iteriramo po kockama u glavnoj petlji i radimo provjeru vidljivosti između trenutne kocke i slijedeće trebamo napraviti još jednu provjeru. Ako bilo koja od tih kocki ima vrijednost *z=0* u tom slučaju treba napraviti posebnu provjera koja uzima u obzir i zidove.

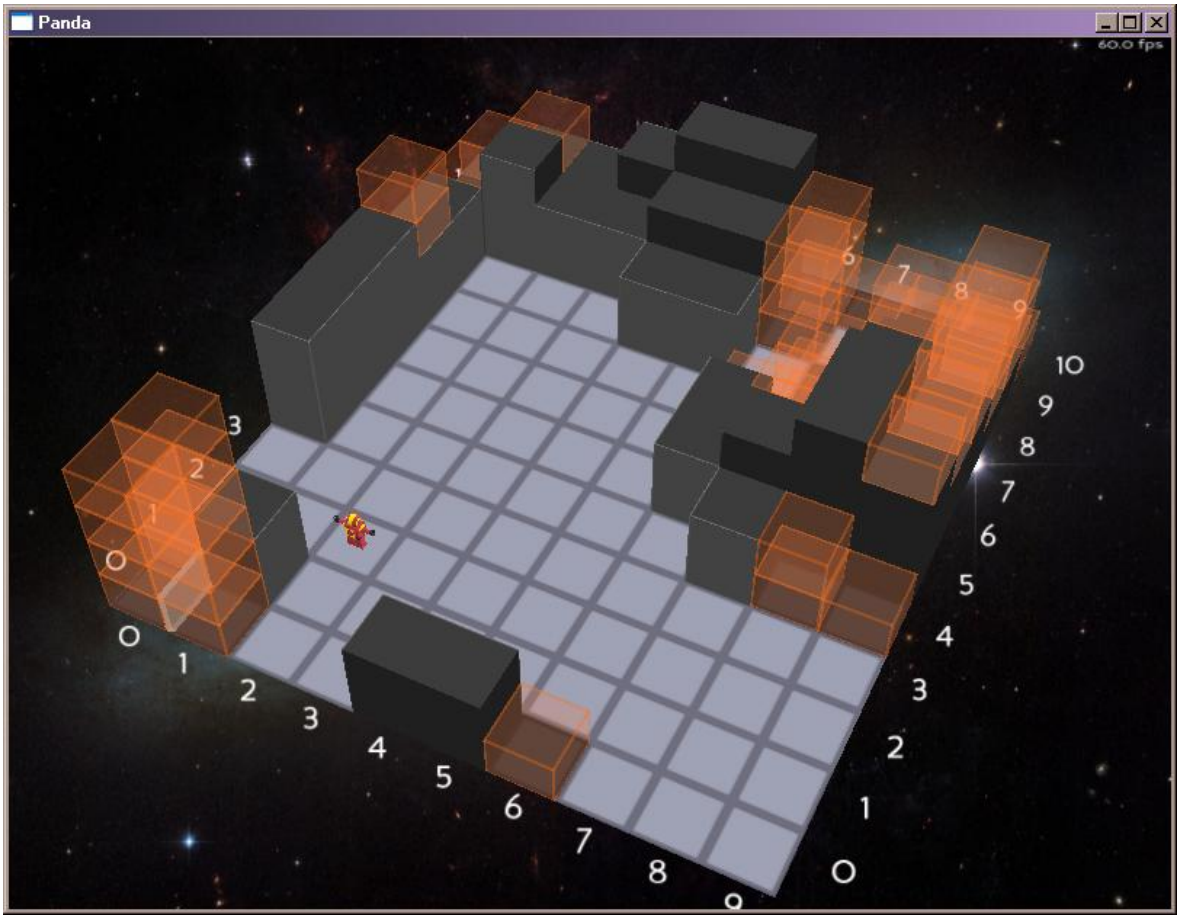
## 5.2. Testni program

Testni program se pokreće naredbom "python tester.py". Da bi se program mogao pokrenuti potrebno je imati instaliran Python 2.7 i Panda3D Runtime 1.0.4.

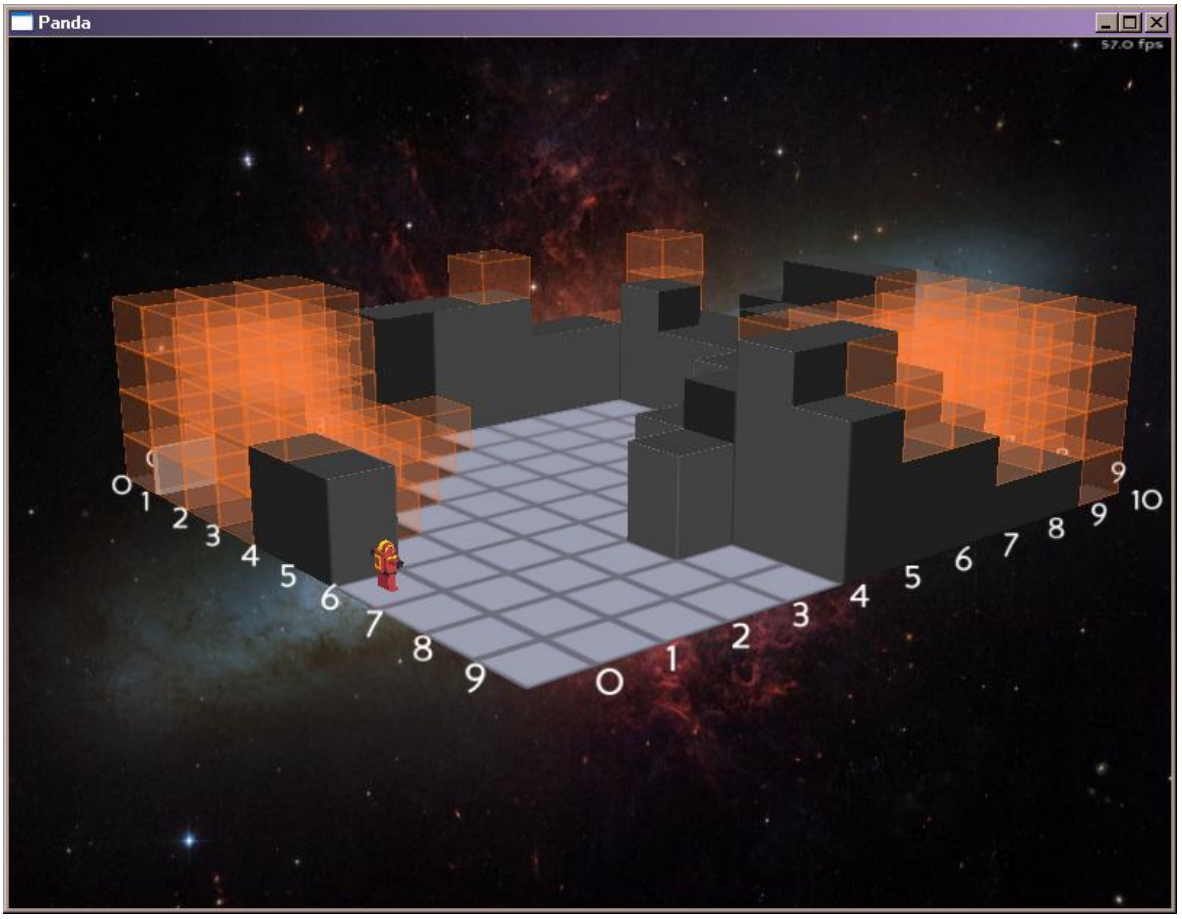
Kada se program pokrene pritiskom na lijevu tipku miša dodaje se nova jedinica na polje iznad kojeg je stisnuta tipka.

Kamera se pomiče pomoću tipki *w,a,s* i *d*, a rotira pomoću miša kada se drži stisnut desni gumb.

Pritiskom na tipku *3* na tipkovnici se određuje trodimenzionalna provjera vidljivosti, dok se s tipkom *2* određuje dvodimenzionalna provjera.




**Sl. 37** Primjer vidljivosti u 3d, narančaste kocke nisu vidljive




**Sl. 38** Još jedan primjer vidljivosti

## 5.3. Pseudokod

Funkcija *udaljenost()* vraća udaljenost između dvije točke.

Pseudokod algoritma modificiranog za trodimenzionalnu provjeru:

**Funkcija** provjeri(x1,y1,z1,x2,y2,z2):
**ako je** z1=0 **i** z2=0:
pozovi 2d provjeru i **vrati** rezultat
absx0 := abs(x2-x1)
absy0 := abs(y2-y1)
absz0 := abs(z2-z1)
dist := udaljenost(x1,y1,x2,y2)

```
ako je absx0>absy0:
    ako je x2<x1:
        zamijeni x1 i x2, y1 i y2, z1 i z2
inače:
    ako je y2<y1:
        zamijeni x1 i x2, y1 i y2, z1 i z2
x := x1
y := y1
z := z1
sgnz0 = sig(z2-z1)
z_d = absz0/dist
ako je absx0>absy0:
    sgny0 := sig(y2-y1)
    y_x := absy0/absx0
    D := y_x–0.5
    ako je dist>absz0:
        Dz := z_d–0.5
    za i od 0 do absx0:
        lastx := x
        lasty := y
        lastz := z
        ako je D>0:
            y := y+sgny0
            D := D-1
    ako je dist>absz0:
        ako je Dz>0:
            z := z + sgnz0
            Dz := Dz – 1
        Dz := Dz + z_d
    inače:
        z := z + z_d * sgnz0
    x := x+1
    D := D+y_x
    testiraj vidljivost((x,y,z), (lastx,lasty,lastz))
inače:
    sgnx0 := sig(x2-x1)
    x_y := absx0/absy0
    D := x_y–0.5
    ako je dist>absz0:
        Dz := z_d–0.5
```

```
        za i od 0 do absy0:
                lastx := x
                lasty := y
                lastz := z
                ako je D>0:
                        x := x+sgny0
                        D := D-1
        ako je dist>absz0:
                ako je Dz>0:
                        z := z + sgnz0
                        Dz := Dz – 1
                Dz := Dz + z_d
        inače:
                z := z + z_d * sgnz0
        y := y+1
        D := D+x_y
        testiraj vidljivost((x,y,z), (lastx,lasty,lastz))
vrati True
```

# 6. Zaključak

U ovom radu je bila razrađena modifikacija Bresenhamovog algoritma za određivanje vidljivosti u virtualnoj sceni. Pošto je glavna prednost Bresenhamovog algoritma njegova brzina izračuna tako su bile razrađene i metode optimizacije brzine. Iako je određivanje vidljivosti ovom metodom brzo i točno ima jedno ograničenje, a to je da se vidljivost pojedine kocke definira kao jedna od dvije vrijednosti: vidljivo ili nevidljivo. Takva apsolutna podjela vidljivosti u nekim specifičnim slučajevima dovodi do neprirodnih rezultata, pa bi daljnja razrada ove metode trebalo biti proširenje da se vidljivost izrazi u postocima. Ta metoda bi zahtijevala skroz drugačiji način izračuna vidljivosti, pomoću kutova susjednih kocaka, ali bi modificirani Bresenhamov algoritam ostao kao osnovna metoda pretraživanja kocaka.

# 7. Literatura


1. http://www.slideshare.net/abhijeetgoel77/visibility-graphics
2. http://www.zemris.fer.hr/predmeti/rg/predavanja/5_skrivanje.pdf
3. http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m13/bsp.html
4. http://medialab.di.unipi.it/web/IUM/Waterloo/node68.html
5. http://www.chadvernon.com/blog/resources/directx9/frustum-culling/
6. http://accu.org/index.php/journals/1550
7. http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

# Tehnike određivanja skrivenih linija i površina

## Sažetak

U ovome radu je predstavljena problematika određivanja vidljivosti u računalnoj grafici. Također su izložene neke od tehnika koje služe za rješavanje problema vidljivosti kao i za uklanjanje skrivenih linija i ploha.

Kao praktični dio rada razrađena je modifikacija Bresenhamovog algoritma u svrhu određivanja vidljivosti u dvodimenzionalnom i trodimenzionalnom prostoru.

**Ključne riječi:** skrivene linije, skrivene površine, vidljivost, Bresenhamov algoritam

# Methods of detecting hidden lines and surfaces

## Abstract

This paper presents the problem with calculations of visibility in computer graphics. It presents some methods that solve both visibility and removal of hidden lines and surfaces.

This paper documents the process of modifying Bresenham's line algorithm for use in visibility calculations in both two dimensional and three dimensional space.

**Keywords:** hidden lines, hidden surfaces, visibility, Bresenham's line algorithm