Osnovni algoritmi rasterske grafike za prikazivanje 2D primitiva

Ciganj, Andrej

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Andrej Ciganj

OSNOVNI ALGORITMI RASTERSKE GRAFIKE ZA PRIKAZIVANJE 2D PRIMITIVA

Diplomski rad

Voditelj rada:
doc. dr. sc. Tina Bosner

Zagreb, srpanj, 2016.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Većina današnjih prikaznih jedinica (ekrani TV-a, računala, mobitela, itd.) je diskretna - rasterska. Slika koju rasterski ekrani prikazuju je sastavljena od malih dijelova koji se nazivaju pikseli.

Za razliku od rasterskih ekrana, u povijesti su se koristili ekrani kojima je slika kontinuirana - vektorska, no zbog raznih nedostataka, brzo su izašli iz upotrebe.

2D slika spremljena u računalu također može biti vektorska ili rasterska. Rasterski zapis ili bitmapa je veći od vektorskog jer sadrži podatke o svakom dijelu slike - pikselu, pa takve slike jednostavno možemo prikazati na ekranu.

Vektorski zapis je manji od rasterskog jer sadrži samo podatke o geometrijskim likovima na slici. Tako nam za trokut bilo koje veličine i oblika trebaju samo koordinate vrhova. Vektorski zapis se najviše koristi u dizajnu, računalnim igrama, CGI (*Computer-Generated Imagery*) i slično. Slike koje su stvorene računalno najčešće su spremljene u vektorskom zapisu.

- Prikaz rasterskih slika na rasterskom ekranu nije nikakav problem - direktno se aktiviraju odgovarajući pikseli ekrana na temelju informacija u rasterskom zapisu
- Prikaz vektorskih slika na vektorskom ekranu, također nije problem - elektronska zraka osvjetljava određeni dio ekrana na temelju informacija u vektorskom zapisu
- Prikaz rasterskih slika na vektorskom ekranu bi bio važan problem kad bismo se služili vektorskim ekranima
- **Problem koji ovaj rad obrađuje je prikaz vektorskih slika na rasterskom ekranu**

U radu je opisan proces dobivanja slike na rasterskom ekranu i neki osnovni algoritmi koji izvode korake tog procesa.

Tehnološkim napretkom, konkretno razvojem računala, računalnih ekrana i računalne grafike dolazi do sve većih zahtjeva na izgled slike na ekranu i brzine iscrtavanja slike. Jedan od primjera su računalne igre koje zahtjevaju sve više prikazanih sličica u sekundi (engl. *frames per second* - FPS) i sve više objekata prikazanih odjednom.

Poglavlje 1

Rasterizacija

Rasterizacija je proces pretvaranja vektorskog zapisa slike u rastersku sliku na ekranu.

Vektorski zapis slike sastoji se od geometrijskih primitiva. Primitive su najjednostavniji geometrijski likovi kojima se služi većina sustava za prikazivanje slike na ekranu. Najčešće korištene 2D geometrijske primitive su:

- točke
- dužine (u radu korišten pojam "linija")
- kružnice ili elipse
- trokuti i ostali mnogokuti.

Jednom definirane primitive (primjerice, točka pomoću x i y koordinate, linija pomoću dvije točke, kružnica pomoću točke i radiusa, itd.) prolaze kroz 2D transformacije. Osnovne transformacije uključuju translaciju, skaliranje, rotaciju i njihovu kompoziciju. Svaka primitiva koja prođe kroz jednu od osnovnih transformacija ostaje primitiva.

Primitive se mogu nalaziti bilo gdje u koordinatnom sustavu, dok je ekran kojim ih prikazujemo ograničen. Stoga prije prikazivanja, umjesto da se troše računalni resursi na primitive koje ionako nećemo prikazati, provodi se obrezivanje (engl. *clipping*) primitiva. U sljedeći korak se puštaju samo one primitive ili samo dijelovi primitiva koji se nalaze unutar granica ekrana.

Posljednji korak rasterizacije je upisivanje bitova u međuspremnik nakon čega vrijednosti bitova direktno aktiviraju pripadajuće piksele. Odabir bitova u međuspremniku naziva se *scan konverzija* (*scan conversion*). Jedan piksel može biti vezan na jedan ili više bitova. U slučaju jednog bita, piksel ima samo informaciju je li ugašen ili upaljen (crn ili bijel). Što je više bitova vezano za piksel, to više boja i intenziteta piksel može prikazati.

U diskretizaciji, pa tako i u scan konverziji javlja se problem *aliasing*-a, ili nazubljenosti. Tehnike kojim se problem aliasing-a ublažuje nazivaju se *antialiasing*.

Zaključno, koraci rasterizacije su:

1. definiranje i transformiranje primitiva
2. obrezivanje
3. scan konverzija i antialiasing

Rad je podijeljen na četiri poglavlja. U drugom je obrađen osnovni dio rasterizacije, scan konverzija, nakon čega se obrađuje obrezivanje i na kraju antialiasing.

Poglavlje 2

Scan konverzija

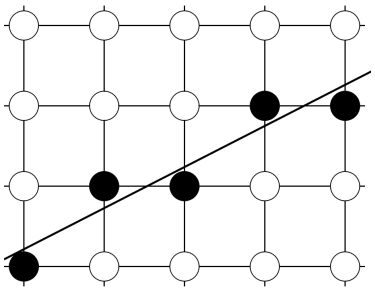
2.1 Scan konverzija točke

Scan konverzija je pretvorba analognog signala u digitalnu sliku. U prvim televizijskim i računalnim CRT (*Cathode Ray Tube*) ekranima, scan konverzija se izvodila hardverski. Tema ovog poglavlja je softverska scan konverzija. Vidjet ćemo kako možemo izračunati koordinate piksela koje moramo aktivirati da bi se željena primitiva prikazala na ekranu.

Krenut ćemo od najjednostavnije primitive, točke. Točka je zadana s njezinim x i y koordinatama. U slučaju cjelobrojnih koordinata, aktivira se piksel kojemu odgovaraju te koordinate, dok u slučaju decimalnih, koordinate se prvo zaokružuju pa se aktivira piksel kojem odgovaraju dobivene koordinate.

2.2 Scan konverzija linije

Metoda kojom se najčešće crtaju linije je implementacija Bresenhamovog postupka. U poglavlju se polako, u nekoliko pokušaja konstruiranja dobrog algoritma, dolazi do njega.



Slika 2.1: Idealna linija aproksimirana pikselima

Prilikom crtanja linije poznate su koordinate početne i završne točke. Točke su zadane na sljedeći način:

- Koordinate x_1, y_1, x_2 i y_2 su **cjelobrojne**.
- Početna točka T_1 ima koordinate (x_1, y_1).
- Završna točka T_2 ima koordinate (x_2, y_2).
- Vrijedi: $x_1 < x_2$ i $y_1 < y_2$.
- Linija ima nagib manji ili jednak 1 (kut manji od 45°)

Ova ograničenja se trenutno koriste zbog jednostavnosti, a kasnije se postupak može lako generalizirati.

Jednadžba pravca kroz dvije točke je:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Uvođenjem oznake a za koeficijent smjera pravca odnosno za tangens kuta između pravca i osi x, i oznake b za odsječak na osi y dobivamo:

$$y = a \cdot x + b \tag{2.1}$$

pri čemu je

$$a = \frac{\Delta y}{\Delta x}, \quad b = -a \cdot x_1 + y_1.$$

gdje su $\Delta x = x_2 - x_1$ i $\Delta y = y_2 - y_1$.

Koeficijenti a i b ovise samo o konstantama pa se stoga mogu izračunati na početku i koristiti tijekom cijelog algoritma. Koristeći formulu 2.1, pravac možemo nacrtati sljedećim trivijalnim algoritmom:

```
void linija (int x1, int y2, int x1, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    double a = dy/(double)dx;
    double b = -a * x1 + y1;
    for (int x = x1; x <= x2 ; x++) {
        double y = a*x + b;
        aktiviraj_piksel(x,zaokruzi(y));
    }
}
```

```
int zaokruzi(double x){
    return floor(x + 0.5);
}
```

U petlji se za svaki x množenjem u aritmetici pomičnog zareza računa odgovarajući y. To je izrazito sporo i potrebno je promjeniti način računanja.

Ispišimo nekoliko koraka petlje da vidimo postoji li neko pravilo kojim se postupak može ubrzati:

$$
\begin{aligned}
y|_{x=x_1} &= a \cdot x_1 + b = y_1 \\
y|_{x=x_1+1} &= a \cdot (x_1 + 1) + b = a \cdot x_1 + b + a = y|_{x=x_1} + a \\
y|_{x=x_1+2} &= a \cdot (x_1 + 2) + b = a \cdot x_1 + b + 2a = y|_{x=x_1+1} + a \\
y|_{x=x_1+3} &= a \cdot (x_1 + 3) + b = a \cdot x_1 + b + 3a = y|_{x=x_1+2} + a \\
&\ldots
\end{aligned}
$$

Zaključujemo da je svaki sljedeći y za a veći od prethodnog. To nam omogućuje da se riješimo zahtjevne operacije množenja. Uzevši u obzir dobiveno, možemo zapisati novu verziju algoritma:

```
void linija(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    double a = dy / (double)dx;
    double y = y1;
    for (int x = x1; x <= x2 ; x++) {
        aktiviraj_piksel(x, zaokruzi(y)) ;
        y += a ;
    }
}
```

Riješili smo se množenja, ali bi bilo dobro kada bismo rijeđe morali pozivati funkciju `zaokruzi(y)`.

2.2.1 Decimalni Bresenhamov algoritam

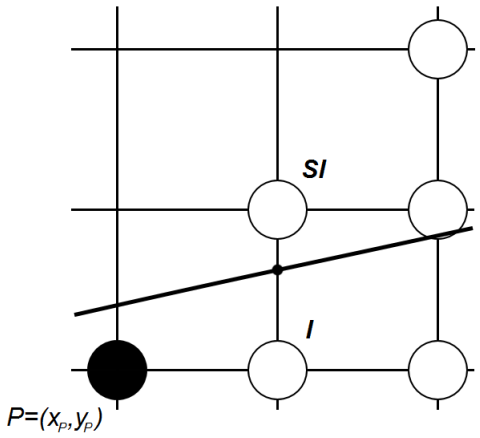
Ponovno ćemo ispisati nekoliko iteracija algoritma da vidimo što se točno događa. Početnu zaokruženu vrijednost y koordinate ne moramo računati. Za $x = x_1$ imamo $y = y_1$. Nakon što aktiviramo piksel na poziciji (x_1, y_1), pomičemo se za jedan piksel udesno. Time se vrijednost y koordinate poveća za koeficijent smjera a. Zbog ograničenja da je pravac pod kutom manjim ili jednakim 45°, vrijednost varijable a je između 0 i 1.

Provedimo nekoliko iteracija za primjer $a = 0.2$:

$$\begin{aligned} y|_{x=x_1} &= y_1 \rightarrow aktiviraj\_piksel(x_1, y_1) \\ y|_{x=x_1+1} &= y|_{x=x_1} + 0.2 \rightarrow aktiviraj\_piksel(x_1 + 1, y_1) \\ y|_{x=x_1+2} &= y|_{x=x_1} + 0.4 \rightarrow aktiviraj\_piksel(x_1 + 2, y_1) \\ y|_{x=x_1+3} &= y|_{x=x_1} + 0.6 \rightarrow aktiviraj\_piksel(x_1 + 3, y_1 + 1) \\ &\dots \end{aligned}$$

Koordinatu y ćemo razdvojiti na cjelobrojni dio, varijablu y, i odgovarajući decimalni dio, varijablu d. Prethodni postupak sada možemo opisati na sljedeći način: aktiviramo piksele na poziciji $(x_1 + k, y)$, pri čemu je $k \in \mathbb{Z} \cap [0, \Delta x]$. Na početku imamo: $y = y_1$ i $d = 0$. Svakim pomakom u desno (po x osi) d uvećamo za a. Kad god d dosegne ili prijeđe 0.5, y uvećamo za 1, a d smanjimo za 1.

Kako bismo vizualno bolje razumjeli zašto oduzimamo 1 od d pogledajmo sliku 2.2.



Slika 2.2: Prethodno odabrani piksel (x_P, y_P) i pikseli **SI** i **I** između kojih algoritam bira u trenutnoj iteraciji

Nakon iteracije u kojoj je odabran piksel (x_P, y_P), algoritam bira između dva piksela: **I** (istok) i **SI** (sjeveroistok). Odabirom piksela **I**, ostaje $y = y_P$, dok odabirom piksela **SI** y postaje $y_P + 1$. Decimalni dio d označava razliku y koordinate idealne linije u $x_P + 1$ i y koordinate piksela odabranog u prošloj iteraciji, y_P. Kada d prijeđe 0.5 idealna je linija bliže pikselu **SI** nego pikselu **I** pa algoritam odabire **SI**, odnosno $(x_P + 1, y_P + 1)$. U našem primjeru to se dogodilo pri izboru piksela u $x_1 + 3$. d se umanjio za 1 jer y koordinatu idealne linije više ne uspoređujemo s y_1 nego s $y_1 + 1$. Možemo reći da 0.6 iznad y_1 postaje -0.4 iznad $y_1 + 1$.

Implementacija ovog algoritma dana je u nastavku:

```
void linija (int x1 ,int y1 ,int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    double a = dy / (double)dx;
    double d = 0.;
    int y = y1;
    for (int x = x1; x <= x2; x++) {
        aktiviraj_piksel(x, y);
        d += a;
        if(d >= 0.5) {
            d -= 1.0;
            y++;
        }
    }
}
```

Do Bresenhamovog algoritma dijeli nas samo jedna sitna modifikacija: varijablu odluke d možemo inicijalizirati na -0.5 i sve usporedbe raditi s nulom.

```
void bresenham_decimalni (int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    double a = dy / (double)dx;
    double d = -0.5;
    int y = y1;
    for (int x = x1 ; x <= x2; x++) {
        aktiviraj_piksel(x, y);
        d += a;
        if(d >= 0.) {
            d -= 1.0;
            y++;
        }
    }
}
```

Problem s trenutnim algoritmom je aritmetika pomičnog zareza. U nastavku je cilj prebaciti Bresenhamov algoritam na cjelobrojnu aritmetiku.

2.2.2 Cjelobrojni Bresenhamov algoritam

Prvo uvođenje decimalnih brojeva dogodilo se kod računanja koeficijenta smjera a prema formuli:

$$a = \frac{\Delta y}{\Delta x} \tag{2.2}$$

Taj smo koeficijent koristili pri izračunu decimalne vrijednosti d. Nakon svakog pomaka po x koordinati za 1, računali smo novi d prema formuli:

$$d_{novi} = d_{stari} + a.$$

Uvrštavanjem izraza 2.2 dobivamo:

$$d_{novi} = d_{stari} + \frac{\Delta y}{\Delta x} = \frac{d_{stari} \cdot \Delta x + \Delta y}{\Delta x}$$

Pomnožimo li dobiveno s nazivnikom:

$$d_{novi} \cdot (\Delta x) = d_{stari} \cdot (\Delta x) + \Delta y.$$

Uvodimo $d' = d \cdot (\Delta x)$ pa izraz prelazi u:

$$d'_{novi} = d'_{stari} + \Delta y.$$

Sada vidimo da umjesto dosadašnje vrijednosti d možemo koristiti d pomnoženu s Δx i tako se riješiti decimalnih brojeva barem u tom dijelu algoritma.

Inicijalni uvjeti još uvijek sadrže decimalne brojeve. Inicijalnu vrijednost varijable d smo postavili na -0.5. Raspisano:

$$\begin{aligned} d_0 &= -\frac{1}{2} \Big| \cdot \Delta x \\ d_0 \cdot (\Delta x) &= -\frac{\Delta x}{2} \\ d'_0 &= -\frac{\Delta x}{2} \end{aligned}$$

Ostala nam je dvojka u nazivniku koje se jednostavno rješavamo množeći sve izraze s 2. Zapišimo konačne izraze:

- Računanje nove vrijednosti varijable d:

$$\begin{aligned} d_{novi} &= d_{stari} + \frac{\Delta y}{\Delta x} = \frac{d_{stari} \cdot \Delta x + \Delta y}{\Delta x} \Big| \cdot 2\Delta x \\ 2 \cdot d_{novi} \cdot \Delta x &= 2d_{stari} \cdot \Delta x + 2\Delta y \\ d'_{novi} &= 2d_{stari} \cdot \Delta x + 2\Delta y \\ d'_{novi} &= d'_{stari} + 2\Delta y \end{aligned}$$

  .

- Inicijalna vrijednost:

$$\begin{aligned} d_0 &= -\frac{1}{2} \Big| \cdot 2\Delta x \\ 2d_0 \cdot \Delta x &= -\Delta x \\ d'_0 &= -\Delta x. \end{aligned}$$

- Oduzimanje jedinice:

$$d_{novi} = d_{stari} - 1 \Big| \cdot 2\Delta x$$
$$2d_{novi} \cdot \Delta x = 2d_{stari} \cdot \Delta x - 2\Delta x$$
$$d'_{novi} = d'_{stari} - 2\Delta x.$$

Sada možemo napisati cjelobrojni Bresenhamov algoritam. Iako smo koristili novu oznaku d' u algoritmu ćemo ipak radi jednostavnosti koristiti standardnu oznaku d, podrazumjevajući da su uzeti u obzir novi izrazi.

```
void bresenham_cjelobrojni (int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int d = -dx;
    int a = 2 * dy;
    int jedinica = 2 * dx;
    int y = y1;
    for (int x = x1; x <= x2; x++) {
        aktiviraj_piksel(x, y);
        d += a;
        if (d >= 0){
            d -= jedinica;
            y++;
        }
    }
}
```

Time smo dobili algoritam koji uzima u obzir samo linije pod kutevima između 0°i 45°. Algoritam proširujemo u nekoliko koraka:

1. **Između 0°i 90°**: ispitujemo je li tangens kuta veći od 1. U slučaju da je manji od 1, provodimo algoritam koji imamo. U slučaju da je veći od 1, zamijenimo uloge koordinatnih osi.

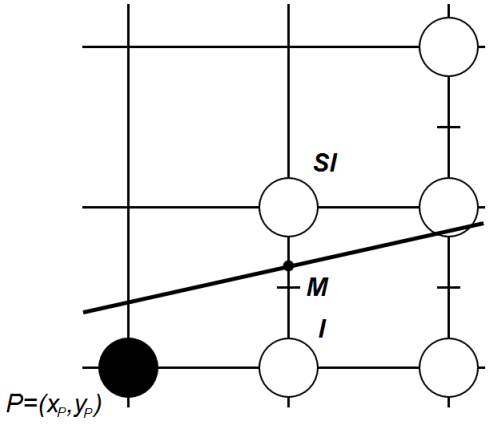
2. **Između -90°i 90°**: ispitujemo je li tangens kuta negativan. U slučaju da je pozitivan, provodimo prvi korak. U slučaju da je negativan, postupak je sličan prvom koraku samo što y u svakoj iteraciji umanjujemo za 1 umjesto povećavanja za 1.

3. **Svi kutevi**: ako su kutevi između 90°i 270°, zamijenimo početnu i završnu točku te provodimo prethodne korake.

Kako bismo postupak scan konverzije linije lakše generalizirali na kružnice i elipse, opisat ćemo još i *metodu srednje točke* koja za liniju i cjelobrojne kružnice odgovara Bresenhamovom postupku, tj. aktivira iste piksele.

2.2.3 Metoda srednje točke

U metodi srednje točke (*midpoint technique*) se promatra leži li srednju točka, tj. polovište dužine koja spaja piksele **I** i **SI**, iznad ili ispod idealne linije. U slučaju da leži iznad, u iteraciji biramo piksel **I**, dok u suprotnom biramo piksel **SI**.



Slika 2.3: Promatramo na kojoj strani idealne linije leži srednja točka M

Umjesto eksplicitnog:

$$y = \frac{\Delta y}{\Delta x}x + b$$

koristit ćemo implicitni oblik jednadžbe pravca:

$$F(x, y) = \Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot b$$

Za funkciju $F(x, y)$ vrijedi:

- $F(x, y) = 0$ na liniji
- $F(x, y) > 0$ za točke ispod linije
- $F(x, y) < 0$ za točke iznad linije

Testiramo srednju točku M tako da računamo vrijednost $F(M) = F(x_P + 1, y_P + \frac{1}{2})$ i odredimo joj predznak. Definiramo $d = F(M)$ koju ćemo nazivati varijabla odluke. Po definiciji je:

$$d = \Delta y(x_P + 1) - \Delta x(y_P + \frac{1}{2}) + \Delta x \cdot b.$$

- ako je $d > 0$ biramo piksel **SI**
- ako je $d < 0$ biramo piksel **I**
- ako je $d = 0$ biramo **I** jer je svejedno koji odaberemo

Promatramo što se događa s vrijednošću d kroz iteracije. Nova lokacija srednje točke M ovisi o izboru piksela u prošloj iteraciji.

Ako je izabran **I**, M se pomiče za 1 u smjeru x. Vrijedi:

$$\begin{aligned} d_{novi} &= F(x_P + 2, y_P + \frac{1}{2}) \\ &= \Delta y(x_P + 2) - \Delta x(y_P + \frac{1}{2}) + \Delta x b \\ &= d_{stari} + \Delta y \end{aligned}$$

i varijablu odluke dobivamo inkrementiranjem tekuće vrijednosti za $\Delta_{\mathbf{I}} = \Delta y$ bez računanja $F(M)$.

Ako je izabran **SI**, M se pomiče za 1 korak u smjeru x i 1 u smjeru y. Vrijedi:

$$\begin{aligned} d_{novi} &= F(x_P + 2, y_P + \frac{3}{2}) \\ &= \Delta y(x_P + 2) - \Delta x(y_P + \frac{3}{2}) + \Delta x b \\ &= d_{stari} + \Delta y - \Delta x \end{aligned}$$

i varijabla odluke se dobiva inkrementiranjem tekuće vrijednosti za $\Delta_{\mathbf{SI}} = \Delta y - \Delta x$.

Inicijalna vrijednost varijable odluke je

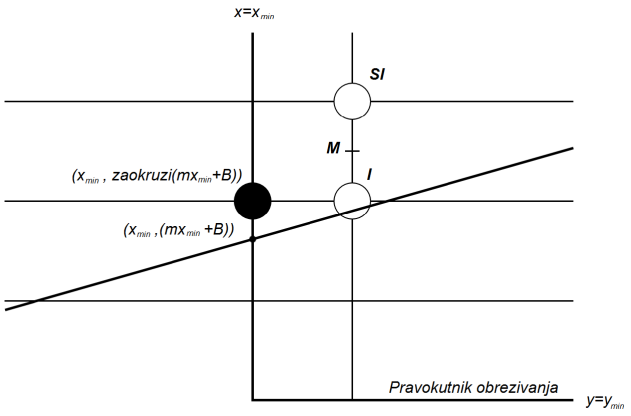
$$\begin{aligned} d_0 &= F(x_1 + 1, y_1 + \frac{1}{2}) \\ &= \Delta y(x_1 + 1) - \Delta x(y_1 + \frac{1}{2}) + \Delta x b \\ &= F(x_1, y_1) + \Delta y - \frac{\Delta x}{2}. \end{aligned}$$

Budući da je (x_1, y_1) točka na liniji, $F(x_1, y_1) = 0$ i $d_0 = \Delta y - \frac{\Delta x}{2}$. Kako bismo izbjegli dijeljenje u d_0, redefinirat ćemo funkciju $F(x, y)$ tako da ju pomnožimo s 2. Ova izmjena neće utjecati na predznak varijable odluke što je jedino važno u algoritmu. Kod algoritma prikazan je u nastavku.

```
void srednja_tocka (int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int d = 2*dy - dx;
    int deltaI = 2 * dy;
    int deltaSI = 2 * (dy - dx);
    int x = x1;
    int y = y1;
    aktiviraj_piksel(x, y);
    while (x < x1) {
        d += a;
        if (d <= 0){
            d += deltaI;
            x++;
        } else {
            d += deltaSI;
            x++;
            y++;
        }
        aktiviraj_piksel(x, y);
    }
}
```

2.2.4 Obrezane linije

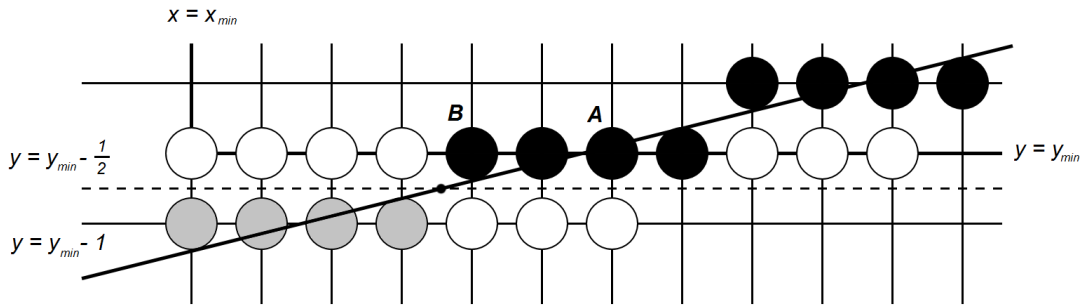
Ako je linija koju crtamo analitički obrezana (vidjeti poglavlje 3), potrebno je modificirati algoritam iscrtavanja.



Slika 2.4: Scan konverzija linije obrezane vertikalnom stranicom pravokutnika

Slika 2.4 prikazuje liniju obrezanu lijevom stranicom pravokutnika obrezivanja, za $x = x_{min}$. Sjecište linije i stranice ima cijelobrojnu x i realnu y koordinatu. Najbliži piksel liniji na stranici pravokutnika obrezivanja je piksel $P(x_{min}, zaokruzi(a \cdot x_{min} + b)$. Potrebno je aktivirati piksel P i inicijalizirati varijablu odluke d na srednju točku između **I** i **SI** desno od piksela P. Time zapravo crtamo originalnu liniju samo što počinjemo od sjecišta. Kada bismo Bresenhamov algoritam ili metodu srednje točke primjenili direktno na liniju od $(x_{min}, zaokruzi(a \cdot x_{min} + b))$ do druge krajnje točke originalne linije, dobili bismo drukčiji nagib.

Kompliciraniji slučaj je obrezivanje horizontalnom linijom.



Slika 2.5: Scan konverzija linije obrezane horizontalnom stranicom pravokutnika

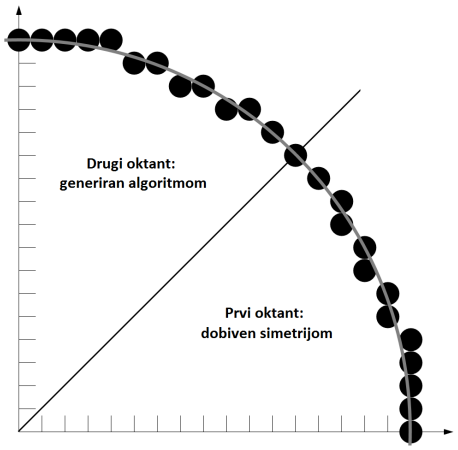
Na slici 2.5 prikazana je linija obrezana horizontalno. Linija je malog nagiba pa više piksela leži na donjoj stranici pravokutnika obrezivanja. Koristeći istu metodu kao u prošlom primjeru, zaokružujući, u ovom slučaju, koordinatu x na sjecištu, iscrtavanje bismo započeli od piksela A umjesto od piksela B. Sa slike vidimo da je piksel B prvi piksel iznad i desno od mjesta u kojem linija prolazi kroz srednju točku $y = y_{min} - \frac{1}{2}$. Dobivamo da je rješenje upravo nalaženje sjecišta linije s horizontalnim pravcem $y = y_{min} - \frac{1}{2}$, tj. piksel B je $(zaokruzi(x|_{y=y_{min}-\frac{1}{2}}), y_{min})$.

2.3 Scan konverzija kružnice

Slično kao u prethodnom algoritmu računamo za kružnicu koja je implicitno zadana s:

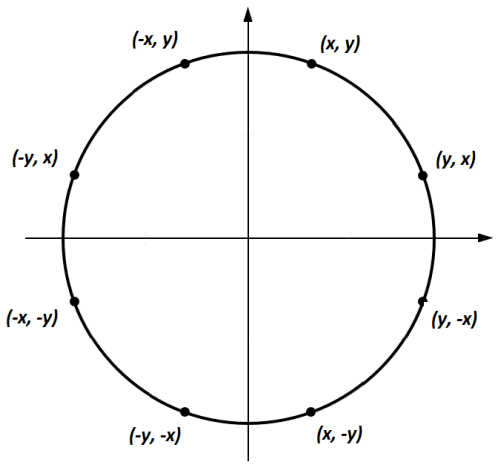
$$F(x, y) = x^2 + y^2 - R^2$$

Budući da na većini kružnice ne vrijedi svojstvo da možemo crtati jedan piksel po stupcu ili retku, algoritam će računati samo piksele drugog oktanta, a ostale crtati preko simetrija.



Slika 2.6: Pikseli prva dva oktanta kružnice

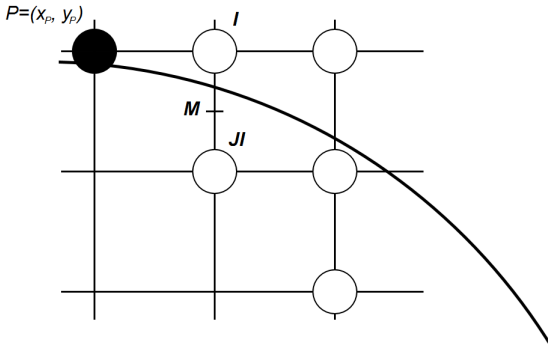
Funkcija koja će koristiti simetrije kružnice je dana u nastavku, a izračunate točke prikazane na slici 2.6. Rubne točke oktanata su specijalni slučajevi.



Slika 2.7: Točke koje se mogu dobiti koristeći simetrije kružnice

```
void tocke_kruznice (int x, int y){
    aktiviraj_piksel( x,  y);
    aktiviraj_piksel( y,  x);
    aktiviraj_piksel( y, -x);
    aktiviraj_piksel( x, -y);
    aktiviraj_piksel(-x, -y);
    aktiviraj_piksel(-y, -x);
    aktiviraj_piksel(-y,  x);
    aktiviraj_piksel(-x,  y);
}
```

Imamo sličan slučaj kao kod linije: funkcija $F(x, y)$ je jednaka 0 na kružnici, pozitivna je izvan kružnice i negativna unutar kružnice. Ako je srednja točka između piksela **I** i piksela **JI** izvan kružnice, bliži piksel kružnici je **JI**, a ako je unutar kružnice, **I**.



Slika 2.8: Algoritam bira između piksela **I** i **JI**

Odlučujemo na temelju varijable odluke, d, koja je vrijednost funkcije F u srednjoj točki:

$$d_{stari} = F(x_P + 1, y_P - \frac{1}{2}) = (x_P + 1)^2 + (y_P - \frac{1}{2})^2 - R^2.$$

Ako je $d_{stari} < 0$ biramo piksel **I**, a srednja točka u sljedećoj iteraciji je pomaknuta za 1 u smjeru x. Tada je

$$d_{novi} = F(x_P + 2, y_P - \frac{1}{2}) = (x_P + 2)^2 + (y_P - \frac{1}{2})^2 - R^2$$

i $d_{novi} = d_{stari} + (2x_P + 3)$, dakle inkrement je $\Delta_{\mathbf{I}} = 2x_P + 3$. Ako je $d_{stari} \geq 0$ biramo piksel **JI**, a srednja točka u sljedećoj iteraciji je pomaknuta za 1 u smjeru x i 1 u smjeru $-y$. Tada je

$$d_{novi} = F(x_P + 2, y_P - \frac{3}{2}) = (x_P + 2)^2 + (y_P - \frac{3}{2})^2 - R^2$$

Budući da je $d_{novi} = d_{stari} + (2x_P - 2y_P + 5)$, inkrement je $\Delta_{\mathbf{JI}} = 2x_P - 2y_P + 5$.

U linearnom slučaj, za liniju, $\Delta_{\mathbf{I}}$ i $\Delta_{\mathbf{SI}}$ bile su konstante. U kvadratnom slučaju, za kružnicu, one variraju iz koraka u korak i funkcije su tekućih vrijednosti x_P i y_P. Budući da te funkcije ovise o (x_P, y_P), točku P nazivamo točka evaluacije. Delta funkcije možemo računati direktno iz vrijednosti x i y piksela izabranog u prethodnoj iteraciji. Ta operacija nije previše skupa jer su funkcije linearne. Ukratko, radimo ista dva koraka u iteraciji kao i za liniju: (1) izaberemo piksel na temelju predznaka varijable odluke d i (2) ažuriramo d s Δ u ovisnosti o odabranom pikselu. Jedina razlika je što sada u točki evaluacije imamo za izračunati linearnu funkciju. Inicijalni uvjeti su:

- početna točka (0,R)
- prva srednja točka $(1, R - \frac{1}{2})$
- vrijednost varijable odluke $d = F(1, R - \frac{1}{2}) = 1 + (R^2 - R + \frac{1}{4}) - R^2 = \frac{5}{4} - R$

Sada možemo direktno implemetirati algoritam. Kod je:

```
void srednja_tocka_kruznica(int radius)
{
    int x = 0;
    int y = radius;
    double d = 5/4. - radius;
    tocke_kruznice(x,y);
    while(y>x){
        if(d<0)
            d+=2.0*x+3.0;
        else{
            d+=2.0*(x-y)+5.0;
            y--;
        }
        x++;
        tocke_kruznice(x,y);
    }
}
```

Primjetimo da je struktura algoritma slična algoritmu srednje točke za liniju. Jedini dio u algoritmu koji ima veze s realnom aritmetikom je inicijalna vrijednost varijable d. Jednostavnom modifikacijom možemo algoritam prebaciti na cjelobrojnu aritmetiku.

Definiramo varijablu $h = d - \frac{1}{4}$ i zamjenimo d u kodu s $h + \frac{1}{4}$. Sada je inicijalizacija $h = 1 - R$, a $d < 0$ postaje $h < \frac{1}{4}$. Međutim, budući da je h inicijaliziran kao cijeli broj i uvećava se samo za cijele brojeve, uspoređivati možemo $h < 0$. Sad smo dobili cjelobrojni algoritam prikazan u nastavku. Umjesto varijable h ipak koristimo varijablu d radi konzistentnosti.

```
void srednja_tocka_kruznica(int radius)
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    tocke_kruznice(x, y);
    while(y > x){
        if(d < 0)
            d += 2.0*x  + 3.0;
        else{
            d += 2.0*(x - y) + 5.0;
            y--;
        }
        x++;
        tocke_kruznice(x, y);
    }
}
```

Taj algoritam možemo ubrzati rješimo li se računanja linearnih delta funkcija. To možemo napraviti tako da izračunamo funkciju dvije uzastopne točke, izračunamo njihovu razliku, koja je za polinome uvijek polinom nižeg reda, i dodamo razliku u svakoj iteraciji.

Ako smo u trenutno iteraciji odabrali piksel **I**, točka evaluacije se pomiče s (x_P, y_P) na $(x_P + 1, y_P)$. Već smo izračunali da je vrijednost $\Delta_{\mathbf{I}_{stari}}$ u (x_P, y_P) jednaka $2x_P + 3$. Dobivamo da je vrijednost

$$\Delta_{\mathbf{I}_{novi}} \text{ u točki } (x_P + 1, y_P) = 2(x_P + 1) + 3.$$

Razlika je $\Delta_{\mathbf{I}_{novi}} - \Delta_{\mathbf{I}_{stari}} = 2$.

Slično, imali smo da je $\Delta_{\mathbf{JI}_{stari}}$ u (x_P, y_P) jednaka $2x_P - 2y_P + 5$, novu dobijemo:

$$\Delta_{\mathbf{JI}_{novi}} \text{ u točki } (x_P + 1, y_P) = 2(x_P + 1) - 2y_P + 5$$

Razlika je $\Delta_{\mathbf{JI}_{novi}} - \Delta_{\mathbf{JI}_{stari}} = 2$.

S druge strane, na ako je odabrani piksel bio **JI**, točka evaluacije se pomakla iz (x_P, y_P) u $(x_P + 1, y_P + 1)$ pa na isti način kao prije dobivamo da su razlike $\Delta_{\mathbf{I}_{novi}} - \Delta_{\mathbf{I}_{stari}} = 2$ i $\Delta_{\mathbf{JI}_{novi}} - \Delta_{\mathbf{JI}_{stari}} = 4$

Modificirani algoritam sad se sastoji od četiri koraka:

1. biranje piksela temeljeno na predznaku varijable odluke d
2. ažuriranje d s $\Delta_{\mathbf{I}}$ ili $\Delta_{\mathbf{JI}}$ izračunatih u prošloj iteraciji

3. ažuriranje $\Delta_{\mathbf{I}}$ i $\Delta_{\mathbf{JI}}$ korištenjem izračunatih inkremenata

4. aktiviranje piksela i prebacivanje na sljedeću iteraciju

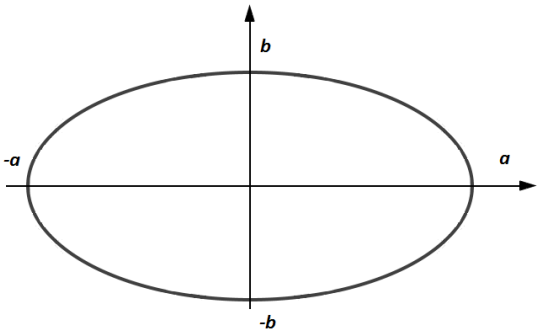
Kod finalnog algoritma je:

```
void srednja_tocka_kruznica(int radius)
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    int deltaI = 3;
    int deltaJI = -2*radius + 5;
    tocke_kruznice(x,y);
    while(y>x){
        if(d<0)
            d += deltaI
            deltaI += 2;
            deltaJI += 2;
        else{
            d += deltaJI;
            deltaI += 2;
            deltaJI += 4;
            y--;
        }
        x++;
        tocke_kruznice(x, y);
    }

}
```

2.4 Scan konverzija elipse

Sličnim postupkom poslužit ćemo se pri računanju piksela elipse.



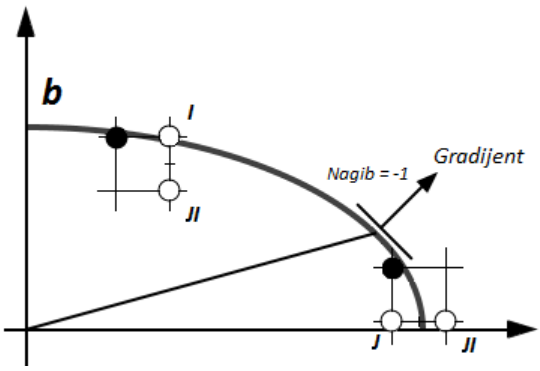
Slika 2.9: Elipsa s poluosima duljine a i b

Implicitna jednadžba elipse s centrom u (0,0) kakvu vidimo na slici 2.9 je:

$$F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0$$

Duljina glavne osi elipse duž x osi iznosi $2a$, a duljina sporedne iznosi $2b$. Kako bismo pojednostavili algoritam, njime računamo samo piksele prvog kvadranta, a ostale crtamo preko simetrija. Standardne elipse koje nemaju centar u ishodištu mogu biti prikazane jednostavnom translacijom nakon obavljene scan konverzije. Ovdje predstavljen algoritam temelji se na algoritmu Da Silve [5].

Podijelit ćemo prvi kvadrant na dvije regije koje su odvojene točkom u kojoj krivulja ima nagib -1. To radimo kako bismo ponovno mogli u svakoj iteraciji birati između dva piksela.



Slika 2.10: Dio elipse s točkom na elipsi u kojoj je nagib tangente -1

Određivanje ove točke nešto je kompleksnije nego kod kružnice. Poslužit ćemo se vektorom okomitim na tangentu krivulje u točki, gradijentom:

$$gradF(x,y) = \frac{\delta F}{\delta x}\vec{i} + \frac{\delta F}{\delta y}\vec{j} = 2b^2x\vec{i} + 2a^2y\vec{j}$$

Granica između dvije regije je točka u kojoj je nagib -1. U toj je točki vrijednost normiranog gradijenta jednaka (1,1). Komponenta $\vec{j}$ gradijenta je veća od komponente $\vec{i}$ u regiji 1, a obrnuto vrijedi za regiju 2. Kad algoritam dođe do točke $P(x_P, y_P)$ kod koje za sljedeću vrijednost varijable odluke $(x_P + 1, y_P - \frac{1}{2})$ vrijedi da je $a^2(y_P - \frac{1}{2}) \leq b^2(x_P + 1)$ (dakle komponenta $\vec{i}$ veća od komponente $\vec{j}$) prebacujemo se na regiju 2.

Slično kao u prijašnjim algoritmima, odlučujemo se za sljedeći piksel na temelju toga koji je bliži elipsi i pri tome koristimo varijablu odluke d.

U regiji 1 izbor će biti između piksela **I** i **JI**. Ako u određenoj iteraciji imamo prethodnu vrijednost d_{stari} u točki $(x_P + 1, y_P - \frac{1}{2})$:

$$d_{stari} = F(x_P + 1, y_P - \frac{1}{2}) = b^2(x_P + 1)^2 + a^2(y_P - \frac{1}{2})^2 - a^2b^2$$

Za piksel **I** računamo:

$$d_{novi} = F(x_P + 2, y_P - \frac{1}{2}) = b^2(x_P + 2)^2 + a^2(y_P - \frac{1}{2})^2 - a^2b^2$$

Budući da je $d_{novi} = d_{stari} + b^2(2x_P + 3)$, pomak u smjeru **I** će biti $\Delta_I = b^2(2x_P + 3)$.

Za piksel **JI** računamo:

$$d_{novi} = F(x_P + 2, y_P - \frac{3}{2}) = b^2(x_P + 2)^2 + a^2(y_P - \frac{3}{2})^2 - a^2b^2$$

Budući da je $d_{novi} = d_{stari} + b^2(2x_P + 3) + a^2(-2y_P + 2)$, pomak u smjeru **JI** će biti $\Delta_{JI} = b^2(2x_P + 3) + a^2(-2y_P + 2)$.

U regiji 2, ako je trenutni piksel (x_P, y_P), varijabla odluke d_2 je $F(x_P + \frac{1}{2}, y_P - 1)$ i provodimo sličnu računicu kao za regiju 1.

Preostaje nam odrediti inicijalne uvjete. Elipsa počinje u točki (0,b) pa je prvo što računamo:

$$F(1, b - \frac{1}{2}) = b^2 + a^2(b - \frac{1}{2})^2 - a^2b^2 = b^2 + a^2(-b + \frac{1}{4})$$

Za svaku iteraciju u regiji 1, osim testiranja varijable odluke d_1 i računanja pomaka, moramo provjeriti trebamo li promjeniti regiju evaluirajući gradijent. Prelaskom u regiju 2 mjenjamo izbor sljedećih piksela s **I** i **JI** na **JI** i **J**. Ako se posljednji piksel koji crtamo u regiji 1 nalazi na (x_P, y_P) inicijaliziramo varijablu d_2 na $F(x_P + \frac{1}{2}, y_P - 1)$. Prestajemo crtati u regiji 2 kad y postane jednak nuli.

```
void elipsa (int a, int b)
{
    int a2 = a * a;
    int b2 = b * b;

    int x = 0;
    int y = b;

    /* Crtamo prve piksele u kvadrantima */
    elipsa_tocke(x,y);

    /* Regija 1 */
    int d1 = round (b2 - (a2 * b) + (0.25 * a2));
    /* Testiramo gradijent */
    while (a2 * (y - 0.5) > b2 * (x + 1)) {
        if (d1 < 0)
            d1 += b2 * (2*x + 3);
        else {
            d1 += b2 * (2*x + 3) + a2 * (-2*y + 2);
            y--;
        }
        x++;
        elipsa_tocke(x,y);
    }

    /* Regija 2 */
    int d2 = round(b2*(x + 0.5)*(x + 0.5) + a2*(y - 1)*(y - 1) - a2*b2);
    while (y > 0) {

        if (d2 > 0)
            d2 += a2*(-2*y + 3);
        else {
            d2 += b2*(2*x + 2) + a2*(-2*y + 3);
            x++;
        }
        y--;
        elipsa_tocke(x,y);
    }
}
```

2.5 Ispunjavanje pravokutnika

Neispunjeni pravokutnik crtamo uz pomoć Bresenhamovog algoritma. Potrebno je nacrtati samo četiri linije. Za ispunjeni pravokutnik potrebno je aktivirati sve unutrašnje piksele.

Jednostavan algoritam za ispunjavanje pravokutnika kojem su stranice paralelne s osima dan je u nastavku. Dijagonala pravokutnika ima krajnje točke (x_{min}, y_{min}) i (x_{max}, y_{max}).

```
void ispunjeni_pravokutnik (int x_min, int x_max, int y_min, int y_max){
    for(int y = y_min; y <= y_max; y++)
        for(int x = x_min; x <= x_max; x++)
            aktiviraj_piksel(x,y);
}
```

Problem kod ovog algoritma je slučaj u kojem dva takva pravokutnika dijele stranicu. Koristeći ovaj algoritam prvo na jednom pa na drugom pravokutniku, aktivirali bismo piksele na zajedničkoj stranici dva puta. Taj problem je dio jednog većeg: definiranje koji su sve pikseli dio neke primitive. Očito su oni koji se nalaze u unutrašnjosti primitive dio te primitive. Problem nastaje kod piksela na rubovima primitive. Da se izbjegne aktiviranje piksela rubova dva ili više puta, potrebno je definirati neko pravilo koje jedinstveno dodjeljuje rubne piksele primitivama. Jedno jednostavno rješenje za pravokutnike je: piksel koji se nalazi na stranici pravokutnika nije dio pravokutnika ako poluravnina definirana tom stranicom, koja sadrži primitivu, leži lijevo ili ispod stranice. Pikseli koji leže na lijevoj ili donjoj stranici će se iscrtati, a na desnoj ili gornjoj neće. Vertikalna zajednička stranica stoga pripada desnijem od dva pravokutnika.

Nekoliko svojstava ovog pravila:

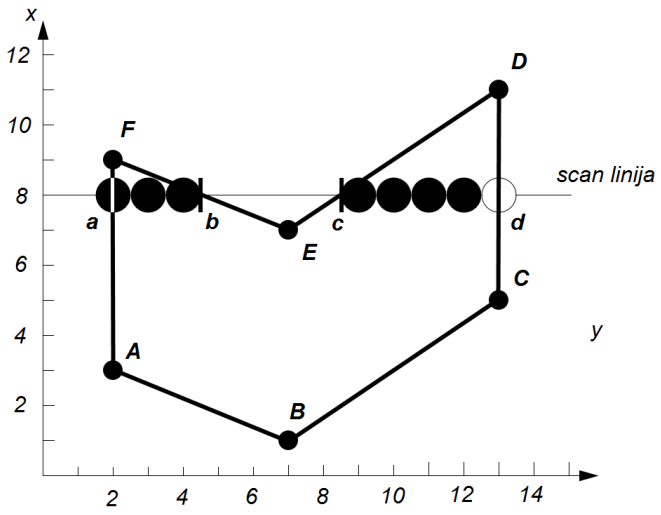
- primjenjivo je na mnogokute
- donji lijevi vrh pravokutnika se još uvijek crta dvaput pa je potrebno posebno pravilo za taj specijalan slučaj
- svaki pravokutnik ostaje bez gornjeg i desnog ruba

Vidimo da ovo pravilo nije savršeno, ali se danas najviše koristi zbog jednostavnosti.

2.6 Ispunjavanje mnogokuta

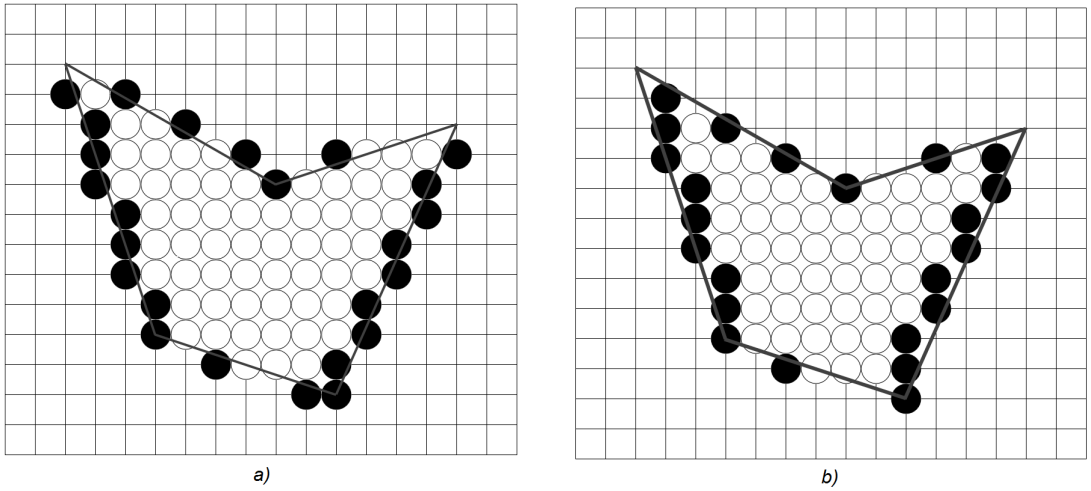
U ovom odjeljku koristit će se pojam *raspona*. Raspon je više piksela koji se u istom retku nalaze jedan za drugim koje možemo zajedno aktivirati. Prvi piksel raspona s lijeve strane i prvi s desne nazivaju se vanjski pikseli raspona.

Algoritam scan konverzije za ispunjavanje mnogokuta opisan u ovom odjeljku primjenjiv je na konveksne mnogokute, konkavne mnogokute, mnogokute koji sami sebe sijeku i koji sadrže rupe. Algoritam nalazi raspone između lijevih i desnih stranica mnogokuta. Vanjski pikseli raspona računaju se kao sjecišta scan linije sa stranicama mnogokuta. *Scan linija* je zamišljena linija koja prolazi kroz središta piksela u istom retku. Slika 2.11 prikazuje mnogokut i jednu scan liniju koja prolazi kroz mnogokut. Sjecišta scan linije sa stranicama FA i CD leže na cjelobrojnim koordinatama dok sjecišta sa EF i DE leže na decimalnim.



Slika 2.11: Mnogokut i scan linija. Pikseli na scan liniji koji pripadaju mnogokutu su označeni crno

Potrebno je odrediti koji su pikseli na svakoj scan liniji dio mnogokuta i aktivirati ih. Ponavljanjem postupka za svaku scan liniju možemo izvršiti scan konverziju cijelog mnogokuta.



Slika 2.12: Dva načina na koja možemo promatrati piksele na rubovima mnogokuta

Slike 2.12 prikazuju vanjske piksele raspona crno, a unutrašnje bijelo. Na slici 2.12(a) rasponi uključuju i piksele koji se nalaze izvan poligona, koji bi bili aktivirani koristimo li primjerice Bresenhamov algoritam za linije na stranicama mnogokuta. Na slici 2.12(b) aktivirani su samo pikseli koji su dio mnogokuta. To je preferirani način scan konverzije budući da ne želimo više puta aktivirati isti piksel.

Kao u Bresenhamovom algoritmu, želimo nekako iskoristiti prethodnu iteraciju - u ovom slučaju prethodnu scan liniju - za računanje vanjskih piksela raspona.
Algoritam će se sastojati od nekoliko koraka:

1. pronađi sjecišta scan linije sa svim stranicama mnogokuta
2. sortiraj sjecišta uzlazno po x koordinati
3. aktiviraj piksele između svakog drugog para sjecišta koji su u unutrašnjosti mnogokuta koristeći bit kojim se pamti parnost: 1 crtaj, 0 ne crtaj - počinje od 0

Prva dva koraka obradit ćemo kasnije i za sad se pozabaviti trećim korakom. Nekoliko se pitanja nameću za teći korak:

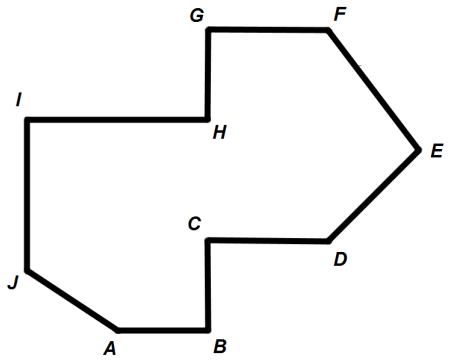
1. Kako odrediti koji je piksel unutrašnji kada je x neka decimalna vrijednost?
2. Što napraviti sa specijalnim slučajem sjecišta na cjelobrojnim koordinatama?
3. Što napraviti sa specijalnim slučajem pod 2. za vrh mnogokuta?
4. Što napraviti sa specijalnim slučajem pod 2. ako je taj vrh - vrh horizontalne stranice?

Odgovor 1: kretanjem u desno po scan liniji, ako smo u mnogokutu i naletimo na decimalno sjecište, zaokružujemo na nižu vrijednost; ako smo izvan mnogokuta, zaokružujemo na višu. Odgovor 2: ako lijevi vanjski piksel rasponu ima cjelobrojnu x koordinatu, definiramo ga kao unutarnji piksel mnogokuta; ako desni vanjski piksel raspona ima cjelobrojnu x koordinatu definiramo ga kao vanjski piksel mnogokuta. Kako bi odgovorili na posljednja dva pitanja, definirat ćemo za određeni vrh mnogokuta da je taj vrh y_{min} za neku stranicu ako je on krajnja točka stranice sa strogo manjom vrijednosti y od druge krajnje točke stranice, a y_{max} ako je ima strogo veću vrijednost y. Odgovor 3: brojimo y_{min} vrh stranice kad računamo parnost, ali ne i y_{max} vrh, tako da se y_{max} vrh crta samo ako je on u isto vrijeme i y_{min} za drugu stranicu kojoj je taj vrh krajnja točka. Npr. vrh A na slici 2.11 je u isto vrijeme y_{min} vrh stranice AF i y_{max} vrh stranice AB pa se broji samo jednom pri računanju parnosti raspona. Odgovor 4: ne brojimo horizontalnu stranicu kad razmatramo koliko vrh doprinosi parnosti.

2.6.1 Horizontalne stranice

Razmotrit ćemo razne slučajeve na lici 2.13 da vidimo zašto ovi odgovori zadovoljavaju. Za horizontalnu stranicu AB vrh A je y_{min} vrh stranice JA, a stranica AB je horizontalna pa ne doprinosi. Bit parnosti se mjenja u 1 i raspon AB je nacrtan. Vertikalna stranica BC ima y_{min} u vrhu B, opet AB ne doprinosi. Bit parnosti se mjenja u 0 i u B se završava raspon koji crtamo. U vrhu J, stranica IJ ima y_{min} vrh, a JA ima y_{max}, bit se mjenja na 1 i crtamo raspon do stranice BC. Kod raspona koji počinje na stranici IJ nema promjene za vrh C jer je C y_{max} vrh strance BC pa se nastavlja do D kad prestaje jer je D y_{min} vrh stranice DE. U vrhu I stranica IJ ima y_{max} vrh, a HI ne doprinosi - bit ostaje 0 i ne iscrtava se HI. U vrhu H stranica GH ima y_{min} pa se raspon od H do stranice FE iscrtava.

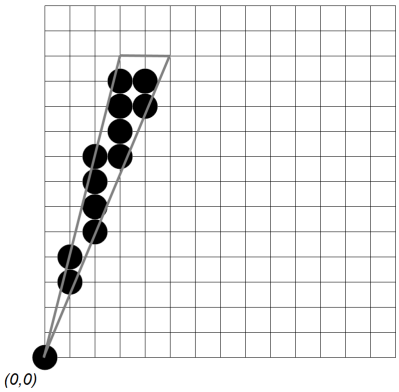
Rezultat je da se gornje stranice ne iscrtavaju, slično pravilu za pravokutnike.



Slika 2.13: Mnogokut s nekoliko horizontalnih stranica

2.6.2 Iveri

Još jedan problem algoritma scan konverzije su iveri. Kada stranice mnogokuta leže dovoljno blizu jedna drugoj stvaraju iver - dio mnogokuta toliko tanak da ne sadrži raspon za svaku scan liniju.



Slika 2.14: Iver nastao pri scan konverziji trokuta s vrhovima (0,0), (3,12) i (5, 12)

Zbog pravila da samo oni pikseli koji leže u unutrašnjosti ili na lijevoj te donjoj stranici, biti će puno scan linija samo s jednim pikselom ili bez ijednog piksela. Problem piksela koji nedostaju je jedan primjer aliasing problema o kojem će više biti riječi u zadnjem poglavlju. Jedan od načina rješavanja tog problema, je ublažavanje pravila i dopuštanje da pikseli na rubovima primitive ili čak vanjski pikseli primitive dobiju intenzitet koji se zasniva na njihovoj udaljenosti od primitive. U tom slučaju više primitiva može doprinositi intenzitetu jednog piksela.

2.6.3 Koherentnost stranica

Korak 1. u algoritmu - računanje sjecišta - mora biti odrađeno brzo. Stoga želimo izbjeći testiranje svake stranice mnogokuta sa svakom scan linijom kako bismo vidjeli postoji li sjecište. Često je bitno provjeriti samo nekoliko stranica za određenu scan liniju. Također, koristit ćemo *koherentnost* stranice: velika je vjerojatnost da većina stranica koje sjeku neku scan liniju i također sjeku scan liniju $i + 1$. Kako s jedne scan linije prelazimo na sljedeću, možemo izračunati x koordinatu novog sjecište na temelju prethodnog, slično kao računanje sljedećeg piksela u Bresenhamovom algoritmu, koristeći:

$$x_{i+1} = x_i + \frac{1}{m}$$

pri čemu je m nagib stranice. U Bresenhamovom algoritmu za scan konverziju linija, izbjegli smo aritmetiku pomičnog zareza računajući cjelobrojnu varijablu odluke i provjeravajući njen predznak za biranje piksela najbližeg liniji. To želimo napraviti i u ovom algoritmu. Razmotrit ćemo lijeve stranice nagiba većeg od 1. Desne stranice i stranice s drugačijim nagibima se na sličan način obrađuju. Horizontalne stranice se rješavaju u zadnjem koraku, a vertikalne su specijalni slučaj.

U (x_{min}, y_{min}) crtamo piksel. Povećavajući y, x koordinata točke na liniji povećava se za $\frac{1}{m}$, pri čemu je

$$m = \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$$

nagib linije.

Rezultat povećanja je da x dobiva decimalni dio koji se može prikazati kao razlomak s nazivnikom $y_{max} - y_{min}$. Kad je decimalni dio x-a jednak nuli, crtamo piksel (x, y) koji leži na liniji, dok za pozitivan decimalni dio x-a zaokružujemo na veći broj kako bi dobili piksel strogo u unutrašnjosti mnogokuta. Kad decimalni dio postane veći od 1, oduzmemo 1 i pomičemo se jedan piksel u desno.

Možemo izbjeći aritmetiku pomičnog zareza pri računanju decimalnog dijela x-a prateći samo vrijednost brojnika i činjenicu da je decimalni dio veći od 1 ako je brojnik veći od nazivnika. Implementiramo ovu tehniku u algoritmu:

```
void scan_lijeve_stranice (int xmin,int ymin, intxmax, int ymax){
    int x = xmin;
    int brojnik = xmax - xmin;
    int nazivnik = ymax  - ymin;
    int inkrement = nazivnik;
    for(int y = ymin; y <= ymax; y++){
        aktiviraj_piksel(x,y);
        inkrement += brojnik;
        if(inkrement > nazivnik){
            x++;
            inkrement -= nazivnik;
        }
    }
}
```

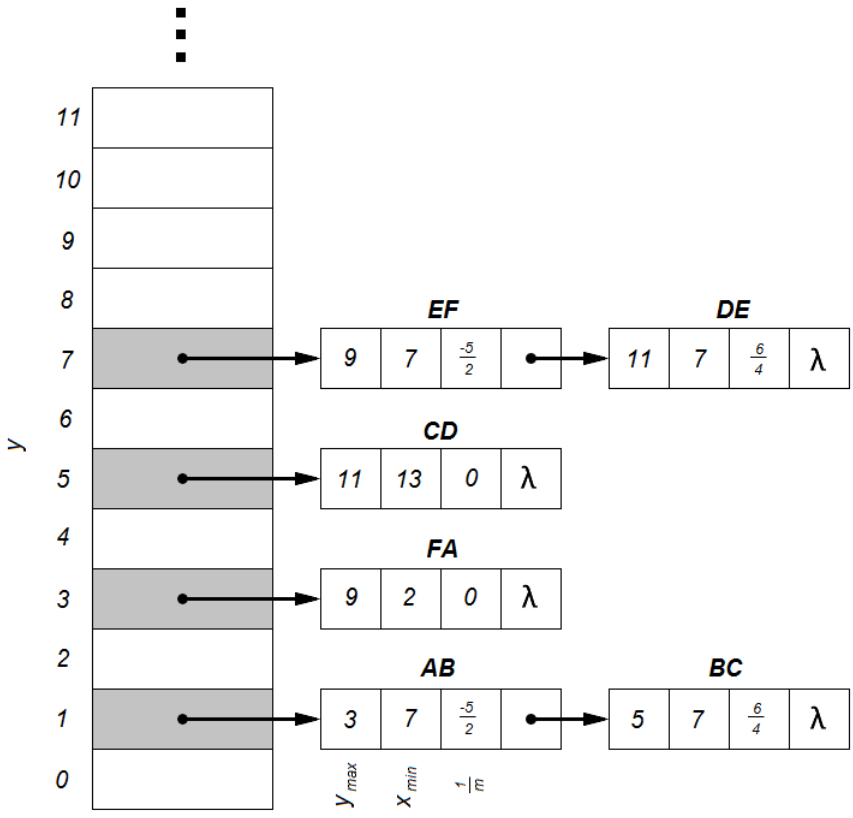
2.6.4 Algoritam scan linija

Sada razvijamo algortiam scan linija koji koristi koherentnost stranice i za svaku scan liniju sprema stranice koje ona sječe i sjecišta s tim stranicama u strukturu podataka koja se naziva *tablica aktivnih stranica* (*Active-Edge Table* - u daljnjem tekstu AET).

Stranice u AET su sortirane po x vrijednostima sjecišta tako da možemo iscrtavati raspone definirane parovima sjecišta. Pomakom na novu scan liniju, $y + 1$, ažuriramo

AET. Prvo se stranice koje su u AET, ali nisu presjećene scan linijom, tj. one kojima je $y_{max} = y$, brišu. Nakon toga se svaka nova stranica presječena sljedećom scan linijom, tj. ona kojoj je $y_{min} = y + 1$, dodaje u AET. Konačno, nove x vrijednosti sjecišta se računaju koristeći inkrementalni algoritam za stranice u AET-u.

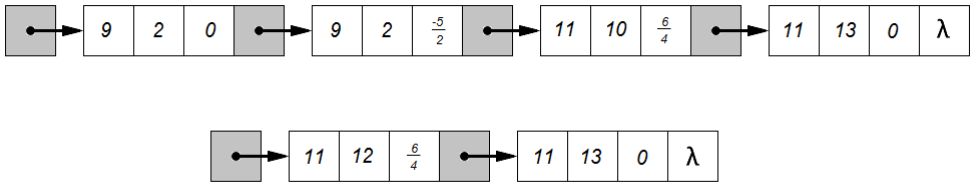
Kako bismo efikasno izveli dodavanje stranica AET-u, stvaramo globalnu *tablicu stranica* (*Edge Table* - u daljnjem tekstu ET) koja sadrži sve stranice sortirane uzlazno po y_{min}. ET se obično stvara koristeći *segmentno sortiranje* (*bucket sort*) s onoliko segmenata koliko je scan linija. U svakom segmentu čuvamo stranice u redosljedu povećanja x koordinate donje krajnje točke. Svaki element ET tablice sadrži y_{max} koordinatu stranice, x koordinatu donje krajnje točke x_{min} i x inkrement koji se koristi pri prelasku na novu scan liniju, $\frac{1}{m}$. Slika 2.15 prikazuje kako bi šest stranica mnogokuta sa slike 2.11 bilo sortirano, a slika 2.16 prikazuje AET za scan linije 9 i 10 za isti mnogokut (u pravoj implementaciji bismo vjerojatno još dodali bit koji nam označava radi li se o lijevoj ili desnoj stranici mnogokuta).



Slika 2.15: ET tablica

Jednom kad imamo ET, koraci algoritma scan linija su:

1. Postavi y na najmanju y koordinatu elementa u prvom nepraznom segmentu ET-a.
2. Inicijaliziraj AET kao praznu tablicu.
3. Ponavljaj dok AET i ET ne ostanu prazni:
   a) Prebaci u AET elemente iz ET za koje je $y_{min} = y$.
   b) Makni iz AET one elemente za koje $y = y_{max}$ (stranice koje nisu uključene u sljedeću scan liniju) i sortiraj AET po x (olakšano time što je ET predsortiran).
   c) Aktiviraj željene piksele na scan liniji y koristeći parove x koordinata iz AET.
   d) Povećaj y za 1, tj. na koordinatu sljedeće scan linije.
   e) Za svaku preostalu stranicu u AET-u koja nije vertikalna, ažuriraj x za novi y.



Slika 2.16: AET tablica

Ovaj algoritam koristi koherentnost stranice za računanje x koordinata sjecišta dok računanje raspone koristi činjenicu da u većini slučajeva nema velike razlike između raspona uzastopnih scan linija. Budući da se sortira mali broj stranica i ponovno sortiranje u koraku 3.a) se radi na uglavnom sortiranoj listi, koristi se *sortiranje umetanjem* (*insertion sort*) ili *bubble sort* koji će u ovom slučaju biti efektivno složenosti O(N). Trokuti i trapezi mogu biti tretirani kao specijalni slučajevi jer imaju maksimalno 2 stranice koje svaka scan linija sječe.

Scan konverzija neispunjenih mnogokuta je samo korištenje Bresenhamova algoritma na njihovim stranicama uz ispitivanje nekoliko specijalnih slučajeva.

Poglavlje 3

Obrezivanje

Kao što je spomenuto na početku, obrezivanje (*clipping*) je odbacivanje primitiva koje se nalaze izvan granica ekrana. Obrezivanje možemo provoditi prije ili za vrijeme scan konverzije. Ono što želimo postići je da uz što manje potrebnih računalnih resursa i vremena odbacimo dijelove one primitive ili dijelove primitiva koje nećemo prikazati na ekranu. Općenito je obrezivanje moguće uz bilo kakav geometrijski lik, ali ekran računala je najćešće pravokutan pa će algoritmi koje ćemo obraditi za obrezivanje koristiti pravokutnik. Taj ćemo pravokutnik nazivati pravokutnik za obrezivanje, pravokutnik obrezivanja ili jednostavno pravokutnik kada je jednoznačno određen.

Kao i kod scan konverzije, bitno nam je obrezivanje linija jer je većina primitiva po dijelovima linija. No prije nego što krenemo s obrezivanjem linija, razmotrit ćemo problem obrezivanja pojedinačnih točaka.

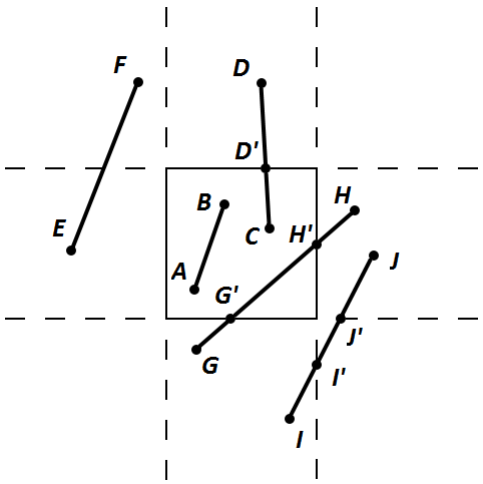
U cijelom poglavlju vrhovi pravokutnika obrezivanja biti će označeni: $T_1(x_{min}, y_{max})$, $T_2(x_{max}, y_{max})$, $T_3(x_{max}, y_{min})$, $T_4(x_{min}, y_{min})$.

Da bi se točka nalazila unutar pravokutnika moraju biti zadovoljene četiri nejednakosti:

$$x_{min} \leqslant x \leqslant x_{max} \qquad i \qquad y_{min} \leqslant y \leqslant y_{max}.$$

Svaku točku i općenito svaku primitivu izvan pravokutnika obrezivanja odbacujemo i ne razmatramo kod scan konverzije.

3.1 Obrezivanje linija



Slika 3.1: Regije obrezivanja

Da bismo obrezali liniju moramo promotriti samo njene krajnje točke.

- Ako obje leže unutar pravokutnika (linija AB na slici 3.1), cijela linija leži unutar pravokutnika i možemo ju trivijalno prihvatiti.
- Ako jedna krajnja točka leži izvan, a druga unutar pravokutnika (linija CD na slici 3.1), linija siječe pravokutnik i moramo računati koordinate sjecišta.
- Ako obje točke leže izvan, linija može sjeći pravokutnik (EF, GH, IJ) te moramo dodatno provjeriti postoje li sjecišta.

Za liniju koju ne možemo trivijalno prihvatiti možemo koristiti jednostavan algoritam koji nalazi sjecišta linije sa svakim pravcem na kojem leži neka stranica pravokutnika i provjerava leži li barem jedno od tih sjecišta na nekoj stranici jer bi to značilo da se dio linije nalazi unutar pravokutnika.

Ovaj pristup zahtjeva korištenje parametarskog zapisa linije kako bismo mogli opisati i vertikalne linije:

$$\begin{aligned} x &= x_0 + t(x_1 - x_0) \\ y &= y_0 + t(y_1 - y_0) \end{aligned} \tag{3.1}$$

Za nalaženje rješenja $t_{stranica}$ i t_{linija}, potrebno je rješiti dva sustava kao 3.1 koji, između ostalog, sadrže množenja i dijeljenja, nakon čega je još potrebno provjeriti jesu li $t_{stranica}$

i t_{linija} u segmentu $[0, 1]$, odnosno leži li sjecište na stranici i na liniji. Taj pristup zahtjeva previše neefikasnog računanja. Vidjet ćemo da postoje efikasniji načini rješavanja tog problema.

3.2 Cohen–Southerland-ov algoritam

U ovom dijelu riječ stranica može predstavljati i pravac na kojoj stranica pravokutnika leži. Cohen-Southerlandov algoritam se sastoji od nekoliko koraka. Prvo se izvode inicijalni testovi na liniji kako bi se utvrdilo je li uopće potrebno računati sjecišta.

1. korak je testiranje nalaze li se obje krajnje točke u pravokutniku. U tom slučaju liniju trivijalno prihvaćamo. Ako se obje točke nalaze u istoj vanjskoj poluravnini, tj. ona poluravnina koja je definirana stranicom pravokutnika i koja ne sadrži pravokutnik, liniju trivijalno odbijamo.

2. korak je, u slučaju da su inicijalni testovi vratili negativne odgovore, nalaženje sjecišta sa stranicama pravokutnika obrezivanja i odbacivanje dijelova linije nakon čega na ostatku linije provodimo algoritam od početka.

Ravninu dijelimo na 4 poluravnine koje ćemo označavati po stranama svijeta: **S** (sjever), **I** (istok), **J** (jug), i **Z** (zapad).

- **S** označava vanjsku poluravninu određenu gornjom ili sjevernom stranicom pravokutnika obrezivanja.

- **I** označava vanjsku poluravninu određenu desnom ili istočnom stranicom pravokutnika obrezivanja.

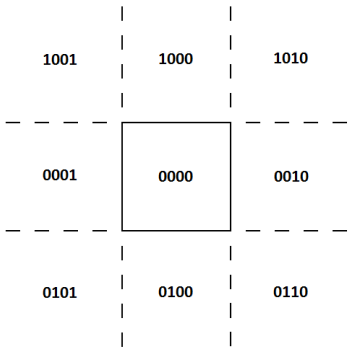
- **J** označava vanjsku poluravninu određenu donjom ili južnom stranicom pravokutnika obrezivanja.

- **Z** označava vanjsku poluravninu određenu lijevom ili zapadnom stranicom pravokutnik aobrezivanja.

U prvom koraku npr. obje krajnje točke linije EF sa slike 3.1 upadaju u **Z** poluravninu jer im je x koordinata manja od x_{min} pa tu liniju možemo trivijalno odbaciti. Slično tome, možemo trivijalno odbaciti linije kojima obje krajnje točke leže u poluravninama **S** ($y > y_{max}$), **I** ($x > x_{max}$) i **J** ($y < y_{min}$).

U drugom koraku dijelimo liniju u dva segmenta odvojena njenim sjecištem s pravcem na kojim leži stranica pravokutnika. Dio linije na unutarnjoj poluravnini stranice postaje linija koju opet testiramo od prvog koraka i postupak ponavljamo dok ne dobijemo dio linije koji možemo trivijalno prihvatiti ili trivijalno odbaciti.

Četiri spomenute poluravnine određuju 9 regija na ravnine kao što vidimo na slici 3.2.



Slika 3.2: Kodovi regija

Svakoj regiji pridružen je 4-bitni kod. Bitovi su postavljen na 1 (istina) ili 0 (laž) što odgovara sljedećim uvjetima:

prvi bit	točka se nalazi u **S** poluravnini	$y > y_{max}$
drugi bit	točka se nalazi u **J** poluravnini	$y < y_{min}$
treći bit	točka se nalazi u **I** poluravnini	$x > x_{max}$
četvrti bit	točka se nalazi u **Z** poluravnini	$x < x_{min}$

Primjerice, regija koja je presjek **S** i **Z** poluravnina ima kod 1001.

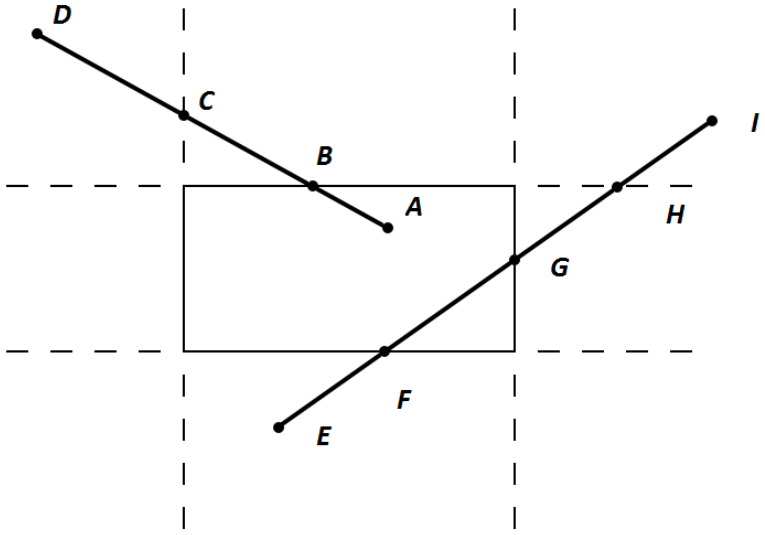
Efikasan način računanja koda točke je korištenje bita predznaka u računalnom zapisu cijelog broja.

- Prvi bit je bit predznaka od $y_{max} - y$ (1 ako je razlika negativna)
- Drugi bit je bit predznaka od $y - y_{min}$
- Treći bit je bit predznaka od $x_{max} - x$
- Četvrti bit je bit predznaka od $x - x_{min}$.

Svakoj krajnjoj točki prvo izračunamo kod. Sljedeći je korak od dva dobivena koda izračunati njihov logički OR i AND:

- Ako logički OR kodova krajnjih točaka vraća kod 0000 to znači da su obje točke unutar pravokutnika i liniju možemo trivijalno prihvatiti.
- Ako logički AND kodova krajnjih točaka vraća kod s barem jednom jedinicom to znači da su obje točke u istoj vanjskoj poluravnini i liniju možemo trivijalno odbaciti.
- U ostalim slučajevima dijelimo liniju u dva segmenta.

Podjelu radimo pomoću stranica koju linija siječe: dio koji se nalazi u vanjskoj poluravnini se odbacuje. Pomaže nam činjenica da bitovi koji su jedinice predstavljaju presječene stranice. Moguće je koristiti bilo koji redosljed ispitivanja presjećenih stranica, ali potrebno je pripaziti da se tijekom cijelog algoritma koristiti isti redosljed. Koristit ćemo redosljed gore-dolje, desno-lijevo koji smo već odabrali pri definiranju bitova i promatrati bitove s lijeva na desno.



Slika 3.3: Primjer obrezivanja dvije linije

Primjer 1: Promotrimo liniju AD na slici 3.3 Kod točke A je 0000, a kod točke D je 1001. Koristimo OR i AND testove i zaključujemo da liniju ne možemo trivijalno odbaciti niti prihvatiti. Algoritam odabire točku D kao vanjsku točku linije jer iz njenog koda vidimo da linija siječe gornju i lijevu stranicu. Po odabranom redosljedu testiranja prvo odbacimo dio linije u S poluravnini. Ono što je preostalo je linija AB. Točki A ne moramo ponovno računati kod, točki B računamo kod 0000. U sljedećoj iteraciji inicijalni testovi trivijalno prihvaćaju liniju AB.

Primjer 2: Obrezivanje linije EI zahtjeva nekoliko iteracija. Kod točke E je 0100 pa je algoritam odabire kao vanjsku točku i obrezuje ju s donjom stranicom na liniju FI. Druga iteracija također trivijalno ne prihvaća niti odbija novu liniju. Kod točke F je 0000, a kod točke I je 1010. Točka I je odabrana kao vanjska i obrezuje se gornjom stranicu. Time dobivamo liniju FH koja ponovno ne prolazi inicijalne testove. Točki H je kod 0010 i liniju FH obrezujemo s desnom stranicom nakon čega je FG trivijalno prihvaćena.

```
typedef unsigned int bitovi;
enum{
    S=0x1,
    J=0x2,
    Z=0x4,
    I=0x8
};
void Cohen_Sutherland (
    double x0, double y0,
    double x1, double y1,
    double xmin, double xmax,
    double ymin, double ymax)
/* Obrezuje se linija s tockama (x0,y0) i (x1,y1) prvaokutnikom s
   dijagonalom od (xmin,ymin) do (xmax,ymax) */
{
    bitovi kod0, kod1, kod_vanjske;
    boolean prihvaceno = FALSE, kraj = FALSE;
    kod0 = racunaj_kod(x0, y0, xmin, xmax, ymin, ymax);
    kod1 = racunaj_kod(x1, y1, xmin, xmax, ymin, ymax);
    do {
        if(!(kod0 | kod1)) {      // trivijalno prihvacanje
           prihvaceno=TRUE;
           kraj=TRUE;
        }
        else if (kod0 & kod1){    // trivijalno odbijanje
           kraj = TRUE;
        }
        else {
            double x,y;

            // barem jedna tocka je izvan pravokutnika
            kod_vanjske=kod0 ? kod0 : kod1;

            /* za nalazenje sjecista koristimo formule:
            y=y0+nagib*(x-x0) i
            x=x0+(1/nagib)*(y-y0) */
            if(kod_vanjske & S){
                // podijeli liniju gornjom stranicom
                x = x0 + (x1 - x0)*(ymax - y0)/(y1 - y0);
                y = ymax;
            }
            if(kod_vanjske & J){
                // podijeli liniju donjom stranicom
                x = x0+(x1-x0)*(ymin - y0)/(y1 - y0);
                y = ymin;
            }
            if(kod_vanjske & I){
```

```
                // podijeli liniju desnom stranicom
                y = y0 + (y1 - y0)*(xmax - x0)/(x1 - x0);
                x = xmax;
            }
            else {
                // podijeli liniju lijevom stranicom
                y = y0 + (y1 - y0)*(xmin - x0)/(x1 - x0);
                x = xmin;
            }
            // dobiveno sjeciste zamjenjuje vanjsku tocku
            if(kod_vanjske == kod0){
                x0 = x;
                y0 = y;
                kod0 = racunaj_kod(x0, y0, xmin, xmax, ymin, ymax);
            }
            else {
                x1 = x;
                y1 = y;
                kod1 = racunaj_kod(x1, y1, xmin, xmax, ymin, ymax);
            }
        }
    } while (kraj==FALSE);
    if(prihvaceno)
        bresenham_cjelobrojni(x0,y0,x1,y1);
}
bitovi racunaj_kod (double x, double y, double xmin, double ymin, double
    xmax, double ymax)
{
   bitovi kod=0;
   if(y>ymax)
       kod |= S;
   else
       if(y<ymin)
           kod |= J;

   if(x>xmax)
       kod |= I;
   else
       if(x<xmin)
           kod |= Z;

   return kod;
}
```

3.3 Parametarski algoritam za obrezivanje linija

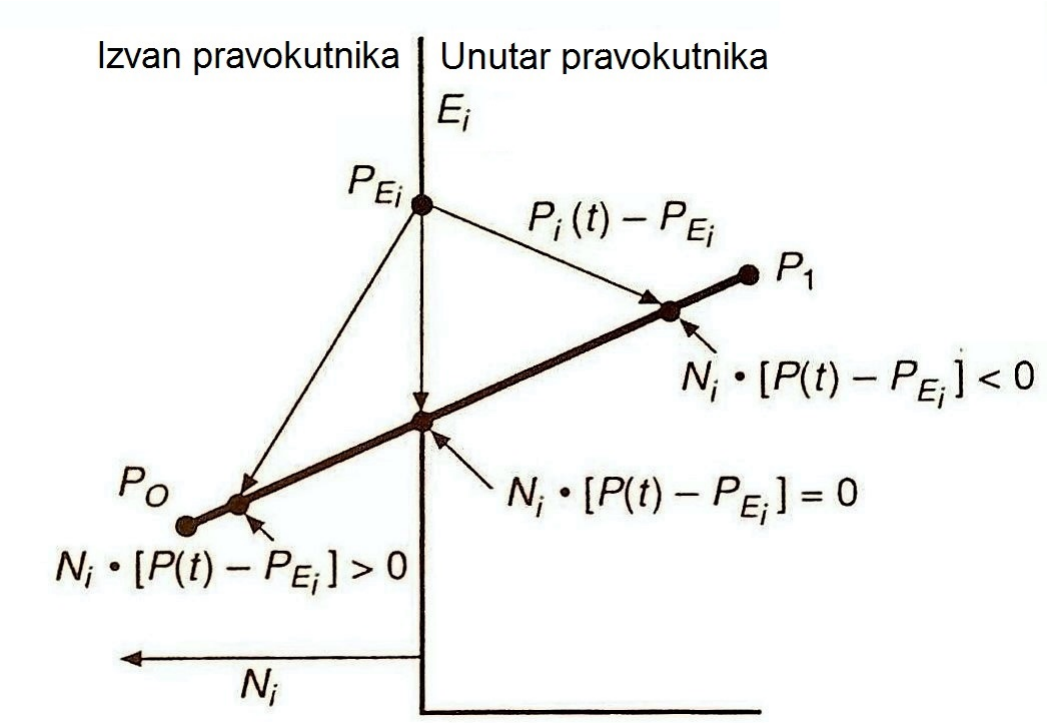
Osim Cohen–Sutherland-ovog algoritma, najčešće se za obrezivanje linija pravokutnikom koristi Liang–Barsky-jev algoritam. To je specijalni slučaj Cyrus–Beck-ovog algoritma za obrezivanje linije konveksnim mnogokutom.

3.3.1 Cyrus–Beck-ov algoritam

Parametarski zapis linije:

$$P(t) = P_0 + (P_1 - P_0)t$$

pri čemu je $t = 0$ u P_0 i $t = 1$ u P_1. Odaberimo proizvoljnu točku P_{E_i} na stranici E_i mnogokuta i promotrimo tri vektora $P(t) - P_{E_i}$ od P_{E_i} do tri točke na liniji P_0P_1.



Slika 3.4: Jednostanim računom možemo odrediti nalazi li se točka linije s vanjske ili s unutarnje poluravnine koju određuje stranica pravokutnika

Te tri točke su: sjecište koje želimo odrediti, krajnja točka linije na unutarnjoj poluravnini stranice (unutar mnogokuta) i krajnja točka linije na vanjskoj poluravnini stranice. Možemo razlikovati u kojem dijelu ravnine se nalazi točka na temelju vrijednosti skalarnog produkta normale N_i stranice E_i s vektorom $P_tP_{E_i}$, odnosno: $N_i \cdot [P(t) - P_{E_i}]$. Ta vrijednost

je negativna za točku na unutrašnjoj poluravnini, 0 za točku na pravcu na kojem se nalazi stranica i pozitivna za točku na vanjskoj poluravnini.

Sada želimo dobiti formulu za t na sjecištu:

$$N_i \cdot [P(t) - P_{E_i}] = 0$$

Uvrstimo P(t):

$$N_i \cdot [P_0 + (P_1 - P_0) \cdot t - P_{E_i}] = 0$$

$$N_i \cdot (P_0 - P_{E_i}) + N_i \cdot (P_1 - P_0) \cdot t = 0$$

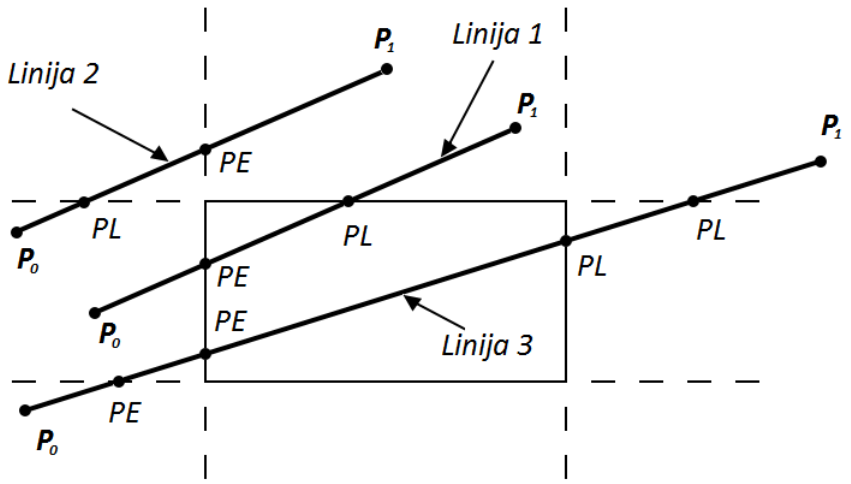
Označit ćemo s $D = (P_1 - P_0)$ vektor od P_0 do P_1 i izraziti t:

$$t = \frac{N_i \cdot (P_0 - P_{E_i})}{-N_i \cdot D} \tag{3.2}$$

Postoje tri slučaja u kojima nazivnik izraza 3.2 može biti nula:

- $N_i = 0$ (normala je 0 - došlo je do greške u algoritmu)
- $D = 0$ ($P_1 = P_0$, radi se o točki)
- $N_i \cdot D = 0$ (stranica E_i i dužina P_0P_1 su paralelne, znači da sjecišta nema pa algoritam nastavlja na sljedeći stranicu mnogokuta)

Izraz 3.2 koristimo za pronalaženje sjecišta linije P_0P_1 i stranica monogokuta. Npr. u slučaju pravokutnika, dobit ćemo četiri vrijednosti t. One koje su unutar segmenta $[0, 1]$ nalaze se na liniji, a ostale zanemarimo. Nakon toga algoritam određuje nalaze li se te točke na stranicama pravokutnika.



Slika 3.5: Određivanje PE i PL sjecišta

Sjecišta na slici 3.5 su označena s PE (*potentially entering* - potencijalno ulazeći) i PL (*potentially leaving* - potencijalno izlazeći): ako "kretanjem" od točke P_0 do točke P_1 linija ulazi u unutrašnju poluravninu stranice sjecište označavamo s PE, a ako izlazimo u vanjsku poluravninu sjecište označavamo s PL. PE i PL sjecišta možemo odrediti preko kuta između P_0P_1 i normale N_i. Ako je kut manji od 90°označujemo s PL, a ako je veći s PE. To se računanje obavlja prilikom računanja nazivnika u izrazu 3.2.

$$N_i \cdot D < 0 \rightarrow \text{PE : kut je veći od } 90^\circ$$
$$N_i \cdot D > 0 \rightarrow \text{PL : kut je manji od } 90^\circ$$

Vidimo na slici 3.5 da je iz linije 3 potrebno obrezati dio koji je između PE sjecišta koji ima najveću vrijednost t (t_E) i PL sjecišta s najmanjom vrijednošću t (t_L). U obzir uzimamo samo točke na liniji, tj. $t \in [0, 1]$ i $t_E < t_L$ (pogledati liniju 2 na slici 3.5 za bolje razumijevanje ovog uvjeta). Sve ostalo odbacujemo. Vrijednosti t_E i t_L dalje koristimo za računanje koordinata sjecišta.

Pseudo kod:

```
unaprijed izračunaj N_i i odaberi P_Ei za svaku stranicu;

za svaku liniju koju obrezujemo {
    ako (P1 == P0)
        nije linija, obrezuj kao točku
    inače {
        t_E = 0;
        t_L = 1;
        za svako potencijalno sjecište sa stranicom {
            ako (N_i · D ≠ 0) {
                računaj t;
                koristi predznak od N_i · D za određivanje PE i PL;
                ako (PE) t_E = max(t_E, t);
                ako (PL) t_L = min(t_L, t);
            {
        {
        ako (t_E > t_L)
            vrati NULL;
        inače
            vrati P(t_E) i P(t_L) kao sjecišta;
    {
{
```

3.3.2 Liang–Barsky-jev algoritam

U tablici u nastavku nalaze se izrazi koje koristi Liang–Barsky-jev algoritam. Točne lokacije točaka na stranicama pravokutnika P_{E_i} nisu važne prilikom računanja pa su označene samo s x i y.

$stranica_i$	normala N_i	P_{E_i}	$P_0 - P_{E_i}$	$t = \frac{N_i \cdot (P_0 - P_{E_i})}{-N_i \cdot D}$
Z	$(-1, 0)$	(x_{min}, y)	$(x_0 - x_{min}, y_0 - y)$	$\frac{-(x_0 - x_{min})}{x_1 - x_0}$
I	$(1, 0)$	(x_{max}, y)	$(x_0 - x_{max}, y_0 - y)$	$\frac{-(x_0 - x_{max})}{x_1 - x_0}$
J	$(0, -1)$	(x, y_{min})	$(x_0 - x, y_0 - y_{min})$	$\frac{-(y_0 - y_{min})}{y_1 - y_0}$
S	$(0, 1)$	(x, y_{max})	$(x_0 - x, y_0 - y_{max})$	$\frac{-(y_0 - y_{max})}{y_1 - y_0}$

U nastavku je dana jedna moguća implementacija algoritma. Procedura poziva funkciju `obrezivanje_t()` koja koristi predznak nazivnika iz (3.2) za određivanje je li sjecište linije i stranice PE ili PL, računa parametarsku vrijednost sjecišta i provjerava postoji li trivijalno odbacivanje kada vrijednost $t_E(t_L)$ prijeđe prethodnu vrijednost $t_L(t_E)$. Spomenuta funkcija također može trivijalno odbaciti liniju ako je ona paralelna sa stranicom i izvan pravokutnika. Samo obrezivanje radi glavna procedura pomičući krajnje točke na posljednje vrijednosti t_E i t_L, ali samo ako je dužina cijela unutar pravokutnika. Ovaj uvjet je zapisan s četiri poziva funkcije `obrezivanje_t()` koja vraća `false` ako je linija odbačena.

```
void liang_barsky(double *x0, double *y0, double *x1,  double *y1,
    boolean *vidljiv)
/* Obrezivanje duzine s krajnjim tockama (x0,y0) i (x1,y1) uspravnim
    pravokutnikom s nasuprotnim vrhovima (xmin,ymin) i (xmax,ymax).
    Koordinate vrhova pravokutnika obrezivanja u ovom slucaju su
    globalne varijable, a mogu se prosljediti u metodu kao parametri.
    Varijabla vidljiva ima vrijednost FALSE ako je cijela linija
    odbacena i koordinate joj se u tom slucaju ne mjenjaju, a TRUE ako
    je vracena obrezana linija.*/
{
     double dx = *x1 - *x0;
     double dy = *y1 - *y0;

     *vidljiva=FALSE;

     /* TRUE ako se tocka nalazi unutar pravokutnika */
     if (dx==0 && dy==0 && obrezivanje_tocke(*x0,*y0))
         *vidljiva=TRUE;
     else {
         double tE = 0.0;
```

```
        double tL = 1.0;
        if (obrezivanje_t(dx, xmin - *x0, &tE, &tL)) &&
            //s unutarnje strane lijeve stranice
            obrezivanje_t(-dx, *x0 - xmax, &tE, &tL) &&
            //s unutarnje strana desne stranice
            obrezivanje_t(dy, ymin - *y0, &tE, &tL) &&
            //s unutarnje strana donje stranice
            obrezivanje_t(-dy, *y0 -ymax, &tE, &tL))
            //s unutarnje strane gornje stranice
            {
            *vidljiva=TRUE;
            /* racuna PL sjeciste ako je tL pomaknut */
            if(tL < 1){
                *x1 = *x0 + tL * dx;
                *y1 = *y0 + tL * dy;
            }
            /* racuna PE sjeciste ako je tE pomaknut */
            if (tE > 0) {
                *x1 += tE * dx;
                *y1 += tE * dy;
            }
        }
    }
}

boolean obrezivanje_t(double nazivnik, double brojnik, double *tE,
    double *tL)
/* Funkcija racuna nove vrijednosti tE ili tL za unutrasnje sjeciste
    duzine i stranice pravokutnika. Parametar nazivnik je nazivnik
    izraza (3.2). Predznak tog parametra odreduje da li je sjeciste PE
    ili PL. Parametar brojnik je brojnik izraza (3.2). Ako je duzina
    odbacena, funkcija vraca FALSE, a ako nije, vraca se TRUE i
    azurirane vrijednosti tE i tL. */
{
    double t;
    if (nazivnik > 0) {         // PE sjeciste
        t = brojnik / nazivnik;
        if (t > *tL)
            return FALSE;
        else if (t > *tE)
            *tE = t;
    } else if (nazivnik < 0) {  // PL sjeciste
        t = brojnik / nazivnik;
        if (t < *tE)
            return FALSE;
        else
            *tL = t;
```

```
    } else if (brojnik > 0)     // duzina je izvan pravokutnika
        return FALSE;
    return TRUE;
}
```

Cohen-Sutherlandov algoritam je efikasan kada se testiranje kodova može izvesti brzo (npr. korištenjem operacija direktno nad bitovima u assembly jeziku) i kada se većina linija može trivijalno prihvatiti ili odbiti.

Parametarski algoritmi za obrezivanje linija su efikasniji kada je potrebno obrezati puno linija jer je računanje koordinata sjecišta odgođeno do onda kad je stvarno potrebno, a testiranja se izvode na parametarskim vrijednostima. Međutim, to računanje parametarskih vrijednosti izvodi se i u slučajevima kada bi linija mogla biti trivijalno prihvaćena ili odbaćena u Cohen-Sutherlandovom algoritmu.

Liang–Barsky-jev algoritam je efikasniji od Cyrus–Beck-ovog algoritma zbog dodatnih testova trivijalnog odbacivanja koje izbjegavaju računanje četiri parametarske vrijednosti za linije koje ne sjeku pravokutnik. Za linije koje ne leže cijele na vanjskoj poluravnini, testovi odbacivanja u Liang–Barsky-evom algoritmu su očito bolji od ponavljanog obrezivanja u Cohen–Sutherland-ovom algoritmu.

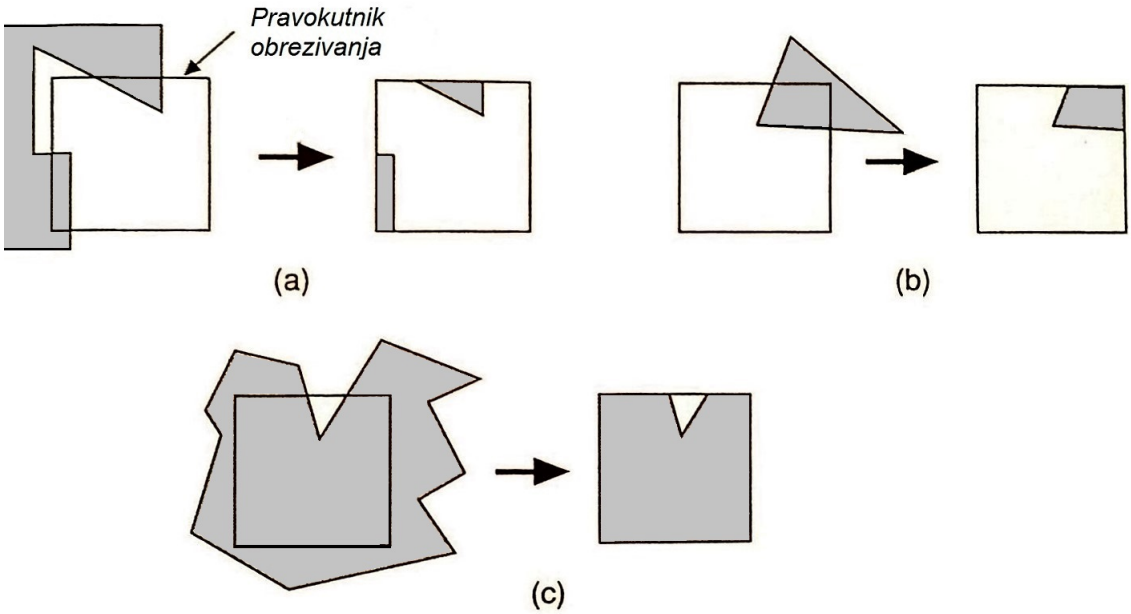
3.4 Obrezivanje kružnice i elipse

Da bismo obrezali kružnicu pravokutnikom možemo napraviti trivijalne testove prihvaćanja i odbacivanja tako da obrežemo kvadrat koji opisuje kružnicu pravokutnikom koristeći algoritam iz sljedećeg odlomka za obrezivanje mnogokuta. Ako kružnica siječe pravokutnik, podjelimo ju u kvadrante i radimo trivijalne testove za svaki kvadrant. Nakon tih testova sljede testovi za oktante i analitičko računanje sjecišta kružnice sa stranicama pravokutnika. Izvršava se scan konverzija lukova s prikladno inicijaliziranim algoritmom za crtanje kružnice. Ako kružnica nije velika, vjerojatno je efikasnije za svaki piksel posebno provjeravati upada li u pravokutnik obrezivanja.

Za obrezivanje elipse testiramo kvadrante pa radimo analitičko računanje sjecišta ili obrezivanje dok radimo scan konverziju.

3.5 Obrezivanje mnogokuta

Algoritam za obrezivanje mnogokuta mora dobro raditi na puno različitih slučajeva kao što vidimo na slici 3.6



Slika 3.6: Nekoliko različitih slučajeva obrezivanja mnogokuta pravokutnikom

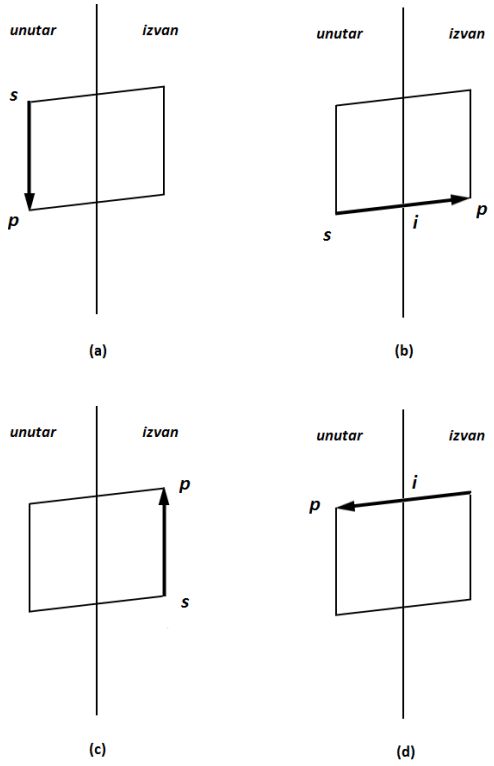
Slučaj (a) je važan jer je konkavni mnogokut moguće obrezati u dva zasebna mnogokuta. Svaka stranica mnogokuta mora biti testirana za svaku stranicu pravokutnika. Obrezivanjem može doći do dodavanja novih stranica te odbacivanja ili dijeljenja postojećih. Obrezivanjem jednog mnogokuta možemo ih dobiti više i potreban nam je organizirani način da rješimo sve te slučajeve.

3.5.1 Sutherland-Hodgmanov algoritam za obrezivanje mnogokuta

Sutherland-Hodgmanov algoritam za obrezivanje mnogokuta koristi pristup *podijeli pa vladaj*: rješava niz jednostavnih i identičnih problema koji, kad ih povežemo, rješavaju ukupni problem. U ovom je slučaju jednostavan problem obrezivanje mnogokuta s obzirom na jednu beskonačnu stranicu za obrezivanje. Rješavajući taj problem s četiri takve beskonačne stranice, pri čemu svaka određuje stranicu pravokutnika, rješava se problem obrezivanja mnogokuta pravokutnikom.

Sutherland-Hodgmanov algoritam nije ograničen samo na obrezivanje pravokutnikom nego možemo njime izvršiti obrezivanje bilo kojim konveksnim mnogokutom. Algoritam prima niz vrhova mnogokuta $v_1, v_2, ..., v_n$. Vrhovi v_i i v_{i+1} te v_n i v_1 definiraju stranice mnogokuta. Algoritam obrezuje jednom beskonačnom stranicom i vraća novi skup vrhova koji se u sljedećoj iteraciji obrezuju sljedećom stranicom itd. Algoritam se u jednoj iteraciji kreće nizom vrhova i za svaku stranicu mnogokuta dodaje nula, jedan ili dva nova vrha u

izlazni niz, tj. niz koji se prosljeđuje sljedećoj iteraciji koja obrezuje mnogokut sljedećom stranicom obrezivanja.



Slika 3.7: Četiri slučaja koja se analiziraju u svakom koraku algoritma.

Promatramo stranicu mnogokuta od vrha s do vrha p, s pretpostavkom da je vrh s bio obrađen u prethodnom koraku.

a) Stranica je cijela unutar granice obrezivanja: dodajemo p u novi niz vrhova.

b) Sjecište i dodajemo u novi niz vrhova jer je u toj točki stranica obrezana.

c) Cijela je stranica izvan granice obrezivanja: ne dodajemo ništa u novi niz.

d) Dodajemo u novi niz sjecište i i vrh p.

Funkcija `Sutherland_Hodgman_obrezivanje()` u prima ulazni niz i stvara novi izlazni niz vrhova. Da bi kod bio jednostavan ne provjeravamo granice niza i koristimo funkciju `izlaz()` da ubacimo novi vrh u `izlazni_niz`. Funkcija `sjeciste()` računa sjecište stranice mnogokuta i stranice obrezivanja. Funkcija `unutra()` vraća `true` ako je

točka na unutarnjoj strani granice. Unutarnja strana je lijevo kada promatramo iz smjera prvog vrha prema drugom vrhu stranice obrezivanja koja leži na granici. Algoritam koristi izraz (3.2). Vrhovi su poredani suprotno od smjera kazaljke na satu.

```
typedef point vrh;
typedef vrh stranica[2];
typedef vrh niz_vrhova[MAX];

static void izlaz(vrh, *int, niz_vrhova);
static boolean unutra (vrh, stranica);
static vertex sjeciste (vrh, vrh, stranica);
void Sutherland_Hodgman_obrezivanje (
    niz_vrhova ulazni_niz,
    niz_vrhova izlazni_niz,
    int ulazna_duljina,
    int *izlazna_duljina,
    stranica granica_obrezivanja)
{
    vrh s, p,      // pocetna i zavrsna tocka stranice mnogokuta
    i;             // sjeciste stranice mnogokuta sa stranicom
       obrezivanja
    *izlazna_duljina = 0;
    s = ulazni_niz[ulazna_duljina - 1];
    for(int j = 0; j < ulazna_duljina; j++){
        p = ulazni_niz[j];
        if(unutra(p, granica_obrezivanja)){
            if(unutra(s, granica_obrezivanja)) // slucaj (a)
                izlaz(p, izlazna_duljina, izlazni_niz);
            else {                             // slucaj(d)
                i = sjeciste(s, p, granica_obrezivanja);
                izlaz(i, izlazna_duljina, izlazni_niz);
                izlaz(p, izlazna_duljina, izlazni_niz);
            }
        } else {
            if(unutra(s, granica_obrezivanja)){ //slucaj (b)
                i = sjeciste(s, p, granica_obrezivanja);
                izlaz(i, izlazna_duljina, izlazni_niz);
            }
        s = p;
        }
    }
}
```

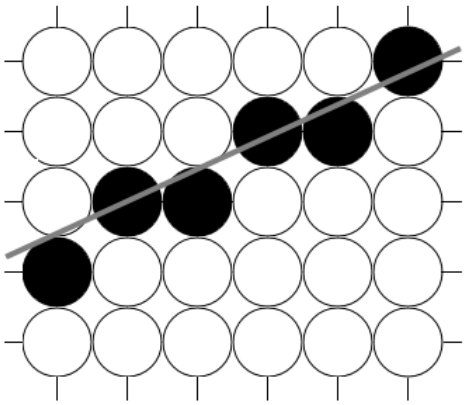
Poglavlje 4

Anti-aliasing

Primitive koje prikazujemo korištenjem obične scan konverzije najčešće su nazubljene jer pikselima pridajemo samo dvije vrijednosti, ili maksimalni intenzitet ili nikakav. Nazubljenost je pojava u rasterizaciji poznata kao aliasing. Primjena postupaka koji umanjuju aliasing naziva se anti-aliasing.

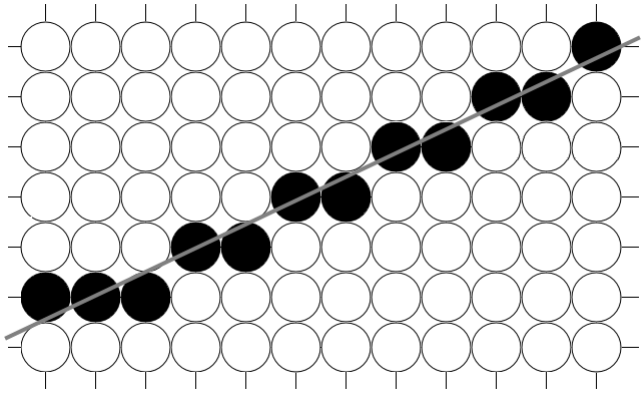
4.1 Povećavanje rezolucije

Kod Bresenhamovog algoritma za crtanje linija nagiba između 0 i 1, kad god iscrtamo piksel u novom stupcu kojem je redak kojeg algoritam bira različit od retka prethodnog piksela, nastaje ”stepenica”, kao što možemo vidjeti preuveličano na slici 4.1. Isto vrijedi za ostale primitive koje crtamo primjenjući samo dvije vrijednosti za intenzitet piksela.



Slika 4.1: Nazubljenost linije

Odlučimo li prikazati istu primitivu na ekranu dvostruke horizontalne i vertikalne rezolucije, stepenica će biti duplo više, ali će biti upola manje veličine, kao što možemo vidjeti na slici 4.2.



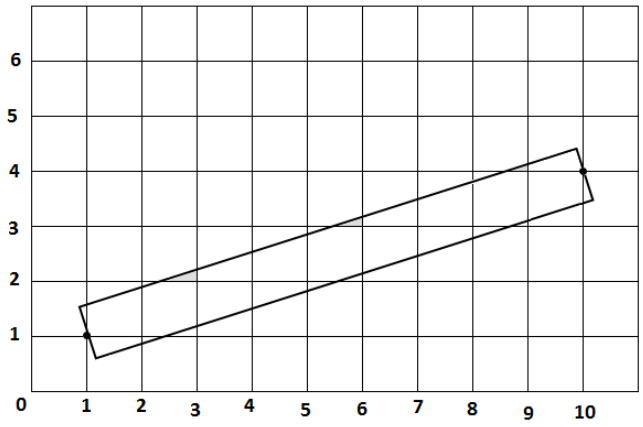
Slika 4.2: Ublažavanje nazubljenosti povećanjem rezolucije

Ovime smo postigli da slika izgleda bolje nauštrb performansi: učetvertostručena je potrebna memorija i vrijeme izvršavanja scan konverzije.

4.2 Netežinsko uzorkovanje područja (*Unweighted Area Sampling* - UAS)

Iako je debljina idealne linije nula, rasterski ekran ne može prikazati ništa tanje od jednog piksela. Stoga ekran umjesto horizontalne ili vertikalne linije prikazuje pravokutnik širine jedan piksel. Za ostale prikazane linije debljina može varirati između 0 i debljine dijagonale piksela.

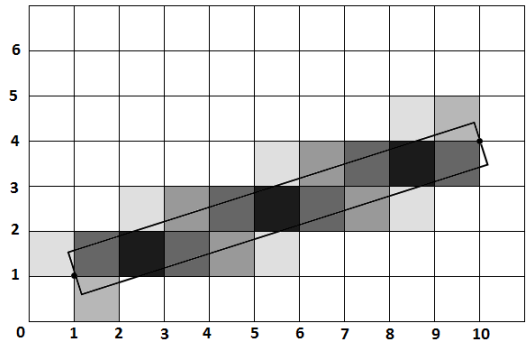
Zamislimo li liniju kao pravokutnik širine jedan piksel, možemo doći do jednog rješenja problema aliasing-a.



Slika 4.3: Linija kao pravokutnik debljine jedan piksel

Umjesto da pridodajemo maksimalan intenzitet točno jednom pikselu u stupcu (govorimo o nagibima u segmentu [0, 1]), intenzitet možemo rasporediti na piksele koji su dijelom pokriveni idealnim pravokutnikom. Za vertikalne i horizontalne linije neće biti razlike koristimo li ovaj pristup ili običnu scan konverziju, i dalje će biti obojan samo jedan piksel u stupcu ili retku.

Konkretno, UAS postavlja intenzitet piksela na vrijednost proporcionalnu dijelu površine piksela koja je prekrivena pravokutnikom (općenito bilo kojom primitivom). Bojamo li piksele u crno, potpuno pokriven piksel bio bi crn, a polovično prekriven bio bi nijansa sive na pola između crne i bijele. Ova tehnika primjenjena na liniju sa slike 4.3 je prikazana na slici 4.4.



Slika 4.4: Primjena uzorkovanja

Primjenom UAS-a ublažuju se nagli prijelazi s piksela maksimalnog i minimalnog intenziteta. Time se postiže zaglađivanje linija i rubova primitive zbog čega slika gledana iz daljine izgleda bolje.

Primjetimo tri svojstva UAS-a:

1. Intenzitet piksela kojeg linija prekriva se smanjuje s udaljenošću centra piksela od linije. Što je primitiva udaljenija od piksela to ona manje utječe na njega.

2. Primitiva ne utječe na piksele koje ne presjeca.

3. Jednake površine prekrivenih područja jednako doprinose intenzitetu, neovisno o tome kolika je udaljenost između centra piksela i prekrivenih područja.

U sljedećem poglavlju opisujemo sličnu antialiasing metodu koja daje vizualno bolje rezultate od UAS-a time što se razlikuje od UAS-a u trećem svojstvu.

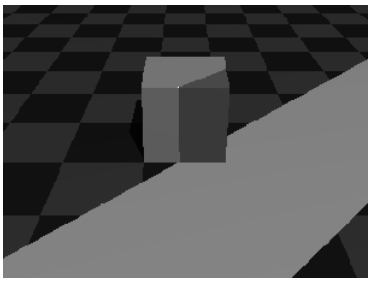
4.3 Težinsko uzorkovanje područja (*Weighted Area Sampling* - WAS)

WAS polazi se od pretpostavke da bi doprinost prekrivenih područja intenzitetu piksela trebao ovisiti o njihovoj udaljenosti od centra piksela.

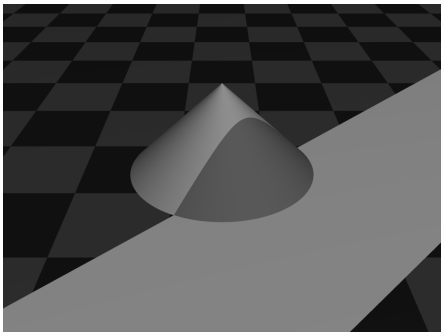
Da bismo zadržali drugo svojstvo UAS metode, moramo promjeniti ”oblik” piksela. Površina koja će predstavljati piksel biti će krug veći od prethodno korištenog kvadrata. Točnije, pokazalo se da najbolje rezultate daje krug radiusa jedan piksel (kružnica prolazi centrima četiri najbliža susjedna piksela).

U nazivu metode je riječ ”težinski” jer definiramo težinsku funkciju (naziva se još i filter funkcija) koja opisuje koliko neka prekrivena točka doprinosi intenzitetu piksela ovisno o udaljenosti te točke od centra piksela. Kod UAS ta je funkcija konstantna, dok se kod WAS linearno smanjuje povećavanjem udaljenosti od centra piksela.

Oblik te funkcije u 3D za UAS je kocka, dok je za WAS stožac. Visina iznad *xy* ravnine je težina točaka ravnine za određeni piksel.



Slika 4.5: Težinska funkcija za UAS . Slika preuzeta iz [3]



Slika 4.6: Težinska funkcija za WAS. Slika preuzeta iz [3]

Intenzitet piksela računa se kao integral težinske funkcije po površini piksela koju prekriva primitiva.

- Za UAS težinska funkcija $W(x, y)$ je konstanta 1 kako bi volumen iznad piksela, pa tako i maksimalni intenzitet piksela, bio jednak 1.
- Za WAS intenzitet piksela se mjeri kao volumen ispod plašta stošca i iznad dijela baze štošca koji je prekriven primitivom. Visina stošca je $h = \frac{3}{\pi}$ kako bi volumen bio 1. Težinsku funkciju pritom definiramo:

$$W(x, y) = h - \frac{(x_c - x)^2 + (y_c - y)^2}{r^2}$$

odnosno:

$$W(x, y) = \frac{3}{\pi} - (x_c - x)^2 + (y_c - y)^2$$

pri čemu je (x_c, y_c) središte piksela,

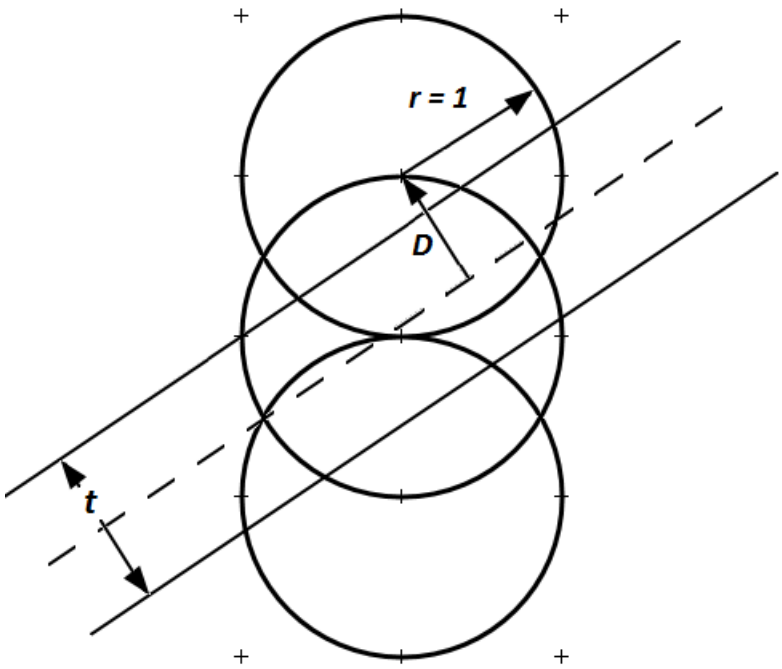
Za razliku od UAS, u WAS metodi vertikalne i horizontalne linije su također prikazane pomoću više piksela u retku ili stupcu jer težinska funkcija piksela doseže do centra susjednog piksela. Piksel kojim prolazi linija debljine jednog piksela više nije punog intenziteta. Ovime se postižu blaže razlike u intenzitetu između susjednih piksela čime nastaje oku ugodnija slika.

4.4 Gupta–Sproull-ovo zaglađivanje linija

Gupta-Sproullov algoritam scan konverzije za linije korisiti unaprijed izračunate intenzitete spremljene u tablicu. U tablici su spremljene vrijednosti volumena računatih za nekoliko diskretnih udaljenosti centra linije od centra piksela.

Originalni članak Gupte i Sproull-a sadrži vrijednosti tablice za stožastu težinsku funkciju za 4-bitni monitor: 24 intenziteta za vrijednosti udaljenosti iz segmenta [0, 1.5]. Spremljene su vrijednosti udaljenosti za 0, $\frac{1}{16}$, ... , $\frac{23}{16}$, a za sve veće udaljenosti intenzitet je 0. Preciznost ne mora biti veća od $\frac{1}{16}$ jer je monitor 4-bitni.

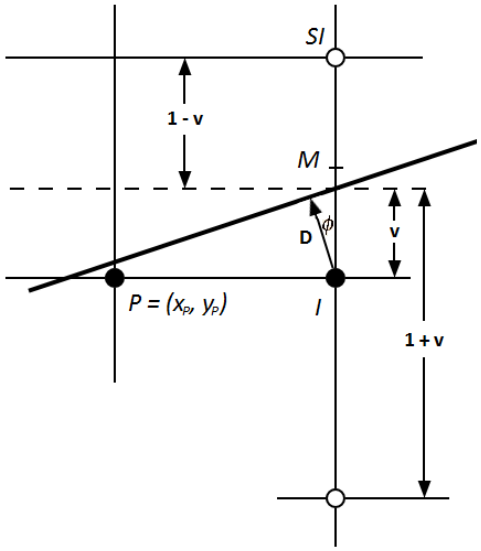
U svakom će stupcu tri piksela biti obojana kao što vidimo na slici 4.7. To vrijedi za većinu linija s nagibom između -1 i 1.



Slika 4.7: Linija debljine jedan piksel i tri piksela kojima će biti pridodan intenzitet

Na slici 4.8 prikazani su pikseli kojima pridodajemo intenzitet i linija. Udaljenost između piksela i linije označavamo s D, debljina linije označena s t je jednaka 1.

Modificirat ćemo Bresenhamov algoritam tako se da prilikom scan konverzije izvede WAS zaglađivanje. Kao i prije, koristimo varijablu odluke, d, za odabiranje sljedećeg piksela (**I** ili **SI**), samo što umjesto aktiviranja piksela koji odaberemo, na temelju težinske funkcije raspoređujemo intenzitet na odabrani i na dva susjedna piksela u stupcu.



Slika 4.8: Udaljenosti od linije do piksela

Vertikalna udaljenost između piksela i točke na liniji, v, je razlika njihovih y koordinata. Udaljenost v je pozitivna ako dužina prolazi iznad piksela i negativna ako prolazi ispod pa se u težinsku funkciju šalje apsolutna vrijednost varijable. Odabrani piksel u stupcu je srednji od tri koja će biti obojana. Gornji je od dužine udaljen za $1 - v$, a donji $1 + v$. Ovo vrijedi za oba slučaja: i za pozitivan i negativan v.

$$D = v \cos \phi = \frac{v\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \tag{4.1}$$

Umjesto direktnog računanja varijable v, računat ćemo ju dok računamo varijablu odluke d kao u Bresenhamovom algoritmu: $d = F(x_P + 1, y_P + \frac{1}{2})$. Općenito, poznavajući vrijednost x koordinate točke na liniji, možemo izračunati y koordinatu koristeći:

$$F(x, y) = 2(ax + by + c) = 0$$

$$y = -\frac{ax + c}{b}$$

Za piksel **I** vrijedi: $x = x_P + 1$, $y = y_P$ i $v = y - y_P$ iz čega sljedi:

$$v = -\frac{a(x_P + 1) + c}{b} - y_P.$$

Množeći obje strane s $-b$ dobivamo:

$$-bv = a(x_P + 1) + by_P + c = \frac{F(x_P + 1, y_P)}{2}.$$

Budući da je $b = -\Delta x$ sljedi da je

$$v\Delta x = \frac{F(x_P + a, y_P)}{2}. \tag{4.2}$$

Budući da je $v\Delta x$ brojnik jednakosti 4.1, a nazivnik je konstanta koja se može unaprijed izračunati, htjeli bismo izračunati $v\Delta x$ iz prethodnog računanja varijable d te izbjeći dijeljenje s 2 da se ne bi morali baviti decimalnim brojevima.

Za piksel **I**,

$$\begin{aligned} 2v\Delta x = F(x_P + 1, y_P) &= 2a(x_P + 1) + 2by_P + 2c \\ &= 2a(x_P + 1) + 2b(y_P + \frac{1}{2}) - b + 2c \\ &= d + \Delta x \end{aligned}$$

pa je

$$D = \frac{d + \Delta x}{2\sqrt{\Delta x^2 + \Delta y^2}}$$

s nazivnikom koji je konstanta.

Odgovarajuće brojnike za piksele $y_P + 1$ i $y_P - 1$ dobijemo: $2(1 - v)\Delta x = 2\Delta x - 2v\Delta x$ i $2(1 + v)\Delta x = 2\Delta x + 2v\Delta x$.

Slično za piksel **SI**:

$$\begin{aligned} 2v\Delta x = F(x_P + 1, y_P + 1) &= 2a(x_P + 1) + 2b(y_P + 1) + 2c \\ &= d - \Delta x \end{aligned}$$

i odgovarajući brojnici su: $2\Delta x - 2v\Delta x$ i $2\Delta x + 2v\Delta x$.

U nastavku je dana jedna implementacija Gupta–Sproull-ovog algoritma koji dodaje zaglađivanje Bresenhamovom algoritmu. Funkcija `aktiviraj_piksel()` zamjenjena je s funkcijom `intenzitet_piksela()` koja se poziva za tri piksela kojima u svakom stupcu pridajemo intenzitet. Funkcija `intenzitet_piksela()` koristi tablicu za pretvaranje udaljenosti u spremljene vrijednosti intenziteta.

```
void intenzitet_piksela(int x, int y, double udaljenost)
{
    // uzimanje vrijednosti iz tablice
        double intenzitet = Filter(Round(fabs(udaljenost)));
    aktiviraj_piksel(x, y, intenzitet);
}
void bresenham_cjelobrojni(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
```

```
    int d = -dx;

    int pomakI = 2 * dy;
    int pomakSI = 2 * (dy - dx);

    int dva_v_dx= 0;
    double invNazivnik = 1.0 / (2.0 * sqrt(dx*dx + dy*dy));
    double dva_dx_invNazivnik = 2 * dx * invNazivnik;

    int y = y1;
    for (int x = x1; x <= x2; x++) {
        // srednji piksel
        intenzitet_piksela(x, y, dva_v_dx * invNazivnik);

        // susjedni pikseli
        intenzitet_piksela(x, y + 1, dva_dx_invNazivnik - dva_v_dx *
            invNazivnik);
        intenzitet_piksela(x, y - 1, dva_dx_invNazivnik + dva_v_dx *
            invNazivnik);

        if(d < 0) {
            // izbor piksela I
            dva_v_dx = y + dx;
            d += pomakI;
        } else {
            // izbor piksela SI
            dva_v_dx = d - dx;
            d += pomakSI;
            y++;
        }
    }
}
```

Algoritam se može proširiti tako da obuhvaća i krajnje točke, za što je potrebna još jedna tablica s vrijednostima intenziteta. Mana Gupta–Sproull-ovog algoritma je upravo potreba korištenja tablice vrijednosti intenziteta jer sadrži vrijednosti samo za linije fiksiranih debljina.

Na sličan način možemo zagladiti tekstualne znakove. Drugi način zaglađivanja znakova je ručno zaglađivanje rubova na njihovim bitmapama.

Bibliografija

[1] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes: *Computer Graphics: Principles and Practice in C*, Addison-Wesley Professional, drugo izdanje (kolovoz 1995.).

[2] Marko Čupić, Željka Mihajlović: *Interaktivna računalna grafika kroz primjere u OpenGL-u*, skripta, (ožujak 2016.).

[3] Andreas Kriegl, Computer Graphics, `http://www.mat.univie.ac.at/~kriegl/Skripten/CG/CG.html` (travanj 2016.)

[4] Tina Bosner, Računalna grafika, `https://web.math.pmf.unizg.hr/nastava/CG/` (lipanj 2016.)

[5] Dilip Da Silva, Raster algorithms for 2D primitives, `https://cs.brown.edu/research/pubs/theses/masters/1989/dasilva.pdf` (lipanj 2016.)

Sažetak

U radu su opisani algoritmi kojima se efikasno mogu prikazivati neke 2D primitive na rasterskoj prikaznoj jedinici kao što je većina modernih računalnih ekrana. Kreće se od najjednostavnijih algoritama za prikazivanje linije i intuitivno dolazi do boljeg rješenja, Bresenhamovog algoritma i metode srednje točke. Metodu srednje točke primjenili smo i za prikazivanje drugih primitiva jer ju je lakše generalizirati nego Bresenhamov algoritam.

Načini rješavanja problema koje su primitive vidljive na ekranu, a koje su izvan granica ekrana, opisani su u sljedećem poglavlju o obrezivanju.

Bresenhamov algoritam se u zadnjem poglavlju modificira kako bi se ublažila jedna nuspojava rasterizacije, nazubljenost.

Summary

This thesis covers numerous algorithms dealing with effective ways of drawing 2D primitives on a raster screen. It starts with simple scan conversion algorithms for lines, which is intuitively built up to Bresenham's solution. Later, the same incremental technique is used in algorithms for scan conversion of other 2D primitives.

Next chapter covers clipping, a way to dismiss parts of the image that lies outside of screen boundaries, thus enabling scan conversion algorithms to deal only with visible parts of primitives to make the process more efficient.

Last chapter deals with solutions to aliasing problem, which arises when trying to approximate ideal primitives with discrete pixels.

Životopis

Rođen sam 4. studenog 1991. godine u Rijeci. Osnovnu školu "Gelsi" pohađam od 1998. do 2002. a osnovnu školu "Turnić" od 2001. do 2006. godine. Iste godine upisujem matematički smjer Gimnazije "Andrije Mohorovičića" Rijeka. Državnu maturu polažem 2010. godine nakon čega upisujem preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. Preddiplomski studij završavam 2013. godine kad upisujem diplomski studij računarstva i matematike na istom fakultetu. Od 2015. godine aktivan sam u studentskoj udruzi BEST Zagreb te sam 2016. glavni organizator projekta EBEC Zagreb Dani Inženjera. Iste godine se zapošljavam u tvrtki Amphinicy Technologies.