модели на реално съществуващи продукти или на такива, които предстои да бъдат произведени. Те могат също така да бъдат напълно абстрактни, каквито са някои математически структури или компютърно синтезираните картини.

Много важно разделение се прави между различните приложения в зависимост от ролята на визуалните образи в тях. В някои приложения като визуализацията на данни, картографията, изкуството и рекламата, визуализираният обект е крайният продукт на съответните програми. В други приложения като автоматизираното проектиране и управлението на процеси, визуализираният обект само представя някаква геометрична информация, която е достатъчна на потребителя да вземе решение за по-нататъшните си действия.

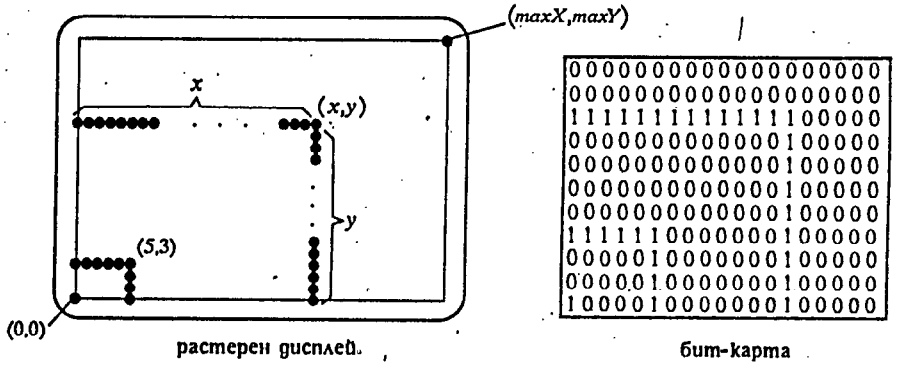
Приложенията могат да се класифицират и по вида на тяхната интерактивност - нуждата им от взаимодействие с потребителя. Тук възможностите са много, вариращи от приложни програми, в които потребителят задава поредица от данни и след това получава визуалния образ на резултата до т.нар. *интерактивно проектиране*, при което потребителят започва работа върху празен екран, създава нови графични елементи, включва стари такива, асемблира ги и ги структурира, така че в резултат да получи образа на желания обект. Едни от най-силно интерактивните приложения са именно системите за автоматизирано проектиране, които са най-тясно свързани с компютърната графика и геометричното моделиране.

Глава 2

Растерна графика

Алгоритмите за растерна графика имат за цел да обслужват визуализацията върху растерни устройства. Необходимостта от специални алгоритми за генериране и манипулиране на растерни изображения се налага поради широкото използване на растерните дисплеи за силно динамични задачи като анимация, симулация в реално време, т.нар. *симулация на реална среда* и др. Тези задачи изискват изпълнение на операциите с голяма скорост, която се постига и когато самите алгоритми са много ефективни и отчитат особеностите на растерните дисплеи. Търсенето на бързодействие е причината и едно от съвременните направления в изследванията в тази област да е създаването на паралелни алгоритми за растерна графика.

В тази глава ще разгледаме най-често използваните алгоритми за генериране на растерни изображения на основните геометрични обекти (графичните примитиви) - отсечка, окръжност, дъга и текстов символ, както и начините за запълване на области, зададени с многоъгълници и окръжности. Ще се спрем накратко на методите за отстраняване на стъпаловидността на обектите (antialiasing), удебеляването и използването на типове линии и образци за запълване. Тук не е отделено внимание на такива теми като растерни трансформации, филтриране, съхраняване и компресиране на растерни изображения, тъй като те са свързани повече с обработката, а не с генерирането на изображения.



Фиг. 2-1

При представянето на растерните дисплеи по-горе казахме, че изображението се описва в т.нар. *бит-карта* или *пикселна карта*, която е една правоъгълна матрица. Елементите на тази матрица (пикселите) са фиксиран брой битове, които задават цвета и интензитета на съответната точка от екрана. Адресацията на всеки пиксел в тази растерна матрица става чрез двойка целочислени координати, всяка от които варира във фиксиран целочислен интервал [0,*maxX*] и [0,*maxY*] съответно. Числата *maxX*+1 и *maxY*+1 са различни за всеки растерен дисплей и отразяват броя на пикселите по всяка от координатните оси.

Координатната система, определена върху растерната мрежа за повечето растерни дисплеи е с координатно начало в горния ляв ъгъл на растера (съответно и на екрана). Навсякъде в тази книга ние обаче ще считаме, че координатната система на растера е декартова, т.е. координатното начало е в долния ляв ъгъл на устройството.

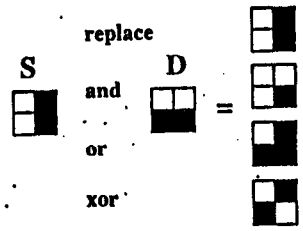
От гледна точка на програмиста обслужването на едно растерно устройство е сравнително елементарно. Пикселите се адресират с техните декартови координати и във всеки пиксел може да се запише дадена стойност, както и да се прочете тази, която вече е записана там. Записването на нова стойност в бит-картата става в няколко различни режима. Режимът определя типа на булевата операция, която се извършва между стойността S, която предстои да се запише и стойността D, която пикселът има преди записването.

D ← S BoolOp D

От 16-те възможни булеви операции се използват най-често само:

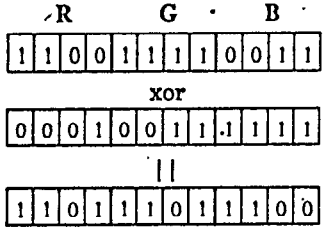
- **replace**: стойността, която пикселът получава е точно тази, която се записва, а предишното състояние на пиксела се игнорира;
- **and, or, xor**: стандартните булеви "и", "или" и "изключващо или".

В огромна част от случаите записването в бит-картата се извършва с операцията **replace**. Затова и в повечето растерни системи се използва *текущ режим на записване*, вместо изрично да се задава операцията, с която всеки пиксел се модифицира. За растерни дисплеи, пикселите на които могат да имат само една от двете стойности 0 и 1 ("черна" и "бяла"), режимите за записване могат да се илюстрират така:



Фиг. 2-2

При растерни дисплеи с *n* състояния за всеки пиксел булевите операции се извършват побитово. Долният пример показва това за една система, в която за всеки от основните цветове (червен - R, зелен - G и син - B) са отделени по четири бита:



Споменатите режими се използват в следните случаи:

replace: за записване на растеризацията на примитив, както и за изтриването му;

and: за избирателно изтриване на пиксели в дадена правоъгълна област (блок от пиксели);

or: за добавяне на блок от пиксели към изображението без да се изтриват тези от тях, за които в блока има записана 1;

xor: за инвертиране на образа.

При повторно изпълнение на последната операция образът възвръща началното си състояние: D ≡ S xor (S xor D). Това е важно свойство, което се използва в много интерактивни похвати, както ще видим в четвърта глава. За нуждите на тази глава можем да смятаме, че разполагаме със следния набор от функции, с които се осъществява казаното дотук:

```
void SetWritingMode(REPLACE|AND|OR|XOR)
int  GetPixel(x,y)
void PutPixel(x,y,value)
```

Последната функция извършва записването на стойността `value` в зададения пиксел съобразно текущия режим на записване. Пикселите в пикселната карта са разположени по редове и затова е подходящо и обработката на група от съседни пиксели да става по редове. Тъй като операциите върху поредица от хоризонтално разположени пиксели, както и върху цял блок могат да се реализират по-ефективно, ще използваме още функциите:

```
void PutPixelRow(x,y,w,value)
void PutPixelBlock(x,y,w,h,buffer)
void GetPixelRow(x1,x2,y,buffer)
void GetPixelBlock(x,y,w,h,buffer)
```

2.1 РАСТЕРИЗИРАНЕ НА ГРАФИЧНИ ПРИМИТИВИ

В тази част ще разгледаме растеризирането на най-често използваните графични примитиви като се спрем на различните начини, по които се прави това. Особено внимание ще обърнем на целочислените алгоритми, които позволяват ефективна реализация. Представените програми лесно могат да бъдат оптимизирани, използвайки ефективни езикови конструкции за постигане на

максимално бърза визуализация, което често е основна цел при разработването на интерактивни системи. Тук целта е по-скоро яснота, затова и не всички програми са написаии оптимално.

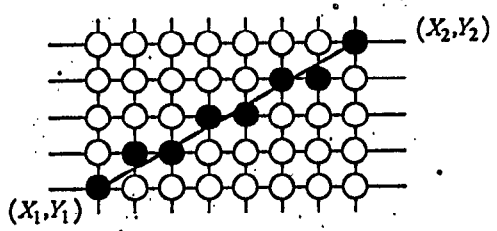
2.1.1 Растеризиране на отсечка

Основната задача при растеризирането на отсечка е да се намерят координатите на точките от растера, които лежат най-близо до зададеиата отсечка и съвкупността от които дава най-добро визуално приближеиие за нея. Нека една отсечка е зададена с координатите на началната и крайиата си точки (X_1,Y_1), (X_2,Y_2), които принадлежат на растера. Уравнението иа една такава невертикалиа отсечка тогава би било:

$$y = m(x - X_1) + Y_1, \qquad m = \frac{dy}{dx} = \frac{Y_2 - Y_1}{X_2 - X_1} \qquad [2.1]$$

РАСТЕРИЗИРАНЕ СЪС ЗАКРЪГЛЯВАНЕ. Ако отсечката има наклои, близък до хоризоитален, т.е. $-1 < m < 1$, то във всеки вертикален стълб на растера между двата й крайии пиксела ще има само по един пиксел от нея. За да намерим всеки от тези пиксели, можем да даваме иа x стойиости X_1, X_1+1, $X_1+2,\ldots, X_2$, при което от [2.1] за y ще получим последователността Y_1, Y_1+m, $Y_1+2m,\ldots, Y_2$. Тъй като m е реално число, то и всички числа от тази редица ще бъдат реални. За да намерим целочислеиата y-координата, ще трябва да закръглим всяко от реалиите числа до най-близкото цяло. Тогава отсечката би могла да бъде представена с пикселите:

$$(x_i, y_i), \qquad x_i = X_1 + i, \qquad y_i = \lfloor Y_1 + i \cdot m + 0.5 \rfloor, \qquad i = 0,1,\ldots,dx$$

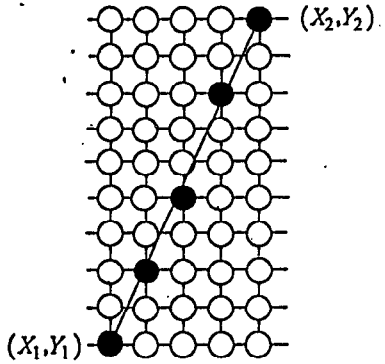


Фиг. 2-3

Растеризацията на отсечки в първи квадрант, чиито иаклои е $m > 1$, по този начии би била неприемлива, тъй като за тях y се мени по-бързо от x и не можем да твърдим, че във всеки вертикален стълб има само по един пиксел от отсечката (фиг. 2-4).

Този проблем лесно може да се реши като просто се разменят местата на x и y. И накрая за да обобщим за отсечка от кой да е квадрант е необходимо да отчитаме, че стъпката с която се изменя i може да бъде и отрицателна (когато $X_2 < X_1$). Би било добре да отбележим, че не е желателно да се разменят началната и крайната точки на отсечката при положение, че $X_2 < X_1$ с цел винаги да растеризираме с положително нарастване по x. Проблем възниква

например при растеризиране на пунктирана отсечка, която е част от начупена линия. Тогава размяната на краищата й би довела до нежелателно прекъсване на пунктира, тъй като поставянето на образеца е ориентирано винаги от началната към крайната точка.



Фиг. 2-4

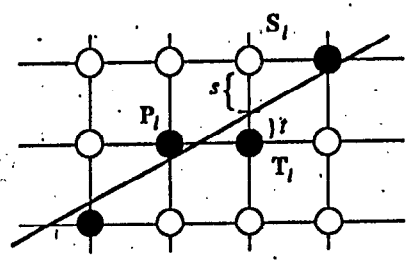
Следната функция ще осъществи растеризирането по описаната схема:

```
void SimpleLine(X1,Y1,X2,Y2,value)
int X1,Y1,X2,Y2; int value;
{int d,dx,dy,x,inty;
 int incX,n,reverse; float y, incY;
   dx=abs(X2-X1); dy=abs(Y2-Y1);
   if (reverse=(dx<dy)) {
      d=X1; X1=Y1; Y1=d;   /* разменяме местата на X и Y */
      d=X2; X2=Y2; Y2=d;   /* ако наклонът е по-голям от */
      d=dx; dx=dy; dy=d;   /* 45 градуса */
   }
   incX=(X1<=X2)?1:-1;     /* нарастването по X */
   incY=((float)dy)/dx;    /* може да е отрицателно */
   x=X1; y=Y1;
   n=dx+1;
   while (n--) {
         inty=(int)y;      /* закръгляване на реалното Y */
         if (reverse) PutPixel(inty,x,value);
                 else PutPixel(x,inty,value);
         x+=incX; y+=incY;
   }
}
```

Използването на числа с плаваща запетая не е най-ефективният начин за растеризиране на отсечка. Естествено е да търсим алгоритъм, който работи само с цели числа, тъй като и всички координати на пиксели са целочислени.

АЛГОРИТЪМ НА БРЕЗЕНХАМ ЗА ОТСЕЧКА. Следният алгоритъм, предложен от Брезенхам (Bresenham) през 1965 год. за управление на плотер при растеризиране на вектори е на практика един от най-често прилаганите целочислени алгоритми. Идеята на този алгоритъм е следната:

Нека за простота разгледаме отсечка, чийто наклон е $0 < m < 1$. Да видим как се осъществява избора на точка от растера на i-тата стъпка в този алгоритъм (фиг. 2-5). Ако на дадена стъпка е била избрана точката $P_i=(x_i,y_i)$, то следващият избор за отсечка с такъв наклон може да бъде или $T_i=(x_i+1,y_i)$ или $S_i=(x_i+1,y_i+1)$.



Фиг. 2-5

Ясно е, че ако $(t-s)<0$, то трябва да изберем T_i, а в противен случай - S_i. Това означава, че можем да използваме тази разлика в качеството на "оценка" за избор на следващия пиксел на всяка стъпка от растеризирането на отсечката. Тя за съжаление е отново реално число, но ние ще покажем, че можем да намерим друга *целочислена* оценка, за която можем да изведем рекуретна зависимост и че тази целочислена оценка е пряко свързана с разликата $(t-s)$.

За да опростим, нека да транслираме отсечката, зададена с уравнение [2.1] така, че началната точка (X_1,Y_1) да съвпадне с (0,0). Тогава можем да запишем следното:

$$y_i+t=y_i+1-s=\frac{dy}{dx}(x_i+1),$$

което би дало следните изрази за s и t:

$$t=\frac{dy}{dx}(x_i+1)-y_i, \qquad s=y_i+1-\frac{dy}{dx}(x_i+1).$$

За разликата $t-s$ ще получим:

$$t-s=2\frac{dy}{dx}(x_i+1)-2y_i-1.$$

Тъй като ще оценяваме дали тази разлика е положителна или не, вместо нея можем да разгледаме $dx(t-s)$, защото $dx>0$. По този начин ще получим целочислена оценка. Нека означим с d_i тази оценка:

$$d_i=dx(t-s)=2dy(x_i+1)-2y_idx-dx. \qquad [2.2]$$

Използвайки горната формула, можем да запишем каква ще бъде оценката и на $(i+1)$-вата стъпка:

$$d_{i+1}=2dy(x_{i+1}+1)-2y_{i+1}dx-dx.$$

За да получим рекурентна зависимост за оценката, нека извадим горните две равенства и използваме, че $x_{i+1}-x_i=1$:

$$d_{i+1}=d_i+2dy(x_{i+1}-x_i)-2dx(y_{i+1}-y_i)=d_i+2dy-2dx(y_{i+1}-y_i).$$

Този резултат вече ни дава основание да направим следното заключение за избора на точка от растера на тази стъпка и за извеждането на оценката за следващата:

Ако $d_i \le 0$, трябва да изберем T_i, а следващата оценка ще е:

$$d_{i+1}=d_i+2dy.$$

Ако $d_i>0$, трябва да изберем S_i и $d_{i+1}=d_i+2dy-2dx$.

От горното следва, че оценката на всяка стъпка може да се пресмята целочислено, което изключва вече необходимостта да се използват числа с плаваща запетая. Използвайки, че началният пиксел съвпада с (0,0), от [2.2] за началната оценка ще получим: $d_0=2dy-dx$.

Показаната тук програма илюстрира използването на този алгоритъм в общия случай. Допълнителното, което е направено, за да се растеризира една произволно наклонена отсечка е следното:

- да се осигури $dx>0$ и $dy>0$;
- да се разменят координатите на крайните пиксели, както и на тези, които се получават при растеризирането, ако отсечката има наклон по-голям от 45 градуса;
- да се предвиди възможността x и y да намаляват вместо само да нарастват - в случай, че някоя от разликите (X_2-X_1) или (Y_2-Y_1) е отрицателна.

```
void BresenhamLine(X1,Y1,X2,Y2,value)
int X1,Y1,X2,Y2; int value;
{int x,y,dx,dy,incX,incY;
 int d,incUP,incDN,reverse,n;
    dx=abs(X2-X1);
    dy=abs(Y2-Y1);
    if (reverse=(dx<dy))
        ExchangeXY(X1,X2,dx,Y1,Y2,dy);
    incUP=-2*dx+2*dy;       /* нарастване при избор на S */
    incDN= 2*dy;            /* нарастване при избор на T */
    incX=(X1<=X2)?1:-1;
    incY=(Y1<=Y2)?1:-1;
    d=-dx+2*dy;
    x=X1; y=Y1;
    n=dx+1;
    while (n--) {
          if (reverse) PutPixel(y,x,value);
                  else PutPixel(x,y,value);
          x+=incX;
          if (d>0) { d+=incUP; y+=incY;
          } else     d+=incDN;
    }
}
```

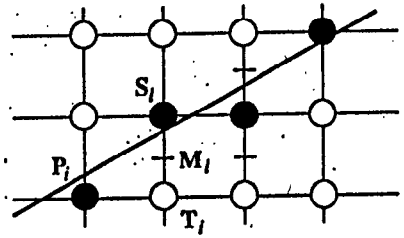
АЛГОРИТЪМ НА СРЕДНАТА ТОЧКА. Ван Аакен (Van Aken) разработва един интересен подход към растеризирането - т.нар. *принцип на средната точка* - използването на който позволява създаването на алгоритми за целочислено растеризиране не само на отсечки и окръжности, но и на конични сечения. Прилагането на този принцип за отсечки води до получаването на същия код като при алгоритъма на Брезенхам. Ние ще се спрем на тази формулировка защото тя ще ни бъде полезна по-нататък.

Друга форма на уравнението на правата [2.1], върху която лежи отсечката е следната:

$$F(x,y) = ax + by + c = dy \cdot x - dx \cdot y + c = 0, \qquad [2.3]$$

където $dy = Y_2 - Y_1$; $dx = X_2 - X_1$; $c = Y_1X_2 - X_1Y_2$.

Тъй като dx и dy са неотрицателни, $F(x,y)$ има положителна стойност за всяка точка разположена под отсечката и отрицателна за точките над нея. Ако на i-тата стъпка в този алгоритъм е била избрана точката $P_i = (x_i, y_i)$, то може да се каже, че следващият избор - $T_i = (x_i+1, y_i)$ или $S_i = (x_i+1, y_i+1)$ зависи от положението на средната им точка $M_i = (x_i+1, y_i+1/2)$ - фиг. 2-6.



Фиг. 2-6

Положението на M_i може веднага да се определи като се пресметне $F(M_i)$ от [2.3] – $F(x_i+1, y_i+1/2)$. Това означава, че е удобно да изберем за оценка числото:

$$d_i = F\left(x_i+1, y_i+\frac{1}{2}\right) = dy \cdot (x_i+1) - dx \cdot \left(y_i+\frac{1}{2}\right) + c. \qquad [2.4]$$

Можем да кажем, че ако $d_i \le 0$, трябва да изберем T_i, а ако $d_i > 0$, ще изберем S_i. При това, ако изберем T_i, т.е. $P_{i+1} = T_i$ то оценката на $(i+1)$-вата стъпка ще е:

$$d_{i+1} = F\left(x_i+2, y_i+\frac{1}{2}\right) = dy \cdot (x_i+2) - dx \cdot \left(y_i+\frac{1}{2}\right) + c. \qquad [2.5]$$

Рекурентната зависимост ще получим, като извадим d_i от d_{i+1} зададени с [2.4] и [2.5]: $d_{i+1} = d_i + dy$.

Ако пък изберем S_i, т.е. $P_{i+1} = S_i$ то новата оценка ще е:

$$d_{i+1} = F\left(x_i+2, y_i+\frac{3}{2}\right) = dy \cdot (x_i+2) - dx \cdot \left(y_i+\frac{3}{2}\right) + c = d_i + dy - dx.$$

Остава да намерим стойността в първата средна точка, която е:

$$d_0 = F\left(x_0+1, y_0+\frac{1}{2}\right) = dy \cdot (x_0+1) - dx \cdot \left(y_0+\frac{1}{2}\right) + c =$$
$$= F(x_0, y_0) + dy - \frac{1}{2}dx = dy - \frac{1}{2}dx$$

За да избегнем дробта в горната формула, можем да умножим навсякъде по 2, което не би променило нищо в разсъжденията ни, но би позволило да използваме целочислена оценка. С други думи, можем да приемем, че уравнението на правата е:

$$F(x,y) = 2(ax + by + c) = 2(dy \cdot x - dx \cdot y + c) = 0. \qquad [2.6]$$

Получихме същия резултат като в алгоритъма на Брезенхам, но чрез различни разсъждения. Те дават по-ясна представа за смисъла на оценката, която въвеждаме и използваме на всяка стъпка от алгоритъма. Нещо повече, тази оценка може да бъде използвана в алгоритмите за отстраняване на стъпаловидността на отсечките, както ще видим по-късно.

АЛГОРИТЪМ НА ПОРЦИИТЕ. Разгледаните дотук алгоритми решават задачата за намиране на една от координатите на един пиксел като знаем другата му координата, така че този пиксел да е най-близко разположен до растеризираната отсечка. В този смисъл всяка итерация води до намирането само на един пиксел от растеризацията.

Ако разгледаме една отсечка в първи октант (т.е. в тази част от I квадрант, в която $x > y$), ще видим, че растерният ѝ образ се състои от поредица от хоризонтални участъци (фиг. 2-7). Тези хоризонтални участъци Брезенхам нарича *порции*. Един друг подход към растеризирането на отсечка е да се получат всички такива порции, като на всяка итерация в алгоритъма се намира съответната поредица от пиксели, а не само един единствен.

Нека за удобство да разгледаме отсечка от първи октант с начало точката (0,0) и крайна точка (H,V). Алгоритъмът на средната точка на практика решава системата от $H+1$ неравенства:

$$y - \frac{1}{2} < \frac{V}{H} \cdot x \le y + \frac{1}{2}, \qquad x = 0,1,2,\ldots,H \qquad [2.7]$$

относно неизвестните y. Нека да модифицираме тази система като я решим относно x. Тя ще се преобразува в система от $V+1$ неравенства спрямо x:

$$\frac{H}{2V}(2y-1) < x \le \frac{H}{2V}(2y+1), \qquad y = 0,1,2,\ldots V,$$

което може да се запише и като:

$$\frac{2H}{2V}y - \frac{H}{2V} < x \le \frac{2H}{2V}y + \frac{H}{2V}, \qquad y = 0,1,2,\ldots V \qquad [2.8]$$

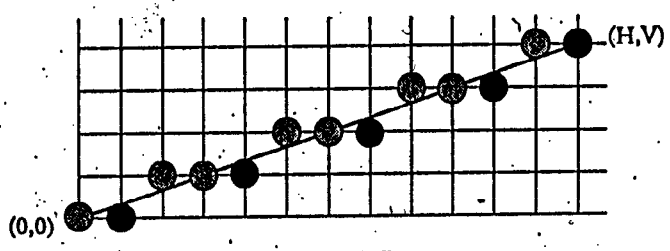
Нека да представим дробите в горната система от неравенства като цяла

част и остатък по модул $2V$, за да можем да извършим нейното решаване в цели числа. Ще получим следното:

$$\frac{H}{2V}=c_0+\frac{r_0}{2V},\qquad \frac{2H}{2V}=c_1+\frac{r_1}{2V} \qquad [2.9]$$

Използвайки [2.9], системата [2.8] може да бъде записана по следния начин:

$$c_1y-c_0+\frac{r_1y-r_0}{2V}<x\le c_1y+c_0+\frac{r_1y+r_0}{2V},\qquad y=0,1,2,\ldots V \qquad [2.10]$$



Фиг. 2-7

Ако разгледаме фиг. 2-7 ще видим, че тези x, за които десните неравенства в тази система се превръщат в равенства, са точно крайните десни пиксели във всяка хоризонтална порция. Те често се наричат *преходни пиксели*. Растеризирането ще извършим, като на всяка итерация намираме преходния пиксел в съответния ред. Можем за отбележим, че най-малкото x, което удовлетворява неравенствата за $y=j+1$, се получава чрез прибавяне на единица към x-координатата на преходния пиксел в неравенството за $y=j$.

Нека разгледаме кога се получава равенство в дясното неравенство от [2.10] за $y=0$. При решаване на уравнението

$$x=c_1y+c_0+\frac{r_1y+r_0}{2V}$$

в цели числа ще видим, че първият преходен пиксел има координати точно $(c_0,0)$.

Всеки следващ преходен пиксел може да бъде получен използвайки същото уравнение след предварително изчисляване на коефициентите в целочисленото разлагане от [2.9]. За целта е необходимо само да въведем една променлива (в представената програма сме използвали името mod), в която да съхраняваме стойността на числителя r_1y+r_0 на дробта в споменатото уравнение и да коригираме цялата част на x всеки път когато тази дроб стане по-голяма от единица.

В така предложения алгоритъм могат да се направят допълнителни подобрения за намаляване на броя на операциите и дори за намаляване на итерациите. Ние ще предоставим възможност на читателя да подобри представения вариант. Много по-сериозно подобрение е да се извлече закономерността в образуването на порциите, за намирането на която напоследък има разработени интересни алгоритми. В тях основно се анализира делимостта на числата H и V.

```
void DrawBresenhamSliceLine(H,V,value)
int H,V,value;
{int r1,c1,y;
 int startX,            /* началната точка на порцията */
 int endX,              /* координатата на преходната точка */
 int mod;               /* числителят в дробта за изчисляване на x */
   r1 = (H+H)%(V+V);
   c1 = (H+H-r1)/(V+V);
   y = 0; startX = 0;
   mod = H%(V+V);
   endX = (H-mod)/(V+V);
   while(1) {
     if (endX<H) {      /* записване на порцията */
        PutPixelRow(startX,endX,y,value);
     } else {           /* това е последната порция */
        PutPixelRow(startX,H,y,value);
        return;
     }
     startX = endX+1;
     endX += c1;
     mod += r1;         /* корекция на числителя */
     if (mod>V+V) { endX++; mod -= V+V; }
     y++;
   }
}
```

2.1.2 Растеризиране на окръжност

Задачата за растеризиране на окръжност е почти толкова важна в компютърната графика, колкото и тази за растеризиране на отсечка поради често налагащото се визуализиране на окръжности. Въпреки, че са разработени алгоритми, приложими за всякакви плоски криви с уравнения $F(x,y)=0$ с непрекъснати първи производни окръжността заслужава специално внимание поради симетричността си. Първо нека разгледаме стандартния алгоритъм, използващ числа с плаваща запетая, след което ще се спрем по-подробно и на целочислените.

Без да ограничаваме общността навсякъде по-долу ще считаме, че работим с централна окръжност: $x^2+y^2=R^2$, чийто радиус е цяло число.

РАСТЕРИЗИРАНЕ СЪС ЗАКРЪГЛЯВАНЕ. Нека най-напред разгледаме само тази част от една централна окръжност, която лежи в първи квадрант. За нея лесно можем да получим явен израз за y:

$$y=\sqrt{R^2-x^2},\qquad x\in[0,R].$$

Ако даваме на x последователно стойности $0,1,\ldots,R$ и намираме y от горното уравнение, след закръгляването му до най-близкото цяло число ще имаме последователност от точки на растера, които апроксимират окръжността.

Както може да се очаква, втората половина от тази четвърт-окръжност изглежда неприемливо поради голямото разстояние между апроксимиращите точки (фиг. 2-8). Това лесно може да се преодолее като се използва симетрията на окръжността отиосно координатните оси и правите $x=y$ и $x=-y$.

Използвайки тази симетрия може да се генерира само 1/8 от окръжността по този алгоритъм, както е показано с програмата по-долу. В нея е отделен случаят $x=y$, защото тогава за всяка от съответните точки ще има по две обръщения към функцията PutPixel, което в режим на записване xor (изключващо "или") би довело до различна осветеност на тези точки. Забележете, че е възможно пикселът с координати (x,y), когато $x=y$ да не е точка от растеризацията на окръжността. По същата причина вън от цикъла е изнесено и записването на първата точка, чиято симетричиа спрямо оста Oy е същата точка.

```
void SimpleCircle(xc,yc,R,value)
int xc,yc,R; int value;
{int x,y;
    x=0; y=R;
    PutPixel(xc,yc+R,value);
    PutPixel(xc,yc-R,value);
    PutPixel(xc+R,yc,value);
    PutPixel(xc-R,yc,value);
    while (x<y) {
        x++;
        y=(int)sqrt((double)(R*R-x*x));
        EightSymmetric(xc,yc,x,y,value);
    }
    if (x==y) FourSymmetric(xc,yc,x,y,value);
}

void EightSymmetric(xc,yc,x,y,value)
int xc,yc,x,y; int value;
{   FourSymmetric(xc,yc,x,y,value);
    FourSymmetric(xc,yc,y,x,value);
}

void FourSymmetric(xc,yc,x,y,value)
int xc,yc,x,y; int value;
{   PutPixel(xc+x,yc+y,value);
    PutPixel(xc-x,yc-y,value);
    PutPixel(xc-x,yc+y,value);
    PutPixel(xc+x,yc-y,value);
}
```

Поради наличието на умножение и извличане на квадратен корен, представеният алгоритъм не е много по-добър от този, който генерира последователност от стойности за координатите чрез вариране на параметъра θ в параметричното уравнение:

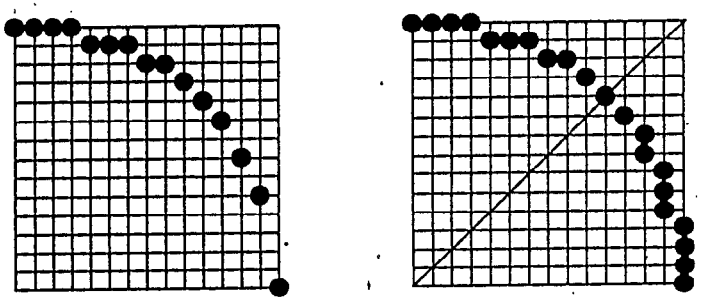
$$\begin{cases} x = X_C + R\cdot\cos\theta \\ y = Y_C + R\cdot\sin\theta \end{cases}$$

Споменатият начии не е за предпочитане ие само поради наличието на тригонометрични функции, но и поради иезависимостта на параметъра от

растера. Използването на тригонометрични функции може да се избегне като се направи следната субституция: $t=\mathrm{tg}(\theta/2)$. Получената нова параметризация:

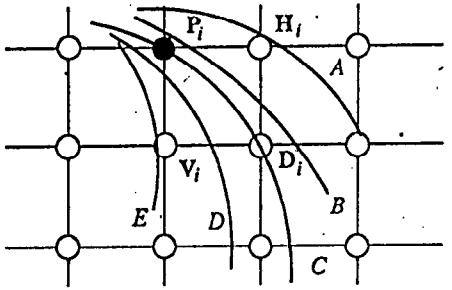
$$x = X_C + R\cdot\frac{1-t^2}{1+t^2} \qquad y = Y_C + R\cdot\frac{2t}{1+t^2}$$

може да се използва за решаването на някои геометрични задачи, но също като горната не може да се свърже директно с растера. Още повече разпределението на генерираните пиксели не е равномерно по протежение на нито една от координатните оси.



Фиг. 2-8

АЛГОРИТЪМ НА БРЕЗЕНХАМ ЗА ОКРЪЖНОСТ. Алгоритъмът на Брезеихам е целочислеи и се стреми да даде оценка за грешката, която се прави на всяка стъпка при избор на една или друга апроксимираща точка от растера. Нека отново разгледаме само първата четвърт от централна окръжност, намираща се в първи квадрант. Да приемем, че на i-тата стъпка в този алгоритъм е била избрана точката $P_i=(x_i,y_i)$. Следващият избор може да бъде само една от точките: $H_i=(x_i+1,y_i)$, $V_i=(x_i,y_i-1)$ или $D_i=(x_i+1,y_i-1)$, показани на фиг 2-9. На същата фигура са показани възможните начини (A,B,C,D,E), по които разглежданата част от четвърт-окръжността може да е разположена спрямо точките от растера.



Фиг. 2-9

Ще въведем оценка за грешката, която се прави при избирането на диагонално разположената точка чрез ориентираното й разстояние до истинската окръжност:

$$\Delta_i = D(\mathrm{D}_i) = (x_i+1)^2 + (y_i-1)^2 - R^2 \qquad [2.11]$$

Случай 1. Нека първо разгледаме какъв избор правим при $\Delta_i < 0$. Тогава окръжността е разположена както в случаите *A* и *B* от горната фигура и е естествено да се избере или $\mathrm{H}_i = (x_i+1, y_i)$, или $\mathrm{D}_i = (x_i+1, y_i-1)$. Ще въведем още една оценка, за да разграничим тези две възможности:

$$\delta_i = D(\mathrm{H}_i) + D(\mathrm{D}_i) = 2(x_i+1)^2 + (y_i-1)^2 + y_i^2 - 2R^2 \qquad [2.12]$$

а/В случая *A* трябва да се избере винаги H_i и както се вижда от фигурата тогава $D(\mathrm{H}_i) \le 0$ и $D(\mathrm{D}_i) < 0$, а следователно и $\delta_i < 0$;

б/В случай *B* е изпълнено $D(\mathrm{H}_i) > 0$ и $D(\mathrm{D}_i) < 0$ - т.е. двата члена на сумата [2.12] имат различни знаци и избраната точка би била H_i само ако $D(\mathrm{H}_i) \le D(\mathrm{D}_i)$.

От казаното тук можем да заключим, че при $\Delta_i < 0$ е необходимо разграничението:

- при $\delta_i < 0$ избираме H_i
- при $\delta_i > 0$ избираме D_i

Тук не използваме равенство, защото $\delta_i \neq 0$: уравнението $\delta_i = 0$, както ще видим по-късно, няма подходящо решение в цели числа. Нека сега запишем неравенството $\delta_i > 0$ като използваме [2.12], допълним до точния квадрат на $y_i - 1$ с прибавяне и изваждане на $(-2y_i+1)$ и след това прегрупираме подходящо членовете му:

$$\delta_i = 2\left[(x_i+1)^2 + (y_i-1)^2 - R^2\right] + 2y_i - 1 = 2(\Delta_i + y_i) - 1 > 0,$$

като за израза в квадратните скоби използваме дефиницията [2.11]. Следователно можем да кажем, че в разглеждания случай ($\Delta_i < 0$) ще избираме D_i само ако:

$$\Delta_i > -y_i + \frac{1}{2}.$$

От това неравенство можем да заключим, че $\delta_i \neq 0$. Тъй като неизвестните в неравенството са цели числа, то би било изпълнено само ако $\Delta_i > -y_i$. Да обобщим този случай:

- при $\Delta_i \le -y_i$ избираме H_i
- при $-y_i < \Delta_i < 0$ избираме D_i

Случай 2. Нека сега приемем, че $\Delta_i > 0$. Това съответства на разположенията *D* и *E* от фиг. 2-9. Както в предния случай, да въведем една нова оценка, за да разграничим тези две възможности:

$$\varepsilon_i = D(\mathrm{D}_i) + D(\mathrm{V}_i) = (x_i+1)^2 + 2(y_i-1)^2 + x_i^2 - 2R^2 \qquad [2.13]$$

Аналогично анализирайки възможностите за избор при *D* и *E*, ще заключим, че:

- при $\varepsilon_i > 0$ избираме V_i, а
- при $\varepsilon_i < 0$ избираме D_i.

Тук отново можем да изпуснем знака за равенство.

Замествайки в първото неравенство с [2.13] и както преди допълвайки до точен квадрат с прибавяне и изваждане на $(2x_i+1)$ получаваме:

$$\varepsilon_i = 2\left[(x_i+1)^2 + (y_i-1)^2 - R^2\right] + 2x_i - 1 = 2(\Delta_i - x_i) - 1 > 0,$$

или

$$\Delta_i > x_i + \frac{1}{2}$$

Както и в предния случай от целочислеността следва, че горното неравенство ще е изпълнено ако $\Delta_i > x_i$, при което трябва да се избере V_i. Обобщавайки ще заключим, че:

- при $\Delta_i > x_i$ избираме V_i, а
- при $0 < \Delta_i \le x_i$ избираме D_i.

Случай 3. Както се вижда ясно на фиг. 2-9 (разположение *C*), при $\Delta_i = 0$, окръжността минава точно през диагонално разположената точка и тя именно трябва да бъде избрана.

След като разгледахме всички възможни случаи за избор на следваща точка на *i*-тата стъпка в растеризирането на окръжността, сега можем да обобщим трите разгледани възможности по следния начин:

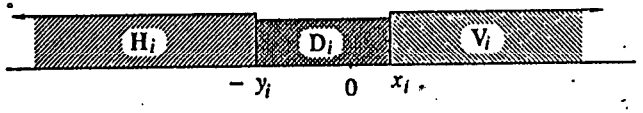
- Ако $\Delta_i < 0$ и $\Delta_i \le -y_i$ избираме точката H_i;
- Ако $\Delta_i > 0$ и $\Delta_i > x_i$ избираме точката V_i;
- Във всички останали случаи ще избираме D_i

Ако вземем предвид, че $y_i \ge 0$ за всяка точка от първата четвърт на окръжността, условието за избор на H_i се свежда до следните две:

- $\Delta_i \le -y_i$ за $y_i > 0$ и
- $\Delta_i < 0$ за $y_i = 0$.

Но $y_i = 0$ само когато е достигнат краят на дъгата, а тогава не е необходимо да продължаваме, т.е. не е необходимо да пресмятаме оценката за следващата стъпка. Следователно условието за избор на H_i е само $\Delta_i \le -y_i$.

Аналогично, тъй като $x_i \ge 0$, условието за избор на V_i се свежда до $\Delta_i > x_i$. Следващата фигура илюстрира направените изводи за избор на следваща точка в зависимост от стойността на Δ_i:



Фиг. 2-10

За оценката Δ_i ще изведем лесно рекурентна зависимост по формулата [2.11]. В началото $x_0=0$ и $y_0=R$ и следователно $\Delta_0=2(1-R)$. На всяка стъпка новата стойност на оценката зависи от предишния избор:

а/Изборът на точката H_i означава, че $x_{i+1}=x_i+1$ и $y_{i+1}=y_i$, а оттам и оценката е:

$$\begin{aligned}\Delta_{i+1} &= (x_{i+1}+1)^2+(y_{i+1}-1)^2-R^2=\left[(x_i+1)+1\right]^2+(y_i-1)^2-R^2\\ &=(x_i+1)^2+(y_i-1)^2-R^2+2(x_i+1)+1 \qquad [2.14]\\ &=\Delta_i+2(x_i+1)+1=\Delta_i+2x_{i+1}+1\end{aligned}$$

б/Изборът на V_i отговаря на $x_{i+1}=x_i$ и $y_{i+1}=y_i-1$, а оценката е:

$$\begin{aligned}\Delta_{i+1} &= (x_{i+1}+1)^2+(y_{i+1}-1)^2-R^2=(x_i+1)^2+\left[(y_i-1)-1\right]^2-R^2\\ &=(x_i+1)^2+(y_i-1)^2-R^2-2(y_i-1)+1 \qquad [2.15]\\ &=\Delta_i-2(y_i-1)+1=\Delta_i-2y_{i+1}+1\end{aligned}$$

в/При избор на точката D_i - $x_{i+1}=x_i+1$ и $y_{i+1}=y_i-1$, а оттам и оценката е:

$$\begin{aligned}\Delta_{i+1} &= (x_{i+1}+1)^2+(y_{i+1}-1)^2-R^2=\left[(x_i+1)+1\right]^2+\left[(y_i-1)-1\right]^2-R^2 \qquad [2.16]\\ &=\Delta_i+2(x_i+1)-2(y_i-1)+1=\Delta_i+2x_{i+1}-2y_{i+1}+1\end{aligned}$$

Следващата програма растеризира цялата окръжност по представения начин, като използва симетрията ѝ отиосно координатните оси. Случаят $y=0$ е изнесен вън от цикъла по същата причиии, поради която направихме това за алгорйтъма, използващ числа с плаваща запетая - за да се записва всеки пиксел само по ведиъж.

Ако искаме да използваме осемстранната симетрия, бихме могли да растеризираме само една осма от окръжността. Тогава вместо три възможности за избор на точки от растера: H_i, D_i и V_i, ще имаме само две: H_i и D_i. Оценката в такъв случай вместо Δ_i ще бъде δ_i, дефииираиа чрез [2.12].

```
void DrawBresenhamCircle(xc,yc,R,value)
int xc,yc,R; int value;
{int x,y,d;
    x=0;y=R;
    d=2-2*R;
    PutPixel(xc,yc+R,value);
    PutPixel(xc,yc-R,value);
    PutPixel(xc+R,yc,value);
    PutPixel(xc-R,yc,value);
    while (1) {
          if (d >-y) {y--; d+=1-2*y;}
          if (d <=x) {x++; d+=1+2*x;}
          if (y) return;
          FourSymmetric(xc,yc,x,y,value);
    }
}
```


Използвайки метода на Брезенхам, Михенер (Michener) предлага алгоритъм само за една осма от окръжността с използване на тази именно оценка. Рекурентните зависимости се извеждат по аналогичен на горния начин, а началната ѝ стойност е:

$$\delta_0=2(x_0+1)^2+(y_0-1)^2+y_0^2-2R^2=2+R^2-2R+1+R^2-2R^2=3-2R,$$

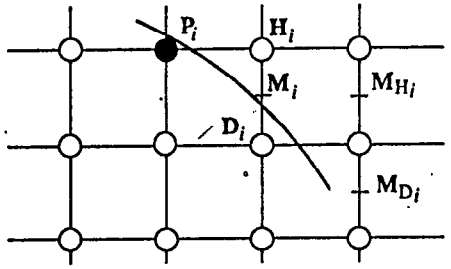
имайки пред вид, че началната точка е $(0,R)$.

Програмата, показаиа по-долу, осъществява растеризирането на пълна окръжност по този именно алгоритъм. В нея е възможно някои от пикселите да се запишат по два пъти. Читателят видя как може да се избягва това когато е необходимо в програмата `SimpleCircle` а и в по-горната реализация на алгоритъма на Брезенхам.

```
void DrawMichenerCircle(xc,yc,R,value)
int xc,yc,R; int value;
{int x,y,d;
    d=3-2*R; y=R;
    EightSymmetric(xc,yc,0,R,value);
    for (x=0; x<y; x++) {
          if (d >= 0) d+=10+4*x-4*(y--);
              else    d+= 6+4*x;
          EightSymmetric(xc,yc,x,y,value);
    }
}
```

АЛГОРИТЪМ НА СРЕДНАТА ТОЧКА ЗА ОКРЪЖНОСТ. Принципът на средната точка, който въведохме в 2.1.1 може да се приложи и при намиране на оценка за избор на следваща точка при дискретизация на окръжност. При това отново ще видим, че полученият алгоритъм почти не се различава от резултата на Брезенхам. Ще разгледаме растеризирането само на дъгата във втори октант на централна окръжност, т.е. когато x се мени от 0 до $R\sqrt{2}$.

Ако избраният пиксел на i-тата стъпка е $P_i=(x_i,y_i)$, следващият избор може да бъде или $H_i=(x_i+1,y_i)$ или $D_i=(x_i+1,y_i-1)$. Отново ще оценим положението на средната точка, за да определим избора.



Фиг. 2-11

Уравнението на окръжността е: $F(x,y)=x^2+y^2-R^2=0$. Функцията $F(x,y)$ има положителни стойности за точките извън окръжността и отрица-

телни за тези във вътрешността ѝ. Ще вземем за оценка стойността на функцията в средната точка:

$$d_i = F\left(x_i+1, y_i-\frac{1}{2}\right) = (x_i+1)+\left(y_i-\frac{1}{2}\right)^2 - R^2.$$

При $d_i \geq 0$, трябва да изберем D_i и оценката на $(i+1)$-вата стъпка ще е:

$$\begin{aligned} d_{i+1} &= F\left(x_i+2, y_i-\frac{3}{2}\right) = (x_i+2)^2+\left(y_i-\frac{3}{2}\right)^2 - R^2 \\ &= (x_i+1)^2+\left(y_i-\frac{1}{2}\right)^2 - R^2+2(x_i+1)-2\left(y_i-\frac{1}{2}\right)+2 \\ &= F\left(x_i+1, y_i-\frac{1}{2}\right)+2x_i-2y_i+5 = d_i+2x_i-2y_i+5 \end{aligned} \qquad [2.17]$$

Ако пък $d_i < 0$, трябва да изберем H_i, а тогава новата оценка ще е:

$$\begin{aligned} d_{i+1} &= F\left(x_i+2, y_i-\frac{1}{2}\right) = (x_i+1)^2+\left(y_i-\frac{1}{2}\right)^2 - R^2+2(x_i+1)+1 \\ &= F\left(x_i+1, y_i-\frac{1}{2}\right)+2x_i+3 = d_i+2x_i+3 \end{aligned} \qquad [2.18]$$

Остава да намерим и началната стойност на оценката в първата средна точка:

$$d_0 = F\left(1, R-\frac{1}{2}\right) = 1-\left(R^2-R-\frac{1}{4}\right)-R^2 = \frac{5}{4}-R$$

Дотук разсъжденията не се отличават от тези направени в извеждането на алгоритъма за отсечка. За да избегнем използването на дроби можем да въведем нова оценка $e = d-\frac{1}{4}$, която има целочислена начална стойност и се изменя целочислено.

Сравненията трябва да се заменят с $e \geq \frac{1}{4}$ и $e < \frac{1}{4}$, но тъй като e е цяло число, то те могат да се извършват и спрямо 0. Забележете, че ако положим

$$e = 2d+\frac{1}{2}$$

и извършваме сравненията спрямо 0, а не спрямо -1/2, ще получим точно алгоритъма на Михенер, показан по-горе.

ЧАСТНИ РАЗЛИКИ ОТ ВТОРИ РЕД. Алгоритмите, които разгледахме дотук, използват частни разлики от първи ред. Така се наричат разликите в изменението на функцията от уравнението на примитива в две съседни точки. Когато уравнението на примитива е линейно (напр. на отсечката), тогава тези частни разлики са константи, но тъй като окръжността се задава с уравнение от втори ред, тези нараствания са линейни функции на x и y - [2.17], [2.18]. При растеризиране на окръжност е по-удобно да се използват частни

разлики от втори ред, които пък са разликите в изменението на първите частни разлики. В този случай те са константи.

Нека означим с d_i^H изменението на d_i при избор на H_i, а с d_i^D нарастването на d_i при избор на D_i. Това са всъщност и частните разлики от втори ред за окръжността. В точката P_i:

$$d_i^H = 2x_i+3, \text{ а } d_i^D = 2x_i-2y_i+5. \qquad [2.19]$$

Когато се избира H_i, въведените втори частни разлики променят стойностите си по следния начин:

$$\begin{aligned} d_{i+1}^H &= 2(x_i+1)+3 = d_i^H+2, \\ d_{i+1}^D &= 2(x_i+1)-2y_i+5 = d_i^D+2. \end{aligned}$$

Аналогично, при избор на D_i тези разлики са:

$$\begin{aligned} d_{i+1}^H &= 2(x_i+1)+3 = d_i^H+2, \\ d_{i+1}^D &= 2(x_i+1)-2(y_i-1)+5 = d_i^D+4. \end{aligned}$$

Началните стойности d_0^H и d_0^D могат да се определят от [2.19] за точката с координати $(0,R)$. Функцията за растеризиране на окръжност, която показваме тук, е написана съобразно изведените формули и използва алгоритъма на средната точка.

```
void DrawMidpoint2OrderDiffCircle(xc,yc,R,value)
int xc,yc,R; int value;
{int x,y, d,dH,dD;
    d=1-R; y=R;
    dH=3; dD=5-2*R;
    EightSymetric(xc,yc,0,R,value);
    for (x=0; x<y; x++) {
        if (d<0) {d+=dH; dH+=2; dD+=2;}
           else  {d+=dD; dH+=2; dD+=4; y--;}
        EightSymetric(xc,yc,x,y,value);
    }
}
```

2.1.3. Растеризиране на дъга от окръжност

В този параграф ще покажем как можем да приложим метода на Брезенхам за растеризиране на произволна част от окръжност. Първото нещо, което е необходимо да си осигурим, е началната и крайната точка на дъгата да бъдат точки от растеризацията на окръжността. Ще представим един много прост, но ефективен начин за това:

Да разгледаме една точка $P = (x,y)$ в първи квадрант. Отдалечеността на тази точка от окръжността (грешката, която се допуска, като се използва тя за апроксимираща точка) се задава като $e(P) = F(x,y) = x^2+y^2-R^2$. При извеждането на алгоритъма на Брезенхам в 2.1.2 видяхме, че грешката в съседните точки се задава като:

$$e(x+1,y)=e(x,y)+2x+1$$
$$e(x,y+1)=e(x,y)+2y+1$$

Да започнем да се движим от т. P в посока към окръжността правейки по една стъпка (само по x или само по y) и пресмятайки новата грешка дотогава докато тази грешка намалява. Посоката на стъпката ще зависи от знака на грешката - ако е положителна (точката е извън окръжността), трябва да се движим наляво и надолу, а ако е отрицателна - надясно и нагоре.

Естествено е да правим стъпка по тази координатна ос, по която бихме получили по-малка нова грешка. Например ако точката е във вътрешността на окръжността, тогава трябва да избираме между $(x+1,y)$ и $(x,y+1)$. Ще изберем $(x+1,y)$ ако:

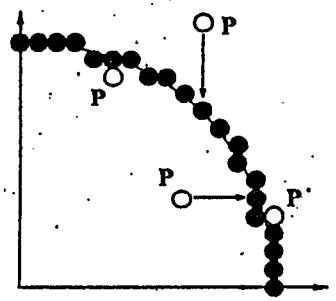
$$|e(x+1,y)|<|e(x,y+1)|$$

Нека приемем, че двете съседни точки, за които пресмятаме грешката, са също във вътрешността на окръжността. В такъв случай можем да се освободим от абсолютните стойности, тъй като и двете грешки ще бъдат отрицателни:

$$-e(x+1,y)<-e(x,y+1) \quad \Rightarrow$$
$$e(x+1,y)>e(x,y+1) \quad \Rightarrow$$
$$e(x,y)+2x+1>e(x,y)+2y+1 \quad \Rightarrow \quad x>y$$

Аналогично ще изберем $(x,y+1)$ ако $x<y$.

Ние си опростихме задачата приемайки, че оценката няма да сменя знака си при движение към окръжността. Дори и с тази уговорка този метод дааа добри резултати, ако точката е достатъчно близко до окръжността, което е вярно в огромната част от случаите (фиг. 2-12). Оставяме на читателя да докаже, че намерената по този начин точка винаги е част от растеризацията на окръжността. За по-отдалечени точки е необходимо да се предвиди придвижване и по диагонал. Представената функция позиционира една точка върху окръжността в общия случай, което налага да сравняваме абсолютните стойности на координатите. Използваният макрос `sign` дава знака на едно число, ако то не е 0, и 0 в противен случай.



Фиг. 2-12


```
void PutPointOnCentralCircle(x,y,R)
int *x,*y,R;
{int e,newe,dx,dy,dir;
    e=(*x)*(*x)-R*R+(*y)*(*y);
    dir=-sign(e);
    while (1) {
        if (abs(*y)>abs(*x)) {
            dx=0; dy=dir*sign(*y);
        } else if (*x) {
            dx=dir*sign(*x); dy=0;
        } else {
            dx=1; dy=0;
        }
        newe=e+2*(*x)*sign(dx)+2*(*y)*sign(dy)+1;
        if (abs(newe)>abs(e)) return;
        (*x)+=dx; (*y)+=dy;
        e=newe;
    }
}
```

За да извършим растеризирането на произволна част от окръжност е необходимо да следваме нарастването на ъгъла, без да използваме симетрията относно координатните оси, тъй като в общия случай дъгите не са симетрични.

Нарастването по x и по y е различно в различните квадранти, както това може да се види от стойностите на `incx` и `incy` в таблицата:

квадрант	*incx*	*incy*
I	-1	1
II	-1	-1
III	1	-1
IV	1	1

Оценката, която се използва в алгоритъма на Брезенхам е стойността на функцията в диагонално разположената точка спрямо текущо избраната, което може да се обобщи за произаолен квадрант като:

$$\Delta_i = D(\mathrm{D}_i)=(x_i+incx)^2+(y_i+incy)^2-R^2$$

Чрез разсъждения подобни на тези за I квадрант, можем да обобщим и рекурентните зависимости за тази оценка:

$\Delta_{i+1}=\Delta_i+ddx$, $\quad ddx=2\cdot incx\cdot x+1$, което е аналогично на [2.14]

$\Delta_{i+1}=\Delta_i+ddy$, $\quad ddy=2\cdot incy\cdot y+1$, аналогично на [2.15] и

$\Delta_{i+1}=\Delta_i+ddx+ddy$, което пък е аналогично на [2.16].

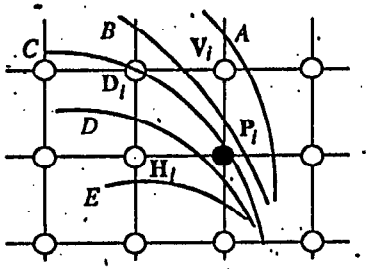
Тези нарастванията съответстват на стъпка или само по x, или само по y, или по диагонал. Необходимо е да отбележим, че ролите на x и y в нашите разсъждения ще са разменени, защото растеризирането ще извършваме по посока на нарастването на ъгъла, което в I квадрант съвпада с нарастването на y и намаляването на x - обратно на това, което разгледахме в 2.1.2. Допълнителната оценка, която въведохме с [2.12] сега е:

$$\delta_i = 2(x_i+incx)^2 + 2(y_i+incy)^2 + x_i^2 - 2R^2$$
$$= 2\left[(x_i+incx)^2 + (y_i+incy)^2 - 2R\right] - 2\cdot incx\cdot x_i - 1 = 2\Delta_i - (2\cdot incx\cdot x_i + 1)$$
$$= 2\Delta_i - ddx$$

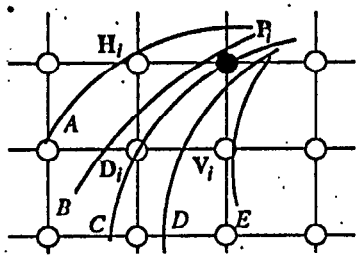
Тъй като $\Delta_i = D(\mathrm{D}_i) = D(\mathrm{P}_i) + ddx + ddy$, неравенството $\delta_i < 0$ ще се преобразува в:

$$2D(\mathrm{P}_i) + 2ddx + 2ddy < ddy \quad \Leftrightarrow \quad D(\mathrm{P}_i) + ddx + ddy < -D(\mathrm{P}_i) - ddx.$$

Сравненията за определяне на посоката, в която се прави всяка следваща стъпка в I и III квадранти, са еднакви. Например при $\delta_i < 0$ и в двата нечетни квадранта трябва да се избере вертикално отместеният пиксел. Не е така обаче в четните квадранти, както може да се види на фиг. 2-13.



I-ви квадрант



II-ри квадрант

$\Delta_i < 0$	$\delta_i = D(\mathrm{D}_i) + D(\mathrm{V}_i) < 0$	$\rightarrow$	V_i	$\Delta_i < 0$	$\delta_i = D(\mathrm{D}_i) + D(\mathrm{H}_i) < 0$	$\rightarrow$	H_i
	> 0	$\rightarrow$	D_i		> 0	$\rightarrow$	D_i
$\Delta_i > 0$	$\varepsilon_i = D(\mathrm{D}_i) + D(\mathrm{H}_i) > 0$	$\rightarrow$	H_i	$\Delta_i > 0$	$\varepsilon_i = D(\mathrm{D}_i) + D(\mathrm{V}_i) > 0$	$\rightarrow$	V_i
	< 0	$\rightarrow$	D_i		< 0	$\rightarrow$	D_i

Фиг. 2-13

Да разгледаме дефинициите на оценките и сравненията за първите два квадранта. Вижда се, че дефинициите на δ_i и ε_i трябва да са разменени за правилната работа на алгоритъма. Заедно с тях са сменени сравненията за оценката на Брезенхам. Например за избор на хоризонтално разположената точка в първи квадрант е необходимо:

$$\Delta_i > 0 \qquad D(\mathrm{D}_i) + D(\mathrm{H}_i) > 0$$

докато за същия избор във втори квадрант трябва да е изпълнено:

$$\Delta_i < 0 \qquad D(\mathrm{D}_i) + D(\mathrm{H}_i) < 0$$

Тази пълна симетричност ни дава основание да запазим дефинициите и сравненията, а да сменим само знака на оценката във втори квадрант. Ето защо алгоритъмът ще работи за всяка точка от окръжността, като ако тя се намира в четвърти квадрант, нейната оценка ще бъде с обратен знак. Остава да видим какво трябва да се промени когато на някоя итерация се достигне до коя да е от координатните оси, т.е. когато се сменя четността на квадранта. Нека да разгледаме прехода от първи във втори квадрант. Най-напред трябва да сменим знака на самата оценка, на втората частна разлика и на нарастването при вертикално движение. Да видим как се променят първите частни разлики *ddx* и *ddy*:

$$ddx' = -2\cdot incx'\cdot x - 1 = -2\cdot incx\cdot x - 1 = -ddx$$
$$ddy' = -2\cdot incy'\cdot y - 1 = -2\cdot(-incy)\cdot y - 1 = ddy - 2$$

В представената програма е използвана тази именно модификация на алгоритъма на Брезенхам, а нарастването на оценката се пресмята чрез втори частни разлики.

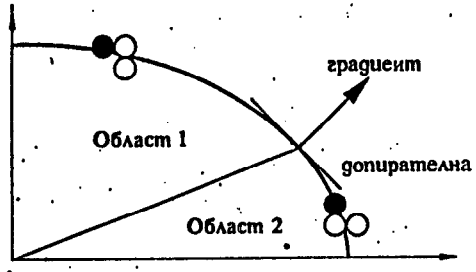
```
void DrawBresenhamArc(x,y,xc,yc,xe,ye,R,value)
int xc,yc,x,y,xe,ye,R; int value;
{int d,ddx,ddy,dd2,incx,incy;
    x-=xc; y-=yc;
    xe-=xc; ye-=yc;
    PutPointOnCentralCircle(&x,&y,R);
    PutPointOnCentralCircle(&xe,&ye,R);
     /* определяне на нарастванията в началния квадрант */
    incx=y?-sign(y):-sign(x);
    incy=x?sign(x):-sign(y);
     /* началните стойности на оценката и нейните нараствания */
    d=x*x+y*y-R*R;
    ddx=2*x*incx+1;
    ddy=2*y*incy+1;
    dd2=2;
    /* в II и IV квадранти разглеждаме оценката с обратен знак */
    if (incx==incy) {d=-d; ddy=-ddy; ddx=-ddx; dd2=-dd2;}
    while (1) {
          PutPixel(xc+x,yc+y,value);
          /* избор на следваща точка и обновяване на оценката */
          if (d+ddx+ddy>-d-ddx) {x+=incx; d+=ddx; ddx+=dd2;}
          if (d+ddx+ddy<-d-ddy) {y+=incy; d+=ddy; ddy+=dd2;}
          if (!x) (     /* достигната е оста Ox */
             d=-d; ddy-=dd2; incy=-incy; ddx=-ddx; dd2=-dd2;
          } else if (!y) {
                        /* достигната е оста Oy */
             d=-d; ddx-=dd2; incx=-incx; ddy=-ddy; dd2=-dd2;
          }
          if (x==xe && y==ye ) return;
    }
}
```

2.1.4 Растеризиране на елипса

Както споменахме по-горе, принципът на средната точка може да се приложи и за конични сечения. Ще покажем как може да се направи това за елипса. Уравнението на една централна елипса е:

$$F(x,y) = \left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1, \quad \text{или} \quad F(x,y) = b^2x^2 + a^2y^2 - a^2b^2 = 0. \qquad [2.20]$$

За разлика от окръжността, тук можем да използваме само 4-странна симетрия. От друга страна, трябва да разделим първи квадрант на две области: в първата, изборът може да се прави между хоризонтално H_i и диагонално D_i разположените съседни пиксели, а във втората - между разположените диагонално и вертикално D_i и V_i. В първата област допирателният вектор $[x,y]$ е такъв, че $x > y$, а във втората: $x < y$.



Фиг. 2-14

Границата между двете области е в точката, в която единичният допирателен вектор е [1,–1]. Това е точката, в която градиентът на F (който е вектор, перпендикулярен на допирателния) има посоката на вектора [1,1]. Тъй като

$$\operatorname{grad} F(x,y)=\left[\frac{\partial F}{\partial x},\frac{\partial F}{\partial y}\right]=\left[2b^2x,2a^2y\right],$$

за граничната точка (x,y) е изпълнено $b^2x=a^2y$.

Подобно на окръжността, функцията $F(x,y)$ има положителни стойности за точките извън елипсата и отрицателни за тези във вътрешността ѝ. Нека разгледаме само първата област. В нея оценката на всяка итерация е стойността на функцията в средната точка спрямо текущо избраната:

$$d_i=F\left(x_i+1,y_i-\frac{1}{2}\right)=b^2(x_i+1)^2+a^2\left(y_i-\frac{1}{2}\right)^2-a^2b^2.$$

При $d_i \geq 0$, трябва да изберем D_i и оценката на $(i+1)$-вата стъпка ще е:

$$d_{i+1}=F\left(x_i+2,y_i-\frac{3}{2}\right)=b^2(x_i+2)^2+a^2\left(y_i-\frac{3}{2}\right)^2-a^2b^2$$

$$=d_i+b^2(2x_i+3)+a^2(-y_i+2).$$

При $d_i < 0$ - избираме H_i. Рекурентната зависимост за оценката в този случай е:

$$d_{i+1}=F\left(x_i+2,y_i-\frac{1}{2}\right)=b^2(x_i+2)^2+a^2\left(y_i-\frac{1}{2}\right)^2-a^2b^2=d_i+b^2(2x_i+3).$$

Началната стойност на оценката в началото на първата област е:

$$d_0=F\left(1,b-\frac{1}{2}\right)=b^2+a^2\left(b-\frac{1}{2}\right)^2-a^2b^2=b^2+a^2\left(-b+\frac{1}{4}\right).$$

Аналогични разсъждения можем да направим и за втората област. За да можем да работим с целочислена оценка, нека да умножим навсякъде по 4, т.е. да използваме оценката $d_i=4F(M_i)$. В показаната програма тази оценка се изчислява директно при започване на растеризирането във всяка от областите, а нейното нарастване се пресмята чрез първи частни разлики. Читателят може да кодира алгоритъма по-ефективно, ако използва втори частни разлики.

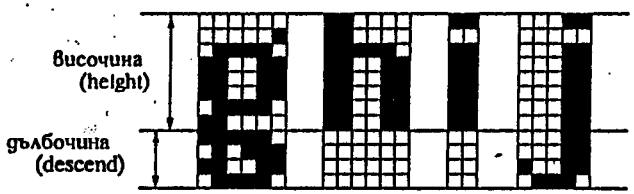
```
void DrawMidpointEllipse(xc,yc,a,b,value)
int xc,yc,a,b; int value;
{int x,y,d, asq,bsq;
    asq=a*a; bsq=b*b;
    x=0; y=b; d=4*bsq-asq*(1-4*b);
    while (asq*y>bsq*x) {                    /* Област 1 */
          FourSymmetric(xc,yc,x,y,value);
          if (d>=0) {
              d+=4*bsq*(2*x+3)+asq*(-2*y+2);
              y--;
          } else d+=4*bsq*(2*x+3);
          x++;
    }
    d=2*bsq*(2*x+1)*(2*x+1)+4*asq*(y-1)*(y-1)-4*asq*bsq;
    while (y>0) {                            /* Област 2 */
          FourSymmetric(xc,yc,x,y,value);
          if (d>=0) {
             d+=4*bsq*(2*x+2)+asq*(-2*y+3);
             x++;
          } else d+=4*asq*(-2*y+3);
          y--;
    }
}
```

Представеният алгоритъм може да се приложи само за елипси, чиито оси съвпадат с координатните. През 1967 год. Питуей (Pitteway) предлага общ алгоритъм за растеризиране на произволно ориентирано конично сечение. В него се разглежда общото уравнение на крива от втора степен и за оценка се взема отново средната точка между двата възможни избора на всяка стъпка. Вторите частни разлики са линейни функции, които могат да се изведат по начин, подобен на този, който прилагахме дотук. Допълнителното, което се прави, е изразяването на компонентите на градиентния вектор чрез първите и втори частни разлики на оценката. Това позволява бързо определяне на границите на октантите, в които става превключване на нарастванията по всяка от координатните оси. Ние няма да се спираме тук на този алгоритъм, но препоръчваме изучаването му от тези читатели, които имат определен интерес към растерната графика.

2.1.5 Растеризиране на символи

Съществуват няколко различни начина за изобразяване на текстови символи върху растерни дисплеи. Изборът на най-подходящия начин зависи от

конкретното приложение и от изчислителната мощност на графичната система. В повечето съвременни работни станции символите се задават с контурите си, представени чрез сплайн-криви на базата на средствата, описани в шеста глава. Изобразяването чрез сплайни е доста трудоемко и затова в персоналните компютри, а и в някои работни станции се използват по-прости методи.



Фиг. 2-15

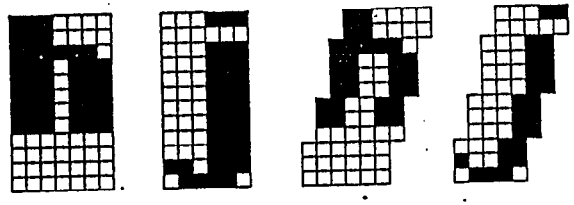
Да разгледаме един от тези методи:

Всеки шрифт се задава с набор от растерни макети за всеки един от символите, от които този шрифт се състои. Тези макети са малки правоъгълни матрици, които често имат фиксирани размери. Височината на всички символни макети е почти винаги еднаква, но ширината им може да е различна. Това е така за т.нар. *пропорционални шрифтове* - тези, за които ширината на буквите например "i" и "W" е различна. Глобални параметри са височината и дълбочината на символа (за тези символи, част от които е разположена под хоризонталния ред).

Там, където в макета стои 1, съответният пиксел трябва да бъде осветен. Вижда се, че е удобно всички символи в шрифта да са кодирани в един общ двумерен масив, като са известни ширините на всеки отделен символ. Изобразяването на един символ може да стане с показаната по-долу програма:

```
typedef struct FontDef {
                char height;
                char descent;
                int width[MAXSYMB];
                int pattern[MAXLEN][MAXHEGHT];
        } FontDef;
FontDef font;
. . .
void DrawTextSymbol(x,y,char,value)
int x,y,value; char char;
{int i,j;
    char-=FirstPrintableASCII;
    for (j=0; j<font.descend+font.height; j++)
        for (i=0; i<font.width[char]; i++)
            if (font.pattern[char+i][j])
                PutPixel(x+i,y-font.descend+j,value);
}
```

Обикновено за удебелени символи или такива в курсив се използва отделен, нов шрифт, но сравнително приемливи резултати могат да се получат чрез изобразяване с отместване (за удебеляване) и чрез трансформиране на макета (за курсив).



Фиг. 2-16

Друг, сравнително прост начин за представяне на символи е всеки от тях да се задава с последователност от вектори, като при изобразяване всеки вектор се растеризира по някой от начините, разгледани в 2.1.1. Този начин е подходящ за приложни програми, в които при мащабиране на текста не е желателно удебеляването му. Такива са например текстовите шрифтове, с които се анотират машинни чертежи. Този метод представлява един елементарен начин за геометрично моделиране, който ще разгледаме по-подробно в пета глава.

2.2 ЗАПЪЛВАНЕ НА ОБЛАСТИ И ГРАФИЧНИ ПРИМИТИВИ

В тази част ще се спрем на запълването на графични примитиви и растерни области, зададени чрез набор от пиксели. Необходимо е да обърнем внимание на разликата между тези две понятия. *Растерна област* или само *област* ще наричаме всяко множество от съседни пиксели, а от графичните примитиви ще разгледаме само затворените - многоъгълник и окръжност.

2.2.1 Запълване на области

В зависимост от това как е дефинирано понятието *съседни пиксели* има два типа области - 4-свързани и 8-свързани. 4-свързана е тази област, всяка точка на която може да бъде съединена с коя да е друга точка от областта (без да се напуска областта) с последователност от пиксели, всеки два съседни от които са разположени непосредствено един над друг или са хоризонтално един до друг (съседи или в реда или в стълба от растера, на който принадлежат). В 8-свързаните области един пиксел има за съседни пиксели още и четирите диагонално разположени пиксела.

В зависимост от това как са зададени, областите биват: *вътрешноопределени* и *граничноопределени*. Всяка област се задава спрямо някакъв фиксиран пиксел P. Вътрешноопределената област е максималното свързано (4- или 8-свързано) множество от пиксели имащи стойността на P. Може да се каже, че ако P има стойност OldValue, то пикселите по границата на областта имат стойности, различни от OldValue. Алгоритмите за запълване на такива области се споменават често под името "flood-fill" алгоритми. Областите, които са зададени чрез стойността BoundaryValue на пикселите по границата се наричат *граничноопределени*. Това е максималното свързано множество от съседни пиксели, съдържащо пиксела P, всеки от които има стойност, различна от предварително зададената BoundaryValue. За тях често се налага ограни-

чението всички пиксели от областта да имат стойност, различна от тази, с която се запълва.

ЗАПЪЛВАНЕ С ИЗПОЛЗВАНЕ НА РЕКУРСИЯ. Най-простите алгоритми за запълване от гледна точка на реализация са тези с използване на рекурсия. За тях е необходимо да е известна поне една точка, която принадлежи на областта, която и служи като входен параметър за програмата. Показаният по-долу фрагмент осъществява запълването на една вътрешноопределена област.

```
void SimpleFloodFill_4(x,y,NewValue,OldValue)
int x,y,NewValue,OldValue;
{   if (GetPixel(x,y)==OldValue) {
       PutPixel(x,y,NewValue);
       SimpleFloodFill_4(x-1,y,NewValue,OldValue);
       SimpleFloodFill_4(x+1,y,NewValue,OldValue);
       SimpleFloodFill_4(x,y-1,NewValue,OldValue);
       SimpleFloodFill_4(x,y+1,NewValue,OldValue);
    }
}
```

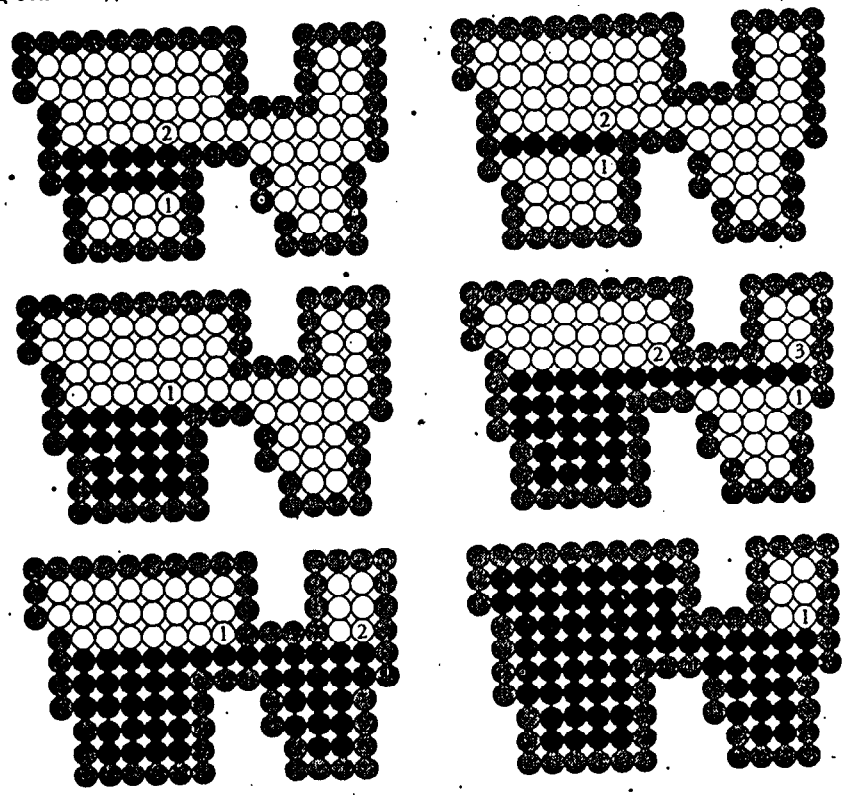
При 8-свързана област е необходимо рекурсивно обръщение и за диагонално разположените пиксели. Обърнете внимание, че този метод не би работил, ако старата и новата стойност на запълване съвпадат. Аналогично на програмата за запълване на вътрешноопределена област, тази за запълване на граничноопределена област има рекурсивно обръщение за всички пиксели, които още не са запълнени и които не принадлежат на границата ѝ.

```
void SimpleBoundaryFill_4(x,y,NewValue,BorderValue)
int x,y,NewValue,BorderValue;
{int value;
    value=GetPixel(x,y);
    if (value!=NewValue && value!=BorderValue) {
       PutPixel(x,y,NewValue);
       SimpleBoundaryFill_4(x-1,y,NewValue,BorderValue);
       SimpleBoundaryFill_4(x+1,y,NewValue,BorderValue);
       SimpleBoundaryFill_4(x,y-1,NewValue,BorderValue);
       SimpleBoundaryFill_4(x,y+1,NewValue,BorderValue);
    }
}
```

Вижда се, че запълването може да се обобщи като се използва множество от няколко вътрешни и няколко гранични цвята.

ЗАПЪЛВАНЕ С ОПТИМАЛНА ДЪЛБОЧИНА НА СТЕКА. Горните алгоритми далеч не са оптимални по отношение на използваните ресурси на компютърната система. Дълбочината на стека на рекурсията може да се намали съществено с предложения по-долу алгоритъм за граничноопределена област. Принципът, на който той работи е следният:

Първо се запълва целият хоризонтален ред, съдържащ началната точка и се определят неговите две граници `Xleft` и `Xright`. После се разглежда редът, който се намира непосредствено под вече запълнения ред. Започвайки от точката, разположена точно под тази с x-координата `Xleft` и движейки се отляво надясно се намира точката, която опира надясно в гранична точка. Тя се записва в стека, прескача се граничната (или граничните) точки, докато се намери нова група от хоризонтално разположени необработени точки от областта върху същия ред. Отново точката, която опира надясно в гранична точка се записва в стека. Тази процедура се повтаря докато разгледаме всички точки наляво от `Xright`. Същото се прави и за реда, намиращ се непосредствено над разглеждания ред. На фиг. 2-17 е показана последователността в запълването на една област и редът, в които точките се поставят в стека. Ще отбележим, че най-дясно разположената точка, която се записва в стека, не е непременно опряна отдясно в границата. Това е възможно в случай, че пикселът, намиращ се точно под (или над) дясната граница `Xright` на текущия ред е незапълнен пиксел от областта, а отдясно на него не стои граничен пиксел. Затова именно запълването на всяка група съседни пиксели от един ред става в двете посоки, започвайки от текущата точка.



Фиг. 2-17

Ще обърнем внимание и на това, че след като се извлече точка от стека, трябва да се провери дали тя не е вече запълнена от предишна итерация. Това може да се случи когато запълваме области, на които контурите са вло-

жени един в друг. По този начин ще се избегне неколкократно запълване на едни и същи пиксели. Забележете, че така предложеният алгоритъм е приложим само за 4-свързана област, тъй като съседните редове се разглеждат само в интервала под най-левия и най-десния запълнени пиксели на текущия ред.

В програмната реализация най-десният от всяка група незапълнени пиксели се намира, като се търси двойката *незапълнен-граничен* пиксели. Ако такава двойка се намери, незапълненият пиксел се акарва в стека. Отделно е разглеждането на пикселите под и над дясната граница. Ако някой от тях е незапълнен, той със сигурност принадлежи на група пиксели, на която още не е намерен най-десният. Следователно, този пиксел трябва също да се сложи в стека.

```
void StackedBoundaryFill_4(x,y,NewValue,BorderValue)
int x,y;
{int nexty;
    pushPoint(x,y);
    while (!isPointStackEmpty()) {
       popPoint(&x,&y);
       if (GetPixel(x,y)==NewValue)
           continue; /* този пиксел вече е запълнен */
       Xleft=Xright=x;
       /* намираме лявата граница на реда */
       while (GetPixel(Xleft-1,y)!=BorderValue ) Xleft--;
       /* намираме дясната граница на реда */
       while (GetPixel(Xright+1,y)!=BorderValue) Xright++;
       PutPixelRow(Xleft,Xright,y,NewValue);
       /* разглеждаме редовете непосредствено под и над текущия ред */
       for (nexty=y-1; nexty<y+2; nexty+=2) {
           p1=GetPixel(Xleft,nexty);
           for (x=Xleft;x<Xright;x++) {
               p2=GetPixel(x+1,nexty);
               /* дали това е двойка "незапълнен-граничен" пиксели */
               if (p1!=BorderValue && p1!=NewValue &&
                   p2==BorderValue) pushPixel(x,nexty);
               p1=p2;
           }
       }
    }
}
```

2.2.2 Запълване на многоъгълник

Запълването на пикселите, заградени от многоъгълник, зададен с последователност от върхове е задача, която се налага да се решава извънредно често в компютърната графика. Например визуализирането на едноцветна стена на обемно тяло се свежда до запълването на проекцията на тази стена, която представлява многоъгълник.

При разработването на ефективни алгоритми за запълване е важно да се подхожда по различен начин в зависимост от това какъв е многоъгълникът. Ако този многоъгълник е правоъгълник със страни, успоредни на координатните оси, неговото запълване би било елементарно и би било неразумно да се използват общи алгоритми. Това е важен принцип в базовите алгоритми за визуализация - да се използват по-прости методи за по-простите случаи. Например в системата PHIGS се прави разлика между обработката на изпъкнали и неизпъкнали многоъгълници. Приложният програмист указва типа на многоъгълника при създаването му, за да може графичната система правилно да подбира най-ефективния алгоритъм за работа с него - в частност за запълването му.

ЗАПЪЛВАНЕ ЧРЕЗ ИНВЕРТИРАНЕ НА ПИКСЕЛИ. Ако разгледаме един ред от растера, можем да кажем, че в общия случай пресечните точки на този ред с ребрата на многоъгълника са четен брой. Всяка двойка такива точки (нечетна и следващата я четна) загражда група от пиксели, които са вътрешни за многоъгълника. Ще наречем пикселите в тези пресечни точки *гранични пиксели*. Задачата ни тогава се свежда до намирането на последователността от двойките гранични пиксели за всеки ред от растера.

Граничните пиксели можем да получим като растеризираме ребрата по някой от стъпковите алгоритми, разгледани в 2.1.1. Ако пикселите от вътрешността на многоъгълника трябва да се установят в новия цвят в режим "изключващо или", можем да използваме един елементарен, но не твърде ефективен начин за запълване:

1. Растеризираме всички ребра и намираме всички гранични пиксели;
2. За всеки граничен пиксел инвертираме стойностите на всички пиксели от същия ред, разположени вдясно от него. Инвертирането се извършва като използваме режим на записване xor с S = 1.

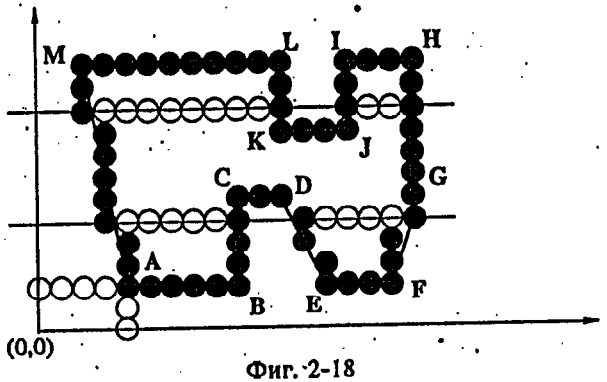
```
void InvertPolyFill(Polygon,N,NewValue)
Point Polygon[]; int N, NewValue;
{. . . /* дефиниране на локалните променливи */
      /* от алгоритъма на Брезенхам */
   SetWritingMode(XOR);
   X1=Polygon[N-1].x; Y1=Polygon[N-1].y;
   for (i=0;i<N;i++) {
      X2=Polygon[i].x; Y2=Polygon[i].y;
      . . .
      /* инициализацията на цикъла в алгоритъма на Брезенхам */
      . . .
      while (n--) {
         if (reverse) PutPixelRow(y,MAX_X,x,value);
                 else PutPixelRow(x,MAX_X,y,value);
      . . .
      }
      X1=X2; Y1=Y2;
   }
   SetWritingMode(REPLACE);
}
```

По този начин всички пиксели между една нечетна и следващата я четна гранични точки ще се инвертират нечетен брой пъти (т.е. в крайна сметка в тях ще се установи новата стойност), а всички пиксели между една четна и следващата я нечетна ще се инвертират четен брой пъти. Тъй като прилагането на "изключващо или" (xor) с една и съща стойност четен брой пъти

води до възстановяване на началното състояние, то пикселите, които не принадлежат на вътрешността на многоъгълника, ще останат непроменени. Остават особените случаи по границата на многоъгълника, но за тях можем да въведем отделно правило, както ще видим по-долу.

Този алгоритъм може да се кодира като инвертирането на реда се прави при последователното получаване на всеки граничен пиксел, растеризирайки контура. Показаната програма е проста модификация на алгоритъма на Бресенхам.

АЛГОРИТЪМ НА СКАНИРАЩИЯ РЕД. По-ефективно е, разбира се, запълването между двойките гранични пиксели да се извършва като итерациите се правят съобразно редовете от растера, а не за всяко едно от ребрата поотделно. В тази част ще разгледаме един алгоритъм за запълване, който се основава на тази именно идея и който може да се приложи не само за многоъгълник, но и за произволна едносвързана област, зададена с вложени контури, както и за самопресичащ се контур. Тъй като запълването се извършва по редове, той носи името *алгоритъм на сканиращия ред*.



Фиг. 2-18

Нека преди да започнем запълването да направим следното:

1. Отстраняваме всички хоризонтални ребра;
2. Подготвяме следната информация за всяко ребро:

`Ymin`: най-малката y-координата на реброто (това е y-координатата на по-долния му край);

`Ymax`: най-голямата y-координата на реброто (това е y-координатата на по-горния край);

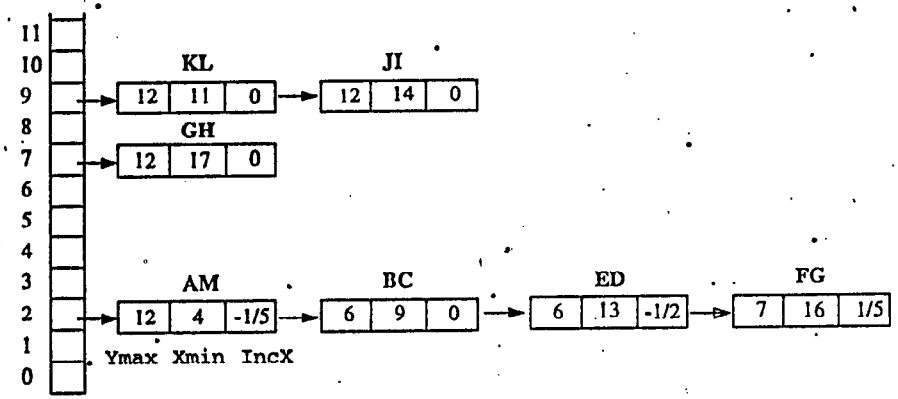
`Xmin`: x-координата на по-ниския край на реброто (това е x-координатата на този край, чиято y-координата е `Ymin`);

`IncX`: изменение по x при нарастването на y с единица.

Последното можем да получим директно от уравнението на правата, носеща реброто, тъй като то не е хоризонтално. Ще използваме уравнение, подобно на [2.1], но спрямо x:

$$x = IncX(y - Y_1) + X_1, \qquad IncX = \frac{dx}{dy} = \frac{X_2 - X_1}{Y_2 - Y_1}$$

3. Тази информация за ребрата подреждаме в таблица, в която за всяко y се съдържа сортиран списък спрямо `Xmin` от всички ребра, чиито `Ymin`=y. Ще наречем тази таблица *таблица на ребрата* - ТР. Отчитайки, че точката A има координати (4,2), многоъгълникът от фиг. 2-18 ще има следната ТР:



Фиг. 2-19

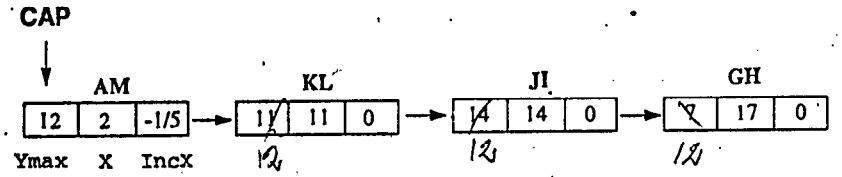
Алгоритъмът на сканиращия ред използва тази таблица, за да поддържа на всяка стъпка (за всеки текущо обработван ред от растера) един списък от всички ребра, които имат пресечни точки с този ред. Този списък ще наричаме *списък на активните ребра* - САР. Всеки елемент от него характеризира по едно ребро и съдържа данни, подобни на тези в ТР:

`Ymax`: най-голямата y-координата на реброто (също като в ТР);

`X` : x-координатата на пресечната точка на реброто с реда;

`IncX`: също като в ТР.

За сканиращия ред y=10 САР за същия многоъгълник (от фиг. 2-18) ще има вида, показан на фиг. 2-20.



Фиг. 2-20

Можем да получим списъка на следващия ред от растера - y=11 - директно от показания САР като за всяко ребро увеличим `X` с `IncX`. Тъй като е възможно от следващия ред да започва ново ребро, то е необходимо да добавим и съответния подсписък от ТР за новото y. Разбира се, ще е необходимо

след това да сортираме списъка, особено ако желаем алгоритъмът да работи за области с неизпъкнали контури. Някои от ребрата могат да не достигат до следващия ред, което лесно можем да установим от техните Ymax стойности. Те пък ще трябва да се изтрият от CAP.

Ето и самия алгоритъм:

1. Построяваме TP за многоъгълника;
2. Инициализираме CAP като празен списък;
3. Даваме на y стойността на най-малкото y в TP;
4. Повтаряме до момента, в който и TP, и CAP станат празни:
   4.1. Добавяме към CAP списъка от TP за текущото y. Това е сливане на два сортирани списъка и може да се осъществи доста ефективно;
   4.2. Запълваме пикселите между всяка двойка от последователни нечетна-четна пресечни точки от CAP;
   4.3. Увеличаваме y с 1;
   4.4. Изтриваме от CAP всички ребра за които Ymax=y;
   4.5. За всяко ребро в CAP увеличаваме X с IncX;
   4.6. Сортираме CAP по нарастване на x. Тъй като предишните две стъпки само леко нарушават сортирането, то тази операция се извършва върху почти сортиран списък.

Този алгоритъм се справя добре с особените случаи, когато броят на пресечните точки на сканиращия ред с ребрата на многоъгълника е нечетен. Това се случва тогава, когато някой от върховете на многоъгълника лежи на реда. Тъй като ребрата са винаги ориентирани (от долния към горния край), а според алгоритъма всички ребра за които Ymax=y се изтриват от CAP, то горният връх на всяко ребро не се счита за пресечна точка. И тъй като всеки връх е край на точно две ребра, то четността винаги ще се запазва. В горния пример, когато сканиращият ред пресича върха G, в CAP ще има отбелязана една пресечна точка, защото G участва в GH като негов долен връх, но не и в FG.

Веднага се вижда, че това правило ще предизвика несиметрично третиране на върхове, които са едновременно долни и едновременно горни краища на своите ребра. Връх, който е долен край и на двете ребра, които се събират в него ще се запълни, докато такъв, който е горен край на две съседни ребра няма да бъде запълнен. Аналогично, хоризонталните ребра ще се растеризират в зависимост от това дали вътрешността на многоъгълника се намира от долната или от горната им страна. Например ребрата AB, EF и KJ от същия пример ще се растеризират, докато CD, ML и IH няма.

Такова разграничение е необходимо да се прави особено когато се визуализира група от многоъгълници, които имат общи страни. При използване на представения алгоритъм общите им хоризонтални ребра ще се растеризират само по веднъж. Това обаче не е вярно за останалите, нехоризонтални ребра. Причината е, че за растеризирането им се използва метод, пригоден за отсечки, в който не се отчита от коя страна е разположена вътрешността на многоъгълника.

За решаването на този проблем първо трябва да се дадат отговори на следните въпроси:

- Кой пиксел да изберем за граничен, когато пресечната точка на едно ребро със сканиращия ред е пиксел от растера? Винаги ли самият пиксел-пресечна точка е най-правилният избор?
- Кой пиксел да изберем за граничен, когато пресечната точка на едно ребро със сканиращия ред НЕ е пиксел от растера, т.е. когато е необходимо закръгляване?

За да отговорим на първия въпрос можем да приемем правилото да включваме всички пиксели, които са начало на интервал за запълване, т.е. стоят на нечетно място в CAP. Това ще означава, че вертикалните страни, които са отляво ще се визуализират, докато тези, които са отдясно няма.

```
typedef Edge *EdgePointer;
typedef struct Edge {
        EdgePointer next;
        int Ymax;
        float X, IncX;
} Edge;

void ScanLineFillPolygon(Polygon,N,NewValue)
Point Polygon[]; int N, NewValue;
{EdgePointer current,previous;
 EdgePointer TP[YMAX_RASTER],CAP;
 int x1,x2,y,ymin;
    CreateTP(TP,Polygon,N,&ymin); y=ymin;
    while (CAP && isEmptyTable(TP,y)) {
          AppendCAP(CAP,TP[y]);
          SortCAPbyX(CAP);
          current=CAP;
          while (current) {
            /* закръгляваме към вътрешността */
                x1=ceil(current->X); current=current->next;
                x2=floor(current->X);
            /* изключваме дясната граница ако е част от растеризацията*/
                if (current->X==x2) x2--;
                if (x1<=x2) PutPixelRow(x1,x2,y,NewValue);
                current=current->next;
          }
          y++; current=CAP; previous=NULL;
          while (current) {
                if (current->Ymax==y)
                   /* изключваме реброто от CAP */
                   if (previous) previous=current->next;
                   else CAP=current->next;
                else
                   /* обновяваме полето X на реброто */
                   current->X+=current->IncX;
                current=current->next;
          }
    }
}
```

Закръгляването ще извършваме в зависимост от това дали пресечната точка е начало на интервал за запълване или не. Ако тя е отляво на такъв

интервал, т.е. стои на нечетно място в CAP - закръгляваме нагоре (към вътрешността), ако пък тази пресечна точка стои на четно място - закръгляваме надолу (отново към вътрешността). Показаната програма илюстрира приетите правила.

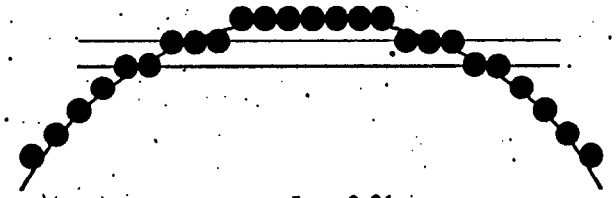
Тъй като IncX е рационално число, алгоритъмът може да се преобразува в целочислен, като се съхраняват отделно числителят и знаменателят на това число. Закръгляването надолу тогава извършваме, като за всяка пресечна точка X пазим и числителя на дробната ѝ част. На всяка стъпка ще прибавяме към него числителя на нарастването и ако той е по-голям от знаменателя, това означава, че цялата част се увеличава с единица, а дробната се намалява със знаменателя на нарастването.

В началото на този параграф отбелязахме, че е необходимо да се опростяват алгоритмите винаги, когато това е възможно. Запълването на изпъкнал многоъгълник е удачно да се прави без да се използват толкова сложни структури от данни, тъй като всеки сканиращ ред пресича изпъкналия многоъгълник точно два пъти. Това важи и за окръжности и елипси, както ще видим по-долу.

Може да се каже, че тъй като всеки многоъгълник може да се представи като множество от изпъкнали, запълването му може да се сведе до запълването на тези многоъгълници. Същото се постига и чрез триангулация на многоъгълника - разбиването му на триъгълници. Това обаче са известни задачи от изчислителната геометрия, чието решаване не е тривиално.

2.2.3 Запълване на окръжност и елипса

Запълването на окръжност може да се извърши, като отново се използва идеята на *сканиращия ред*. Както казахме, в този случай сканиращият ред пресича окръжността точно два пъти и при това двете пресечни точки са симетрично разположени спрямо центъра ѝ.



Фиг. 2-21

Можем да използваме алгоритъма на Брезенхам за растеризиране на окръжността, при който на всяка стъпка се осветява пиксел или от същия ред или от този под него. Трябва да вземем предвид, че при генериране на най-горната (и респективно най-долната) ѝ част - когато $|y| > x$ - на един ред могат да се получат по няколко пиксела от всяка страна както е показано на фиг. 2-21.

От тях за десен граничен пиксел трябва да изберем този пиксел от дясната група, който е разположен най-отдясно и се намира във вътрешността на

окръжността. Левият граничен пиксел можем да получим чрез симетрия. За да определим дали пикселът е вътрешен за окръжността, можем да проверим знака на $F(x,y)$ в тази точка. В алгоритъма на Брезенхам стойността на тази функция можем да получим директно от оценката, дефинираиа с [2.11]. От тази дефиниция директно следва:

$$F(x_i, y_i) = d_i - 2x_i + 2y_i - 2.$$

Може да се случи така, че от цялата група десни гранични пиксели нито един да не е вътрешен. Тогава за граничен пиксел в сканиращия ред трябва да изберем този от двата съседни пиксела, който има най-голяма отрицателна стойност F.

В случаите, когато граничният пиксел лежи точно върху окръжността, ще приложим правилото, прието в предния параграф, а именно да включваме левите и да изключваме десните граници. Това може да е неприемливо за някои приложения, тъй като не се запазва симетричността, която е важна характеристика на окръжността.

2.2.4 Запълване с образец

Образецът е най-често малка правоъгълна $M \times N$-матрица от нули и единици, подобна на тази, използвана за визуализиране на текстови символи. Образецът може да съдържа не само 0 и 1, но и стойности, с които се дефинира цвят, макар че това се прави много рядко. Запълването на една област с образец се извършва като всеки неин пиксел се осветява съобразно стойността в съответен нему елемент от образеца. Това съответствие е различно в различните растерни системи. Възможни са следните варианти:

- *Образецът е свързан с екрана* (или с текущия прозорец върху екрана). Това означава, че елементът с координати (0,0) от образеца съответства на началото на координатната система. Всеки вътрешен пиксел от областта с координати (x,y) ще има за съответен елемента със същите координати, но взети всяка по модул размера на образеца по тази ос, а именно: ($x \bmod M$, $y \bmod N$);
- *Образецът е свързан с областта*, т.е. началото му съвпада с някоя характерна нейна точка. Типичен проблем в този случай е изборът на такава опорна точка, особено когато става дума за произволен многоъгълник. Най-лесно е да се вземе първият връх от списъка върхове или най-левият и най-долен такъв. Съответствието между пикселите тогава ще е:

$$(x,y) \rightarrow ((x-x_0) \bmod M,\ (y-y_0) \bmod N)$$

Запълването е различно и в зависимост от това дали всички пиксели от областта трябва да получат нова стойност. То може да бъде:

- Прозрачно запълване: Запълват се само тези пиксели, чиито съответни елементи от образеца са ненулеви:

```
if (pattern[x%M][y%N]) PutPixel(x,y,value);
```

- Плътно запълване: Запълват се всички пиксели - тези, за които в образеца стои 0, се запълват със стойността на фона (в нашия пример - стойността на променливата background), а останалите - с основния цвят (foreground):

```
PutPixel(x,y,pattern[x%M][y%N]?foreground:background);
```

При плътно запълване е възможно да се записват пикселите не един по един, а на групи от по цял хоризонтален ред от образеца.

Един друг начин за генериране на примитив, запълнен с образец е първо да се растеризира той в правоъгълна работна област с размерите на правоъгълната му обвивка (минималния изправен правоъгълник, които изцяло обхваща примитива). След запълването в тази работна област тя се визуализира върху екрана. Това е два пъти по-сложно от разгледания преди малко начин, но е полезно ако:

- съответният примитив ще се генерира много често, какъвто е случаят с текстови символи и икони; и/или
- графичната система има ефективна реализация на функцията за запис на блок от пиксели върху екрана; и/или
- запълване с два цвята, всеки от които е различен от фоновия.

2.3 ВИЗУАЛНИ АТРИБУТИ НА ГРАФИЧНИТЕ ПРИМИТИВИ

Дотук разгледахме растеризирането на отсечка и окръжност с плътна линия, която има дебелина един пиксел. В чертожните системи също толкова често се използват пунктирани, тънки и дебели линии за улесняване на интерпретацията на чертежа.

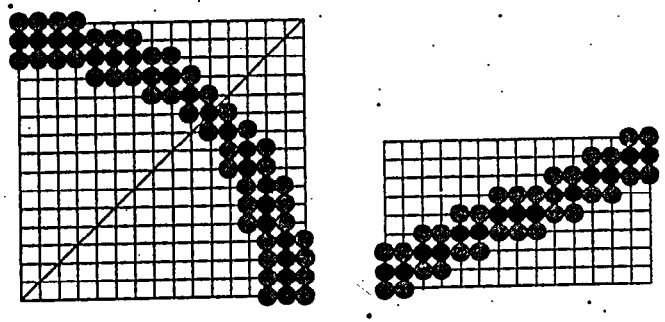
Възможностите на растерните дисплеи за визуализация на чертежи съвсем не отговарят на възможностите, които един чертожник има върху простия бял лист. Това обаче не пречи да се разработват средства за имитиране на най-важните графични атрибути от инженерните чертежи, така че създаваните визуални образи да могат да се възприемат и интерпретират като чертежи.

В тази част ще разгледаме няколко основни метода за растеризиране на линии с различна дебелина и тип. Ще се спрем и на съчетаването на тези атрибути и ще посочим някои особености на растеризирането на начупени линии, които са съставени от удебелени и/или пунктирани примитиви.

2.3.1 Удебеляване на примитиви

Има няколко основни метода за удебеляване на примитиви. Най-важните от тях са: повторение на пиксели; имитиране на следата, оставяна от движещо се перо с определено сечение; запълване на областта между два образа на примитива, отместени един от друг на разстояние, съответстващо на неговата дебелина. Всеки от тези методи има своите предимства и недостатъци, но основната разлика е в това как се прави балансът между *добър визуален резултат* и *приемлива изчислителна сложност*.

УДЕБЕЛЯВАНЕ ЧРЕЗ ПОВТОРЕНИЕ НА ПИКСЕЛИ. Това е най-простият метод, който има и минимална изчислителна сложност, защото е елементарно разширение на алгоритъма за растеризиране. Нека да разгледаме отсечка, чийто наклон е не по-голям от 45 градуса. Във всеки стълб на растера има най-много по един пиксел от тази отсечка. Естествено удебеляване би се получило, ако при растеризирането ѝ вместо по един пиксел осветяваме симетрично и неговите съседни във всеки стълб. Аналогично, когато отсечката има наклон по-голям от 45 градуса ще повтаряме пикселите по редове. При окръжността повторението ще се извършва в реда или в стълба в зависимост от това в кой квадрант се намира пикселът.



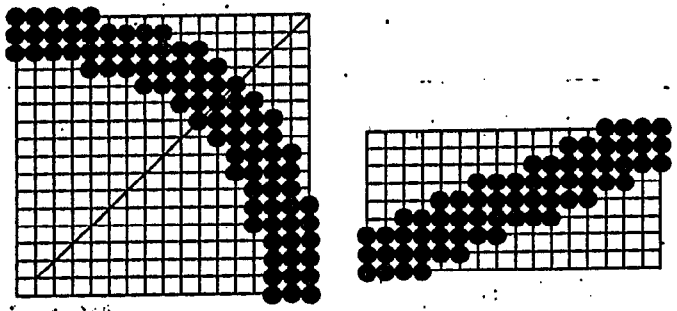
Фиг. 2-22

Типични проблеми при този подход са следните:

- Краищата на примитивите са хоризонтално или вертикално отрязани.
- Има забележимо изтъняване при увеличаване на наклона (от 0 към 45 градуса), което е особено ярко изразено в точките на превключване на повторението от ред към стълб в окръжността (в границите на октантите).
- Върховете на един удебелен по този начин многоъгълник няма да изглеждат добре. Например ъглите на едии правоъгълник, страните на който са удебелени чрез повторение на пиксели ще изглеждат поръбени, защото хоризонталните страни са удебелени вертикално, а вертикалните - хоризонтално.
- Удебеляването няма да е симетрично, ако зададената дебелина определя четен брой пиксели във всеки стълб (или ред).

ДВИЖЕЩО СЕ ПЕРО. Имитирането на следата, която оставя перо със зададен профил при движението си по пикселите на растеризацията е много често използван подход. На всяка стъпка профилът се разполага така, че центърът му да е в пиксела, получен при растеризирането. Тук естествено възникват няколко въпроса:

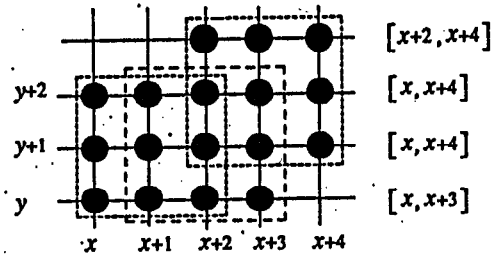
- Какво е най-подходящото сечение на перото?
- Трябва ли сечението да се ориентира спрямо примитива, който се удебелява?



Фиг. 2-23

Обикновено се избира правоъгълно сечение, което има винаги една и съща ориентация, т.е. не е свързано с примитива. За разлика от предния подход, това би довело до по-добре оформени краища, макар и значително удебелени. Самото удебеляване отново не е равномерно за всички наклони, като този път най-тънки са хоризонталните и вертикални участъци. Последното може да се избегне, като се ориентира сечението или се избере сечение с формата на кръг, което е симетрично. Допълнителен проблем при използването на перо с кръгло сечение е, че алгоритъмът трябва да е пригоден за запълване на кръг с радиус, който не е цяло число, а именно ($R=W/2$).

Този метод може лесно да се съчетае с алгоритъма на Брезенхам, като на всяка стъпка от растеризирането се запълва сечение с център генерирания пиксел. Тогава възниква проблемът, че ще има пиксели, които ще се запълват многократно. Както казахме и преди, това трябва да се избягва, особено в режимите **and**, **or** и **xor**. Едно разрешение е да се използва принципът на сканиращия ред, т.е. записването на пиксели да се извършва ред по ред, след като за всеки ред се намерят лявата и дясна граници на групата съседни пиксели.

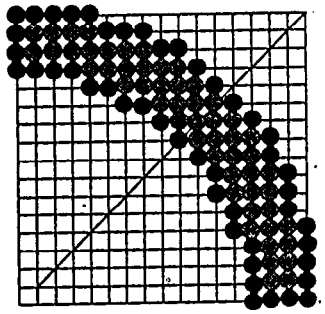


Фиг. 2-24

При удебеляване на отсечка на всяка стъпка ще получаваме интервали за няколко съседни реда, които трябва да обединяваме с интервалите, получени при налагане на сечението върху предходните пиксели от растеризацията. При окръжност, както и при многоъгълник, за всеки сканиращ ред ще има набор от интервали, което води до използването на същите структури като в алгоритъма на сканиращия ред.

УДЕБЕЛЯВАНЕ ЧРЕЗ ЗАПЪЛВАНЕ. Този метод дава най-добри визуални резултати и може да се реализира чрез създаване на област, която впоследствие се запълва по някой от разгледаните в предната част методи. Контурът на областта се получава при отместване на примитива по посоката на нормалата му на $W/2$ и $-W/2$. Ако примитивът е граница на друга област, то граници на удебеляването могат да бъдат самият той и образът, отместен на W към вътрешността на тази област. Така би се решил и проблемът за четното удебеляване.

Генерирането на отместени образи на елипси води до решаване на уравнения от 8-ма степен, затова разширяването и свиването им се апроксимира чрез модифициране на всяка от полуосите с $W/2$.



Фиг. 2-25

КРАЙНИ ТОЧКИ И ВЪРХОВЕ НА МНОГОЪГЪЛНИЦИ. Важен проблем на удебеляването е оформянето на крайните точки на примитивите и върховете на многоъгълниците. На фигурата по-долу са показани някои от възможностите да се направи това за една отсечка. Изборът на типа на удебеляването на крайната точка зависи както от нуждите на приложната програма, така и от това дали примитивът е самостоятелен или е свързан с други. В повечето съвременни графични системи се предлагат по няколко възможности, от които потребителят или приложният програмист избира тази, която е най-подходяща за конкретния случай.

Фиг. 2-26

Оформянето на върховете на един многоъгълник може естествено да се контролира чрез избора на типа на краищата или чрез отсичане на удебеляването по ъглополовящата на върха. Последното се нуждае от допълнителен контрол – например при скосяване на много остър връх.

Накрая ще отбележим, че общият проблем на удебеляването е подобен на задачата за построяване на еквидистанта на даден геометричен елемент (елемент, отстоящ на зададено разстояние от разглеждания), която като аналитична задача заслужава да бъде разгледана отделно.

2.3.2 Използване на типове линии

Изобразяването на примитив със зададен тип линия (пунктирана, централна, осева и т.н.) прилича много на запълването на област с образец. По тази причина можем да използваме същия термин, само че този образец сега ще е линеен - вектор състоящ се от нули и единици. Ето как могат да се представят няколко различни типа линии: пунктирана (от дълги тирета), точкова, осева (тире и точка) и пунктирана (от къси тирета). Дължината на вектора-образец в показания пример е 12 пиксела:

```
Pattern dashed    = {1,1,1,1,1,1,0,0,0,0,0,0},
        dotted    = {1,0,1,0,1,0,1,0,1,0,1,0},
        dashdot   = {1,1,1,1,1,1,0,0,0,1,0,0},
        shortdash = {1,1,1,0,0,0,1,1,1,0,0,0};
```

Аналогично на запълването на област с образец, типовете линии могат да се използват в два режима: прозрачно запълване и плътно запълване, които се реализират по следния начин:

```
if (pattern[i%N]) PutPixel(x,y,value)
```

или

```
PutPixel(x,y,pattern[i%N]?foreground:background);
```

В практиката най-често се използва прозрачно запълване, което означава, че пикселите, намиращи се между тиретата на една пунктирана отсечка няма да бъдат променяни. Много растерни системи предлагат само едната възможност (като пример ще посочим, че Windows-NT предлага само плътно запълване при работа с типове линии).

Най-лесно за кодиране е за променлива i да се избере нарастването (по x или по y в зависимост от наклона) в алгоритъма за растеризиране. Това обаче би довело до различна дължиина на пунктира за отсечки с различни наклони, което за някои приложения може да е неприемливо.

Друг проблем е, че удебеляването на пунктираните линии не може да се извърши при съчетаването на този с някой от разгледаните по-горе методи без да се рискува за някои случаи да се получи лош визуален резултат. При визуализиране на инженерни чертежи е най-добре една пунктирана отсечка да се разбие на множеството от малки отсечки, съставящи пунктира, и всяка от тях да се удебелява поотделно.

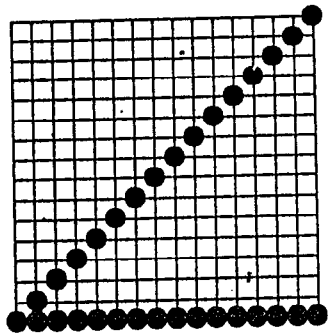
За разлика от образеца при запълване на области, този се привързва към обекта - почти винаги към неговата начална точка, а не към екрана. Единственото изключение са начупените линии, при които образецът трябва се привързва към началото на начупената, а не към началото на всяка нейна отсечка.

2.4 ИЗГЛАЖДАНЕ НА РАСТЕРИЗАЦИЯТА

Всички разгледани дотук алгоритми генерират примитиви, които имат малко или повече стъпаловидна форма. Особено отчетливо това се вижда върху дисплеи с ниска разрешаваща способност (със сравнително малък брой пиксели върху единица площ). Причината за това е, че алгоритмите са основани на принципа "всичко или нищо", т.е. пикселите са или осветени или не. Това, разбира се, е и единствената възможност за устройства, в чиято пикселна карта има отделен само по един бит за пиксел.

Тъй като използването на дисплеи с повече от една *равнини* е вече масово, тук ще разгледаме няколко метода за получаване на *изгладени* примитиви чрез осветяване на поредицата от пиксели, така че всеки от тях да има различен интензитет.

Първо е необходимо да отбележим, че не само пикселите, но и самите примитиви светят с определен интензитет. Нормално е интензитетът на един линеен елемент - отсечка, дъга, окръжност - да бъде пропорционален на неговата дължина (ако дебелината му е само един пиксел). Запълнените елементи пък ще имат интензитет, съответстващ на площта им. Това далеч не е така в разгледаните досега случаи. Една отсечка с наклон 45 градуса например ще има същия брой пиксели както и нейната проекция върху оста x, т.е. интензитетът и на двете отсечки ще е един и същ. Дължините им обаче съвсем не са равни и за да бъде визуалният образ правилен и по-точно - за да не изглежда хоризонталната отсечка по-дебела от наклонената, е необходимо всеки от пикселите на последната да свети с интензитет $\sqrt{2}$.



Фиг. 2-27

Това определено няма да е достатъчно за да изглежда тя по-плътна. За да се отстрани стъпаловидността ѝ ще е необходимо във всеки вертикален стълб от растера да се осветят по няколко пиксела. Интензитетът на всеки един от тях трябва да е пропорционален на частта от отсечката, която припокрива този пиксел. За да прецизираме последното е необходимо първо да изясним няколко неща:

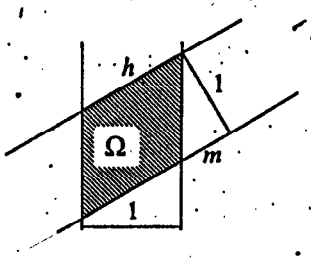
- Каква е формата и големината на един пиксел?
- Каква е формата на един примитив с дебелина един пиксел?

Възможни са няколко различни отговора на тези въпроси. На фигурите досега изобразявахме пикселите като кръгове, за да може да ги отличаваме от координатната мрежа. По-правилно е да третираме всеки пиксел като квадрат със страна 1, с чиито координати (x,y) ще отбелязваме центъра му. Това означава, че хоризонталната и вертикалната отсечки са за нас правоъгълници с ширина един пиксел. По този начин естествено стигаме до извода да разглеждаме всяка растерна отсечка като такъв правоъгълник. Аналогично, растерната окръжност ще е пръстенът, заграден между две концентрични окръжности, разликата между радиусите на които е 1.

Фиг. 2-28

2.4.1 Изглаждане чрез оценка на припокритата площ

Така дефинираните растерни примитиви вече имат площи и следователно интензитетът на един пиксел може да се определи като площта на тази част, която примитивът припокрива от него. Нека вземем една отсечка с наклон между 0 и 45 градуса. Да оставим настрана въпроса за растеризирането на крайните ѝ точки и да разгледаме кой да е друг неин вътрешен пиксел. От всеки вертикален стълб на растера отсечката отсича определена площ, която трябва да се разпредели между два или три пиксела от този стълб. Тази площ можем да изразим чрез наклона на отсечката:



$$\Omega = 1 \cdot h = \sqrt{m^2+1} = \frac{\sqrt{dx^2+dy^2}}{dx}$$

Фиг. 2-29

Ще се опитаме да изразим площта, която се припокрива от отделните пиксели, чрез оценката $d = F(M)$ от алгоритъма на средната точка, където функцията на отсечката е дефинирана с [2.6]. Това би ни помогнало да използваме представените вече алгоритми и за изглаждане.

Първо нека се спрем на случая, в който отсечката припокрива само 2 пиксела от един растерен стълб. Разстоянието от средната точка до отсечката (по-точно до средната линия на правоъгълника, представящ отсечката) е пряко свързано със стойността на оценката:

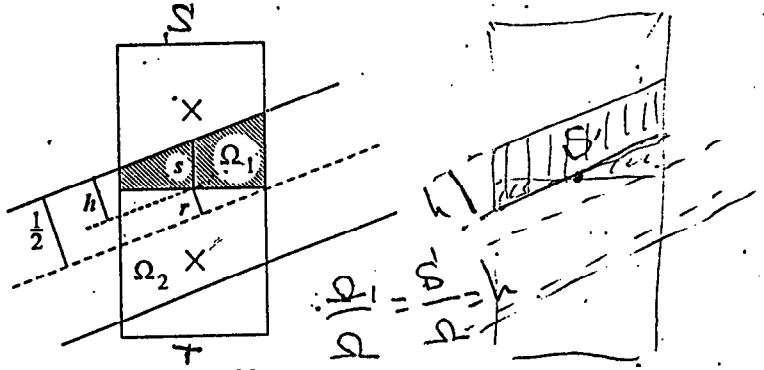
$$r = \frac{F\left(x_i+1, y_i+\frac{1}{2}\right)}{2\sqrt{dx^2+dy^2}} = \frac{d}{2\sqrt{dx^2+dy^2}}.$$

Площта на трапеца, който се отсича от горния пиксел, е равна на дължината на средната му отсечка (тъй като височината му е 1 пиксел), която пък можем да изразим чрез наклона на отсечката, която изглаждаме:

$$\Omega_1 = 1 \cdot s = \frac{\sqrt{dx^2+dy^2}}{dx} h = \frac{\sqrt{dx^2+dy^2}}{dx}\left(\frac{1}{2} - r\right) = \frac{\sqrt{dx^2+dy^2}}{2dx}\left(1 - \frac{d}{\sqrt{dx^2+dy^2}}\right)$$

$$\Omega_2 = \frac{\sqrt{dx^2+dy^2}}{2dx}\left(1 + \frac{d}{\sqrt{dx^2+dy^2}}\right)$$

Доста по-сложен е случаят, когато отсечката се разполага върху три пиксела от един растерен стълб. Тогава припокритите площи няма да имат трапецовидна форма. Зависимостта между оценката d и тези площи ще бъде квадратична, което би затруднило пресмятането им. Поради тази причина можем да разпределяме площта само между двата пиксела T и S - (виж фиг. 2-6), които се разглеждат на всяка стъпка от алгоритъма и да включваме трети пиксел само ако някоя от площите стане по-голяма от 1 (може да се види, че това е по-силно от условието допълващата площ да е по-малка от 0). Просто правило, което можем да приемем, е да осветяваме пиксела над разглежданата двойка с интензитет пропорционален на площта, с която по-горният пиксел превишаваща 1. Аналогично ще осветяваме пиксела под двойката пропорционално на площта, с която по-долният от двойката надхвърля 1.



Фиг. 2-30

Тъй като припокритите площи се получават в интервала $[0,\sqrt{2}]$, за да ги изобразим върху множеството на възможните интензитети $[0, J_{max}]$ трябва да разделим на $\sqrt{2}$ и умножим по J_{max} изведените формули за Ω_1 и Ω_2.

Това дава достатъчно добра апроксимация на интензитетите на припокритите площи:

$$I_{1,2} = \frac{J_{max} \cdot \Omega}{2\sqrt{2}} \left(1 \pm \frac{d}{\sqrt{dx^2 + dy^2}} \right).$$

Промяната, която е необходимо да се направи в главния цикъл на програмата, реализираща алгоритъма на средната точка е показана във фрагмента по-долу. Оценката на припокритата площ е удобен метод за изглаждане на контура на запълнен многоъгълник. В този случай трябва да се оцени площта или само на един или на два пиксела, които едно ребро от контура пресича.

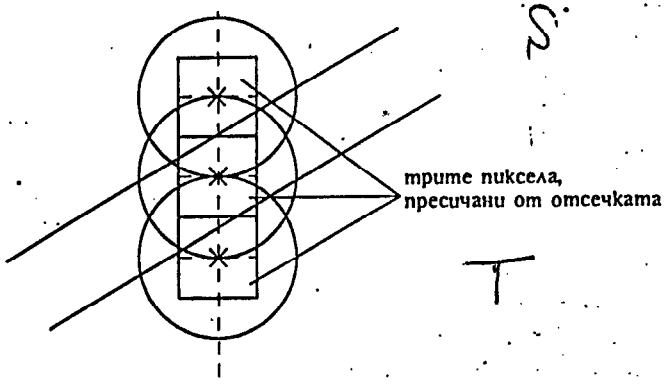
Накрая ще отбележим, че тази сравнително нетривиална апроксимация има смисъл само ако броят на различните интензитети, които можем да зададем на един пиксел е сравнително голям. Това далеч не е така при конвенционалните растерни дисплеи. Ако работим с растерен дисплей, в който за всеки пиксел са отделени по 2 бита (4 възможности), най-удачно е за апроксимация на интензитета да се използва самата оценка *d*.

```
norm=1.0/sqrt((double)(dx*dx+dy*dy));
area=1/(norm*dx);
sqrt2=sqrt(2.0);
while (n--) {
      x+=incX;
      if (d>0) { v[1]=area*(1+d*norm)/2;
                 d+=incUP; y+=incY; other=2; empty=0;
      } else   { v[1]=area*(1-d*norm)/2;
                 d+=incDN; other=0; empty=2;
      }
      if (v[1]>1) { v[empty]=v[1]-1;
                    v[1]=1-v[empty];
                    v[other]=area-v[1]-v[empty];
      }   else    { v[empty]=0;
                    v[other]=area-v[1];
      }
      for (i=0, yc=y+incY; i<3; i++, yc-=incY) {
          if (v[i])
              if (reverse)
                  PutPixel(yc,x,(int)(MAXINTENS*(v[i]/sqrt2)));
              else
                  PutPixel(x,yc,(int)(MAXINTENS*(v[i]/sqrt2)));
      }
}
. . .
```

2.4.2 Изглаждане чрез отчитане на разстоянието до примитива

Можем да кажем, че съществува пряко пропорционална зависимост между осветеността на един пиксел и неговата близост до елемента, който той апроксимира. Друг начин за определяне на интензитета на един пиксел е да намерим разстоянието от неговия център до елемента и да му зададем интензитет обратно пропорционален на това разстояние. Естествено не бива да проверяваме разстоянието на всички пиксели от растера, затова първата задача е

да изберем максималното разстояние, което ще предизвиква осветяване на пиксела. Обикновено се избира това разстояние да е 1 пиксел. Можем да си мислим, че в центъра на всеки пиксел е разположен кръг с радиус 1 и пикселът ще бъде осветен съобразно площта, която отсечката припокрива от този кръг.



Фиг. 2-31

На фиг. 2-31 се вижда, че отсечката под ъгъл не повече от 45 градуса ще пресича най-много три такива кръга. Освен избраната точка в алгоритъма за растеризиране, отсечката пресича и кръговете на двата съседни пиксела от същия растерен стълб. Както и в предния случай, ще изразим разстоянието от избрания пиксел до отсечката чрез оценката *d*.

Нека на *i*-тата стъпка предстои да се избере пикселът T (фиг. 2-6). Разстоянието от неговия център ще е нормираната стойност на функцията на отсечката в този пиксел:

$$D_T = \frac{F(x_i+1, y_i)}{2\sqrt{dx^2+dy^2}}, \quad \text{но} \quad F(x_i+1, y_i) = 2dy(x_i+1) - 2dx \cdot y_i + 2c$$

$$= 2dy(x_i+1) - 2dx\left(y_i + \frac{1}{2}\right) + dx + 2c$$

$$\Rightarrow \quad D_T = \frac{d+dx}{2\sqrt{dx^2+dy^2}}. \qquad = F\left(x_i+1, y_i+\frac{1}{2}\right) + dx = d + dx$$

Освен този пиксел, правоъгълникът на отсечката ще пресича и двата му съседни пиксела, както видяхме по-горе. Разстоянието от съседните й две точки (тази над нея е отбелязана с T+1, а тази под нея - с T–1) можем да изразим чрез вече полученото разстояние D_T:

$$D_{T+1} = \frac{F(x_i+1, y_i+1)}{2\sqrt{dx^2+dy^2}} = \frac{d-dx}{2\sqrt{dx^2+dy^2}} = D_T - \frac{dx}{\sqrt{dx^2+dy^2}}$$

$$D_{T-1} = \frac{F(x_i+1, y_i-1)}{2\sqrt{dx^2+dy^2}} = \frac{d+dx}{2\sqrt{dx^2+dy^2}} = D_T + \frac{dx}{\sqrt{dx^2+dy^2}}$$

Аналогично, ако пикселът избран на i-тата стъпка е S, ще получим следното:

$$D_S = \frac{d-dx}{2\sqrt{dx^2+dy^2}}, \quad D_{S+1} = D_S - \frac{d-dx}{2\sqrt{dx^2+dy^2}}, \quad D_{S-1} = D_S + \frac{d-dx}{2\sqrt{dx^2+dy^2}}.$$

Трябва да отбележим, че всички изчислени по този начин разстояния са ориентирани и затова е необходимо да вземаме тяхната абсолютна стойност при определянето на интензитета.

Естествено е, интензитетът, съответстващ на определено разстояние, да не се изчислява всеки път - за всеки пиксел и за всяка нова отсечка. Тъй като броят на различните интензитети е краен и дори ограничен, възможно е да се състави таблица, в която на всяко от предварително фиксиран набор от разстояния е съпоставен интензитет. Това е и основната идея в алгоритъма на Гупта-Спрул (Gupta-Sproull), където подходящо закръгленото разстояние е входен индекс в таблица от предварително изчислени интензитети. В този алгоритъм, точно както и в алгоритъма на Брезенхам, на всяка стъпка получаваме само по един пиксел (който избираме съобразно знака на оценката), но вместо да осветяваме него, ще осветим пиксела, който трябва да изберем като следващ, заедно с неговите два вертикални съседа.

```
#define DistPixel(x,y,D) \\
        PutPixel(x,y,IntTable(RoundDist(fabs(D))))

void SampledDistanceAntialiasedLine(X1,Y1,X2,Y2)
int X1,Y1, X2,Y2;
{. . . /* променливите в алгоритъма на Брезенхам */
 float D,diff,norm;
    . . .
    /* инициализацията в алгоритъма на Брезенхам */
    . . .
    n=dx;
    norm=1/(2.0*sqrt((double)(dx*dx+dy*dy)));
    diff=2*dx*norm;
    if (reverse) DistPixel(Y1,X1,0.);
        else     DistPixel(X1,Y1,0.);
    while (n--) {
          x+=incX;
          if (d>0) { D=(d-dx)*norm;
                     d+=incUP; y+=incY;
          }   else { D=(d+dx)*norm;
                     d+=incDN;
          }
          if (reverse) {
              DistPixel(y,x,D);
              DistPixel(y+1,x,D-diff);
              DistPixel(y-1,x,D+diff);
          } else {
              DistPixel(x,y,D);
              DistPixel(x,y+1,D-diff);
              DistPixel(x,y-1,D+diff);
          }
    }
}
```


Подобна стратегия може да се приложи почти буквално за изглаждане на окръжност използвайки симетрията ѝ, както това бе показано при растеризирането ѝ. Когато се изглажда контурът на многоъгълник, трябва да се отчете от коя страна се намира вътрешността му, за да не се разглеждат пикселите, попадащи там.

Задачи

2.1 Докажете, че разстоянието от всеки избран пиксел в алгоритъма на средната точка до отсечката, която се растеризира, е винаги $< 1/2$.

2.2 Може ли да използвате симетричността на една отсечка и да я растеризирате едновременно от двата ѝ края към центъра, използвайки една единствена оценка?

2.3 Напишете програма, която растеризира отсечка, чиито нараствания по всяка от осите не са взаимно прости числа. Приемете, че знаете един техен общ делител.

2.4 Напишете програмата за растеризацня на отсечка, като използвате алгоритъма на порциите в общия случай.

2.5 Напишете програма, която растеризира контура на многоъгълник, така че никой пиксел да не се записва по два пъти. Това може да е извънредно важно при използване на режим на растерно записване "изключващо или".

2.6 Напишете програма за ефективна растеризация на окръжност използвайки параметричното ѝ уравнение, без да пресмятате на всяка стъпка тригонометрични функции.

2.7 Напишете програма за растеризация на окръжност чрез растеризация на отсечките (използвайки алгоритъма на Брезенхам) на правилния многоъгълник, който апроксимира тази окръжност.

2.8 Използвайки казаното в началото на 2.1.3 предложете алгоритъм за позициониране на точка върху окръжност, като анализирате грешката и в диагонално разположената точка и отчетете възможността за смяна на знака на грешката при движение.

2.9 Напишете програма за растеризация на дъга от елипса с оси, успоредни на координатните.

2.10 Напишете програма за растеризиране на изправен правоъгълник със заоблени върхове като знаете координатите на две диагонално разположени точки и радиуса на заобляне. Осигурете записването на стойността на всеки пиксел само по веднъж.

2.11 Напишете програма за запълването на изправен правоъгълник със заоблени върхове.

2.12 Добавете към програмата за запълване на многоъгълници възможност за отстраняване стъпаловидността на контура му.

2.13 Модифицирайте алгоритъма за запълване на многоъгълник, така че границите му да се растеризират независимо от това в какво положение спрямо вътрешността му се намират.

2.14 При запълване на многоъгълник, който има много близки един до друг ръбове, които при растеризация имат съвпадащи точки, но спрямо които вътрешността се намира от различни страни по правилото, прието в алгоритъма на сканиращия ред може да се случи в един ред началната точка да се намира след крайната. Модифицирайте програмата, така че да се справя и с този частен случай.