1. РАСТРОВЫЕ АЛГОРИТМЫ

1.1. Растровое представление отрезка. Алгоритм Брезенхейма

Подавляющее число графических устройств являются растровыми, представляя изображение в виде прямоугольной матрицы пикселов (растра), и большинство графических библиотек содержат внутри себя достаточное количество простейших растровых алгоритмов, таких как переведение идеального объекта (отрезка, окружности и др.) в их растровые образы; обработка растровых изображений.

Тем не менее часто возникает необходимость и явного построения растровых алгоритмов.

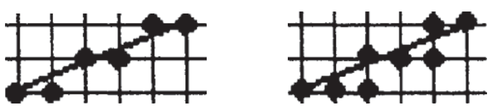
Достаточно важным понятием для растровой сетки является связность – возможность соединения двух пикселов растровой линией, т. е. последовательным набором пикселов. Возникает вопрос, когда пикселы (x_1,y_1) и (x_2,y_2) можно считать соседними.

Вводится два понятия связности:

4-связность: пикселы считаются соседними, если либо их x-координаты, либо их y-координаты отличаются на единицу: $|x_1 - x_2| + |y_1 - y_2| \le 1$;

8-связность: пикселы считаются соседними, если их x-координаты и координаты отличаются не более чем на единицу: $|x_1 - x_2| \le 1,\ |y_1 - y_2| \le 1$.

Понятие 4-связности является более сильным: любые два 4-связных пиксела являются и 8-связными, но не наоборот. На рис. 1.1 изображены 8-связная линия (а) и 4-связная линия (б).



а) б)

Рис. 1.1. Растровое изображение отрезка: а) 8-связная линия, б) 4-связная линия

В качестве линии на растровой сетке выступает набор пикселов $P_1,P_2,...,P_n$, где любые два пиксела P_i, P_{i+1} являются соседними в смысле заданной связности.

Замечание. *Так как понятие линии базируется на понятии связности, то естественным образом возникает понятие 4- и 8-связных линий. Поэтому, когда говорится о растровом представлении (например отрезка), следует ясно понимать, о каком именно представлении идет речь. В общем случае растровое представление объекта не является единственным и возможны различные способы его построения.*

Рассмотрим задачу построения растрового изображения отрезка, соединяющего точки $A(x_a, y_a)$ и $B(x_b, y_b)$. Для простоты будем считать, что $0 \le y_b - y_a \le x_b - x_a$. Тогда отрезок описывается уравнением

$$y = y_a + \frac{y_b - y_a}{x_b - x_a} \cdot (x - x_a),\ x \in [x_a, x_b] \text{ или } y = kx + b,$$

где

$$k = \frac{y_b - y_a}{x_b - x_a},$$

$$b = y_a - kx_a.$$

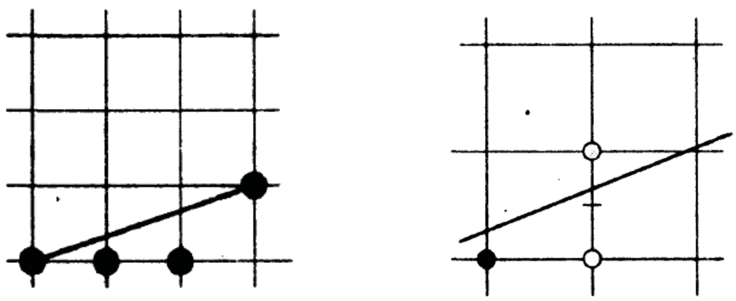
Отсюда получаем простейший алгоритм растрового представления отрезка:

```
void line (int xa, int ya, int xb, int yb, int color)
{
   double k = ((double)(yb-ya))/(xb-xa);
   double b = ya - k*xa;
   for (int x = xa; x <= xb; x++) putpixel (x, (int)( k*x + b ), color);
}
```

Вычисления значений функции $y = kx + b$ можно избежать, используя в цикле реккуррентные соотношения, так как при изменении *x* на 1 значение *y* изменяется на *k*.

```
void line (int xa, int ya, int xb, int yb, int color)
{
   double k = ((double)(yb-ya))/(xb-xa);
   double y = ya;
   for (int x = xa; x <= xb; x++, y += k ) putpixel ( x, (int) y, color);
}
```

Однако получение целой части *y* может приводить к не всегда корректному изображению (рис. 1.2).



а) б)

Рис. 1.2. Растровое изображение отрезка

Улучшить внешний вид получаемого отрезка можно за счет округления значений *y* до ближайшего целого. Фактически это означает, что из двух возможных кандидатов (пикселов, расположенных друг над другом так, что прямая проходит между ними) всегда выбирается тот пиксел, который лежит ближе к изображаемой прямой (рис. 1.2). Для этого достаточно сравнить дробную часть *y* с 1/2.

Пусть $x_0=x_a$, $y_0=y_a$, …, $x_n=x_b$, $y_n=y_b$ – последовательность изображаемых пикселей, причем $x_{i+1}-x_i=1$. Тогда каждому значению x_i соответствует число kx_i+b.

Обозначим через c_i дробную часть соответствующего значения функции $kx_i+b-c_i= \ kx_i+b$.

Тогда, если $c_i \le 1/2$, положим $y_i=[kx_i+b]$, в противном случае – $y_i=[kx_i+b]+1$.

Рассмотрим, как изменяется величина c_i при переходе от x_i, к следующему значению x_{i+1}.

Само значение функции при этом изменяется на *k*. Если $c_i+k \le 1/2$, то $c_{i+1}=c_i+k$, $y_{i+1}=y$.

В противном случае необходимо увеличить *y* на единицу, и тогда приходим к следующим соотношениям: $c_{i+1}=c_i+k-1$, $y_{i+1}=y_i+1$, так как $kx_i+b=y_i+c_i$, $kx_{i+1}+b=y_{i+1}+c_{i+1}$, а y_{i+1} – целочисленная величина.

Заметим, что $c_0=0$, так как точка (x_0, y_0) лежит на прямой $y = kx + b$.

Приходим к следующей программе:

```
void line (int xa, int ya, int xb, int yb, int color)
{
   double k = ((double)(yb-ya))/(xb-xa);
   double c = 0;
   int   y = ya;

   putpixel ( xa, ya, color);
   for (int x = xa + 1; x <= xb; x++ )
   {
     if (( c += k ) > 0.5 )
     {
       c-=1;
       y++;
     }
     putpixel ( x, y, color ); }
}
```

Замечание. *Выбор точки можно трактовать и так: рассматривается середина отрезка между возможными кандидатами и проверяется, где (выше или ниже этой середины) лежит точка пересечения отрезка прямой, после чего выбирается соответствующий пиксел. Это метод срединной точки (midpoint algorithm).*

Сравнивать с нулем удобнее, чем с 1/2, поэтому введем новую вспомогательную величину $d_i = 2c_i - 1$, заметив, что $d_i = 2k - 1$ (так как $c_i = k$). Получаем следующую программу:

```
void line (int xa, int ya, int xb, int yb, int color)
{
   double к = ((double)(yb-ya))/(xb-xa);
   double d = 2*k-1;
   int   y = ya;

   putpixel ( xa, ya, color);
   for (int x = xa + 1; x <= xb; x++ )
   {
     if (d > 0 ) {
      d += 2*k - 2;
      У++;
     }
     else
      d += 2*k; putpixel (x, y, color);
   }
}
```

Несмотря на то что и входные данные являются целочисленными величинами и все операции ведутся на целочисленной решетке, алгоритм использует операции с вещественными числами. Чтобы избавиться от необходимости их использования, заметим, что все вещественные числа, присутствующие в алгоритме, являются числами вида $\frac{p}{\Delta x}$, $p \in Z$. Поэтому если домножить величины d_i, и k на $\Delta x = x_b - x_a$, то в результате останутся только целые числа. Тем самым мы приходим к алгоритму Брезенхейма.

```
// Простейший алгоритм Брезенхейма 0 <= y2 - y1 <= x2 - x1
void line (int xa, int ya, int xb, int yb, int color)
{
   int dx = xb - xa;
   int dy = yb - ya;
   int d = ( dy « 1 ) - dx;
   int d1 = dy « 1;
   int d2 = (dy-dx)«1;

   putpixel ( xa, ya, color);
   for (int x = xa + 1, y = y1; x <= xb; x++ )
   {
     if ( d > 0 ) {
       d += d2;
       y += 1;
     }
     else
       d+=d1;

     putpixel ( x, y, color);
```

```
   }
}
```

Известно, что этот алгоритм дает наилучшее растровое приближение отрезка. Из предложенного примера несложно написать функцию для построения 4-связной развертки отрезка.

```
void line_4 (int x1, int y1, int x2, int y2, int color)
{
   int dx = x2 - x1;
   int dy = y2 - y1;
   int d = 0;
   int d1 = dy « 1;
   int d2 = - ( dx « 1 );

   putpixel ( x1, y1, color);
   for (int x = x1, y = y1, i = 1; i <= dx + dy; i++ )
   {
     if ( d > 0 ) {
       d += d2; У+= 1;
     }
     else {
       d+=d1; x += 1;
     }
     putpixel ( x, y, color);
   }
}
```

Общий случай произвольного отрезка легко сводится к рассмотренному выше; следует только иметь в виду, что при выполнении неравенства $|\Delta y| \leq |\Delta x|$ необходимо x и y поменять местами. Полный текст соответствующей программы приводится ниже.

```
void line (int x1, int y1, int x2, int y2, int color)
{
   int dx = abs ( x2 - x1 );
   int dy = abs ( y2 - y1 );
   int sx = x2 >= x1 ? 1 : -1;
   int sy = y2 >= y1 ? 1 : -1;

   if ( dy <= dx ) {
     int d = ( dy « 1 ) - dx;
     int d1 = dy « 1;
     int d2 = (dy-dx)«1;

     putpixel ( x1, y1, color);
     for (int x=x1 +sx, y=y1, i=1; i <= dx; i++, x+=sx) {
       if (d > 0 ) {
         d += d2; У += sy;
       }
       else
         d+=d1;
       putpixel ( x, y, color);
     }
   }
```

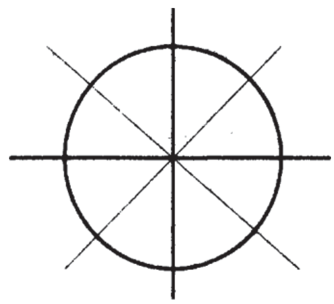
```
   else {
     int d = ( dx « 1 ) - dy;
     int d1 = dx « 1;
     int d2 = (dx-dy)«1;

     putpixel ( x1, y1, color);
    for (int x=x1, y=y1+sy, i=1; i <= dy; i++, y+=sy) {
      if ( d > 0 ) {
        d += d2; x += sx;
      }
      else
        d +=d1;
      putpixel ( x, y, color);
    }
  }//else
}
```

1.2. Растровая развертка окружности

Для упрощения алгоритма растровой развертки стандартной окружности можно пользоваться ее симметрией относительно координатных осей и прямых $y = \pm x$ (в случае, когда центр окружности не совпадает с началом координат, эти прямые необходимо сдвинуть так, чтобы они прошли через центр окружности). Тем самым достаточно построить растровое представление для 1/8 части окружности, а все оставшиеся точки получить симметрией. С этой целью введем следующую процедуру:

```
...
int xCenter;
int yCenter;
void CirclePoints (int x, int y, int color)
{
   putpixel ( xCenter + x, yCenter + y, color);
   putpixel ( xCenter + y, yCenter + x, color);
   putpixel ( xCenter + y, yCenter - x, color);
   putpixel ( xCenter + x, yCenter - y, color);
   putpixel ( xCenter - x, yCenter - y, color);
   putpixel ( xCenter - y, yCenter - x, color);
   putpixel ( xCenter - y, yCenter + x, color);
   putpixel ( xCenter - x, yCenter + y, color);
}
```



Рассмотрим участок окружности из второго октанта $x \in \left[0, R/\sqrt{2} \right]$, $y \in \left[R/\sqrt{2}, R \right]$. Особенностью данного участка является то обстоятельство, что угловой коэффициент касательной к окружности не превосходит 1 по модулю, а точнее, лежит между минус единицей и нулем. Применим к этому участку алгоритм средней точки (midpoint algorithm).

Функция $F(x,y)=x^2 + y^2 - R^2$, определяющая окружность, обращается в нуль на самой окружности, отрицательна внутри окружности и положительна вне ее.