Лекция 1.2. Алгоритмы растретизации

Предположим, что мы имеем своей задачей отображение на экране монитора некоторой линии - прямой, либо кривой, не имеет значения. Данная линия для нас будет являться неким математическим примитивом, задаваемым аналитически (при помощи некой формулы), либо геометрически (своими концами в случае простейшего отрезка).

Имея векторный монитор, отображение данной линии свелось бы к описанию некой команды, следуя которой луч электронов, управляемый магнитным полем, оставил бы на люминофоре след характерной формы. Однако в нашем случае все может оказаться не столь просто, так как большинство мониторов на сегодняшний день - растровые.

В таком случае, задача отображения некоторого графического объекта (вне зависимости от природы его описания) на экране растрового монитора будет сводиться к задаче поиска ограниченного набора растровых точек, наилучшим образом его аппроксимирующих. Подобную задачу в дальнейшем будем называть задачей растеризации или просто растеризацией.

Растеризация отрезка

Т.к. мы имеем дело с точечно-рисующими устройствами, то получить четкую кривую, описанную непрерывно дифференцируемой функцией, практически невозможно, за исключением тех случаев, когда мы работаем с отрезками, параллельными осям координат или являющими биссектрисами углов, образованных осями координат.

Отрезок является простейшим геометрическим примитивом (после точки), задача отображения которого на экране является задачей растеризации. В дальнейшем под отрезком будем понимать ограниченный сегмент прямой линии, задаваемый координатами своих концов. Так же, будем считать, что любая точка растра может иметь лишь два уровня интенсивности.

Критерии допустимости данных алгоритмов:

- отрезок должен начинаться и заканчиваться в заданных точках;
- отрезок должен казаться прямым;

- интенсивность отрезка должна быть постоянна на протяжении всего отрезка и не меняться в зависимости от угла наклона отрезка;
- алгоритм растеризации должен работать максимально быстро;
- должны быть поддержаны требуемые атрибуты

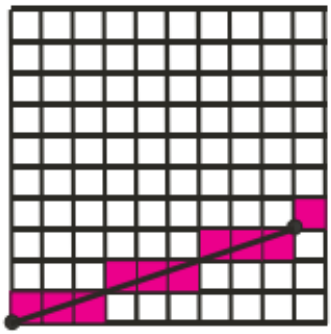
Условия 2 и 3 будет выполнить крайне сложно, так как мы имеем дело с набором дискретных точек. Эффективность алгоритма растеризации, в свою очередь, по большей части будет зависеть от возможности отказаться от применения операций с плавающей точкой.

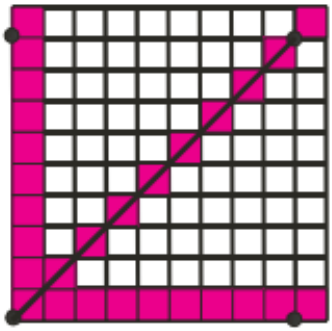
Положим в дальнейшем, что отрезок будет задан следующим образом:

- начальная точка будет располагаться в центре координат (0,0);
- конечная точка - в координатах (a,b), причем и a и b будут не отрицательными, причем a > b;

Данный случай задания отрезка будем называть в дальнейшем каноническим. Все далее описанные алгоритмы будут рассматривать подобный случай. Не трудно заметить, что для других случаев, модификация алгоритма может быть произведена методом приращения координаты, либо изменения знака на противоположный.

Условие a > b приводит нас к появлению еще одного понятия - так называемой основной оси. Основной осью принимается та, проекция на которую будет содержать большее количество точек. К примеру, на следующем рисунке, в качестве основной оси будет выбрана Ox:





В случае, если отрезок располагается диагонально, как на следующем рисунке, выбор основной оси ложится на плечи реализующего алгоритма

Для эффективной растеризации отрезка достаточно определить столько растровых точек, сколько содержит проекция на его основную ось.

В дополнение к основной оси иная ось обычно именуется *дополнительной осью.*

Математическое представление

Простейший способ растеризации отрезка приводит нас к математическому пониманию отрезка, а именно, к функции вида:

$$y = kx + b$$

В каноническом случае k будет определяться как:

$$к = b / a$$

В таком случае общий алгоритм растеризации с использованием математического представления будет выглядеть следующим образом:

```
void MathRasterize ( int a, int b )
{
    float k = b / a;
    int x = 0; float y = 0;
    while ( x <= a )
    {
        pixel ( x, round ( y ) ); x ++; y += k;
    }
}
```

В описанном выше листинге функция pixel высвечивает на экране точку в передаваемых ей координатах. Так как координата y является целочисленной, до использования, ее значение должно быть округлено по правилам округления.

Рассмотрим задачу построения растрового изображения отрезка, соединяющего точки:

(x_1, y_1) и (x_2, y_2).

Для простоты будем считать, что

$0 <= y_2 - y_1 <= x_2 - x_1$.

Тогда отрезок описывается следующим уравнением:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1), x \in [x_1, x_2]$$

или

$$y = kx + b$$

Тогда простейший алгоритм будет выглядеть следующим образом:

```
void DrawLine( int x1, int y1, int x2, int y2 )
{
    float y; int x;
    for( x = x1; x <= x2; x++ )
    {
        y = y1 + ( x - x 1 ) * ( y2 - y1 ) / ( x2 - x1 ); SetPixel( x, y );
    }
}
```

Этот алгоритм можно немного улучшить, если заранее, один раз вычислить тангенс угла наклона отрезка.

```
void DrawLine( int x1, int y1, int x2, int y2 )
{
    float y, m;
    int x;
    m = ( y2 - y1 ) / ( x2 - x1 );
    y = y1;
    for( x = x1; x <= x2; x++ )
    {
        SetPixel( x, y );
        y += m;
    }
}
```

Основные недостатки этих алгоритмов очевидны. Это использование операций с вещественными числами и слишком сложные вычисления для построения обычного отрезка.

Этот алгоритм можно немного улучшить, если заранее, один раз вычислить тангенс угла наклона отрезка.

```
void DrawLine( int x1, int y1, int x2, int y2 )
{
    float y, m;
    int x;
    m = ( y2 - y1 ) / ( x2 - x1 );
    y = y1;
    for( x = x1; x <= x2; x++ )
    {
        SetPixel( x, y );
        y += m;
    }
}
```

Основные недостатки этих алгоритмов очевидны. Это использование операций с вещественными числами и слишком сложные вычисления для построения обычного отрезка.

ЦДА - Цифровой Дифференциальный Анализатор (DDA - Digital Differential Analyzer)

Многие алгоритмы вычерчивания отрезков и кривых используют пошаговый принцип, суть которого состоит в том, что результат вычисления координат высвечиваемого пикселя зависит от результатов, полученных на предыдущем шаге. Алгоритм вычерчивания отрезков по методу цифрового дифференциального анализатора, кроме этого, основан также на достаточно общем принципе, известном в математике: изучение какого-либо явления на основе дифференциального уравнения или системы таких уравнений, описывающей это явление.

Следующий алгоритм упрощает работу, связанную с выбором основной и дополнительной осей. Так же он практически не чувствителен к тому, является ли данный случай описания отрезка каноническим или нет. Не смотря на то, что его эффективность не слишком высока по отношению к другим алгоритмам, его часто используют в качестве просто реализуемого.

Идея алгоритма заключается в пошаговом поиске растровых точек, т.е. поиск каждой следующей точки будет производиться, отталкиваясь от предшествующей. На каждом шаге значение каждой из координат изменяется на постоянное приращение, ей соответствующее.

Поскольку прямая линия на плоскости описывается уравнением вида:

$$ax + by + c = 0,$$

где ***a, b, c*** - коэффициенты этого уравнения, то производная *dy/dx* является постоянной. Заменив дифференциалы конечными разностями, получим:

$$\frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}, \quad (1)$$

где x_1, y_1 *и* x_2, y_2 - координаты начальной и конечной точек отрезка.

Ордината очередного пикселя y_{i+1} может быть вычислена по известной ординате предыдущего пикселя y_i следующим образом:

$$y_{i+1} = y_i + D_y$$

Подставляя ***Dy*** из (1), получим:

Остается определить величину приращения Dx. Как правило, большее из приращений (Dx или Dy) выбирается в качестве единицы растра, а приращение вдоль другой координатной оси подлежит определению. Если же поступить по-другому (меньшее из приращений взять равным единице), то отрезок на экране может получиться «дырявым», то есть состоящим из отдельных точек, не расположенных вплотную друг к другу.

$$y_{i+1} = y_i + \frac{y_2 - y_1}{x_2 - x_1}\Delta x, \quad (2)$$

Алгоритм разложения отрезка в растр по методу ЦДА может быть записан следующим образом:

1. Ввод исходных данных x1, y1, x2, y2.
2. Проверка вырожденности отрезка. Если отрезок вырожден, то высвечивание точки и переход к п. 7.

3. Вычисление L: = |x2-x1|, если |x2-x1|> = |y2-y1|

L: = |y2-y1|, если |y2-y1|>|x2-x1|

4. Вычисление dx: =(x2-x1)/L, dy:=(y2-y1)/L.

5. Задание для координат текущей точки начальных значений:

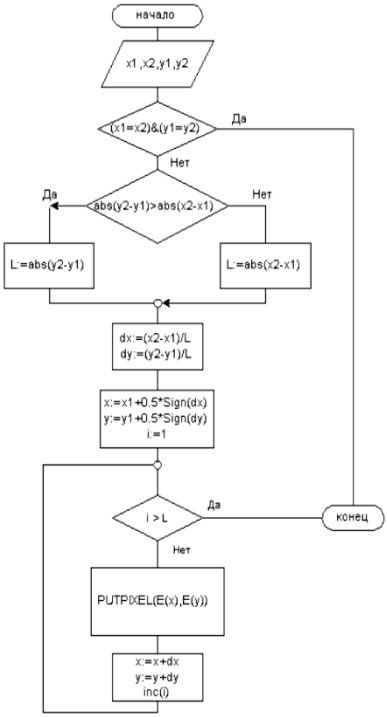
x := x1 + 0.5*Sign(dx), y := y1 + 0.5*Sign(dy),

где Sign - функция, возвращающая -1,0,1 для отрицательного, нулевого и положительного аргумента соответственно, использование этой функции делает алгоритм пригодным для всех квадрантов.

6. Цикл от i:=1 до i=L+1 с шагом 1:

высвечивание точки с текущими координатами (E(x),E(y)), где E - операция округления до ближайшего меньшего целого; вычисление координат следующей точки: x:=x+dx, y:=y+dy.

7. Конец.



Основные недостатки этих алгоритмов очевидны. Это использование операций с вещественными числами и слишком сложные вычисления для построения обычного отрезка.

Программная реализация ЦДА

```
void DDARasterize ( int x1, int y1, int x2, int y2 )
{
    int xLength = x2 - x1;
    int yLength = y2 - y1;
    int numSteps;
    if ( abs( xLength ) > abs( yLength ) ) numSteps = abs( xLength );
    else
    numSteps = abs( yLength );
    float dx = xLength / numSteps; float dy = yLength / numSteps;
    float x = x1;
    pixel ( x, y );
    for ( int k = 0; k < numSteps; k ++ )
    {
        x += dx; y += dy;
        pixel ( x, y );
    }
}
```

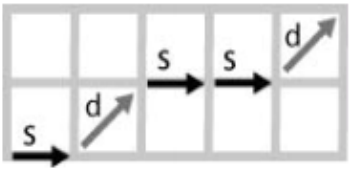
Вещественный алгоритм Брезенхема

Прежде, чем перейти к разбору данного алгоритма необходимо придти к некоторому пониманию о том, каким образом можно описать процесс изображения отрезка при его растеризации. В процессе растеризации отрезка в каноническом случае каждая следующая точка растра может находиться по отношению к текущей в двух положениях - либо правее, либо правее и выше.

Будем рассматривать процесс растеризации пошагово, то есть значение на каждом следующем шаге будет зависеть от значения, полученного на предыдущем. В таком случае, набор растровых точек для некоторого отрезка мы можем описать следующим образом:

sdssd

Данная запись будет соответствовать изображенной на следующей иллюстрации:



В вышеописанной схеме элемент s означает движение по прямой к следующей точке, а d - движение по диагонали.

Таким образом, на каждом новом шаге необходимо будет определить, какая из точек будет высвечена на экране - правая или правая верхняя. Предположим, что у нас имеются две потенциальные токи P_1 (0,1) и P_2 (1,1) . Для определения того, какая из этих двух точек станет следующей необходимо сравнить расстояния от них до точки пересечения отрезка с прямой x = 1:

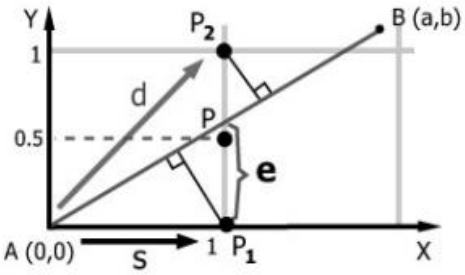
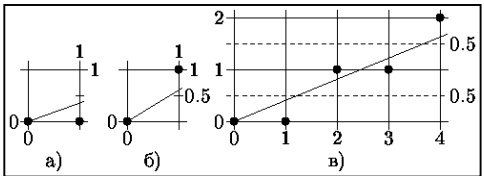


Рис. 3.

Величина e получила название ошибки. На каждом шаге эта ошибка накапливается. До тех пор, пока величина ошибки меньше ½ , производятся переходы типа s. Как только ошибка становится больше ½ осуществляется переход типа d с последующей нормализацией ошибки.

```
void FloatRasterize ( int a, int b )
{
    float e = b / a;
    float deS = b / a;
    float deD = b / a - 1;
    int x = 0; int y = 0;
    while ( x <= a )
    {
        pixel ( x, y ); x ++;
        if ( e > 1/2 )
        {
            // переход d
            y ++;
            e += deD;
        }
        else
        {
            // переход s
            e += deS;
        }
    }
}
```



Несмотря на быстродействие алгоритма, он все еще работает с вещественными числами, что привело к появлению более совершенного алгоритма.

Введем обозначения:

fi - значение ошибки на очередном i-м шаге,

Yu - ордината идеального отрезка,

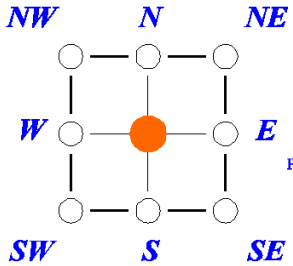
Yi - ордината пикселя, выбранного для аппроксимации отрезка на том же i-м шаге,

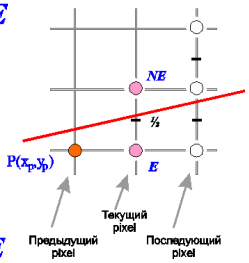
m - тангенс угла наклона отрезка, вычисляемый по формуле:

m = dy / dx

Т.к на первом шаге высвечивается пиксель с начальными координатами, то для него *f=0*, поэтому задаваемое предварительно значение:

f = m - 0.5





- **fi = 0 –** значит точка (x, y) лежит на отрезке
- **fi < 0 –** выше отрезка (NE)
- **fi > 0 –** ниже отрезка (SE)

f = m - 0.5

является фактически ошибкой для следующего шага.

Ошибка на очередном шаге вычисляется как:

$$f_{i+1} = y_{ui+1} - y_{i+1} = y_{ui} + m - y_{i+1} = f_i + m$$

Поскольку в алгоритме и программе не надо сохранять значения ошибок для всех шагов, то последнее выражение можно записать как:

$$f_{i+1} = f_i + m$$

f < 0 - выбирается пиксель с той же ординатой

f >= 0 пиксель с ординатой, на единицу большей NE

Поскольку предварительное значение ошибки вычисляется заранее, то есть *f+m* вычислено на предыдущем шаге, то во втором случае останется только вычесть единицу из значения ошибки:

f = f - 1.

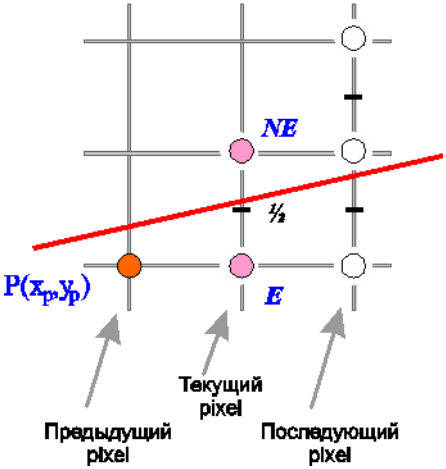
Алгоритм

1. Ввод исходных данных **x0, x1, y0, y1**.
2. Определение вырожденности отрезка. Если отрезок вырожден, то ставим точку **(x, y)** и выходим.
3. Вычисляем приращения **dx = x1-x0** и **dy = y1-y0**.
4. Инициализируем начальное значение ошибки **f = dy/dx – 0.5**.
5. Инициализируем координаты первого пиксела **x = x0** и **y = y0**.

6. Пока не достигнут конец линии:
1. Ставим точку **(x, y)**.
2. Изменяем координату вдоль основной оси **x = x + sign (dx)**.
3. Если **f >= 0**, тогда:

Изменяем координату вдоль не основной оси **y = y + sign (dy)**.
Корректируем ошибку **f = f – 1**.

4. Вычисляем новую ошибку **f = f + dy/dx**



Целочисленный алгоритм Брезенхема

Основная проблема вещественного алгоритма заключалась в том, что он использовал вещественные числа при подсчете и анализе ошибки. Следующий алгоритм Брезенхема был призван исправить подобную ситуацию, заменив вещественную арифметику целочисленной.

Вещественные числа в предыдущем алгоритме появляются в двух местах - при сравнении значения ошибки с 1/2 и при вычислении приращения ошибки. Для устранения вещественной арифметики предлагалось:

- уменьшить ошибку на 1/2 и теперь сравнивать ее значение с 0;
- домножить ошибку и приращение ошибки на 2a;

```
void IntegerRasterize( int a, int b )
{
    int e = 2b - a;
    int deS = 2b;
    int deD = 2b - 2a;
    int x = 0;
    int y = 0;
    while ( x <= a )
    {
        pixel ( x, y ); x ++;
        if ( e > 0 )
        {
            // переход d
            y ++; e += deD;
        }
        else
        {
            // переход s
            e += deS;
        }
    }
}
```

Как не трудно заметить, модифицированный алгоритм не содержит вещественной арифметики вообще, что приводит к повышению его эффективности по отношению к другим алгоритмам.

Алгоритм Брезенхема был создан им для вывода отрезков на цифровых инкрементальных графопостроителях, которые могли осуществлять лишь простые единичные сдвиги печатающей головки. Дальнейшая оптимизация может быть произведена, если заметить, что отрезок симметричен относительно прямой, проходящей перпендикулярно ему через его середину; в этом случае можно начинать рисовать сразу с двух концов, что сократит число итераций цикла в алгоритме вдвое.

Целочисленный алгоритм Брезенхема

- Быстродействие алгоритма можно увеличить, если использовать только целочисленную арифметику и исключить деление.

$$f = dy / dx - 0.5$$

- Умножим обе части на 2*dx

$$2 \cdot dx \cdot f = 2 \cdot dy - dx$$

Обозначив $fч = 2*dx*f$, окончательно получим:

$$fч = 2 \cdot dy - dx$$

Новое значение ошибки:

$$fч = fч + 2 \cdot dy$$

Таким образом

- Инициализация ошибки

$$f = 2 * dy - dx$$

- Корректировка ошибки

$$f = f - 2 * dx$$

- Вычисление ошибки

$$f = f + 2 * dy$$

Развитие алгоритма Брезенхема

В 1975 году П.Гарднер предложил новый подход к задаче построения отрезка. Он основан на симметрии отрезка.

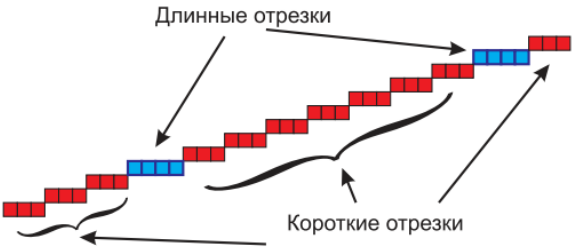
В 1987 году алгоритм вычисления нескольких пикселов был описан Кс.Ву и Дж.Рокне.

1990 г. Рокне - по его новому методу половина отрезка строилась двушаговым алгоритмом Кс. Ву, а вторая половина симметрично отображалась.

В 1997 году Джим Чен доказал, что если m – Наибольший Общий Делитель $dy=y2-y1$ и $dx=x2-x1$, то исходный отрезок состоит из m идентичных частей.

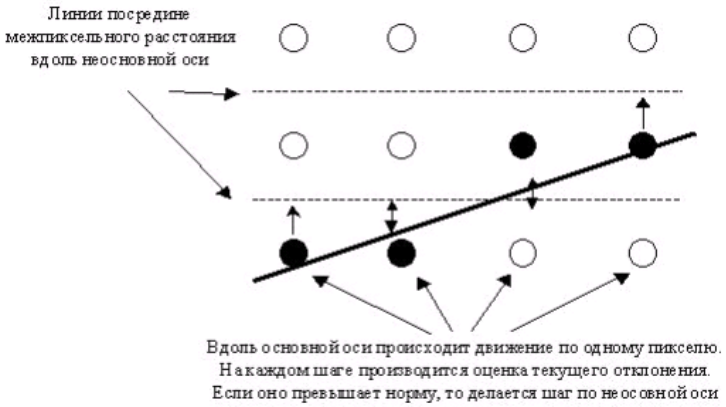
В 1999 году французы J.Boyer и J.J.Bourdin систематизировали уже известные свойства растеризованного отрезка и ввели несколько новых, которые позволили им создать алгоритм, превосходящий алгоритм Бразенхейма по скорости более чем в 20 раз.

Растеризованная линия представляет собой последовательность из коротких горизонтальных отрезков, называемых spans (“прядь”) отрезки. В каждый отрезок входят не более двух видов отрезков различной длины. Каждый последующий отрезок расположен на единичку выше предыдущего. Новые свойства, доказанные J.Boyer и J.J.Bourdin позволили определить длину, количество и порядок чередования отрезков. Таким образом, их алгоритм состоит из двух стадий. Во время первой определяется длины отрезков, а затем заполняется массив последовательности отрезков. Вторая стадия представляет собой собственно рисование линии: программа «проходит» по массиву и выводит соответствующие отрезки на монитор.

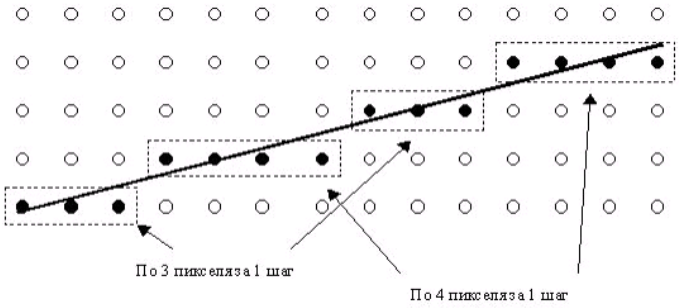


Прорисовка линий Брезенхема по отрезкам

Основа алгоритма Брезенхема - это движение вдоль основной оси по одному пикселю и поддержание текущего отклонения от идеальной прямой в заданных пределах. Если текущее отклонение превышает норму, то делается шаг по неосновной оси, чтобы уменьшить отклонение. Рис. 1 показывает стандартный алгоритм Брезенхема в действии. Главный момент здесь - это то, что вычисление и проверка производится на каждом шаге.



Когда программа проводит линию в 35 пикселей по X и 10 пикселей по Y, у вас уже есть масса доступной информации, определенная часть которой игнорируется стандартным алгоритмов Брезенхема. Например, поскольку наклон линии находится в пределах 1/3 - 1/4, то вы можете точно сказать, что составляющие линию ряды точек с неизменной неосновной координатой будут состоять из 3 или 4 пикселей.



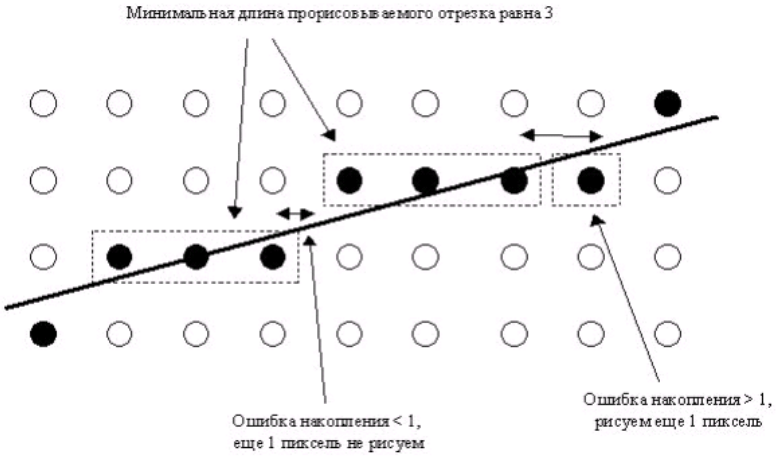
Таким образом, больше нет необходимости вычислять текущее отклонение каждой точки. Достаточно вычислить отклонение один раз за отрезок, чтобы посмотреть, из скольких точек этот отрезок состоит. В результате мы уменьшаем здесь число вычислений примерно на 70%. Для контроля размещения точек необходимо периодически принимать

решения относительно их расстановки и делать это желательно как можно быстрее и как можно чаще. Можно принимать решения для каждой точки, как в стандартном алгоритме Брезенхема, но большинство проверок в этом случае излишни - еще не начав рисовать линию, мы уже будем знать, какие точки проверять не нужно. Алгоритм с переменной длиной отрезков - это в точности алгоритм Брезенхема, но с минимальным числом проверок.

Реализация прорисовки по отрезкам

Мы уже знаем, как определить две возможные длины, которые будут иметь наши отрезки линии. Но как определить, какой отрезок будет иметь какую длину? Это несложно. Допустим, наклон нашей прямой равен 1/3.5, так что ось X - основная.

Минимально возможная длина отрезка, параллельного оси X, равна int(XDelta/YDelta) (здесь XDelta - длина проекции на ось X, YDelta - длина проекции на ось Y). Максимально возможная длина на единицу больше минимальной. Хитрость состоит в том, как разместить отрезки с минимальной и максимальной длиной, чтобы вместе они образовывали рисуемую линию.



На каждый шаг по неосновной оси мы ставим, по меньшей мере, три пикселя вдоль основной. После этого нам нужно решить, ставить ли

четвертый или переходить на следующую координату по оси X. Если мы подсчитаем, какую ошибку отклонения мы накопили, то на основании этого значения сможем сказать, нужен ли нам в отрезке дополнительный пиксель или хватит трех.

Это не сложный процесс, в который входит одно целочисленное сложение, одно вычитание и одна проверка, и еще остаются всевозможные оптимизации. Например, мы можем рисовать отрезки переменной длины по парам. Для вышерассмотренной прямой с наклоном 1/3.5 мы можем с помощью одной функции провести отрезки минимальной длины, а потом вызвать вторую и доставить недостающие точки. Эти функции не должны содержать циклов, а просто последовательность команд, предназначенных для прорисовки нужного числа пикселей. Поскольку мы контролируем накопленную ошибку, мы можем сразу выбрать, какую функцию из двух нам вызвать.

Растеризация окружности по методу Брезенхема

Как известно, окружность можно описать аналитически в следующем виде:

$$R^2 = x^2 + y^2$$

Будем полагать, что центр окружности будет лежать в центре системы координат. Тогда задача о растеризации окружности может быть решена «в лоб» через нахождение всех

$$y = y(x) = \pm\sqrt{x^2 + y^2}\ .$$

Однако при этом мы неизбежно столкнемся со следующими неприятностями:

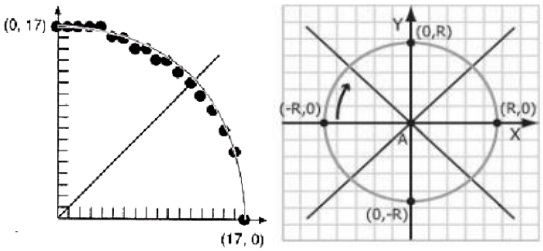
- чрезмерно сложные математические операции (извлечение квадратного корня, к примеру);
- неравномерная освещенность окружности вследствие неравномерного распределения точек;

Эти проблемы можно попытаться решить, перейдя от классического определения окружности к способу задания ее в полярных координатах:

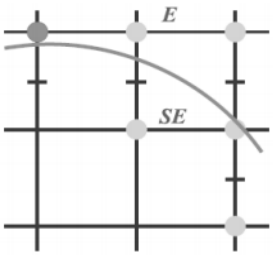
$$x = R\cos\theta \ \text{ и } \ y = R\sin\theta$$

Если в этом случае на каждом шаге изменять угол на одну и ту же величину, то в идеале мы получим равномерное распределение пикселей. Однако в этом случае достаточно непросто подобрать зависимость данного угла от радиуса, чтобы толщина окружности оставалась постоянной, и не появлялись пробелы. Вдобавок, вычисления синуса и косинуса все еще трудоемки.

Задачу растеризации окружности можно существенно упростить, если принять, что окружность по своей природе является центрально-симметричной фигурой, а следовательно, достаточно провести растеризацию лишь восьмой ее части. Остальные семь частей могут быть получены методом зеркального отражения.



Для растеризации окружности будем использовать левый верхний сегмент первого октанта. Начальной точкой для растеризации выберем точку, в которой *y* максимален, а саму растеризацию будем проводить по шагам. В таком случае на каждом следующем шаге координата *x* будет увеличиваться на 1, а *y* либо оставаться прежней, либо уменьшаться на 1, т.е. следующей точкой может быть либо *E*, либо *SE:*



Для определения, какая из двух этих точек будет являться следующей достаточно использовать координаты средней между ними точки (x+1,y-0.5). Эти координаты необходимо подставить в уравнение:

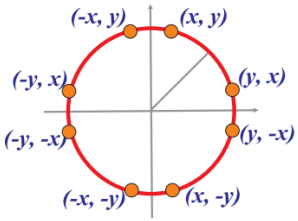
$$\boldsymbol{d}\,(x, \boldsymbol{y}) = \boldsymbol{x}^2 + \boldsymbol{y}^2 - \boldsymbol{R}^2$$

И проанализировать его знак:

d = 0 - линия окружности проходит точно посередине - выбор точки за реализацией;

d > 0 - срединная точка вне окружности, следовательно, следующей станет точка SE;

d < 0 - срединная точка внутри окружности, следовательно, следующей станет точка E;



Пусть текущая точка имеет координаты ***(Xp,Yp)***

Изначально в самой верхней точке дуги значение d_0 будет соответствовать:

$$d_0 = d(1, R - 0,5) = 1 + (R^2 - R + 1/4) - R^2 = 5/4 - R$$

В случае E-перехода оно изменится на величину:

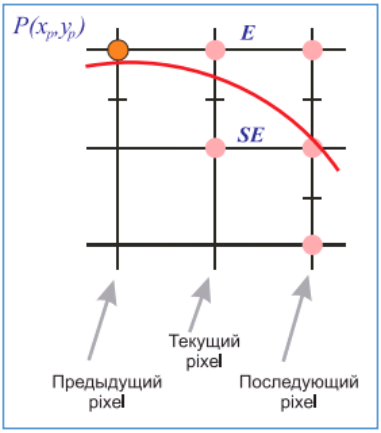
$$d_{new} = d(x_p + 2, y_p - 1/2) = d_{old} + (2x_p + 3)$$

$$\Delta d_E = 2x_p + 3$$

В случае перехода по SE:

$$d_{new} = d(x_p + 2, y_p - 3/2) = d_{old} + (2x_p - 2y_p + 5)$$

$$\Delta d_{SE} = 2x_p - 2y_p + 5$$



Таким образом, начиная с самого первого пикселя окружности, пока не нарисована дуга, нам необходимо проверять знак d и выбирать новый пиксель, при этом увеличивая d либо на одну, либо на другую величину.

Использование вещественных величин в алгоритме нежелательно, вследствие чего предлагается перейти к целочисленной версии алгоритма. Для этого достаточно перейти от величины d к величине $h = d - 1/4$. Тогда $h_0 = 1 - R$. Но в этом случае в процессе алгоритма значения d приходилось бы сравнивать с -1/4, а не с нулем. Однако, так как значения d всегда будут целые, то от -1/4 можно легко перейти к 0 без ущерба работоспособности алгоритма.

Как и в случае отрезка, будем использовать задание окружности в неявном виде с помощью функции f(x,y):

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

Если f=0, то точка с данными координатами (x,y) расположена на окружности, если больше нуля - то вне окружности. А если меньше нуля - то внутри окружности.

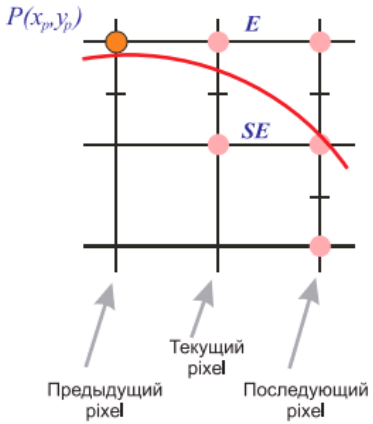
Пусть поставленная точка имеет координаты:

$$(x_p, y_p)$$

Вычислим значение в соответствующей ей срединной точке:

$$d_{old} = F(x_p + 1, y_p - 1/2)$$

Если это d меньше нуля, т.е. окружность проходит выше срединной точки, то выбирается пиксел E, иначе выбирается пиксел SE.



Рассмотрим два случая (для двух различных выборов пиксела):

Если выбран пиксел E:

$$d_{new} = f(x_p + 2, y_p - 1/2) = d_{old} + (2x_p + 3)$$
$$\Delta d_E = (2x_p + 3)$$

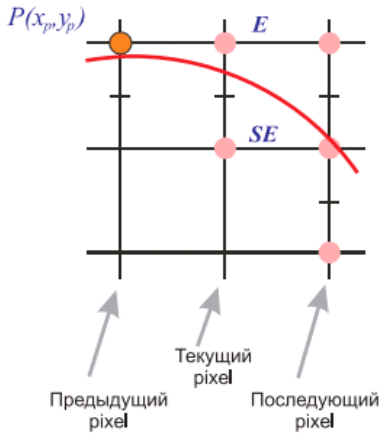
Если выбрали пиксел SE, то он имеет координаты (x+1, y-1), а значит, соответствующая ему срединная точка (x+2,y-3/2). И тогда:

$$d_{new} = d_{old} + (2x_p - 2y_p + 5)$$
$$\Delta d_{SE} = (2x_p - 2y_p + 5)$$

Таким образом, начиная с самой первой точки (верхняя точка окружности), мы определяем значение d для каждого нового пиксела, и, сравнив его значение с нулем, строим следующий пиксел в дуге окружности.

Рассчитаем значение d в начальной точке дуги (0,R):

$$f(1, R-1/2) = 1 + (R^2 - R + 1/4) - R^2 = 5/4 - R$$
$$d_0 = 5/4 - R$$



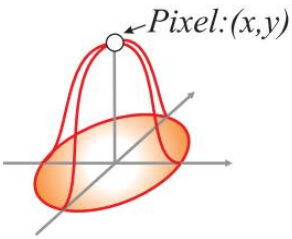
Лестничный эффект (aliasing)

Все растровые алгоритмы построения геометрических примитивов подвержены одному общему серьезному недостатку: ступенчатости линий. Он возникает как следствие самой природы растрового дисплея, в котором все изображение строится из дискретного набора точек. Эта ступенчатость хорошо заметна для глаза, и в результате, изображение теряет в реалистичности. Однако существует один механизм, который призван бороться с этим дефектом. Он получил название anti-aliasing.

Конечно, полностью устранить ступенчатость он не в силах, для этого необходимо отказаться от растровых мониторов. Однако с его помощью можно создать особую иллюзию, которая глазом будет восприниматься как гладкая линия. Как мы видим, на рисунке, линия размывается, и кажется гладкой. Таким образом, мы выигрываем в визуальном качестве изображения, но несколько теряем в его четкости.

Пиксел мы обычно представляем в виде небольшого квадратика, что на самом деле не совсем верно с точки зрения дисплея. Пиксел, на самом деле, имеет форму круга, с максимальной яркостью в центре, которая плавно убывает к краям. Радиус этого круга примерно равен расстоянию между пикселами, т.е. может быть принят за 1. При этом на мониторе пикселы частично накладываются друг на друга. Таким

образом, если мы нарисуем график распределения яркости монитора, то каждый пиксел будет иметь форму конуса.



Пусть наша линия имеет ширину равную 1. На графике распределения яркости, линия в идеале должна представляться в виде параллелепипеда. (Т.е. иметь повсюду одинаковую яркость). Наша «линия» (параллелепипед) будет пересекаться с «пикселами» (представляемыми в форме конуса).

Причем можно утверждать, что объем, отсекаемый «линией» от «пиксела», равен той доли общей яркости линии, которую должен внести данный пиксел. Вообще говоря, этот объем зависит от расстояния между пикселом и линией. Тогда яркость пиксела можно задавать как некоторую функцию F от ширины линии и расстояния до пиксела. Расстояния задается в долях единицы (принимая за единицу расстояние между пикселами), обычно в шестнадцатых.

Тогда яркость пиксела можно задавать как некоторую функцию F от ширины линии и расстояния до пиксела. Расстояния задается в долях единицы (принимая за единицу расстояние между пикселами), обычно в шестнадцатых. В этом случае удобнее всего задавать эту функцию в виде небольшой таблицы:

|p|	f(p,1)
0/16	0.78
1/16	0.775
2/16	0.760
….	
16/16	0.11

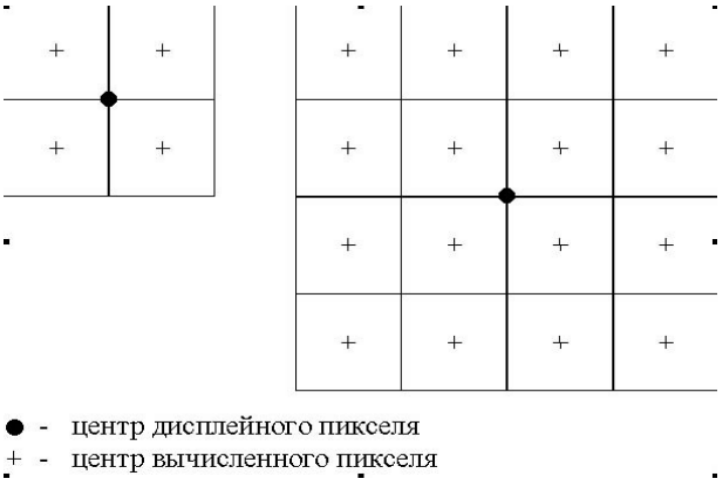
Механизм anti-aliasing помимо устранения ступенчатости линий позволяет устранить появление некоторых неприятных артефактов, возникающий в растровых алгоритмах. Например, при построении отрезков прямых по методу Бразенхейма, горизонтальный и диагональный отрезки подсвечиваются с существенно различной яркостью (вплоть до 1.5 раз). При сглаживании их яркости выравниваются.

Можно классифицировать искажения следующим образом:

- ступенчатость ребер;
- некорректная визуализация тонких деталей или фактуры;
- некорректное отображение мелких объектов, особенно в анимационной последовательности кадров.

Увеличение математического разрешения

Один из методов устранения искажений связан с увеличением разрешения растра. Таким образом учитываются более мелкие детали. Однако, всегда существует предел повышения разрешающей способности на уровне аппаратных средств компьютера. Такое ограничение предполагает, что растр надо вычислять с более высоким разрешением, а изображать с более низким, используя усреднение некоторого типа для получения атрибутов пикселя с более низким разрешением. При таком подходе каждый дисплейный пиксель делится на подпиксели в процессе формирования растра более высокого разрешения. Для получения атрибутов пикселя определяются атрибуты в центре каждого подпикселя, которые затем усредняются (см. рис.). Можно получить лучшие результаты, если рассматривать больше подпикселей и учитывать их влияние с помощью весов при определении атрибутов.



Аппроксимация полутонами

Рассмотрим метод, называемый аппроксимацией полутонами. Метод использует минимальное число уровней интенсивности, обычно черный и белый, для получения нескольких полутонов серого или уровней интенсивности. Несколько пикселей объединяются в конфигурации. Хотя при этом ухудшается разрешение, однако визуальные свойства изображения улучшаются, если разрешение изображения меньше разрешения дисплея. Число доступных уровней интенсивности, полученных таким образом, можно увеличить с помощью увеличения размера клетки.