

# Растровые алгоритмы

# План

- Терминология
- Заливка – три алгоритма
- Алгоритм Брезенхема – растеризация

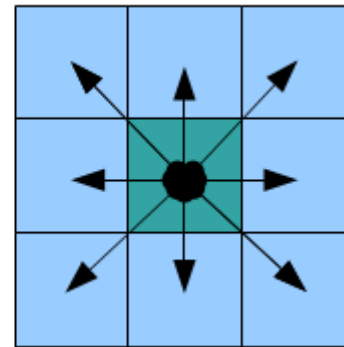
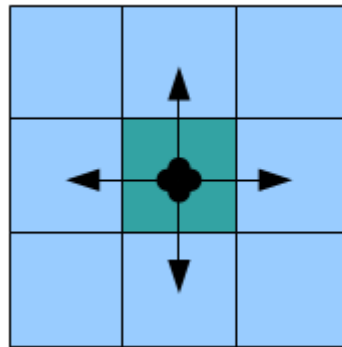
# Термины

- pixels - сокращение от picture elements
- pixel map - пиксельная карта
- bitmap - побитовое отображение или битовая карта

# Смежность точек

4-смежность

8-смежность



$$|x_1 - x_2| + |y_1 - y_2| \leq 1$$

$$|x_1 - x_2| \leq 1 \text{ и } |y_2 - y_1| \leq 1$$

4-смежность сильнее, чем 8-смежность

# Связность точек

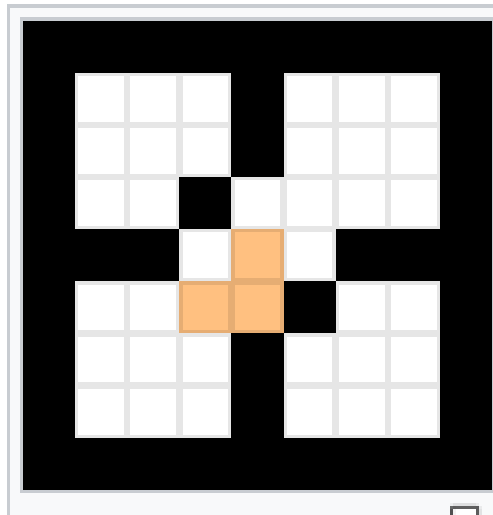
Назовем два пиксела *4-связными*, если существует соединяющий их непрерывный путь из 4-смежных пикселов.

Два пиксела называются *8-связными*, если существует соединяющий их непрерывный путь из 8-смежных пикселов.

# Задание областей

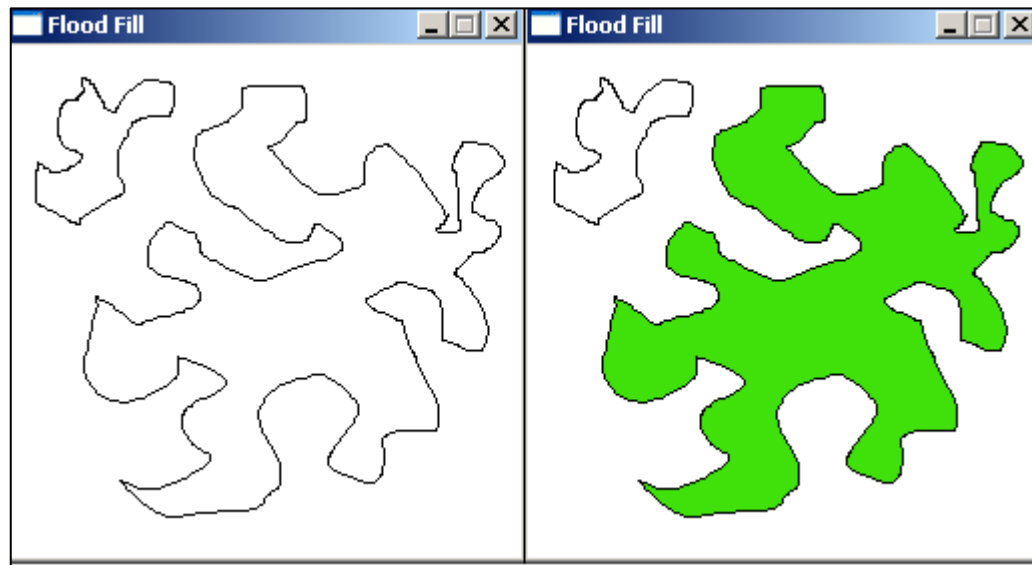
- *Пиксельно-определенная область* характеризуется текущими цветами в пиксельной карте.
- Различают *внутренне-определенную* область и *гранично-определенную*.
- Если граница области 4-связная, то внутренность 8-связная, и наоборот.

# Заливка



Как будет выглядеть результат при 4-х связной и 8-ми связной заливке?

# Задача заливки



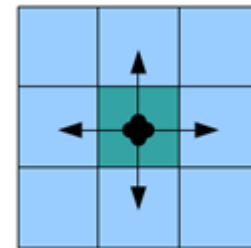
Граница в ряде мест может обладать толщиной большей, чем один пиксел.

Одни алгоритмы хорошо воспринимают толстые границы, а другие в этих случаях «приходят в замешательство».



# Простой рекурсивный алгоритм заливки

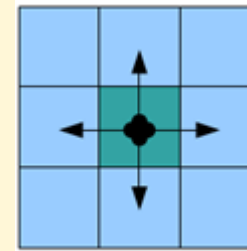
- Выбранная внутренняя точка становится текущей.
- Если текущая точка еще не закрашена и подходит по условию для закрашивания, то
  - Закрашиваем ее и
  - Применяем рекурсивный вызов данной функции для точек, смежных с текущей



# Простой рекурсивный алгоритм заливки

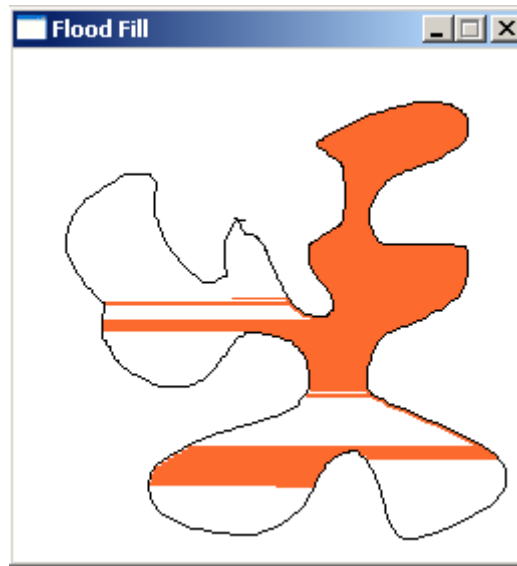
```
//Recursive 4-way floodfill, crashes if recursion stack is full
void floodFill4(int x, int y, int newColor, int oldColor)
{
    if(x >= 0 && x < w && y >= 0 && y < h && screenBuffer[x][y] == oldColor &&
screenBuffer[x][y] != newColor)
    {
        screenBuffer[x][y] = newColor; //set color before starting recursion

        floodFill4(x + 1, y,    newColor, oldColor);
        floodFill4(x - 1, y,    newColor, oldColor);
        floodFill4(x,    y + 1, newColor, oldColor);
        floodFill4(x,    y - 1, newColor, oldColor);
    }
}
```



# Недостаток

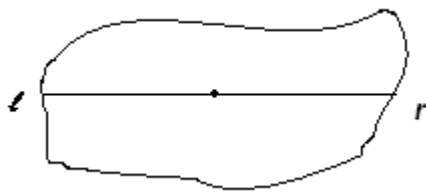
- Многие пикселы проверяются многократно, что требует огромного количества вызовов функции. Стек рекурсии может стать очень глубоким, даже для простой области. Поэтому велика вероятность переполнения стека.



- Существуют более эффективные методы заливки.

# Рекурсивный алгоритм заливки на основе серий пикселов (линий)

- Выбранная внутренняя точка становится текущей.
- Если текущая точка еще не закрашена и подходит по условию для закрашивания, то
  - Для текущей точки находим левую и правую границу. Рисуем линию от левой границы до правой границы, не включая саму границу.
  - В цикле от левой до правой границы (не включая саму границу) вызываем эту же функцию рекурсивно для всех точек, лежащих выше текущей на один пиксел.
  - Выполняем аналогичный цикл для всех точек, лежащих ниже текущей на один пиксел.



# Алгоритм заливки с предварительным выделением границы

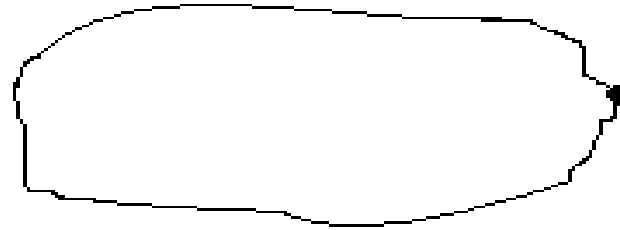
Процесс разбивается на 3 отдельные стадии.

1. Создание упорядоченного списка граничных пикселов (отслеживание или выделение границы).
2. Обследование внутренней части для обнаружения отверстий. Обследование производится посредством сканирования внутренних отрезков между парами граничных пикселов с одинаковыми значениями координаты  $y$  (от правой до левой границы). Если в области обнаруживается отверстие (еще не занесенные в список точки внутренней границы), то происходит занесение точек внутренней границы в общий список с сохранением упорядоченности. Процесс прекращается, когда вся внутренняя область обследована.
3. Соединение точек в списке слева направо горизонтальными прямыми (заполнение области).



# Создание списка граничных пикселов

- $x$ ,  $y$  и флаг



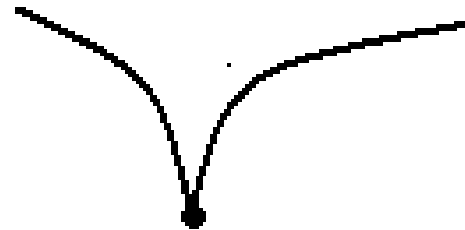
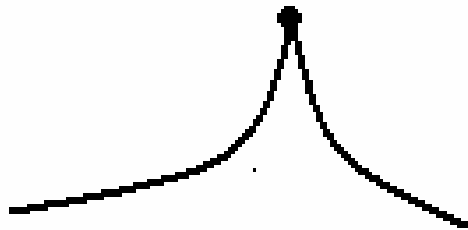
- *Обход начинается с начальной точки и продолжается по часовой стрелке.*
- *Внутренность области всегда находится справа от направления движения. Если пиксел не соседствует с внутренней частью области, то алгоритм не считает его граничным.*

# Выбор следующей точки

3а	2а	1а	а
4а	Xа	0а	а
5а	6а	7а	а

- В самый первый раз обход начинаем вниз. Проверяем, закрашена ли точка цветом границы. Если нет, то поиск закрашенной цветом границы точки продолжаем против часовой стрелки.
- В список заносим, сохраняя упорядоченность по  $y$ , если же  $y$ -ки имеют одинаковые значения, то по  $x$ . Выбор следующей точки  $(i+1)$ -ой, где  $i > 1$ , на 90 градусов по часовой стрелке от того направления, по которому мы туда пришли.

# Классификация точки: левая или правая?





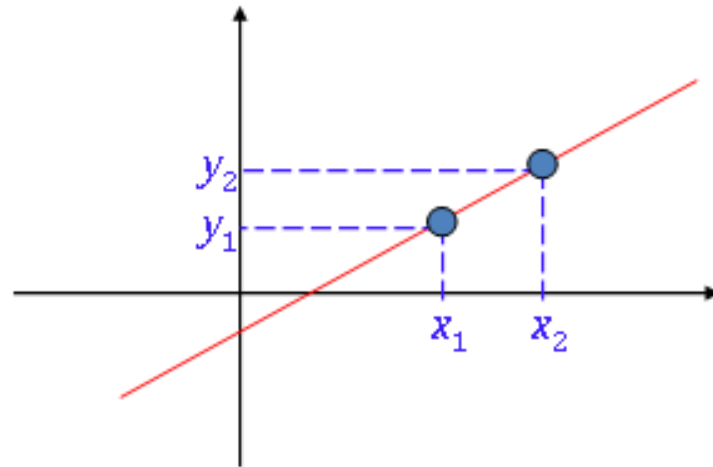
# Сравнение алгоритмов

Давайте обсудим

# Растиризация

- Процесс определения пикселов, наилучшим образом аппроксимирующих заданный отрезок, называется разложением в растр (растиризацией).

# Построение линии



$$y = mx + c$$

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

# Jack Elton Bresenham

b. 1937

IBM /

Winthrop University

Bresenham's Line  
Algorithm (1962)

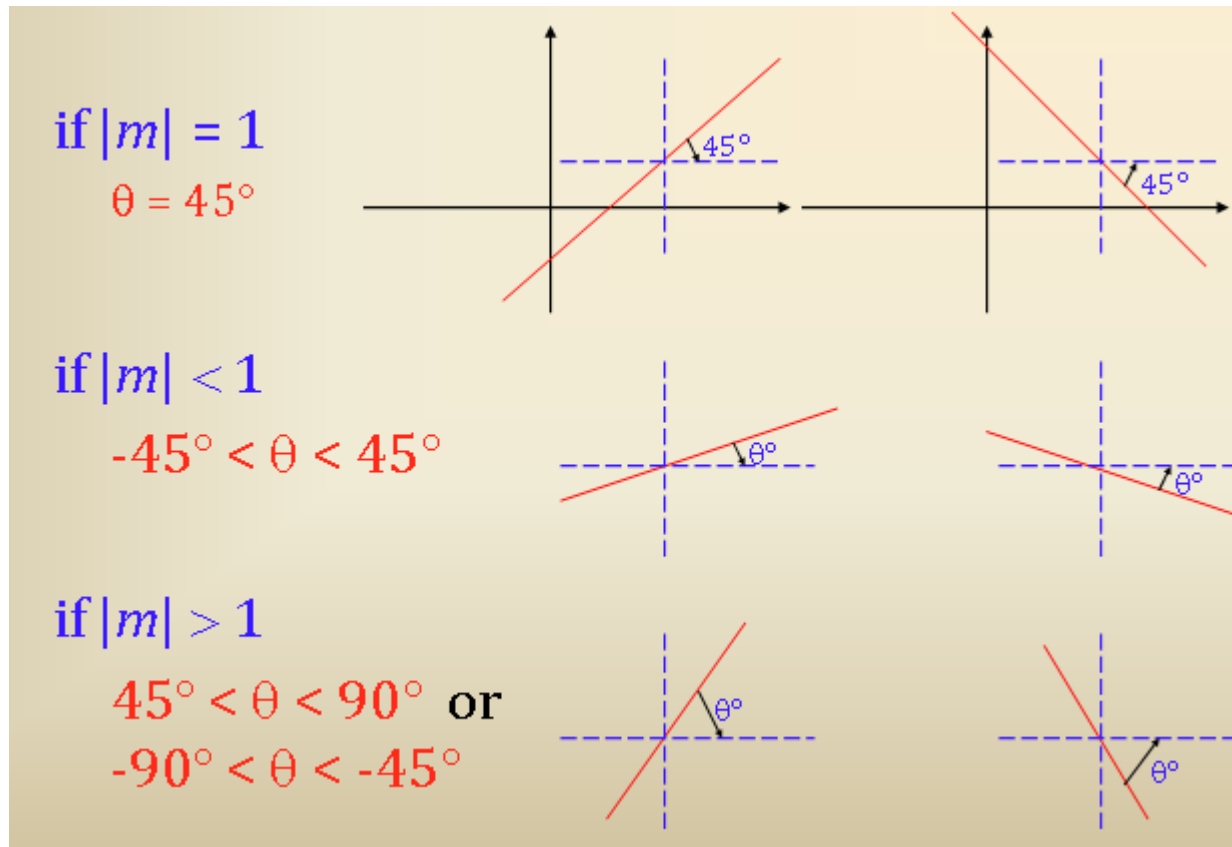


# Алгоритм Брезенхейма построения линий



Алгоритм, предложенный Дж. Э. Брезенхеймом в 1962, был опубликован в 1965 г.

# Углы наклона



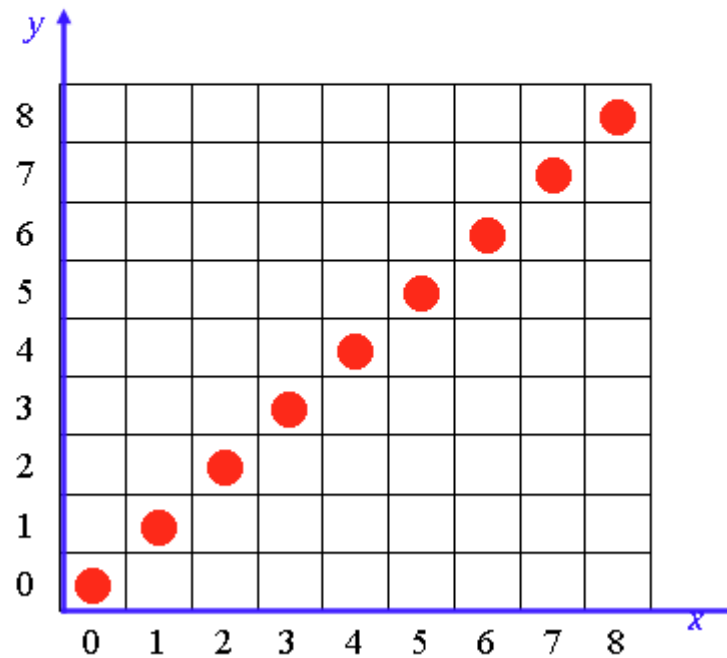
$$|m| = 1$$

$$y = x$$

$$m = 1$$

$$c = 0$$

x	y
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8

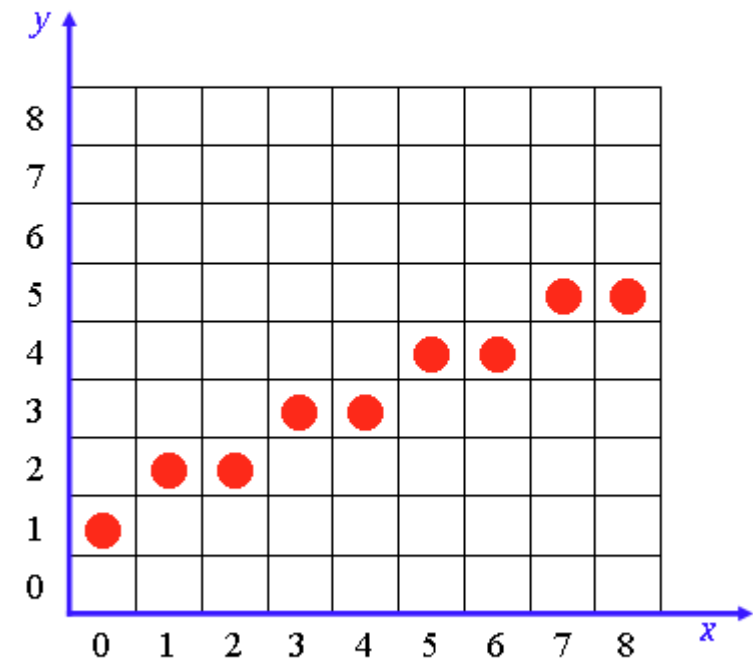


$$|m| < 1$$

$$y = \frac{1}{2}x + 1$$

$$m = \frac{1}{2}$$
$$c = 1$$

x	y	round(y)
0	1	1
1	1.5	2
2	2	2
3	2.5	3
4	3	3
5	3.5	4
6	4	4
7	4.5	5



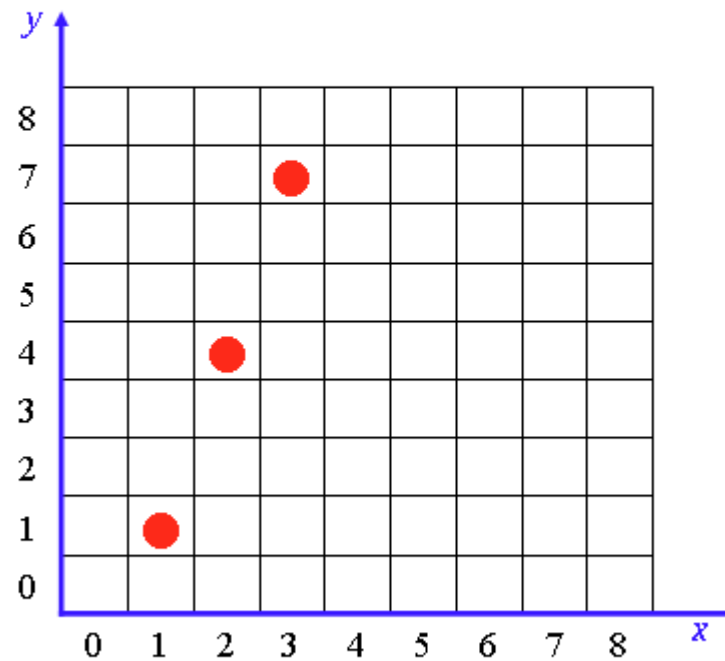


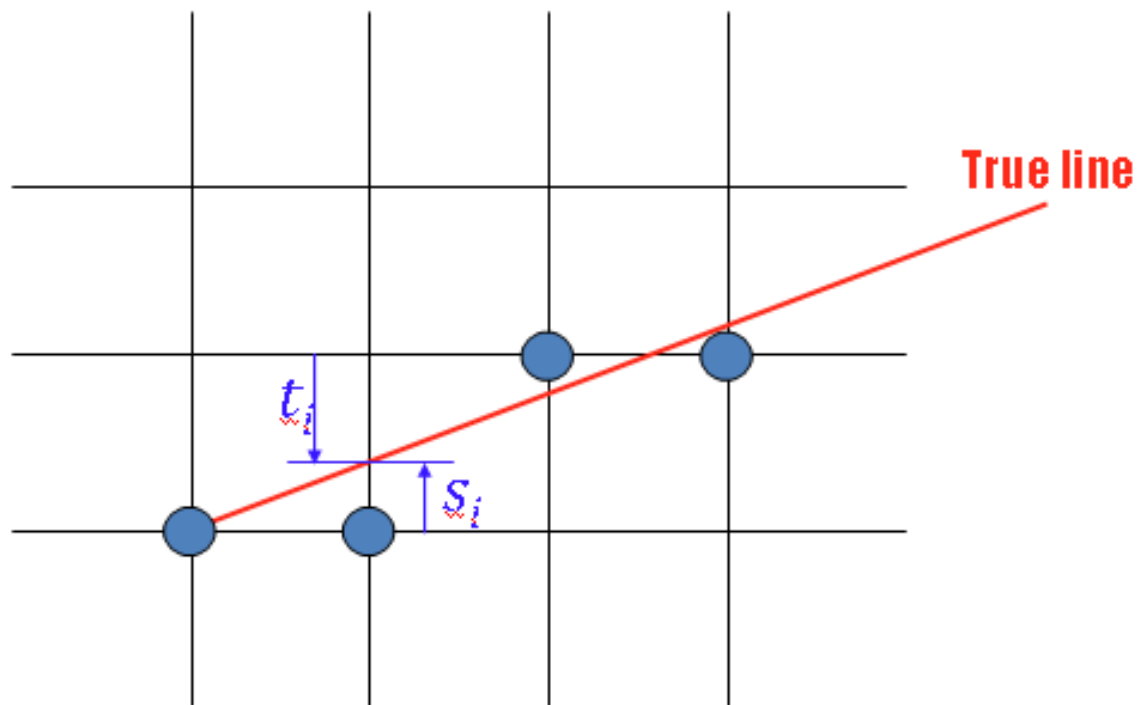
$$|m| > 1$$

$$y = 3x - 2$$

$$m = 3$$
$$c = -2$$

x	y	<u>round(y)</u>
0	-2	-2
1	1	1
2	4	4
3	7	7
4	10	10
5	13	13
6	16	16
7	19	outside 19



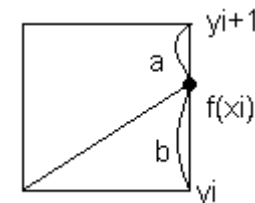
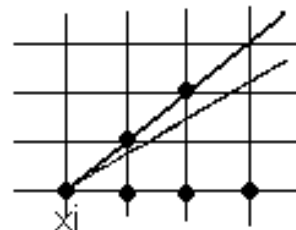


# Вывод

$$y = (dy/dx) * x,$$

где  $dy = y_2 - y_1$  и

$$dx = x_2 - x_1$$



- Выпишем формулы для расстояний  $a$  и  $b$ :

$$a = y_{i+1} - f_i = y_{i+1} - (dy/dx) * x_i$$

$$b = f_i - y_i = (dy/dx) * x_i - y_i$$

- Если расстояние  $a$  меньше расстояния  $b$ , то к точной прямой ближе  $y_{i+1}$ , если  $b$  меньше  $a$ , то ближе  $y_i$ .

if  $b - a > 0$  then

$$y_{next} = y_{i+1}$$

else

$$y_{next} = y_i$$

# Алгебраические преобразования

$$a = y_{i+1} - f_i = y_{i+1} - (dy/dx) \cdot x_i$$

$$b = f_i - y_i = (dy/dx) \cdot x_i - y_i$$

$$b - a = 2(dy/dx)x_i - y_i - y_i - 1 = 2(dy/dx)x_i - 2y_i - 1$$

- Так как  $x_1$  меньше  $x_2$ , то приращение  $dx$  всегда будет  $>0$ .  
Следовательно, вместо  $(b - a)$  можно использовать  $(b - a)dx$ .

$$(b - a)dx = 2dyx_i - 2y_i dx - dx = 2(dyx_i - y_i dx) - dx$$

- Обозначим:

$$(b - a) dx = d_i$$

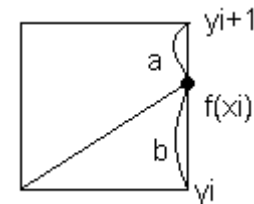
- Тогда в итоге получили алгоритм:

**if  $d_i > 0$  then**

**$y_{next} = y_{i+1}$**

**else**

**$y_{next} = y_i$**



# Алгебраические преобразования

$$d_i = 2(dyx_i - y_i dx) - dx$$

Для итеративного вычисления  $d_{i+1}$  необходимо знать, как получить это значение из  $d_i$

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

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

- Следовательно,  $d_{i+1}$  может быть получено путем прибавления к  $d_i$  одной из двух констант.

**if  $d_i > 0$  then**

$$d_{i+1} = d_i + 2dy - 2dx$$

**else**

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

# Как вычислить $d_1$ ?

- выпишем формулу:

$$d_i = 2dyx_i - 2y_i dx - dx$$

- Вспомним, что первый пиксел по договоренности имеет координаты (0,0). Подставив  $x_i = 1$ , а  $y_i = 0$ , получим
- $d_1 = 2dy - dx$ .

➤ For a line with gradient  $\leq 1$

$$d_0 = 2dy - dx$$

if  $d_i < 0$  then

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

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

if  $d_i \geq 0$  then

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

$$d_{i+1} = d_i + 2(dy - dx)$$

$$x_{i+1} = x_i + 1$$

➤ For a line with gradient  $> 1$

$$d_0 = 2dx - dy$$

if  $d_i < 0$  then

$$x_{i+1} = x_i$$

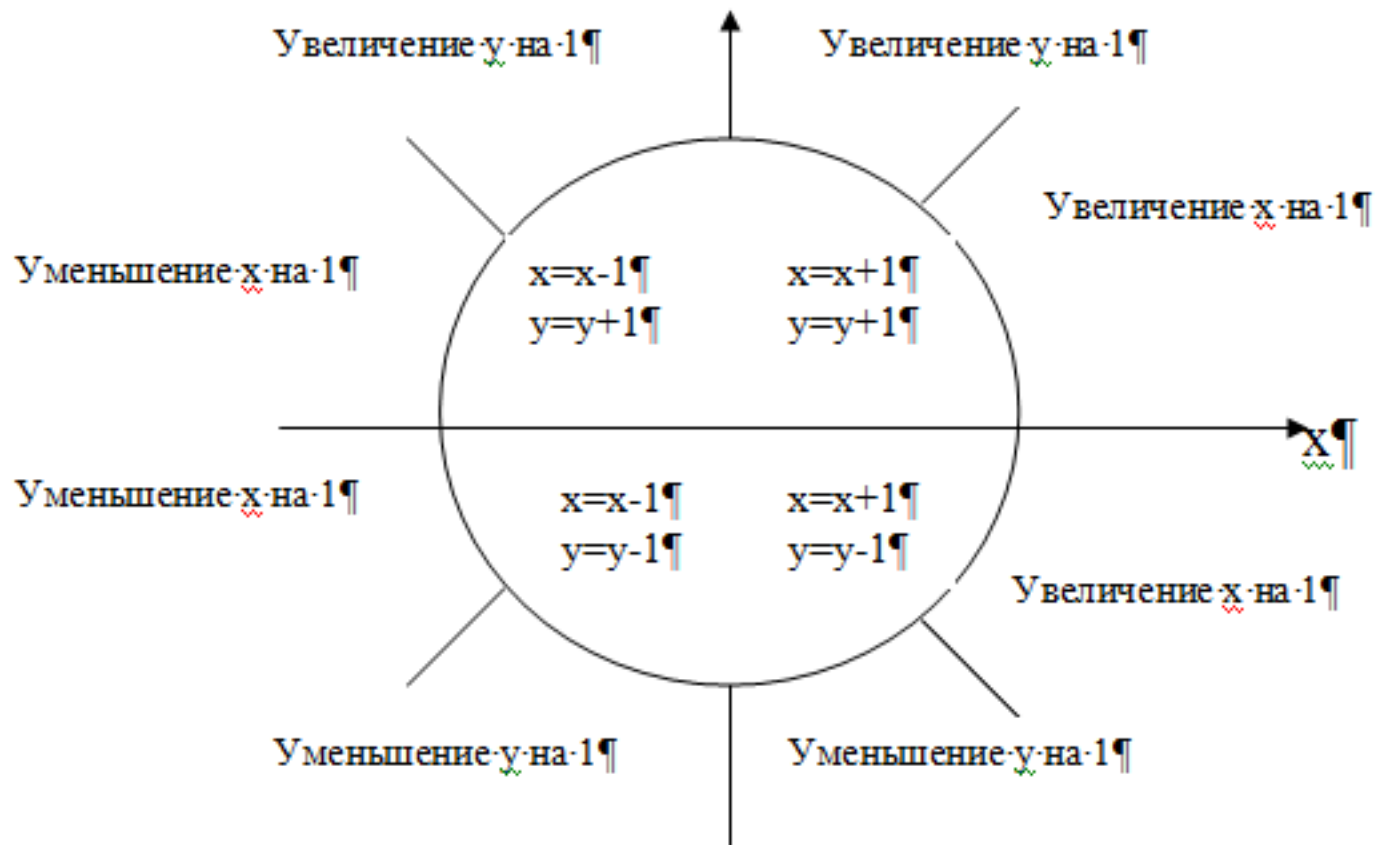
$$d_{i+1} = d_i + 2dx$$

if  $d_i \geq 0$  then

$$x_{i+1} = x_i + 1$$

$$d_{i+1} = d_i + 2(dx - dy)$$

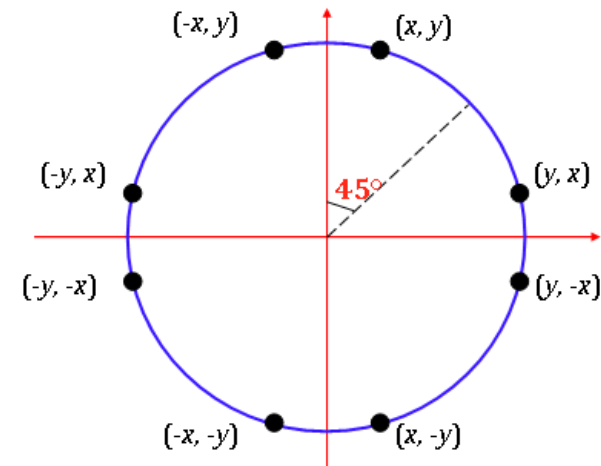
$$y_{i+1} = y_i + 1$$





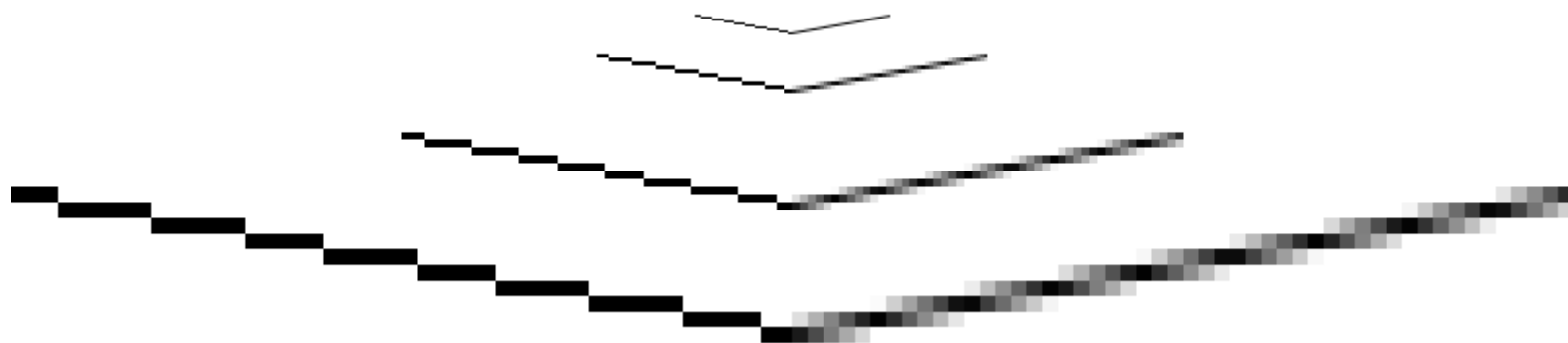
# Модификация для рисования окружностей

```
void BresenhamCircle(int x0, int y0, int radius) {  
    int x = radius; int y = 0;  
    int radiusError = 1 - x;  
    while (x >= y) {  
        DrawPoint(x + x0, y + y0); DrawPoint(y + x0, x + y0);  
        DrawPoint(-x + x0, y + y0); DrawPoint(-y + x0, x + y0);  
        DrawPoint(-x + x0, -y + y0); DrawPoint(-y + x0, -x + y0);  
        DrawPoint(x + x0, -y + y0); DrawPoint(y + x0, -x + y0);  
        y++;  
        if (radiusError < 0)  
            radiusError += 2 * y + 1;  
        else  
            x--; radiusError += 2 * (y - x + 1)  
    }  
}
```

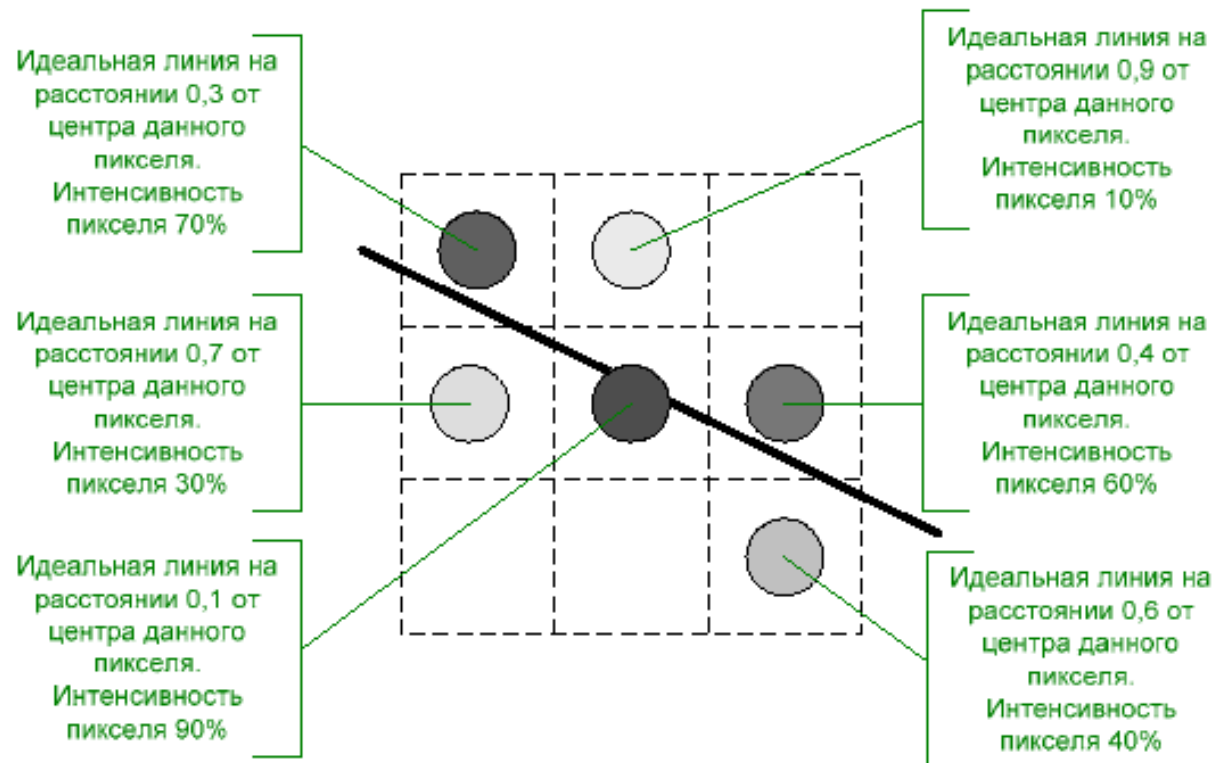


# Сглаживание - Алгоритм Ву (У Сяолиня)

"An Efficient Antialiasing Technique" in the July 1991 issue of Computer Graphics,  
"Fast Antialiasing" in the June 1992 issue of Dr. Dobb's Journal



# Сглаживание - Алгоритм Ву



# Сглаживание - Алгоритм Ву

```
DrawPoint(steeper, x1, y1, 1);  
// Последний аргумент — интенсивность в долях единицы  
float dx = x1 - x0; float dy = y1 - y0;  
float gradient = dy / dx;  
float y = y0 + gradient;  
for (var x = x0 + 1; x <= x1 - 1; x++) {  
    DrawPoint(steeper, x, (int)y, 1 - (y - (int)y));  
    DrawPoint(steeper, x, (int)y + 1, y - (int)y);  
    y += gradient;  
}
```