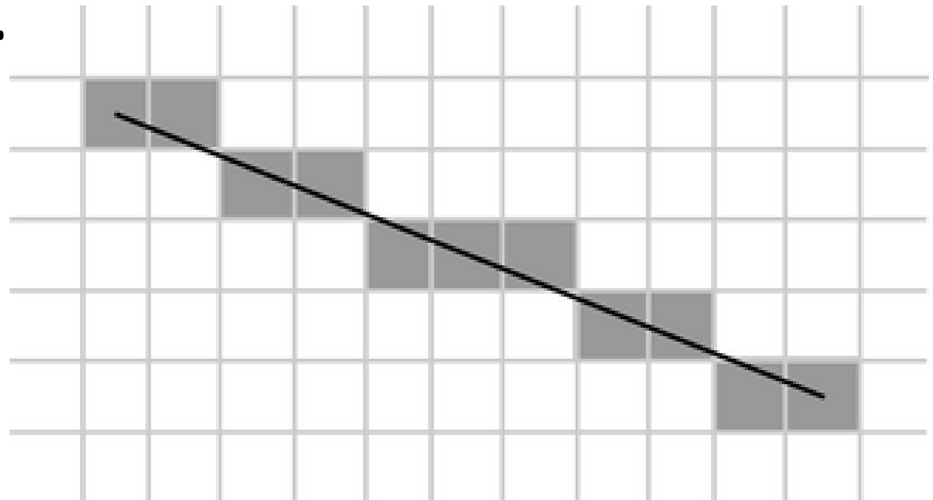
Растровые алгоритмы

Лекция 3

Растровые алгоритмы
Алгоритмы растеризации
Алгоритмы обработки растровых изображений
Алгоритмы перевода графических примитивов в растровую форму
Алгоритмы заполнения областей и многоугольников
Регулировка яркости и контрастности
Масштабирование изображений
Геометрические преобразования
Алгоритмы фильтрации

Растеризация (Rasterisation)

- **Растеризация** — это перевод двухмерного изображения, описанного векторным форматом в пиксели или точки, для вывода на дисплей или принтер. Процесс, обратный векторизации.
- **Растеризация отрезка** — это задача определения какие точки двумерного растра нужно закрасить, чтобы получить близкое приближение прямой линии между двумя заданными точками.



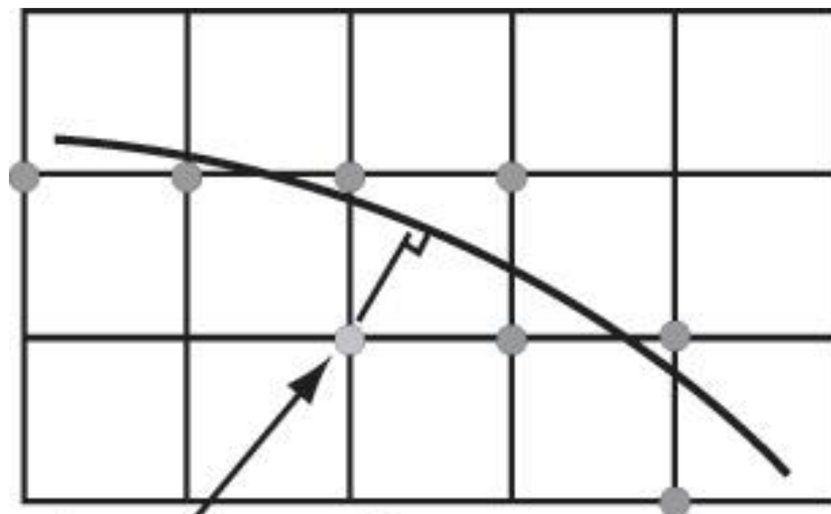
Примитивы

- *Точки*
- *Линии*
- *Прямоугольники (со сторонами, параллельными границам экрана)*
- *Многоугольники*
- *Шрифты*
- *Заливка областей*
- *Плоское отсечение*

Перевод графических примитивов в растровую форму

Перевод графических примитивов в растровую форму

- Пусть у нас есть некоторая кривая, и мы хотим построить ее изображение на растровой решетке.
- Возникает вопрос: какие из ближайших пикселей следует закрашивать? В данной лекции рассмотрим случай построения на монохромном растре, когда возможны только два уровня интенсивности закраски пикселя - "полностью закрашен" или "полностью не закрашен".

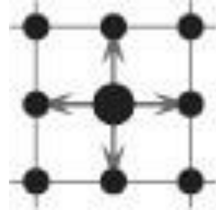


Связность

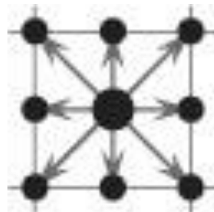
Связность – возможность соединения двух пикселей растровой линией, т.е. последовательным набором пикселей.

Пусть (x_0, y_0) - фиксированный пиксель, а (x, y) - некоторый другой пиксель на плоскости. Тогда для определения их связности вводятся следующие понятия:

4-связность
$|x-x_0|+|y-y_0| = 1$

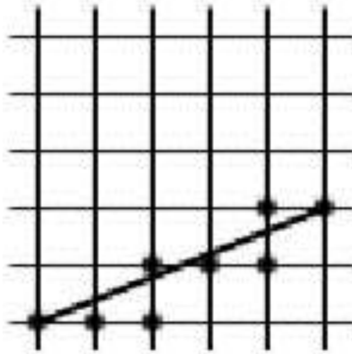


8-связность
$\max\{|x-x_0|, |y-y_0|\} = 1$

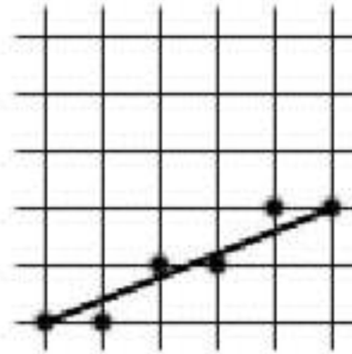


Четырёх и восьми связность

Четырехсвязная линия



Восьмисвязная линия



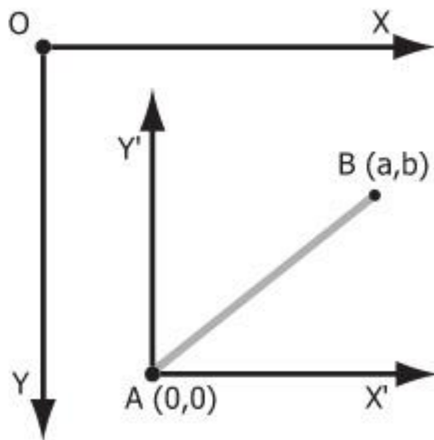
При переводе объектов в растровое представление существуют, алгоритмы, использующие как четырех, так и восьми связность.

Растрирование отрезка

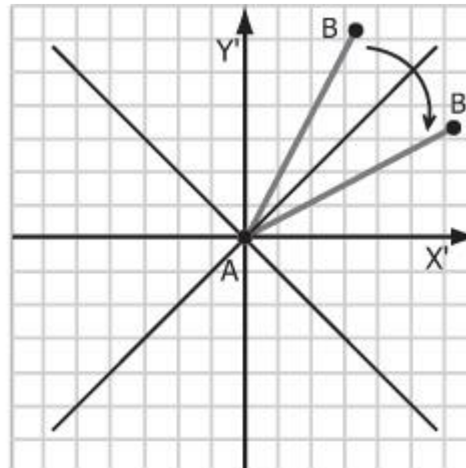
Растрирование отрезка

Пусть наш отрезок - это AB. Перейдем от системы координат Oxy к Ax'y' .

Отрезок может лежать в любом из 8 октантов, но всегда существуют симметрии относительно осей, разделяющих эти октанты, симметрии определяются матрицами $\begin{pmatrix} \pm 1 & 0 \\ 0 & \pm 1 \end{pmatrix}$ и $\begin{pmatrix} 0 & \pm 1 \\ \pm 1 & 0 \end{pmatrix}$,
позволяющими свести задачу к случаю отрезка, лежащего в первом октанте.



1



2

Простейший алгоритм растрирования отрезка

Рассмотрим задачу построения растрового изображения отрезка, соединяющего точки $A(x_a, y_a)$ и $B(x_b, y_b)$.

Для простоты будем считать, что $0 \le y_b - y_a \le x_b - x_a$,

т.е. отрезок находится в первом октанте.

$$\frac{y_b - y_a}{x_b - x_a} \qquad 0$$

Тогда отрезок описывается уравнением:

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

где b можно приравнять 0, перенеся отрезок в начало координат.

Простейший алгоритм растрирования отрезка

```
void line(int x1, int y1, int x2, int y2)
{
    int dx = x2 - x1;
    int dy = y2 - y1;

        float k = dy / dx;
        int x = x1;
        int y = y1;

        while (x <= x2)
        {
            Draw(x, y);
            y = (int)k*x;
            x++;
        }
}
```

Рекуррентный алгоритм растрирования отрезка

Вычислений значений функции ***y = kx*** можно избежать, используя в цикле рекуррентные соотношения, так как при изменении ***x*** на **1** значение ***y*** меняется на ***k***.

Рекуррентный алгоритм растрирования отрезка

```
void line(int x1, int y1, int x2, int y2)
{
        int dx = x2 - x1;
        int dy = y2 - y1;

        float k = dy / dx;
        int x = x1;
        int y = y1;

        while (x <= x2)
        {
            Draw(x, y);
            y += (int)k;
            x++;
        }
}
```

Недостатки рассмотренных алгоритмов

1. Выполняют операции над числами с плавающей точкой, а желательно было бы работать с целочисленной арифметикой;

2. На каждом шаге выполняется операция округления, что также снижает быстродействие.

3. Перевод чисел с плавающей точкой в целочисленные происходит с погрешностью, что приводит к тому , что конечное значение ***y*** не совпадет с ***y2****.*

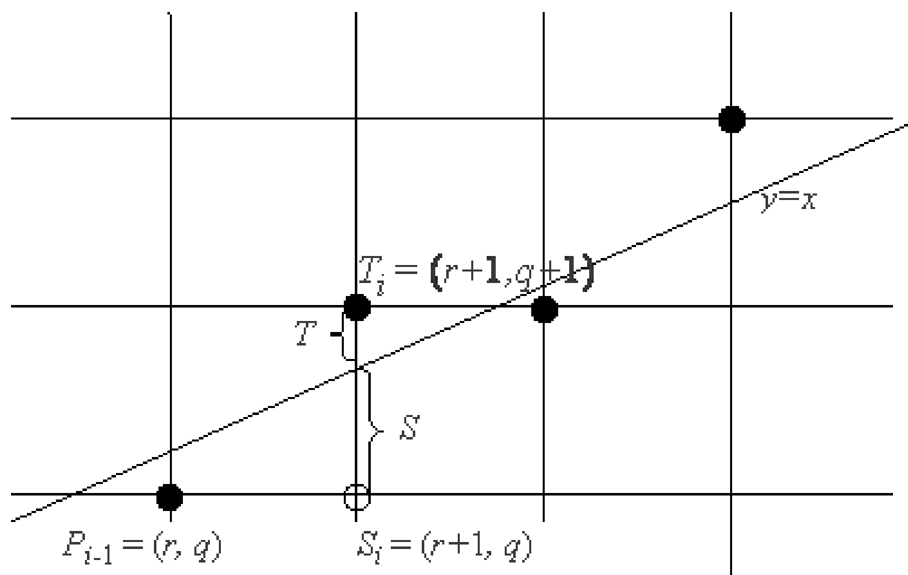
Алгоритм Брезенхейма растеризации отрезка

Эти недостатки устранены в следующем алгоритме **Брезенхейма**.

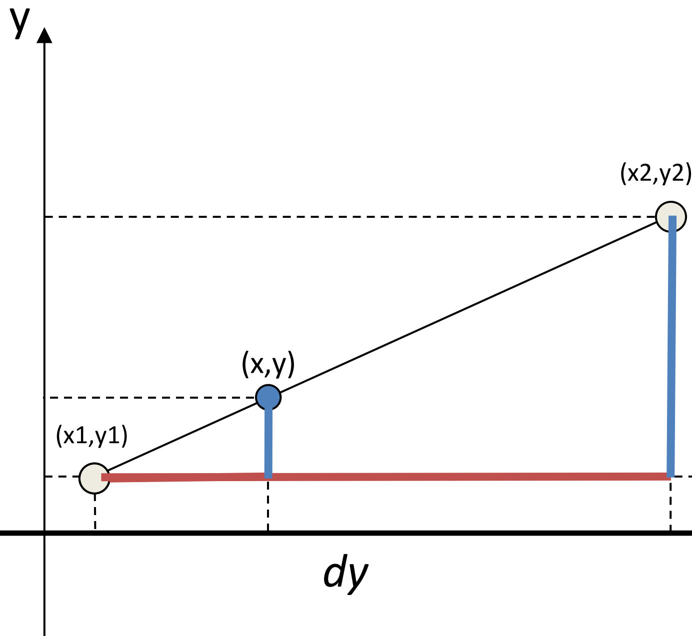
Как и в предыдущем случае, будем считать, что тангенс угла наклона отрезка принимает значение в диапазоне от 0 до 1, т.е. прямая находится в 1-ом октанте.

Алгоритм Брезенхейма растеризации отрезка

- Рассмотрим *i*-й шаг алгоритма. На этом этапе пиксель $P_{i\text{-}1}$ уже найден как ближайший к реальному отрезку. Требуется определить, какой из пикселов (T_i или S_i) будет установлен следующим.



Алгоритм Брезенхейма (метод центральной точки)



$$\frac{x - x1}{x2 - x1} = \frac{y - y1}{y2 - y1}$$

$$(x - x1)\cdot(y2 - y1) - (y - y1)\cdot(x2 - x1) = 0$$

$$dy \cdot x - dx \cdot y - (x1 \cdot dy - y1 \cdot dx) = 0$$

$$f(x, y) = dy \cdot x - dx \cdot y - (x1 \cdot dy - y1 \cdot dx)$$

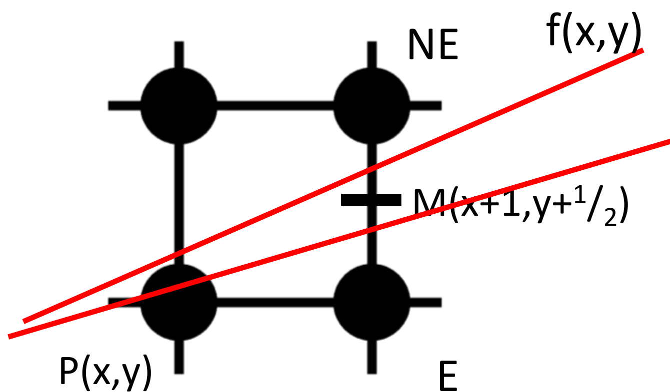
$$\begin{cases} f(x, y) > 0 \\ f(x, y) = 0 \\ f(x, y) < 0 \end{cases}$$

точка *(x,y)* «ниже» прямой

точка *(x,y)* «лежит» на прямой

точка *(x,y)* «выше» прямой

Алгоритм Брезенхейма (метод центральной точки)



Подставляем точку *M* в функцию *f*:

- если *f*(*M*) < 0 выбираем точку *NE*
- если *f*(*M*) >= 0 выбираем точку *E*

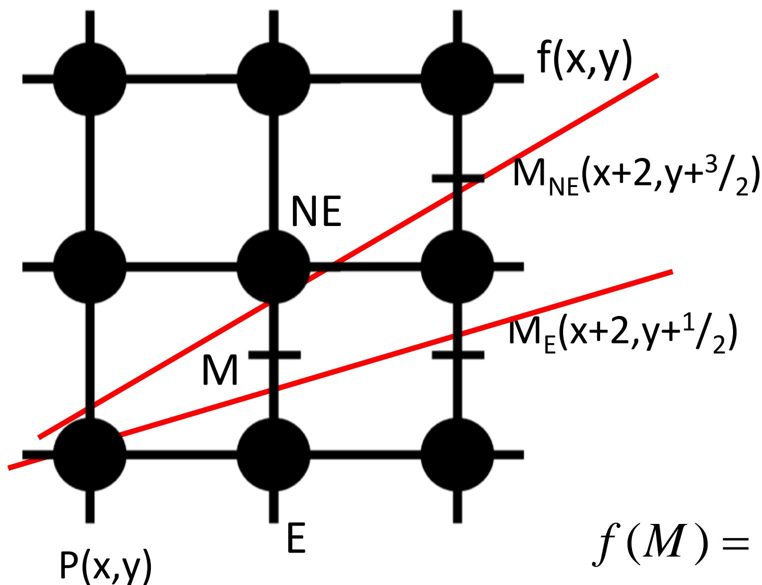
```
int x, y;

x = x1; y = y1;

SetPixel(x, y);
count = dx;
while (count > 0)
{
  count = count - 1;

  if (f(x + 1, y + 0.5) > 0)
    y = y + 1;
  x = x + 1;
  SetPixel(x, y);
}

/* f(x, y) =
 *    dy * x - dx * y - (x1 * dy - y1 * dx)
 * dx = x2 - x1; dy = y2 - y1;
 */
```

Каждый раз вычислять *f*(*x*,*y*)

В коде ось y направлена вниз, поэтому знак “>”

Алгоритм Брезенхейма (метод центральной точки)

$$f(x,y) = dy \cdot x - dx \cdot y - (x1 \cdot dy - y1 \cdot dx)$$



Подставляем точку *M* в функцию *f*:

- если *f*(*M*) < 0 выбираем точку *NE*
- если *f*(*M*) >= 0 выбираем точку *E*

Изменения значения *f*(*M*) при переходе к новым точкам (*E* или *NE*):

$$f(M) = f(x+1, y+{}^1\!/_2) = dy \cdot (x+1) - dx \cdot (y+{}^1\!/_2) - C =$$

$$dy \cdot x + dy - dx \cdot y - {}^{dx}\!/_2 - C$$

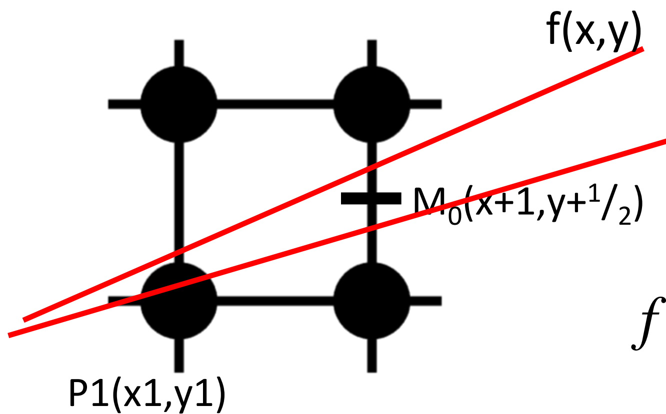
$$f(M_E) = f(x+2, y+{}^1\!/_2) = dy \cdot (x+2) - dx \cdot (y+{}^1\!/_2) - C =$$

$$dy \cdot x + 2 \cdot dy - dx \cdot y - {}^{dx}\!/_2 - C = \underline{f(M) + dy}$$

$$f(M_{NE}) = f(x+2, y+{}^3\!/_2) = dy \cdot (x+2) - dx \cdot (y+{}^3\!/_2) - C =$$

$$dy \cdot x + 2 \cdot dy - dx \cdot y - {}^{3 \cdot dx}\!/_2 - C = \underline{f(M) + dy - dx}$$

Алгоритм Брезенхейма (метод центральной точки)



Известны приращения *f*.

Найдем первоначальное значение для точки (*x*1,*y*1)

$$f(M_0) = f(x1+1, y1+ \tfrac{1}{2}) =$$

$$dy \cdot (x1+1) - dx \cdot (y1 + \tfrac{1}{2}) - (x1 \cdot dy - y1 \cdot dx) =$$

$$\cancel{dy \cdot x1} + dy - \cancel{dx \cdot y1} - \tfrac{1}{2} dx - \cancel{dy \cdot x1} + \cancel{dx \cdot y1} =$$

$$dy - \tfrac{dx}{2}$$

Алгоритм Брезенхейма (метод центральной точки)

Сохранились вещественные числа.

Сделаем замену: 2f = e

Тогда помеченные строки изменяться на:

```
e = 2 * dy - dx;

e > 0

e = e + 2 * dy - 2 *dx;

e = e + 2 * dy
```

и e – целое число.

```
int x, y, dx, dy;
float f;

dx = x2 - x1;
dy = y2 - y1;
f = dy - dx / 2.0;

x = x1; y = y1;
SetPixel(x, y);
count = dx;
while (count > 0)
{
  count = count - 1;

  if (f > 0)
  {
    y = y + 1;
    f = f + (dy - dx);
  }
  else
    f = f + dy;
  x = x + 1;
  SetPixel(x, y);
}
```

Алгоритм Брезенхейма (метод центральной точки)

```
int x, y, dx, dy, incrE, incrNE, e;

dx = x2 - x1;
dy = y2 - y1;
e = 2 * dy - dx;
incrE = 2 * dy;
incrNE = 2 * dy - 2 * dx;

x = x1; y = y1;
SetPixel(x, y);
count = dx;
while (count > 0)
{
  count = count - 1;

  if ( e > 0)
  {
    y = y + 1;
    e = e + incrNE;
  }
  else
    e = e + incrE;
  x = x + 1;
  SetPixel(x, y);
}
```

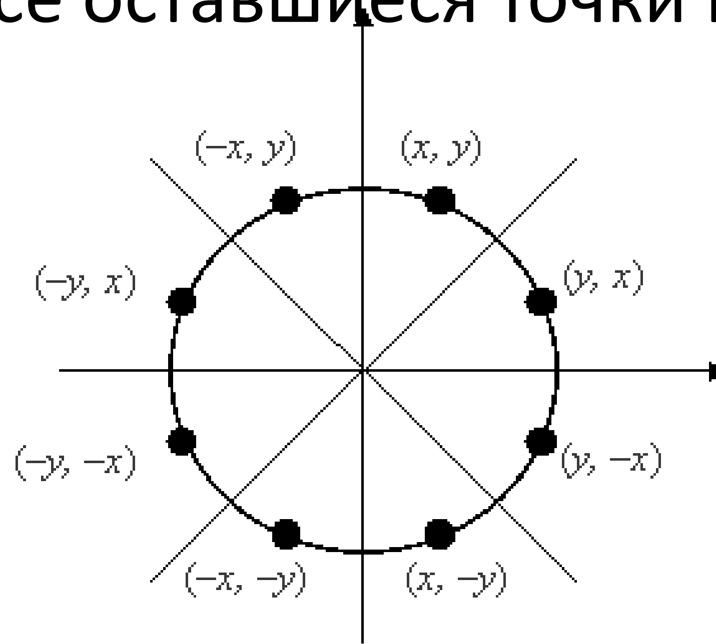
Алгоритм Брезенхейма растеризации отрезка

Преимуществом алгоритма является то, что для работы алгоритма требуются минимальные арифметические возможности: сложение, вычитание и сдвиг влево для умножения на 2.

Растровая развертка окружности

Для упрощения алгоритма растровой развёртки стандартной окружности можно воспользоваться её симметрией относительно координатных осей и прямых $y = \pm x$;

в случае, когда центр окружности не совпадает с началом координат, эти прямые необходимо сдвинуть параллельно так, чтобы они прошли через центр окружности. Тем самым достаточно построить растровое представление для **1/8 части** окружности, а все оставшиеся точки получить симметрией.



Растрирование окружности (1)

- Существует несколько очень простых, но не эффективных способов преобразования окружностей в растровую форму.
- Например, уравнение записывается как $x^2 + y^2 = R^2$. Решая это уравнение относительно *y*, получим

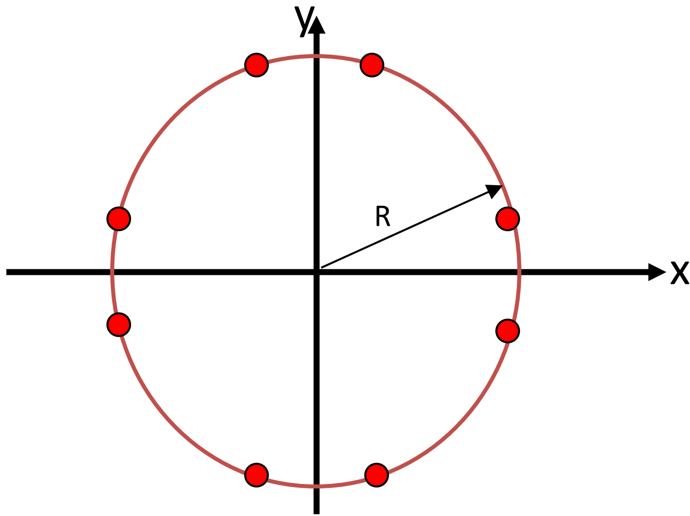
$$y = \pm \sqrt{R^2 - x^2}$$

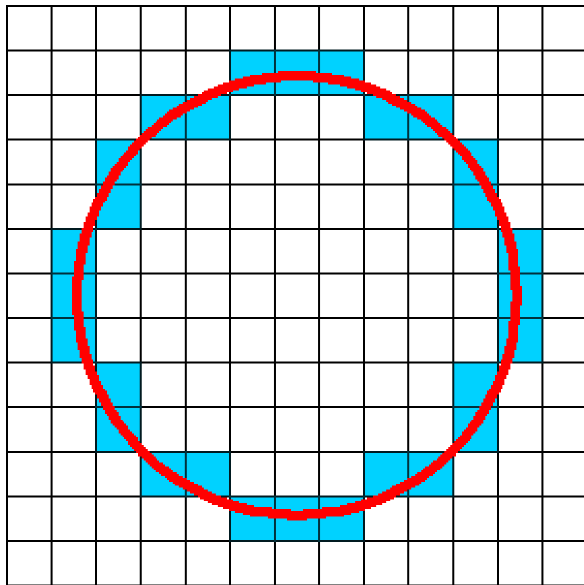
Чтобы изобразить четвертую часть окружности, будем изменять *x* с единичным шагом от 0 до *R* и на каждом шаге вычислять *y*.

Растрирование окружности (2)

- Вторым простым методом растровой развертки окружности является использование вычислений *x* и *y* по формулам ***x* = *R* cos α**, ***y* = *R* sin α** при пошаговом изменении угла **α от 0° до 90°.**

Растрирование окружности



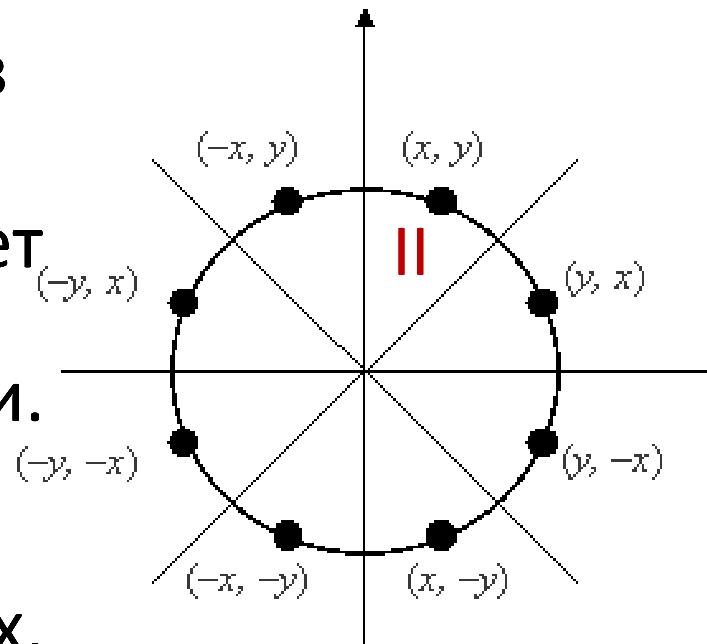


```
SetPixel4(x, y):
  SetPixel(Cx, Cy + R);
  SetPixel(Cx, Cy - R);
  SetPixel(Cx + R, Cy);
  SetPixel(Cx - R, Cy);

SetPixel8(x, y):
  SetPixel(Cx + x, Cy + y);
  SetPixel(Cx - x, Cy + y);
  SetPixel(Cx + x, Cy - y);
  SetPixel(Cx - x, Cy - y);
  SetPixel(Cx + y, Cy + x);
  SetPixel(Cx - y, Cy + x);
  SetPixel(Cx + y, Cy - x);
  SetPixel(Cx - y, Cy - x);
```

(*Cx*, *Cy*) – координаты центра окружности

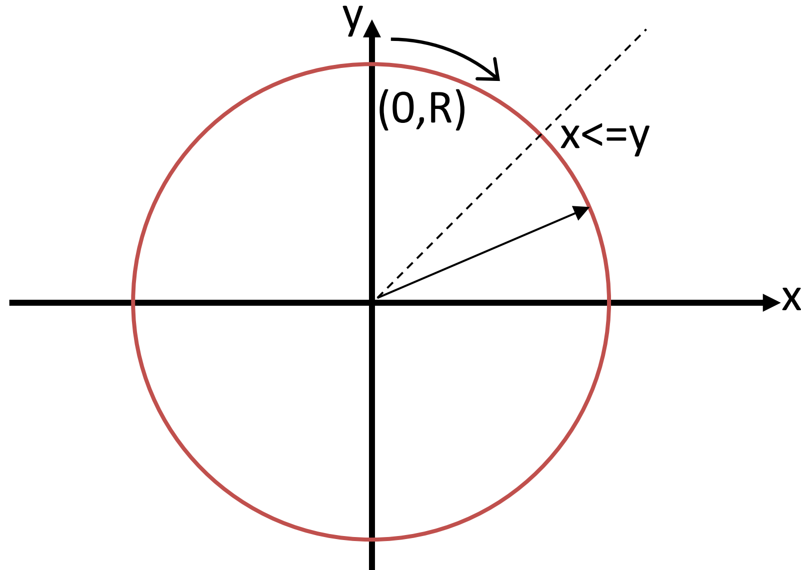
Алгоритм Брезенхейма для растрирования окружности

Рассмотрим участок окружности из второго октанта $x \in [0, R/\sqrt{2}]$.



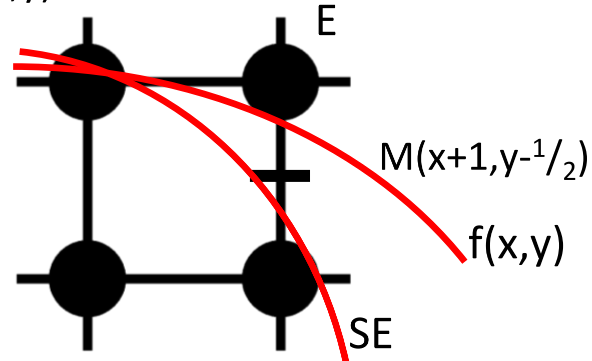
На каждом шаге алгоритм выбирает точку P_i (x_i, y_i), которая является ближайшей к истинной окружности. Идея алгоритма заключается в выборе ближайшей точки при помощи управляющих переменных, значения которых можно вычислить в пошаговом режиме с использованием небольшого числа сложений, вычитаний и сдвигов.

Окружность: Алгоритм Брезенхейма (метод центральной точки)



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

$$\begin{cases} f(x,y) > 0 & \text{точка } (x,y) \text{ вне круга} \\ f(x,y) = 0 & \text{точка } (x,y) \text{ на окружности} \\ f(x,y) < 0 & \text{точка } (x,y) \text{ внутри круга} \end{cases}$$

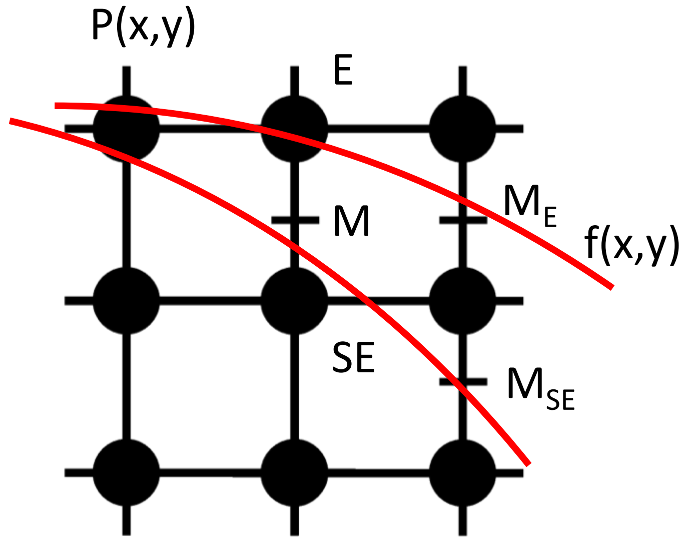



```
int x, y;

x = 0; y = R;

SetPixel4(x, y);
while (x <= y)
{
  if (f(x + 1, y - 0.5) > 0)
    y = y - 1;
  x = x + 1;
  SetPixel8(x, y);
}
```

Подставляем точку M в функцию f:

- если f(M) >= 0 выбираем точку SE
- если f(M) < 0 выбираем точку E

Окружность: Алгоритм Брезенхейма (метод центральной точки)



Изменения значения f(M) при переходе к новым точкам (E или SE):

$$f(M) = f(x+1, y-1/2) = (x+1)^2 + (y-1/2)^2 - R^2 =$$

$$x^2 + 2x + 1 + y^2 - y + 1/4 - R^2$$

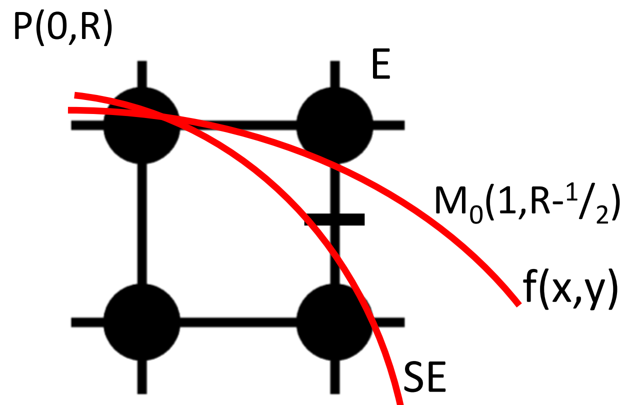
$$f(M_E) = f(x+2, y-1/2) = (x+2)^2 + (y-1/2)^2 - R^2 =$$

$$x^2 + 4x + 4 + y^2 - y + 1/4 - R^2 = f(M) + 2x + 3$$

$$f(M_{NE}) = f(x+2, y-3/2) = (x+2)^2 + (y-3/2)^2 - R^2 =$$

$$x^2 + 4x + 4 + y^2 - 3y + 9/4 - R^2 = f(M) + 2x - 2y + 5$$

Окружность: Алгоритм Брезенхейма (метод центральной точки)



Определили приращения f.
Найдем первоначальное значение для точки (x1,y1)

$$f(M_0) = f(1, R - \tfrac{1}{2}) = 1^2 + (R - \tfrac{1}{2})^2 - R^2 =$$

$$1 + R^2 - R + \tfrac{1}{4} - R^2 = \tfrac{5}{4} - R$$

Все приращения - целые. Сравнение f с 0 строгое: ‘<‘.
Поэтому из первоначального f можно вычесть $^1/_4$.

```
int x, y, f = 1 - R;

x = 0; y = R;

SetPixel4(x, y);
while (x <= y)
{
  if (f > 0)
  {
    y = y - 1;
    f = f + 2 * (x - y) + 5;
  }
  else
    f = f + 2 * x + 3;
  x = x + 1;
  SetPixel8(x, y);
}
```

Окружность: Алгоритм Брезенхейма (метод центральной точки)

Дополнительная оптимизация:

Просчитаем изменение приращений по направлениям E и SE (`incrE=2*x+3` и `incrSE=2*(x-y)+5`) для избавления от доступа к переменным.

• Если выбрана точка E, то ‘x’ увеличивается на 1 и:

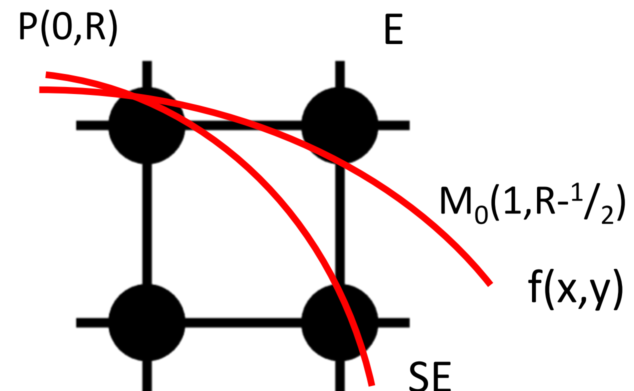
`incrE=incrE+2` и `incrSE=incrSE+2`

• Если выбрана точка SE, то ‘x’ увеличивается на 1, ‘y’ уменьшается на 1 и:

`incrE=incrE+2` и `incrSE=incrSE+4`

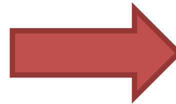
Изначальные значения:

`incrE=3` и `incrSE=5-2*R`

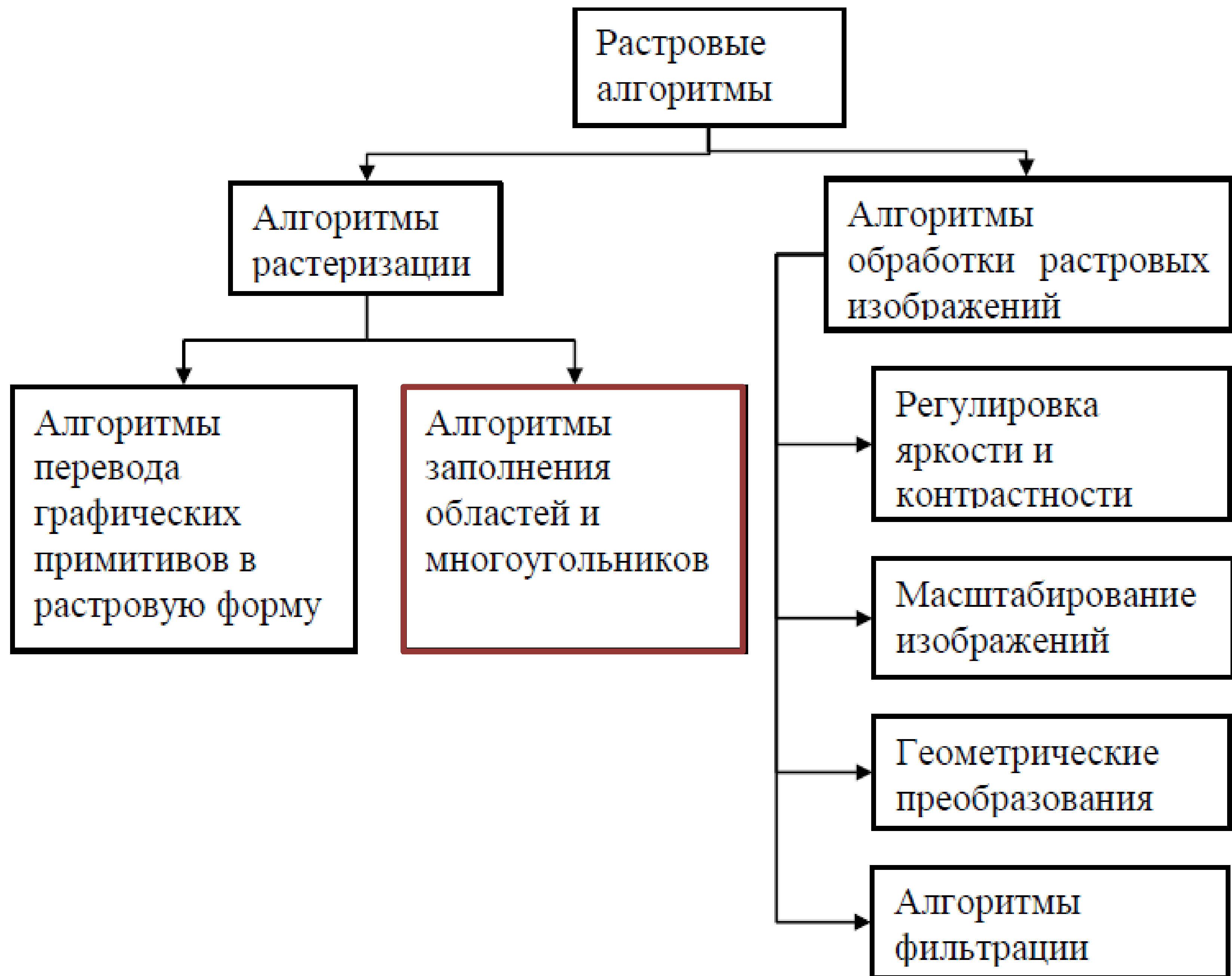


Окружность: Алгоритм Брезенхейма (метод центральной точки)

```
int x, y, f = 1 - R;

x = 0; y = R;

SetPixel4(x, y);
while (x <= y)
{
  if (f > 0)
  {
    y = y - 1;
    f = f + 2 * (x - y) + 5;
  }
  else
    f = f + 2 * x + 3;
  x = x + 1;
  SetPixel8(x, y);
}
```



```
int
  x = 0, y = R,
  f = 1 - R,
  incrE = 3,
  incrSE = 5 - 2 * R;

SetPixel4(x, y);
while (x <= y)
{
  if (f > 0)
  {
    y = y - 1;
    f = f + incrSE;
    incrSE = incrSE + 4;
  }
  else
  {
    f = f + incrE;
    incrSE = incrSE + 2;
  }
  incrE = incrE + 2;
  x = x + 1;
  SetPixel8(x, y);
}
```


Растровые алгоритмы
Алгоритмы растеризации
Алгоритмы обработки растровых изображений
Алгоритмы перевода графических примитивов в растровую форму
Алгоритмы заполнения областей и многоугольников
Регулировка яркости и контрастности
Масштабирование изображений
Геометрические преобразования
Алгоритмы фильтрации

Заполнение многоугольников

Заполнение многоугольников

1. Простейший способ закраски многоугольника состоит в проверке принадлежности каждой точки этому многоугольнику.

2. Более эффективные алгоритмы используют тот факт, что соседние пиксели, вероятно, имеют одинаковые характеристики (***пространственная когерентность***).

В случае с многоугольником когерентность пикселей определяется вдоль сканирующей строки. Сканирующие строки обычно изменяются от «верха» многоугольника до его «низа».

Заполнение многоугольников

Алгоритм

1. Ищутся пересечения при каждом значении *y.*

2. Точки пересечения сортируется по *x* в порядке возрастания.

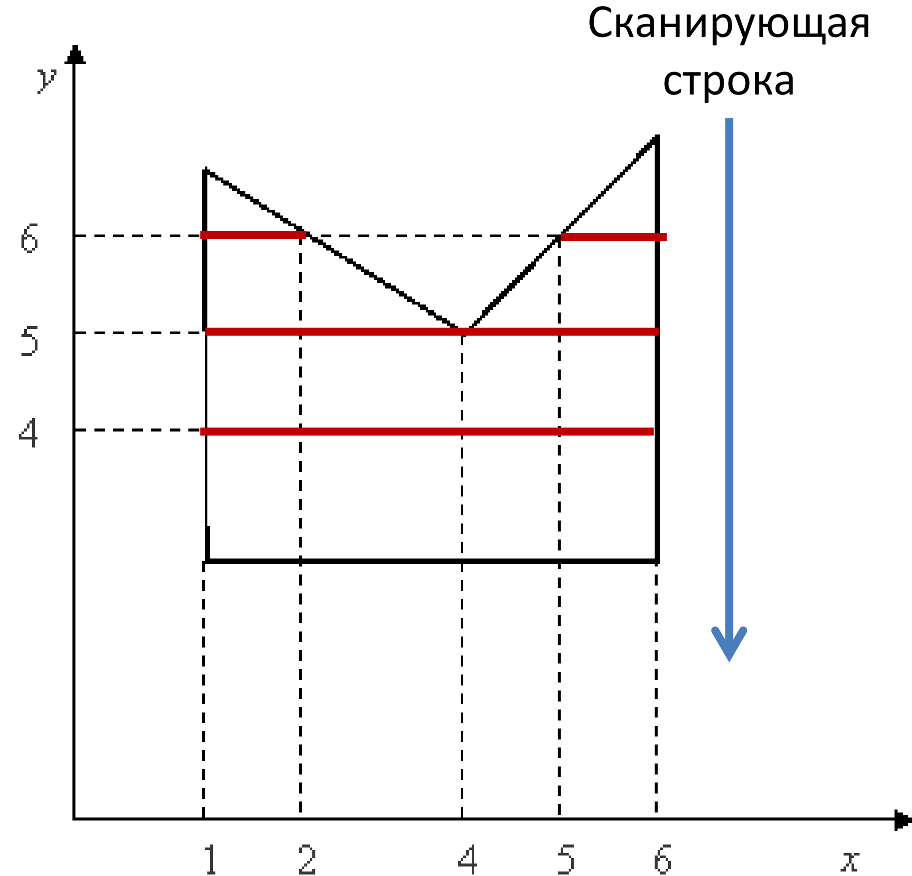
2. Между парами точек пересечения закрашиваются все пиксели.

Пример

1. $y = 4$: $x = 1$ и $x = 6$ =>(1, 6).

2. $y = 6$: $x = 1$; $x = 2$; $x = 5$; $x = 6$.
=>(1, 2) и (5, 6).

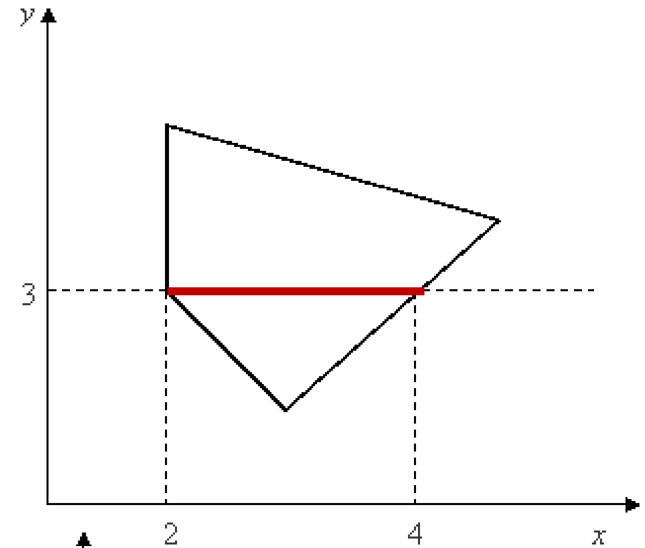
3. $y = 5$ x= (1, 4, 4, 6) =>(1, 4) и (4, 6).



Заполнение многоугольников

y = 3 x= (2, 2, 4).

Если вершину учитывать дважды => (2, 2). Следовательно, при пересечении вершины сканирующей строкой она должна учитываться единожды => (2, 4).



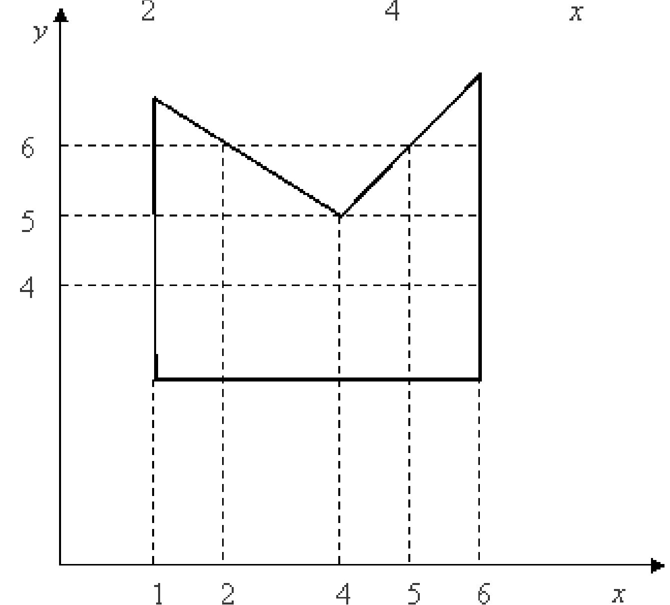
Правило: если вершина является *мин* или *мах*, то точку учитывать 2 раза, если нет, то 1 раз.

Условие нахождения локального минимума или максимума:

Если у обоих концов сторон угла координаты *y* больше, чем *y* вершины пересечения, то вершина – локальный **минимум**.

Если меньше, то вершина пересечения – локальный **максимум**.

Иначе – **промежуточная вершина**.



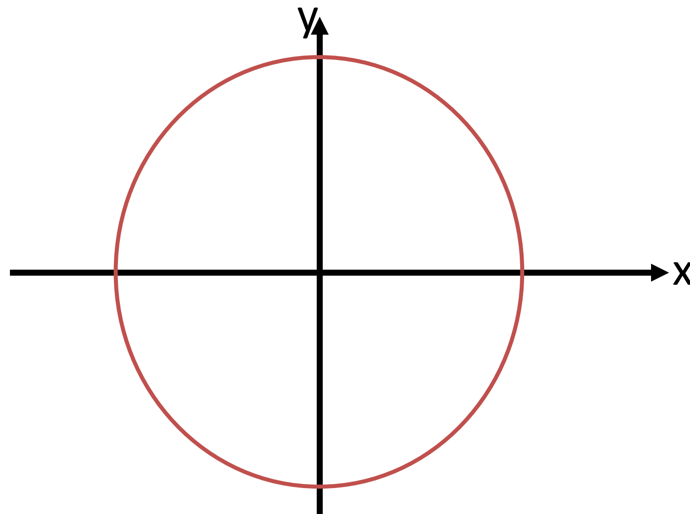
Заполнение многоугольников. САР

Для ускорения работы алгоритма используется список активных ребер (САР), который содержит те **ребра** многоугольника, которые **пересекают сканирующую строку**.

При пересечении очередной сканирующей строки **вершины** многоугольника, из САР удаляются ребра, которые находятся выше, и добавляются концы, которые пересекает сканирующая строка.

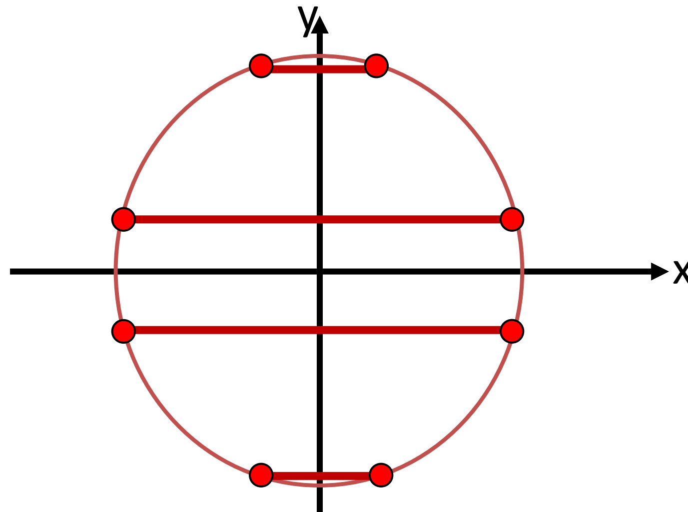
При работе алгоритма находятся пересечения сканирующей строки только с ребрами из САР.

Как закрасить окружность?



Как закрасить окружность?

Во время ее отрисовки, проводя горизонтальные линии между соответствующими точками:

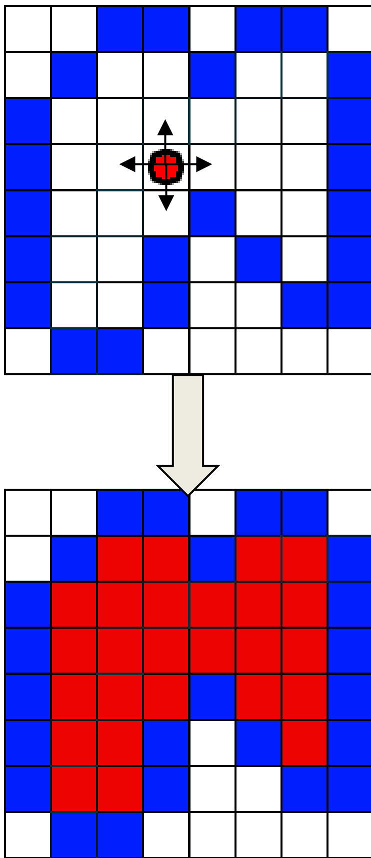


Закраска области, заданной цветом границы

Закраска произвольной области, заданной цветом границы

- Рассмотрим область, ограниченную набором пикселей заданного цвета и точку (x, y), лежащую внутри этой области.

Простейший рекурсивный алгоритм закраски



```
void PixelFill (int x, int y, int border_color, int color)
{
        int c = getpixel(x, y);
        if ((c != border_color) && (c != color))
        {
                putpixel(x, y, color);
                PixelFill(x – 1, y, border_color, color);
                PixelFill(x + 1, y, border_color, color);
                PixelFill(x, y – 1, border_color, color);
                PixelFill(x, y + 1, border_color, color);
        }
}
```

Простейший рекурсивный алгоритм

Этот алгоритм является слишком неэффективным, так как для всякого уже отрисованного пикселя функция вызывается ещё 4 раза и, кроме того, этот алгоритм требует слишком большого объёма стека из-за большой глубины рекурсии.

Для решения задачи закраски области предпочтительнее алгоритмы, способные **обрабатывать** сразу целые **группы пикселей**, т. е. использовать их «связность».

Если данный пиксель принадлежит области, то, скорее всего, его ближайшие соседи также принадлежат данной области.

Алгоритм закрашивания линиями

Группой таких **пикселов** обычно выступает **полоса**, **определяемая правым пикселем**. **Для хранения правых определяющих пикселов используется стек**.

1. Сначала заполняется горизонтальная полоса пикселей, содержащих начальную точку.

2. Затем, чтобы найти самый правый пиксель каждой строки, справа налево проверяется строка, предыдущая по отношению к только что заполненной полосе. Адреса найденных пикселов заносятся в стек.

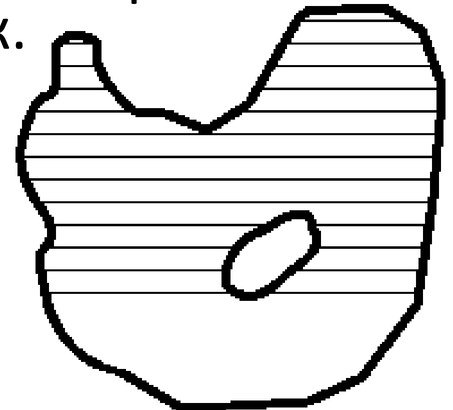
3. То же самое выполняется и для строки, следующей и за последней заполненной полосой.

Когда строка обработана таким способом, в качестве новой начальной точки используется пиксель, адрес которого берется из стека. Для него повторяется вся описанная процедура. Алгоритм заканчивает свою работу, если стек пуст.

- Рекурсивный, но глубина рекурсии пропорциональна лишь числу пикселов в линии. В случае создания стека достаточно хранить 1 пиксел на каждый закрашиваемый участок

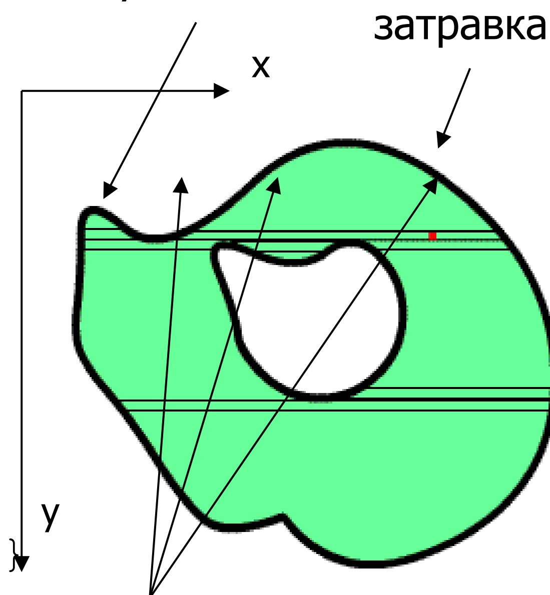
Закраска области с использованием сканирующих строк

1. Поместим затравочную точку в стек.
2. Извлекаем координаты точки с вершины стека в переменные *(x,y)*
3. Заполняем максимально возможный интервал, в котором находится точка, вправо и влево вплоть до достижения граничных точек.
4. Запоминаем крайнюю левую x_l и крайнюю правую x_r абсциссы заполненного интервала.
5. В соседних строках над и под интервалом (x_l, x_r) находим незаполненные к настоящему моменту внутренние точки области, которые объединены в интервалы, а в правый конец каждого такого интервала помещаем на стек.
6. Если стек не пуст, то переходим к пункту 2.

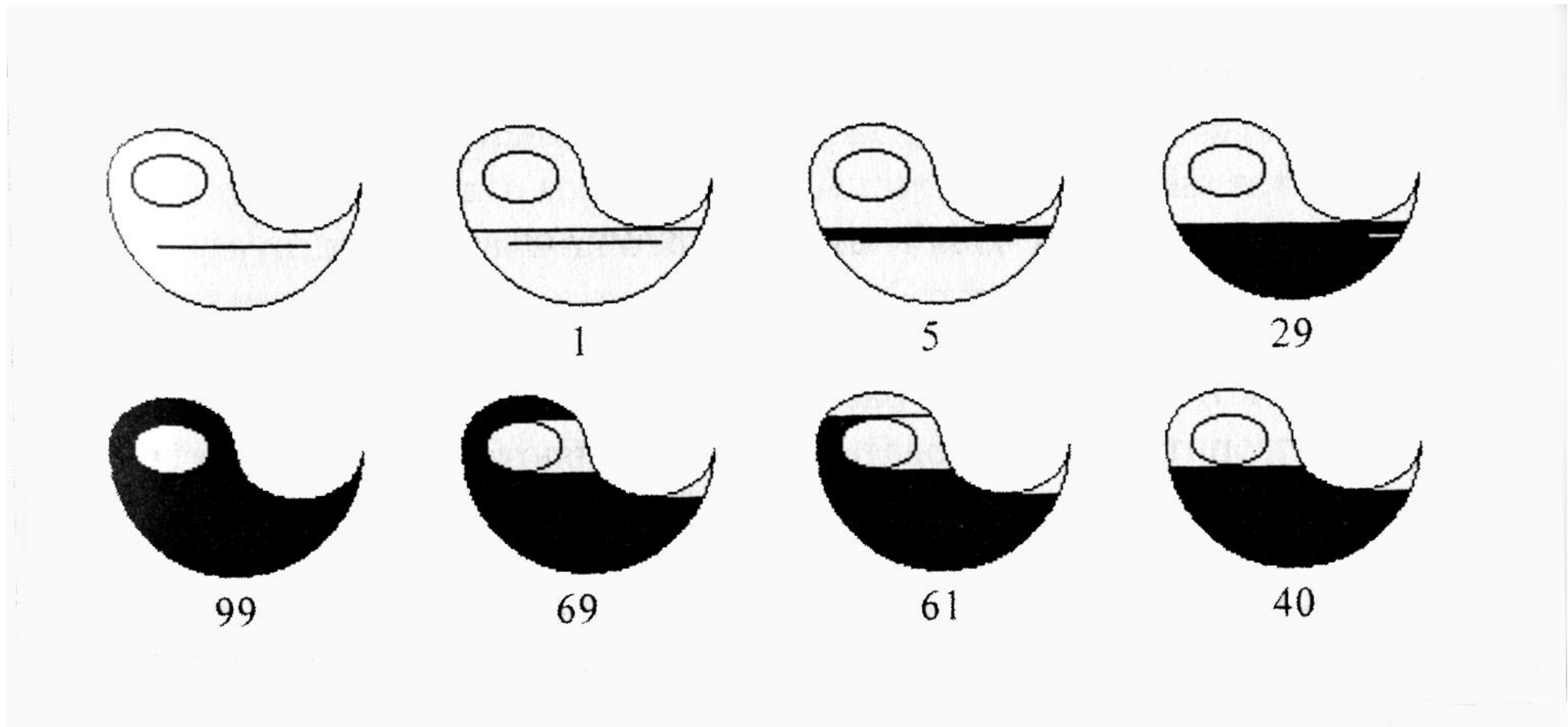


Алгоритмы закрашивания линиями

```
int lineFill(int x, int y,int dy,int preXL, int preXR)
{
int xl=x, xr=x; int c;
//левая и правая граница текущей горизонтали
do {xl--; c=getPixel(xl,y):} while (c!=BORDER);
do {xr++; c=getPixel(xr,y):} while (c!=BORDER);
xl++; xr--;

Line(xl,y,xr,y,fillC); //закрашиваем горизонталь

//Рекурсии:
//1)проверка и закраска над горизонталью
for (x=xl; x<=xr; x++) {c=getPixel(x,y+dy);
    if (c!=BORDER) x=lineFill(x,y+dy,dy,xl,xr);}

// 2)оглянулись и закрасили левее предыдущей
// левой и правее предыдущей правой границ
for (x=xl; x<=preXL; x++) {c=getPixel(x,y-dy);
    if (c!=BORDER) x=lineFill(x,y-dy,-dy,xl,xr);}
for (x=preXR; x<=xr; x++) {c=getPixel(x,y-dy);
    if (c!=BORDER) x=lineFill(x,y-dy,-dy,xl,xr);}
return xr; }
Обращение в затравке: lineFill(xSt,ySt,1,xSt,xSt);
```



Алгоритм закрашивания линиями



Закрашивание контура линиями

Отсечение многоугольников

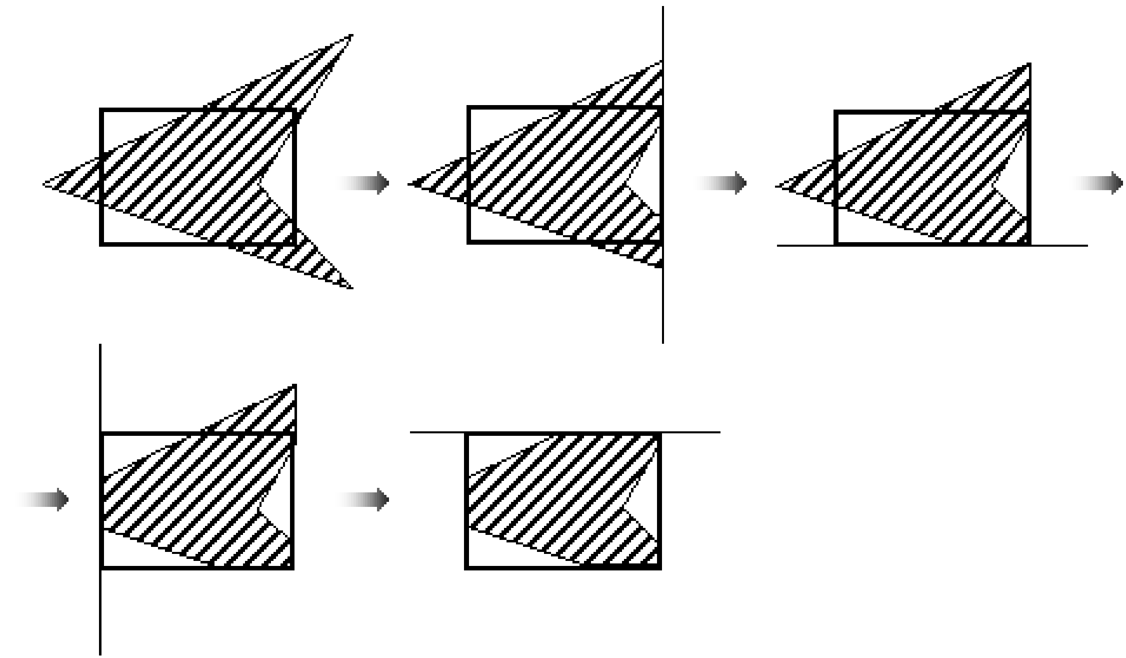
Отсечение многоугольников

Отсечение многоугольников чаше всего проводится для отбрасывания частей многоугольника, выходящих за границу прямоугольной области, которая определяет экран или область окна.

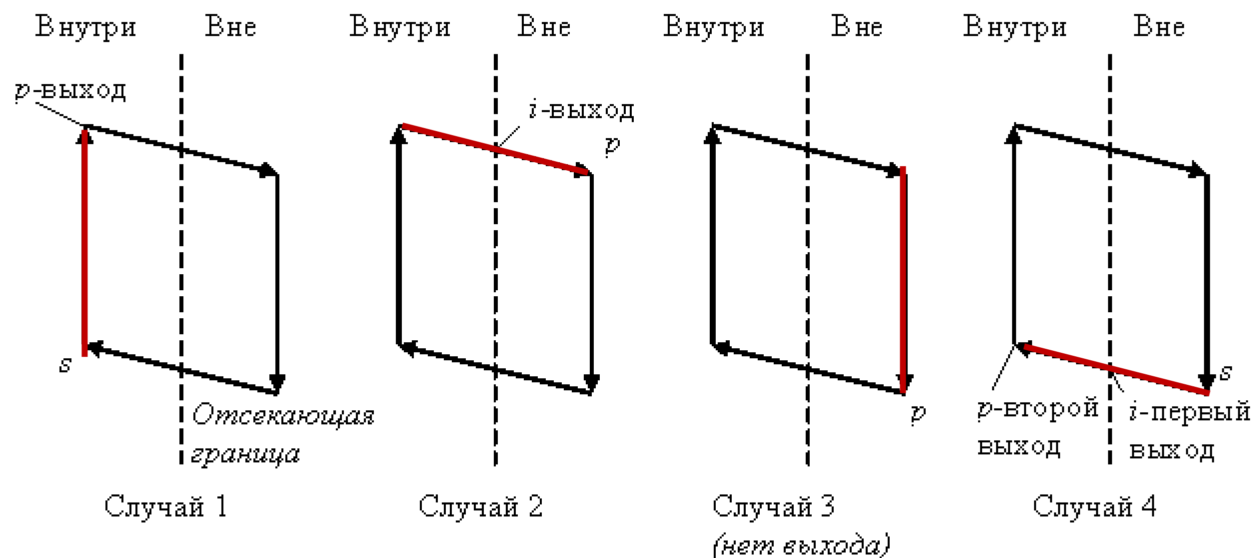
Однако отсечение может проводиться относительно и другого многоугольника. При этом порождается новый многоугольник или несколько новых многоугольников.

Рассмотрим алгоритм **Сазерленда-Ходгмана**. В алгоритме используется стратегия, которая позволяет решение общей задачи свести к решению ряда простых и похожих подзадач.

Примером такой подзадачи является отсечение многоугольника относительно одной отсекающей границы. Последовательное решение четырех таких задач позволяет провести отсечение относительно прямоугольной области.



- На вход алгоритма поступает последовательность вершин многоугольника $V_1, V_2, \ldots, V_n$. Ребра многоугольника проходят от V_i к V_{i+1} от V_n к V_1. С помощью алгоритма производится отсечение относительно ребра и выводится другая последовательность вершин, описывающая усеченный многоугольник.
- Алгоритм «обходит» вокруг многоугольника от V_n к V_1 и обратно к V_n, проверяя на каждом шаге соотношение между последовательными вершинами и отсекающей границей. Необходимо проанализировать четыре случая:



- В первом случае ребро полностью лежит внутри отсекающей границы, к выходному списку добавляется вершина *p*.
- Во втором случае в качестве вершин выводится точка пересечения *i*, поскольку ребро пересекает границу, а начальная точка была выведена при анализе первого случая.
- В третьем случае обе вершины находятся за пределами границы и ни одна из них не выводится.
- В четвертом случае к выходному списку добавляется и точка пересечения *i,* и вершина *p*.