***Алгоритмы*** ***растровой графики***

**Напомним: Типы графики.**

- Управляется командами отображения, нарисованными в виде штрихов

move(x, y), char("A"), line(x, y)…

На основе геометрических примитивов

line, polygon, circle…

- Общий формат файла: SVG (масштабируемая векторная графика)

- Масштабируемое разрешение, поскольку определяется математическими формулами

- Идеально подходит для каллиграфии/типографики, а также геометрических или высокоструктурированных изображений.

- Также известен как растровое изображение или фиксированный массив пикселей, каждый из которых имеет определенную интенсивность, прозрачность и оттенок.

Отсутствие семантики, самая низкая форма представления

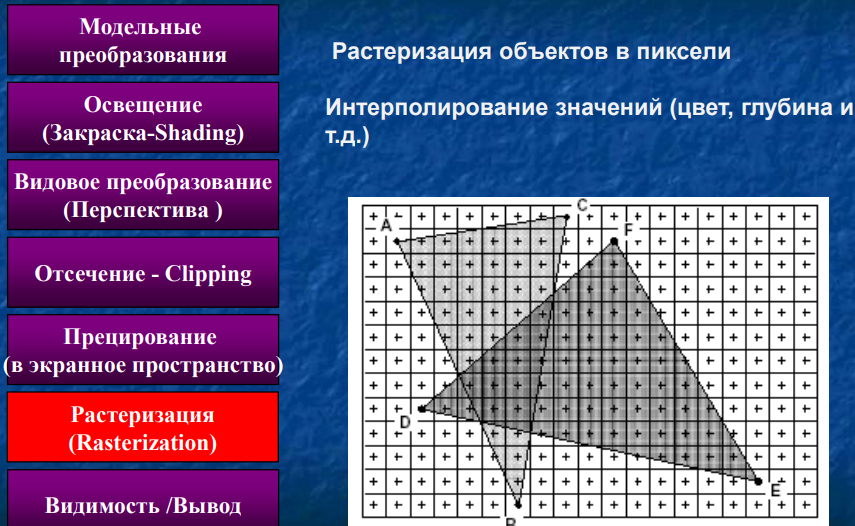
- Фиксированное разрешение \

Распространенные форматы файлов: JPEG, BMP, GIF, TIFF

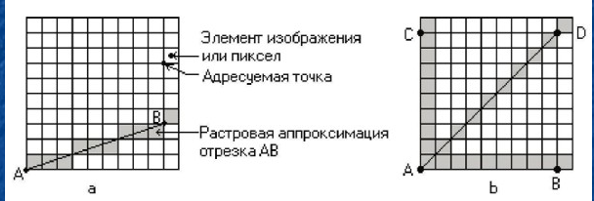
- Обратите внимание на «неровности» (ошибки сглаживания) из-за дискретной выборки непрерывных примитивов.

- Идеально подходит для естественных изображений, ТВ, дисплеев, лазерных принтеров

- Photoshop, MS Paint



**Генерация линий**



*При генерации линий*

* Генерируемый отрезок должен быть прямым
* Начинаться и заканчиваться в заданных точках.
* Яркость отрезка по всей длине должна быть одинаковая и не

зависеть от угла и наклона

* Алгоритм рисования линий должен быть быстрым

Все 4 условия одновременно не могут быть соблюдены

**Преобразует**

* **линии и треугольники**
* **с вершинами с плавающей запятой**

**в**

* **координаты в пикселях**
* **с целыми координатами**
* **в координаты области просмотра (экрана)**

Прежде чем приступать к обсуждению конкретных алгоритмов рисования отрезков, полезно рассмотреть общие требования к таким алгоритмам и ответить на вопрос, каковы желаемые характеристики изображения.

Очевидно, что отрезки должны выглядеть прямыми, начинаться и

заканчиваться в заданных точках.

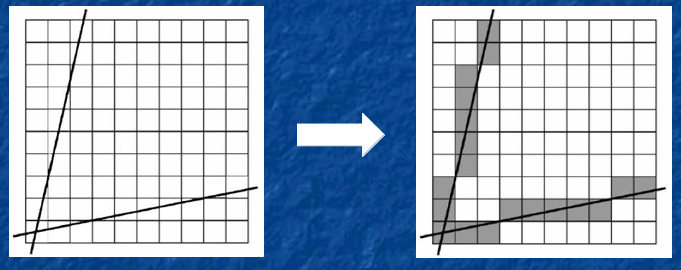
Далее, яркость вдоль отрезка должна быть постоянной и не зависеть от длины и наклона.

Наконец, рисовать нужно быстро. Как это часто бывает, не все из

перечисленных критериев могут быть полностью удовлетворены.

Сама природа растрового дисплея исключает генерацию абсолютно прямых линий (кроме ряда специальных случаев), равно как и точное совпадение начала и конца отрезка с заданными точками. Тем не менее, при достаточно высоком разрешении дисплея можно получить приемлемую аппроксимацию.

* нахождение приближенной длины отрезка,
* сведение вычислений к минимуму,
* предпочтение целой арифметики,
* реализация алгоритмов на аппаратном или микропрограммном уровне.



**Генерация векторов**

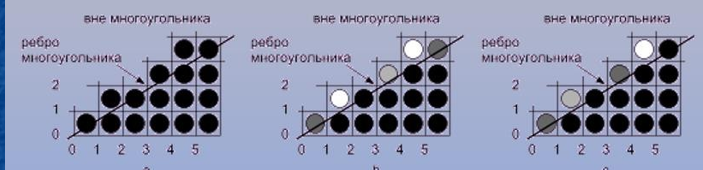
**Постоянная вдоль всего отрезка яркость достигается лишь при проведении горизонтальных, вертикальных и наклоненных под углом 45 ° прямых. Для всех других ориентации разложение в растр приведет к неравномерной яркости, как это показано на рис.**

**Даже для частных случаев яркость зависит от наклона: заметим, например, что расстояние между соседними пикселами для отрезка под углом 45° больше, чем для вертикальных и горизонтальных прямых.**

**Поэтому вертикальные и горизонтальные отрезки будут выглядеть ярче, чем наклонные.**

**Праcтой метод устранения лестничного эффекта**

* **Наиболее заметно ухудшает качество изображения ступенчатость. Имеется следующие способы борьбы со ступенчатостью :**
  + увеличение пространственного разрешения за счет усовершенствования аппаратуры,
  + трактовка пикселя не как точки, а как площадки конечного размера, яркость которой зависит от размера площади пикселя, занятой изображением отрезка,
  + "размывание" резкой границы, за счет частичной подсветки пикселов, примыкающих к формируемому отрезку. Понятно, что при этом ухудшается пространственное разрешение изображения..

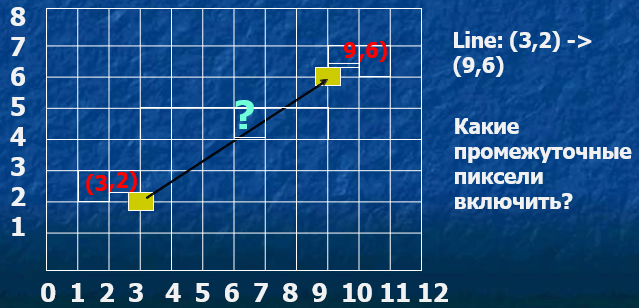


**Алгоритм рисования линии**

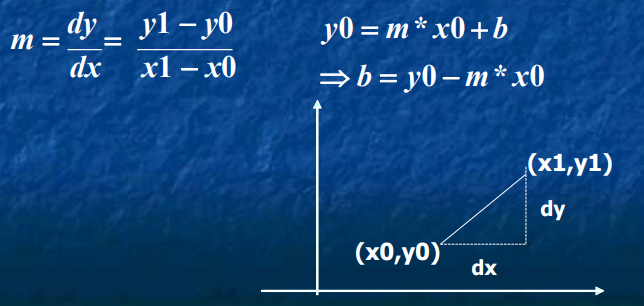
* **●** Значения пикселей (x, y) ограничены целочисленными значениями
* ● Вычисленные промежуточные значения могут быть плавающими
* ● Округление может потребоваться. Например. (10.48, 20.51) округляется до (10, 21)
* ● Округленное значение пикселя находится вне фактического пути линии (неровный !!)
* ● Наклонные линии в конечном итоге имеют зазубрины
* ● Вертикальные, горизонтальные линии, без зазубрин

Программист определяет (x, y) конечных пикселей

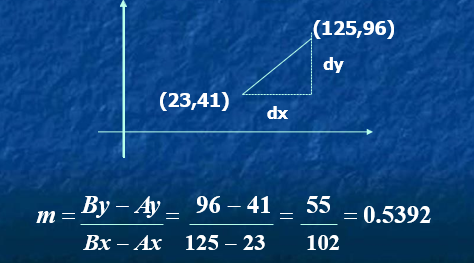
Нужен алгоритм определения пикселей на пути линии



* **Линейное уравнение пересечения**
  + **y = mx + b**
  + **Даны 2 конечные точки (x0,y0), (x1, y1), как посчитать m и b?**



* **Численный пример нахождения m :**
  + **(Ax, Ay) = (23, 41), (Bx, By) = (125, 96)**



**Цифровой дифференциальный анализатор (DDA)**

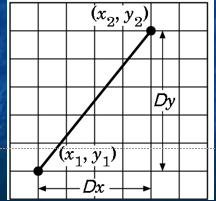
**Алгоритм DDA-линии** растеризует отрезок прямой между двумя заданными точками, используя вычисления в числах с плавающей запятой или целых числах.

С помощью DDА решается дифференциальное уравнение

отрезка, имеющее вид:



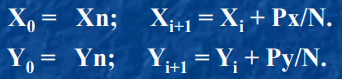
где Py = Yk - Yn - приращение координат отрезка по оси Y, а Px = Xk - Xn - приращение координат отрезка по оси X**.**



При этом DDА формирует дискретную аппроксимацию непрерывного решения этого дифференциального уравнения.

В обычном DDА тем или иным образом определяется количество узлов N, используемых для аппроксимации отрезка.

Затем за N циклов вычисляются координаты очередных узлов:

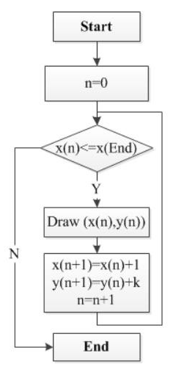


Получаемые значения Xi, Yi преобразуются в целочисленные значения координаты очередного подсвечиваемого пиксела либо округлением, либо отбрасыванием дробной части.

Генератор векторов, использующий этот алгоритм, имеет тот недостаток, что точки могут прописываться дважды, что увеличивает время построения.

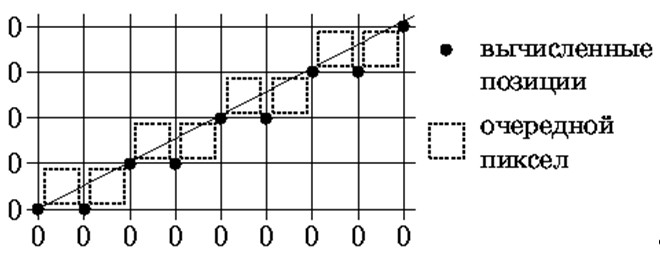
Кроме того из-за независимого вычисления обеих координат нет предпочтительных направлений и построенные отрезки кажутся не очень красивыми.

Проблема заключается в необходимости деления (Py/Px) и сложения вещественных чисел.



**DDA Алгоритм рисования линии**

**Пример генерации отрезка по алгоритму ЦДА**



**Однако, в некоторых случаях в отрезке, разложенным в растр простейшим методом, могут появиться разрывы, например, если выбрать такой отрезок, у которого разброс по оси Y больше разброса по оси X в несколько раз.**

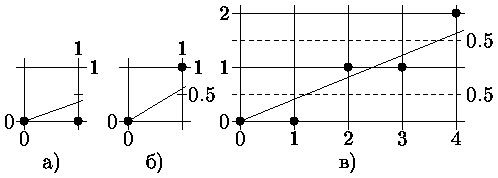
**Вдобавок предложенный алгоритм имеет тот недостаток, что он использует вещественную арифметику.**

**Алгоритм Брезенхэма для отрезков прямых**

Это алгоритм, определяющий, какие точки двумерного растра нужно закрасить, чтобы получить близкое приближение прямой линии между двумя заданными точками.

Это один из старейших алгоритмов в машинной графике — он был разработан Джеком Е. Брезенхемом (Jack E. Bresenham) в компании IBM в 1962 году. Алгоритм широко используется, в частности, для рисования линий на экране компьютера.

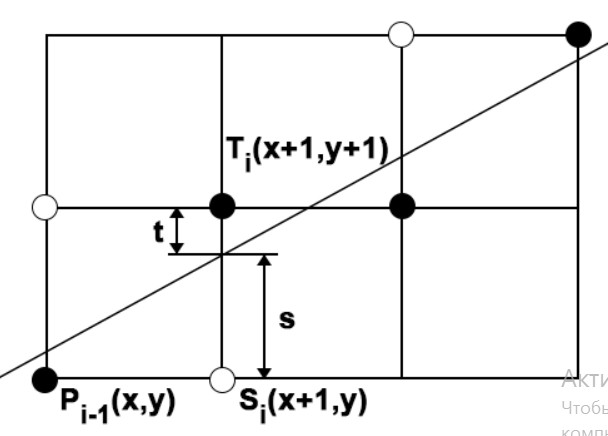
* **Исключите добавление с плавающей запятой из DDA**
* **Предположим снова 0 ≤ m ≤ 1**
* **Предположим, что пиксели центрируются посередине между целыми числами.**



В алгоритме используется управляющая переменная *di*, которая на

каждом шаге пропорциональна разности между *s* и *t*

На рисунке приведен *i*-ый шаг, когда пиксель *Pi-1* уже найден как ближайший к реальному изображаемому отрезку, и теперь требуется определить, какой из пикселей должен быть установлен следующим: *Ti* или *Si*.



**Общий алгоритм Брезенхема**

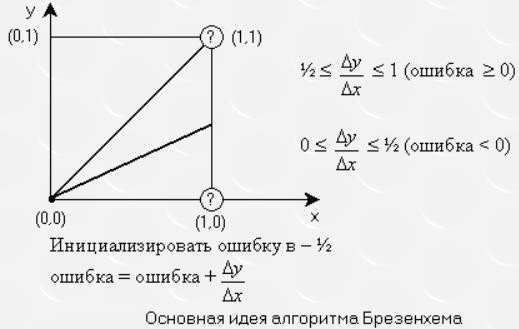
Из рисунка можно заметить, что если угловой коэффициент отрезка из точки (0, 0) больше чем ½,то его пересечение с прямой х = 1 будет расположено ближе к прямой y = 1, чем к прямой y = 0.

Следовательно, точка растра (1, 1) лучше аппроксимирует ход отрезка, чем точка (1, 0).

Если угловой коэффициент

меньше ½, то верно обратное. Для углового коэффициента, равного ½, нет какого-либо предпочтительного выбора.

В данном случае алгоритм выбирает точку (1, 1).



Так как желательно проверять только знак ошибки, то она первоначально устанавливается равной -½.

Таким образом, если угловой коэффициент отрезка больше или равен ½, то величина ошибки в следующей точке растра может быть вычислена как е = -½ + Δy/Δx.

**Недостаток** такого вычисления ошибки в том, что она требует использования арифметики с

плавающей точкой и деления для вычисления углового коэффициента.

Быстродействие алгоритма можно увеличить, если исполь- зовать только целочисленную арифметику и исключить деление

**Растеризация линий.**

**Модификация алгоритма Брезенхэма со сглаживанием границы**

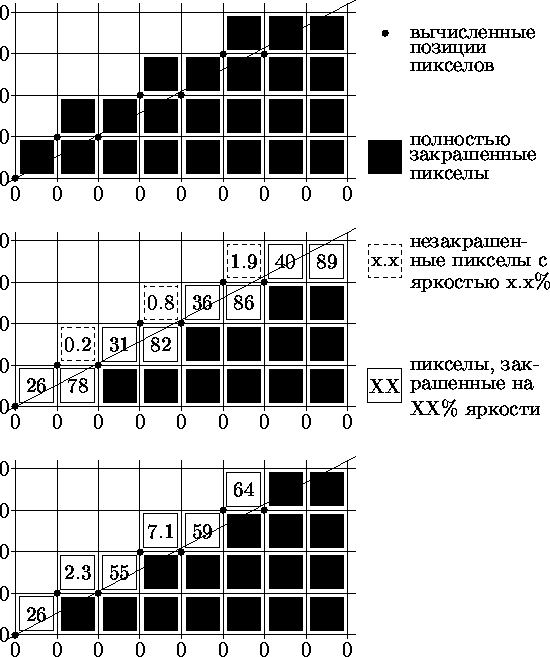
**Модификация** **растеризации линии по Брезенхэму с целью** **сглаживания границы:**

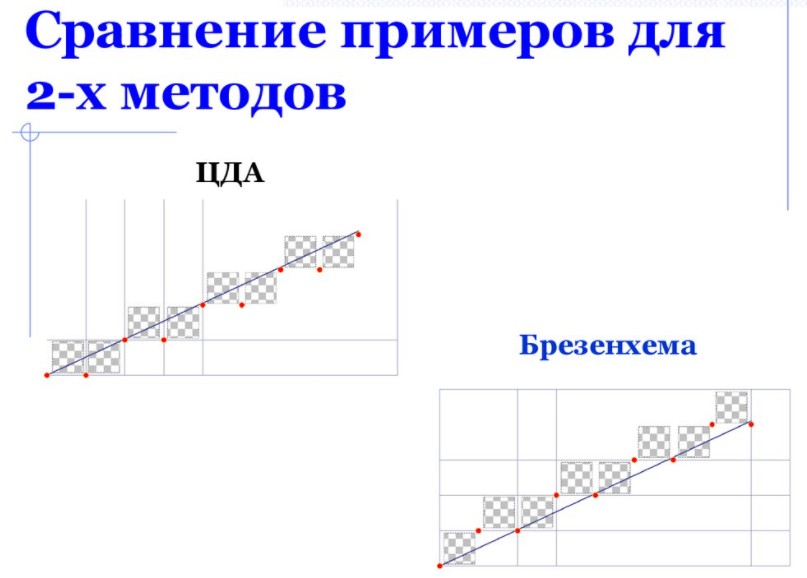
**Полная закраска**

**Неполная закраска 8- связный вариант**

**Неполная закраска, 4- связный** **вариант**

**Для оценки процента закраски** **(p) может быть использована величина функции ошибки: p=100**×**(d-err)/d**





**Растровая развертка сплошных**

Генерацию сплошных областей из простых описаний ребер или вершин будем называть растровой разверткой сплошных областей, заполнением многоугольников или заполнением контуров.

Для этого можно использовать несколько методов, которые

обычно делятся на 2 категории:

* растровая развертка
* затравочное заполнение.

В методах растровой развертки пытаются определить в порядке сканирования строк, лежит ли точка внутри многоугольника или контура.

Эти алгоритмы обычно идут от "верха" многоугольника к "низу".

В методах затравочного заполнения предполагается, что известна некоторая точка (затравка) внутри замкнутого контура.

В алгоритмах ищут точки, соседние с затравочной и расположенные внутри контура.

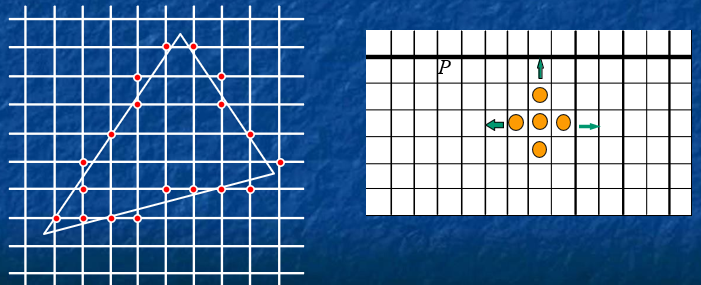
Если соседняя точка расположена не внутри контура, то значит обнаружена граница контура.

Если же точка расположена внутри контура, то она становится затравочной точкой и поиск продолжается рекурсивно.

Подобные алгоритмы применимы только к растровым устройствам.

**Алгоритмы заполнения**

* Сканирование строк
* Метод затравочного заполнения



**Заполнение многоугольника в порядке сканирования строк**

* В порядке сканирующихся строк экранным лучом определить принадлежность точки многоугольнику и вывести с заданными атрибутами
* Повторяя данный процесс для каждой сканирующей строки, пересекающей многоугольник, мы преобразуем в растровую форму весь многоугольник.
* Однако такой процесс неэффективен, поскольку последовательность соседних пикселей обычно принадлежит многоугольнику.
* Можно разработать тест на принадлежность внутренней части, если воспользоваться тем фактом, что соседние пикселы, вероятно, имеют **одинаковые характеристики** (кроме пикселов граничных ребер).

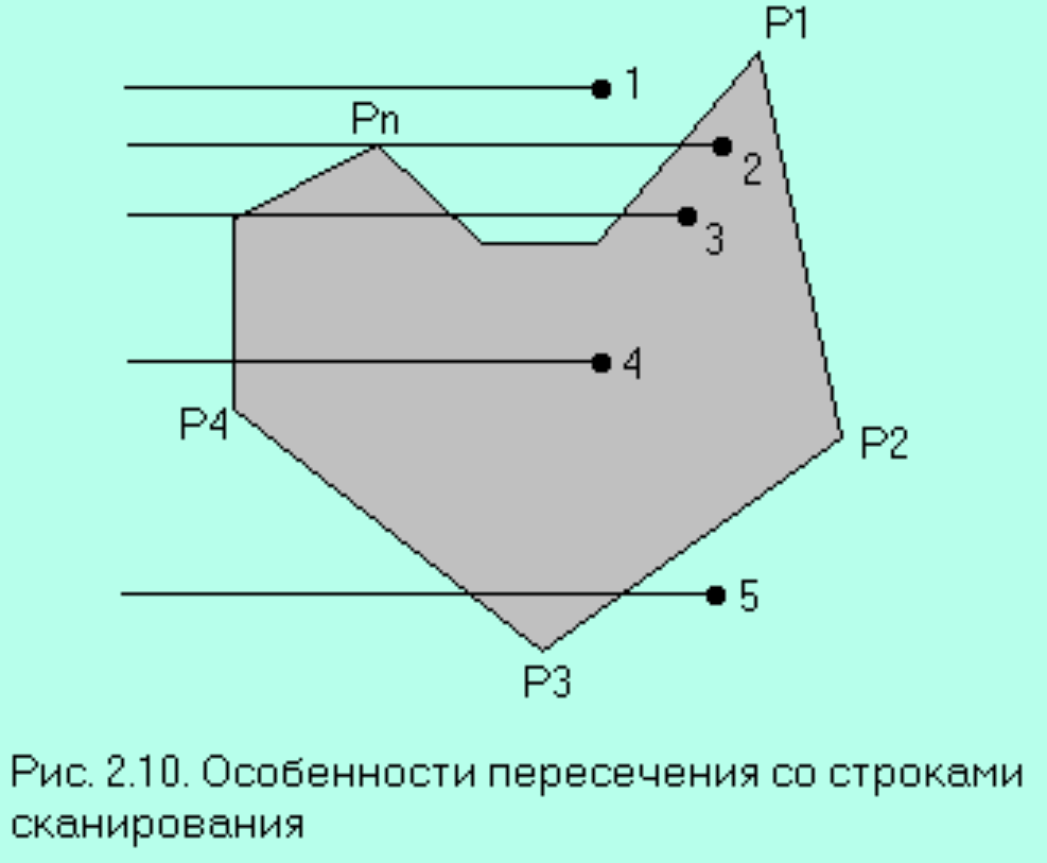
Это свойство называется ***пространственной*** ***когерентностью.***

* Для растровых графических устройств соседние пикселы на сканирующей строке, вероятно, имеют одинаковые характеристики. Это **когерентность растровых строк.**

Характеристики пикселов на данной строке изменяются только там, где ребро многоугольника пересекает строку. Эти пересечения делят сканирующую строку на области.

В результате возникают следующие варианты:

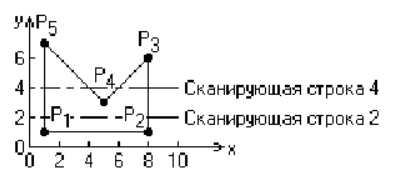
* Нет пересечений с многоугольником ==> точка А внешняя (точка 1).
* Нечетное число пересечений ==> **точка А внутренняя (точки 3 и 4).**
* Четное число пересечений ==> **точка А внешняя (точка 5).**

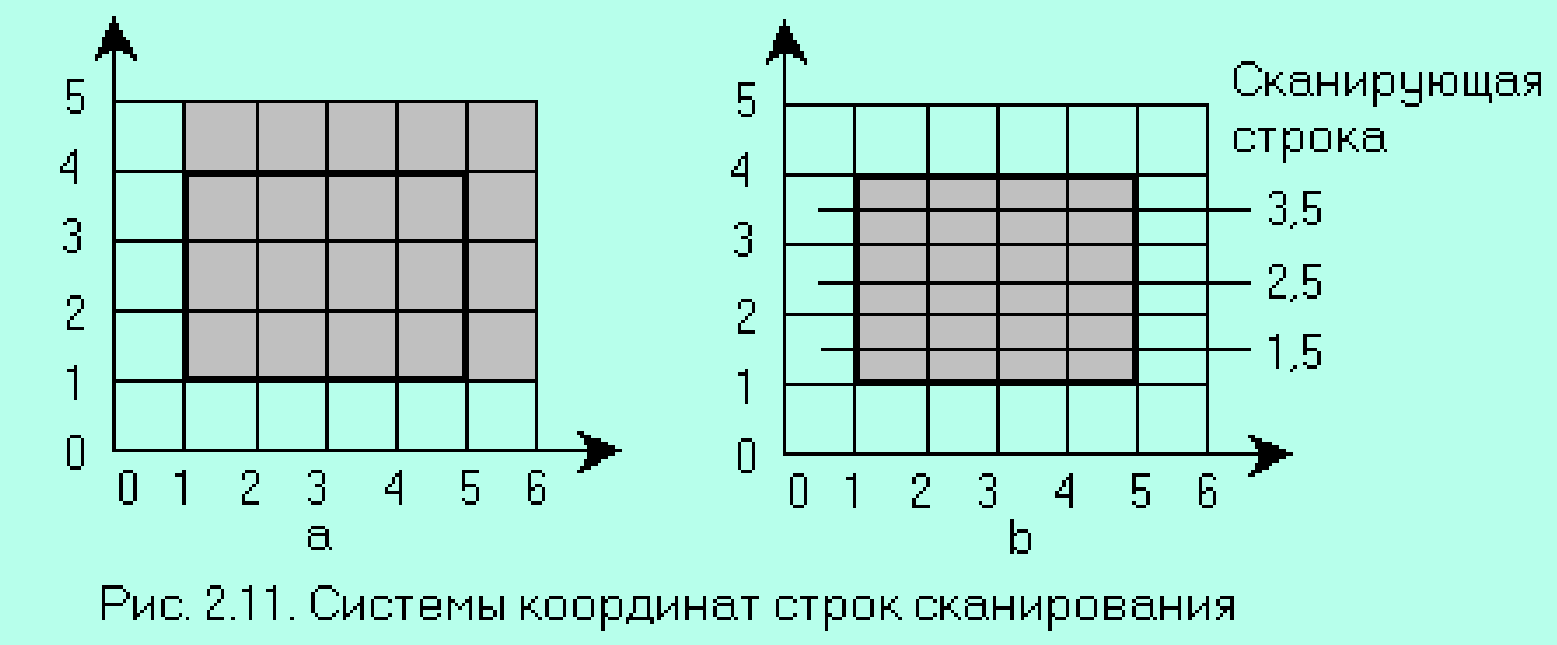


* При определении интенсивности, цвета и оттенка пикселов на сканирующей строке рассматриваются пары отсортированных точек пересечений.
* Для каждого интервала, задаваемого парой пересечений, используется интенсивность или цвет заполняемого многоугольника.

Для интервалов между парами пересечений и крайних (от начала строки до первой

точки пересечения и от последней точки пересе- чения до конца строки) используется фоновая интенсивность или цвет.

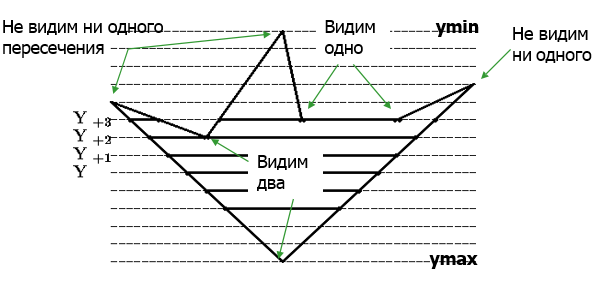




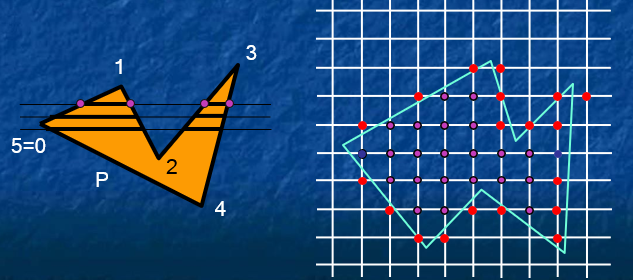
Заметим, что площадь(рис.а), покрываемая активированными пикселами, равна 20, в то время как настоящая площадь (рис.в) прямоугольника равна 12

Данный простой алгоритм работоспособен, но генерирует двух и трехкратное занесение части пикселов. Возможны также ошибки из-за попадания пересечения точно в место соединения двух ребер. Проблему решает следующее правило:

при подсчете пересечений игнорировать пересечения строки развертки с верхней конечной точкой**Р1**ребра, горизо**Р**нт**2**альные ребра также полностью игнорировать

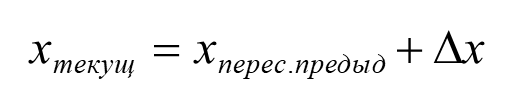


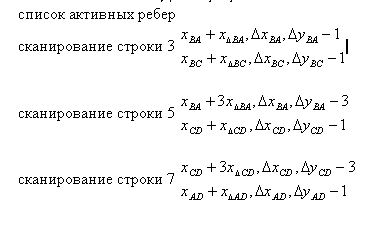
* растровая строка: линия пикселов в изображении
  + набор пикселей в границах многоугольника вдоль горизонтальных линий



**Алгоритм построчного сканирования с групповой** **сортировкой.**

* ***Алгоритм***
* Список активных ребер для каждой сканирующей строки формируется добавлением информации из каждой у группы соответствующей сканирующей строке.
* Координаты х точек пересечения сортируется в порядке сканирования, а ребра из списка активных ребер преобразуются в растровую форму.
* Для каждого отрезка список активных ребер число пересечения со сканирующей строкой уменьшается на единицу. Если отрицательный то отрезок исключается из списка.
* Для каждого отрезка координата х получается добавлением





**Пиксельное определенные области**

* **Определение: Область R - это множество всех пикселей, имеющих цвет C, которые связаны с данным пикселем S**
* **4-связность - соседями точки считаются**

**точки, имеющие с данной точкой одну общую сторону.**

* **4 смежных: пиксели, которые лежат рядом друг с другом**

**по горизонтали или вертикали, НЕ по диагонали**

* **8-связанный: непрерывный путь 8 смежных пикселей, их соединяющих**
* **8-смежный: пиксели, которые лежат рядом друг с другом по горизонтали, вертикали ИЛИ по диагонали**

**Алгоритмы закрашивания.**

**Алгоритмы вывода фигур**

**Фигура (region) -плоский геометрический объект, состоящий из линий контура и точек заполнения, содержащихся внутри контура.**

**Поэтому две задачи:**

1. **нарисовать контур;**
2. **построить точки заполнения.**

**В выводе точек заполнения различают:**

1. **закрашивание (от внутренней точки -**

**затравки) до границы заданного цвета,**

**либо закрашивание пикселов, принадлежащих заданному цветовому диапазону; ~pixel-defined regions**

1. **закрашивание контура, заданного математически ~ symbolic-defined regions.**

**Затравочный пиксел для фигур заданных цветовой границей или цветовым диапазоном, как правило, вносится пользователем.**

**Внутренний пиксел для фигур заданных математически, при необходимости, может быть определен либо на основе решения задачи пересечения фигуры с тестовой горизонталью, либо вычислением суммарного угла при обходе всех**

**вершин фигуры лучем из тестируемой точки (=360 градусов).**

**Алгоритмы заполнения с затравкой**

* В осаждавшихся выше алгоритмах заполнение происходит в порядке

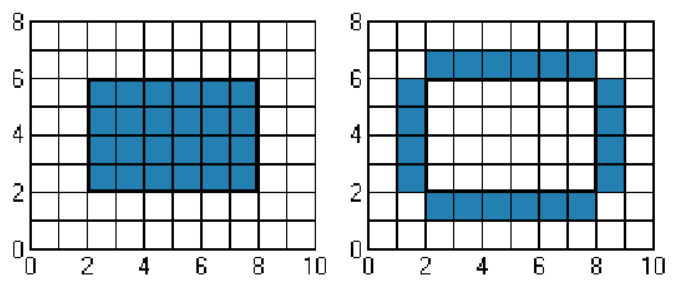
сканирования.

Иной подход используется в алгоритмах заполнения с затравкой.

В них предполагается, что известен хотя бы один пиксел из

внутренней области многоугольника.

Алгоритм пытается найти и закрасить все другие пикселы, принадлежащие внутренней области.



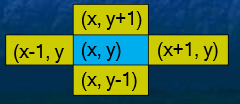
* Области могут быть либо **внутренние**, либо **гранично-определенные.**

**Рекурсивный алгоритм заливки**

* **Начинается с начального пикселя цвета, intColor**
* **Рекурсивно установить 4-х связных соседей в**

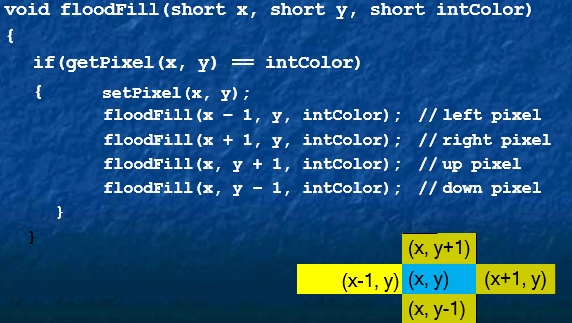
**newColor**

* **Заливка: заполнение области newColor**
* **Основная идея:**
  + **начать с «начального» пикселя (x, y)**
  + **Если (x, y) имеет цвет intColor, заменяем его на newColor**
  + **Выполните рекурсивно для каждого из 4 соседей**



**Рекурсивный алгоритм заливки**

* **Примечание: getPixel (x, y) используется для опроса цвета пикселя в точке (x, y)**



**Рекурсивный алгоритм Flood-Fill**

* **Рекурсивная заливка слепа**
* **Некоторые пиксели проверены несколько раз**
* **Когерентность области - это вероятность того, что внутренний пиксель, скорее всего, соседствует с другим внутренним пикселем**
* **Согласованность может быть использована для улучшения производительности алгоритма**
* **Перспектива: группа смежных пикселей, лежащих на одной линии сканирования**

**Заполнение областей, определенных полигонами**

**Полигон - это упорядоченный список вершин, как показано на следующем рисунке.**

**Для заливки полигонов определенными цветами необходимо определить пиксели, попадающие на границу многоугольника, и пиксели, которые попадают внутрь многоугольника**

Проблема: заданный областью полигон с вершинами Pi = (Xi, Yi), для i = 1… N, определяя последовательность вершин P

**Этот алгоритм выбирает точку внутри объекта и начинает заполняться, пока не достигнет границы объекта.**

**Цвет границы и цвет, который мы заполняем, должны быть**

**разными, чтобы этот алгоритм работал.**

**В этом алгоритме мы предполагаем, что цвет границы одинаков для всего объекта.**

**Алгоритм заполнения границы может быть реализован с помощью 4-х связанных или 8-ми связанных пикселей.**

* **Pseudocode:**

**for(each scan Line L)**

**{**

**Find intersections of L with all edges of P Sort the intersections by increasing x-value**

**Fill pixel runs between all pairs of intersections**

**}**

**4-связанный многоугольник**

В этой технике используются 4 связанных пикселя, как показано на рисунке.

Мы помещаем пиксели выше, ниже, справа и слева от текущих пикселей, и этот процесс будет продол- жаться, пока мы не найдем границу с другим цветом. **Алгоритм**

* **Шаг 1** - Инициализировать значение материала точки
* sееdХ, sееdY, fcolor и dcol.
* **Шаг 2** - Определите граничные значения многоугольника.
* **Шаг 3** - Проверьте, имеет ли текущая начальная точка цвет по умолчанию, затем повторите шаги 4 и 5, пока не будут достигнуты граничные пиксели.
* If getpixel(x, y) = dcol then repeat step 4 and 5

**8-связанный многоугольник**

В этой технике используются 8 связных пикселей как показано на рисунке.

Мы помещаем пиксели

выше, ниже, справа и слева от текущих пикселей как мы делали в 4-х связной технике.

В дополнение к этому мы также помещаем

пиксели в диагонали, чтобы покрыть всю область текущего

пикселя.

Этот процесс будет продолжаться, пока мы не найдем границу с другим цветом.

**Алгоритм**

**Шаг 1 - Инициализировать значение**

**материала точки**

**sееdХ, sееdY, fcolor и dcol.**

**Шаг 2 - Определите граничные значения**

**многоугольника**

* **Шаг 3** - Проверьте, имеет ли текущая начальная точка цвет по умолчанию, затем повторите шаги 4 и 5, пока не будут достигнуты граничные пиксели
* If getpixel(x,y) = dcol then repeat step 4 and 5
* **Шаг 4** - Измените цвет по умолчанию с цветом заливки в начальной точке.
* setPixel(seedx, seedy, fcol)
* **Шаг 5** - Рекурсивно выполните процедуру с четырьмя точками соседства