

Растровые алгоритмы

Компьютерная графика

Лектор: Демяненко Я.М., 4-й курс ФИИТ, 2026

План

Терминология

Заливка – три
алгоритма

Алгоритм
Брезенхема –
растеризация

Растеризация
граней

Термины

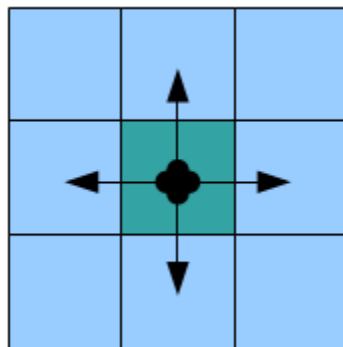
pixels - сокращение
от picture elements

pixel map -
пиксельная карта

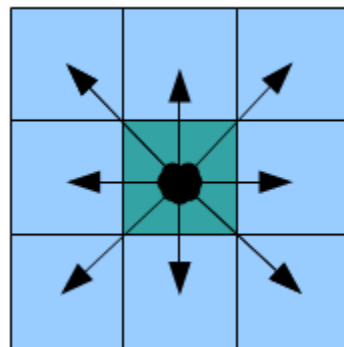
bitmap - побитовое
отображение или
битовая карта

Смежность точек

4-смежность



8-смежность



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

4-смежность сильнее, чем 8-смежность

Связность точек

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

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

Задание областей

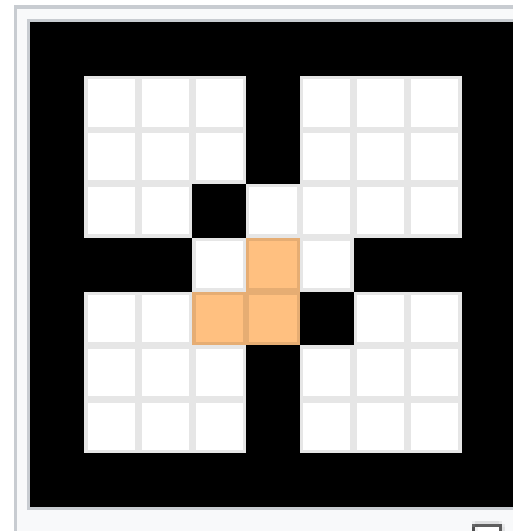
Пиксельно-определенная область
характеризуется текущими цветами в
пиксельной карте

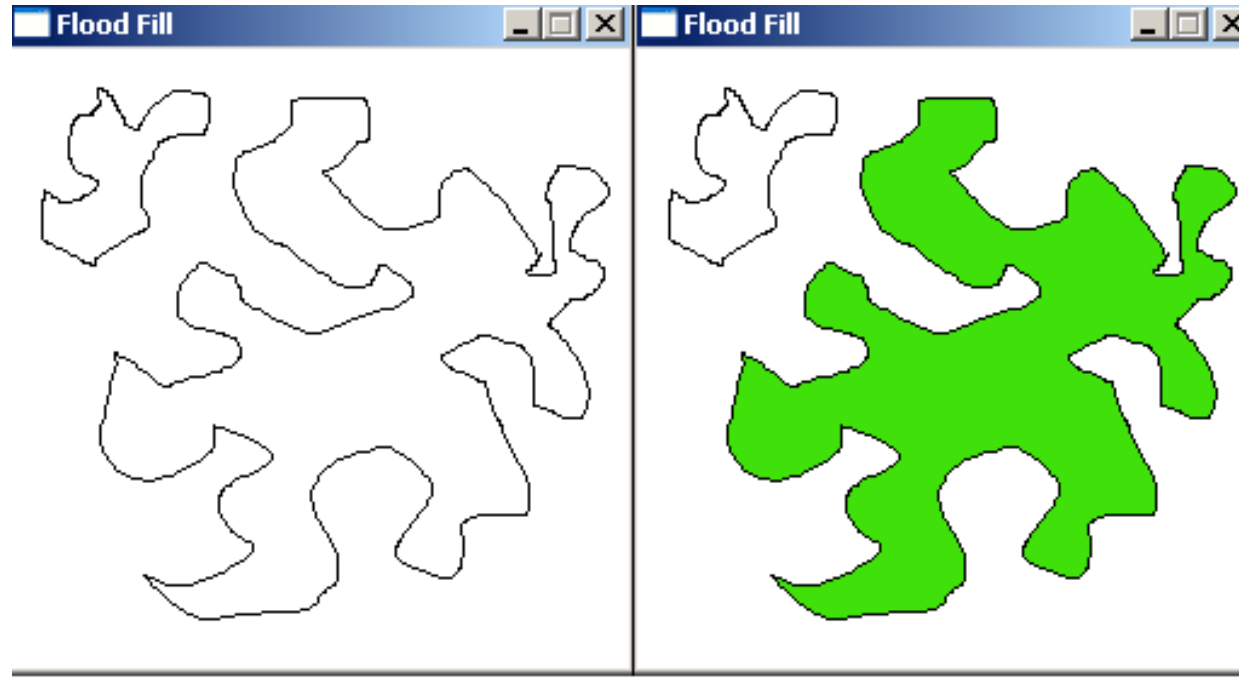
*Различают внутренне-определенную область и
гранично-определенную*

Если граница области 4-х связная, то
внутренность 8-ми связная, и наоборот

Заливка

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



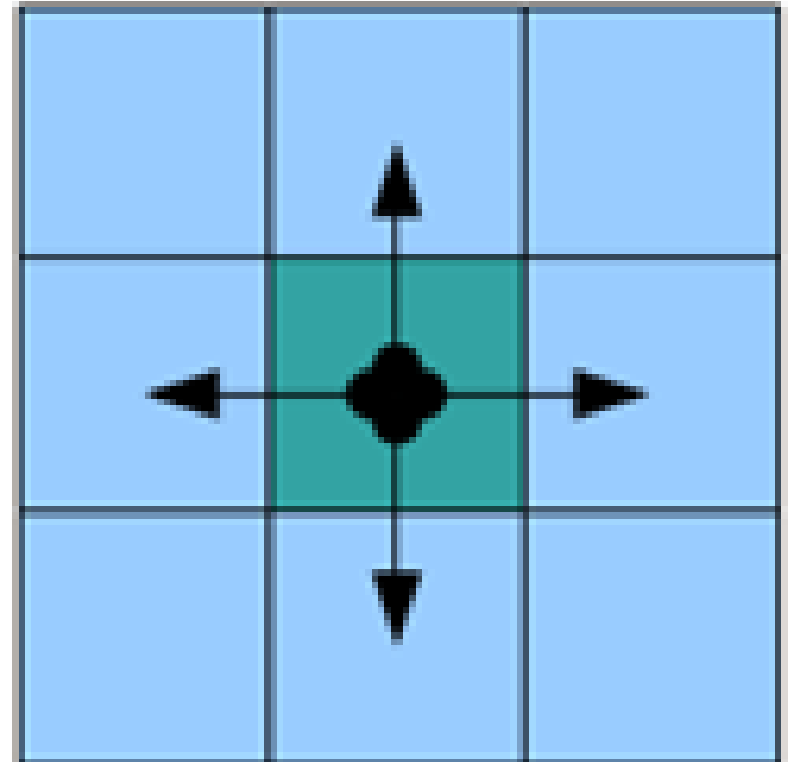


- Граница в ряде мест может обладать толщиной большей, чем один пиксел
- Одни алгоритмы хорошо воспринимают толстые границы, а другие в этих случаях «приходят в замешательство»

Задача заливки

Простой рекурсивный алгоритм заливки

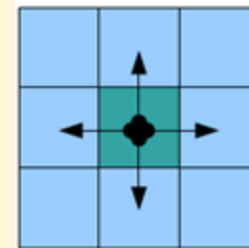
- Выбранная внутренняя точка становится текущей.
- Если текущая точка еще не закрашена и подходит по условию для закрашивания, то
 - закрашиваем ее и
 - применяем рекурсивный вызов данной функции для точек, смежных с текущей



Код простого рекурсивного алгоритма 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);
    }
}
```

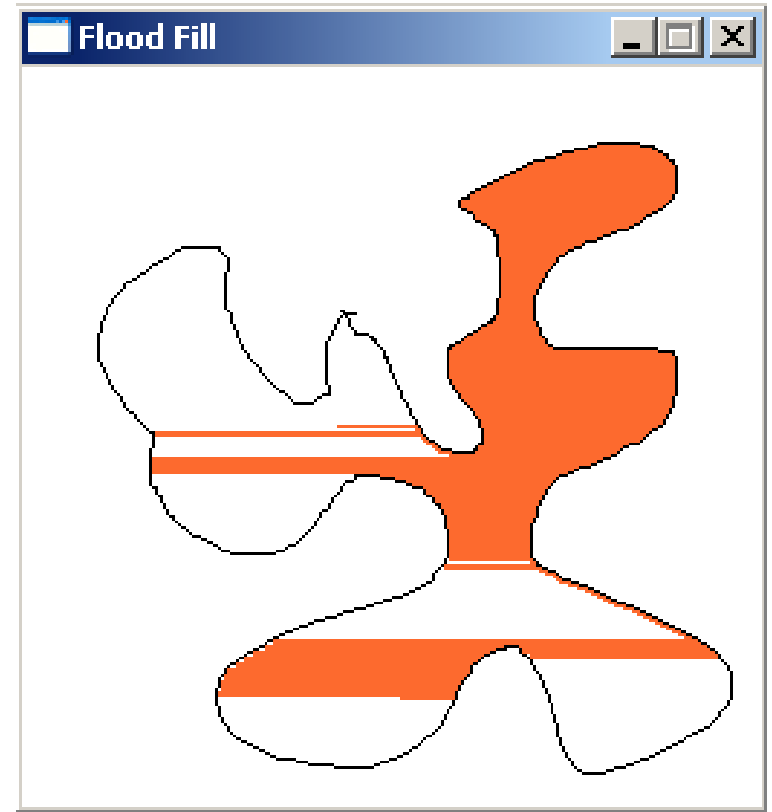


Недостаток

Многие пиксели проверяются многократно, что требует огромного количества вызовов функции

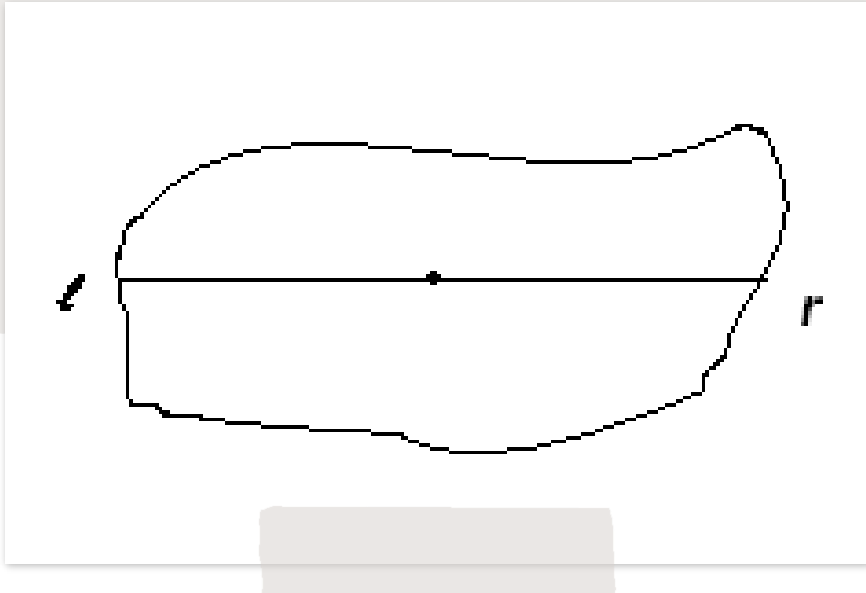
Стек рекурсии может стать очень глубоким, даже для простой области

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 выделением границы (общая идея)

Три стадии:

- **Выделение границы** — обход по часовой стрелке, создание упорядоченного списка граничных пикселей
- **Обследование** внутренней части — сканирование отрезков между парами граничных точек (по y), поиск отверстий
- **Заливка** — соединение точек списка слева направо горизонтальными прямыми

Ключевая идея:

Сначала строим контур,
потом заливаем его построчно

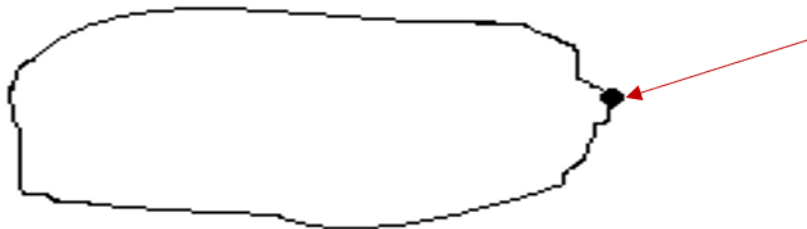
Это позволяет не проверять пиксели
многократно (в отличие от рекурсии)



Детали: обход границы и классификация точек

Обход границы:

- Начальная точка → обход **по часовой стрелке**
- Внутренность области всегда **справа** от направления движения
- Если пиксел не соседствует с внутренней частью — **не считается граничным**



Начальная точка

Выбор точки:

- Первая раз — **вниз (6)**
- Следующая точка — **на 90° по часовой стрелке** от направления, по которому пришли, если не граничная, то далее против часовой стрелки поиск граничной
- В список заносим с упорядоченностью: **по y, потом по x**

Классификация:

- Левая граница** — внутренность справа
- Правая граница** — внутренность слева
- Это нужно для **построчного заполнения** (соединяем левую и правую точки на одном y)

3	2	1	
4	X	0	
5	6	7	

Сравнение алгоритмов

Рекурсивный	С сериями	С границей	
Память	 Стек	 Стек	 Список
Скорость	 Медленно	 Быстро	 Очень быстро
Толстые границы			
Сложность			

Давайте обсудим

1. Почему рекурсивная заливка переполняет стек?
2. В чём главное преимущество заливки с сериями?
3. Почему заливка с границей работает с толстыми границами?
4. Какой алгоритм в вашем редакторе?

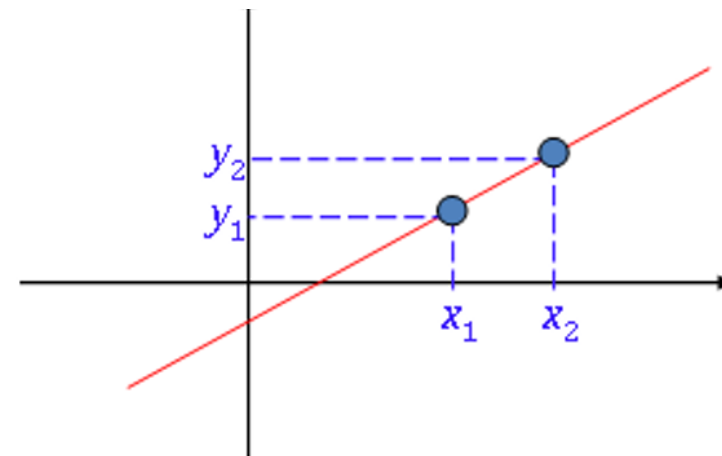
Нет «лучшего» алгоритма — есть подходящий для задачи

Растеризация

- Процесс, обратный векторизации
- Растеризация — это перевод изображения, описанного векторным форматом в пиксели или точки, для вывода на дисплей или принтер
- Растеризация, или метод сканирования строк (англ. scanline rendering) — одна из групп методов рендеринга
- **Растеризация** треугольников — **важнейшая часть рендеринга 3D** объектов на дискретную (с пикселями) плоскость экрана

Растреризация отрезка

- Процесс определения пикселей, наилучшим образом аппроксимирующих заданный отрезок, называется разложением в растр (растреризацией)



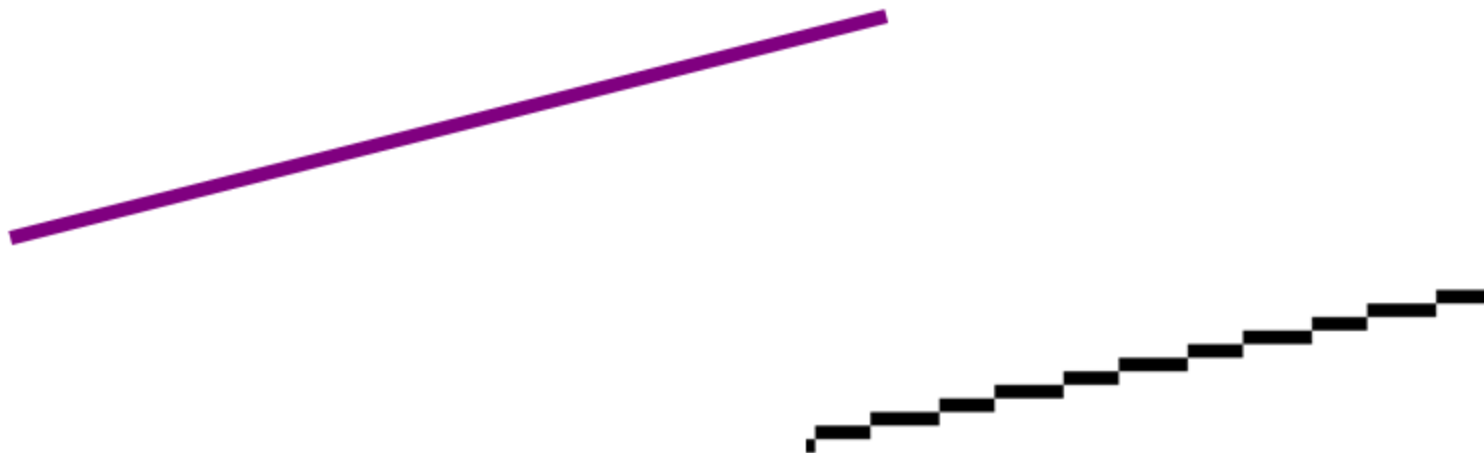
$$y = mx + c$$

Джек Элтон Брезенхэм (Jack Elton Bresenham)

- 1937 г. (88 лет)
- Алгоритм Брезенхема (1962 год)



Алгоритм Брезенхейма построения линий



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

https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

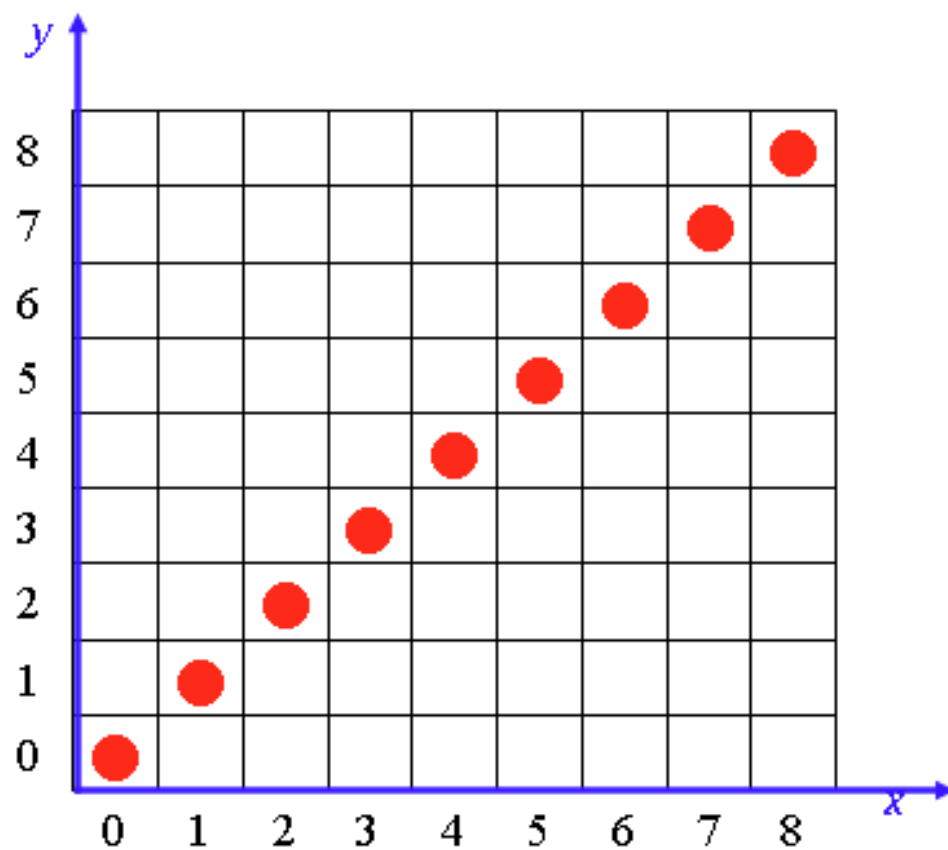
$$|m| = 1$$

$$y = x$$

$$m = 1$$

$$c = 0$$

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



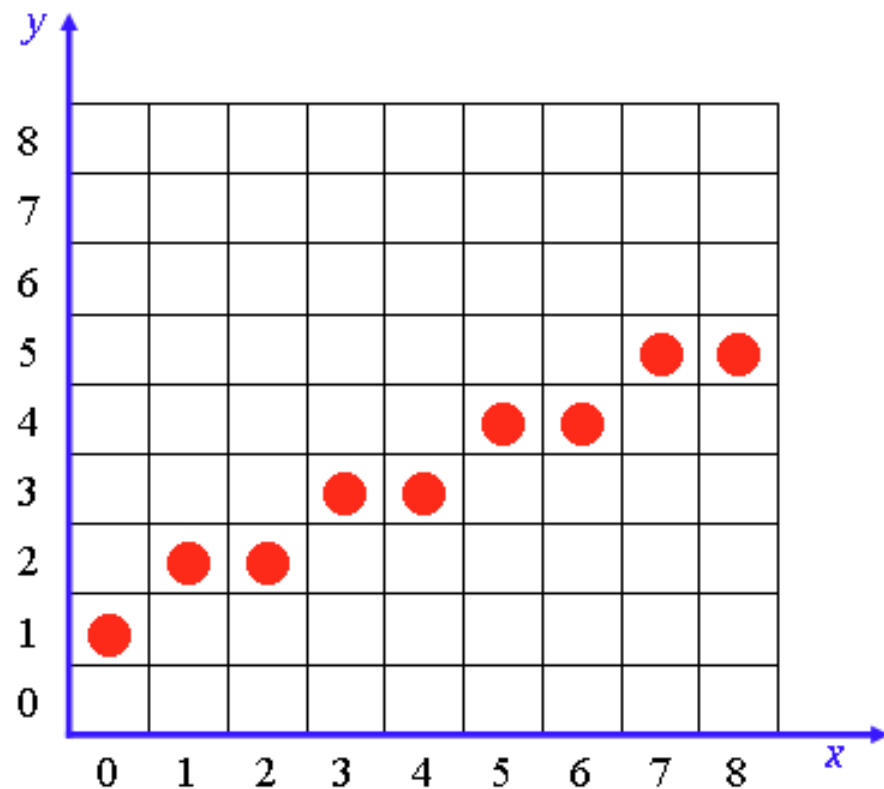
$$|m| < 1$$

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

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

$$c = 1$$

x	y	<u>round(y)</u>
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



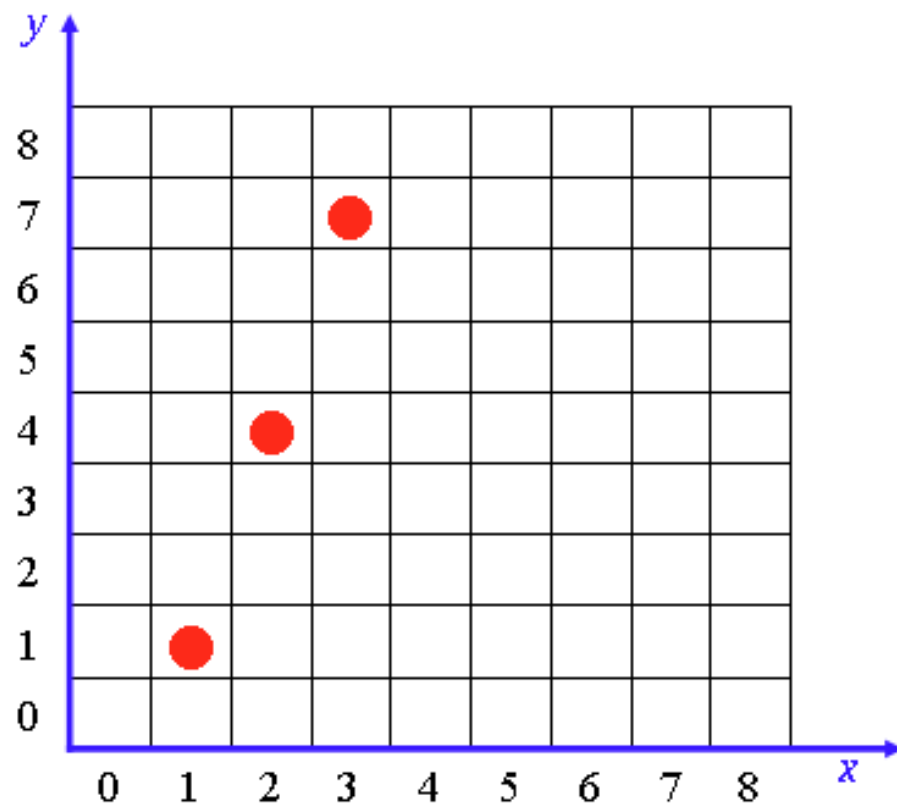
$$|m| > 1$$

$$y = 3x - 2$$

$$m = 3$$

$$c = -2$$

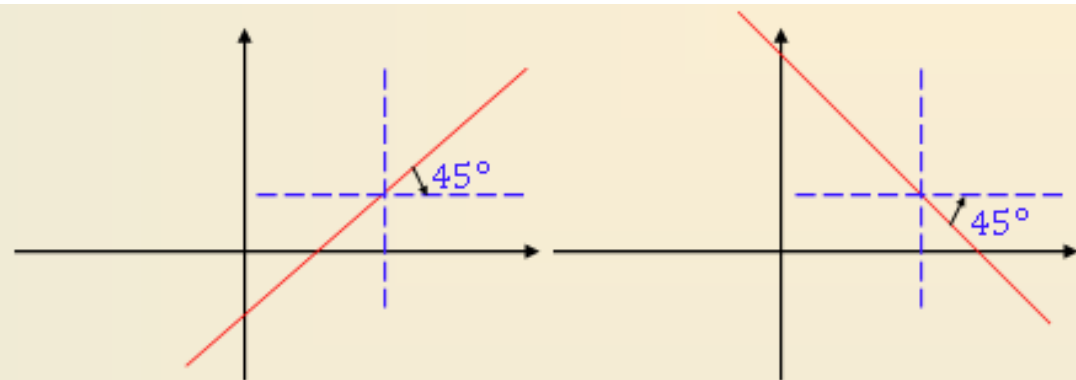
x	y	round(y)
0	-2	-2
1	1	1
2	4	4
3	7	7
4	10	10
5	13	13
6	16	16
7	19	outside 19



Углы наклона

if $|m| = 1$

$$\theta = 45^\circ$$



if $|m| < 1$

$$-45^\circ < \theta < 45^\circ$$

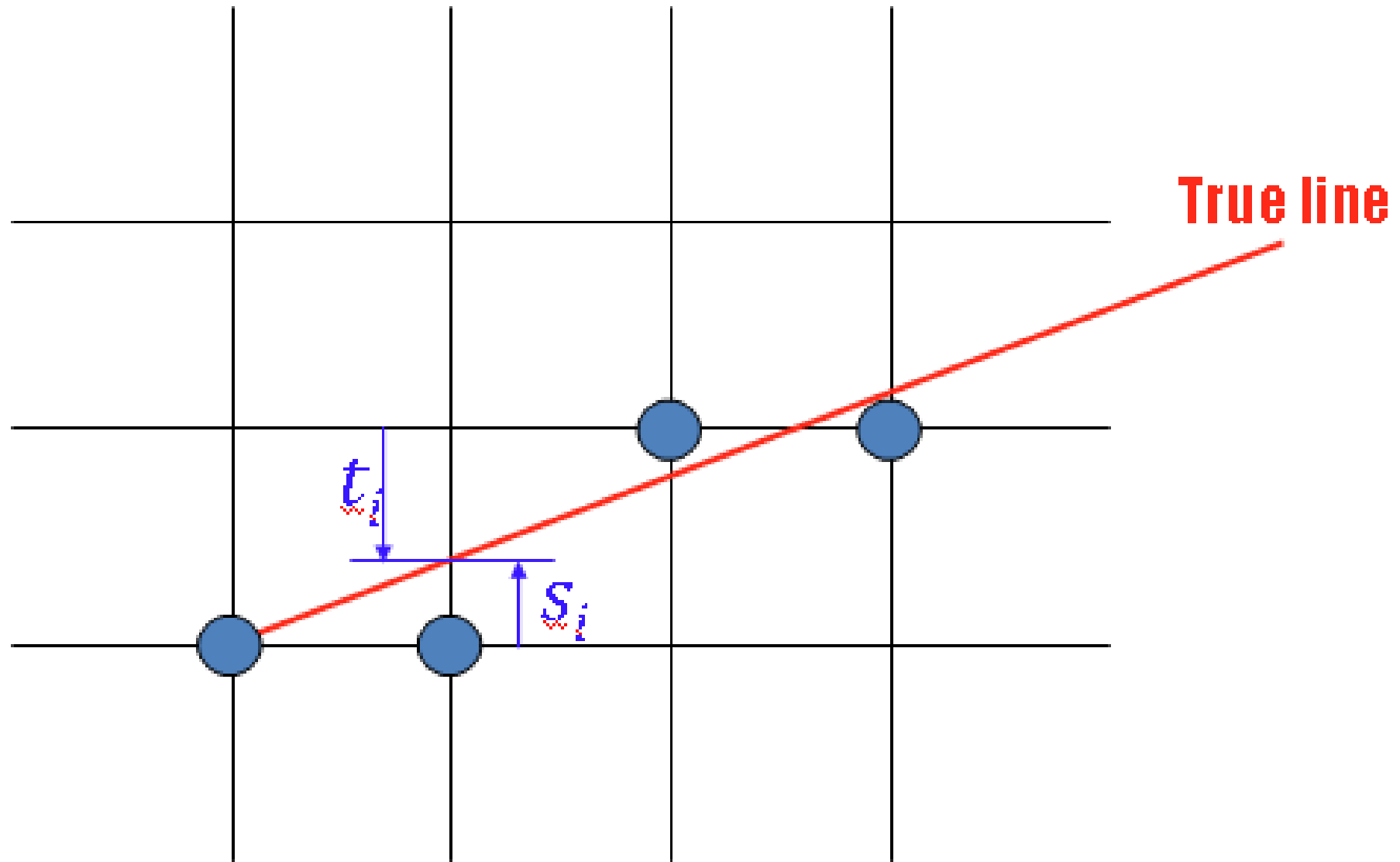


if $|m| > 1$

$$45^\circ < \theta < 90^\circ \text{ or}$$

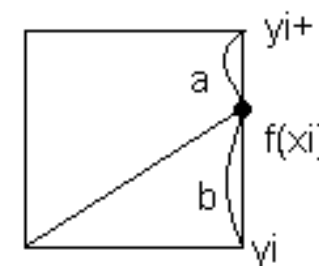
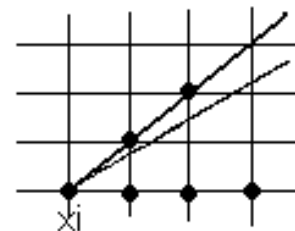
$$-90^\circ < \theta < -45^\circ$$





Вывод формул

$y = (dy/dx) * x$, 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 1

$$a = y_i + 1 - f_i = y_i + 1 - (dy/dx) * x_i$$

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

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

Так как x_1 меньше x_2 , то приращение dx всегда будет >0 .

Следовательно, вместо $(b - a)$ можно использовать $(b - a)dx$.

$$(b - a)dx = 2dyx_i - 2y_idx - dx = 2(dy x_i - y_i dx) - dx$$

Обозначим: $(b - a) dx = d_i$

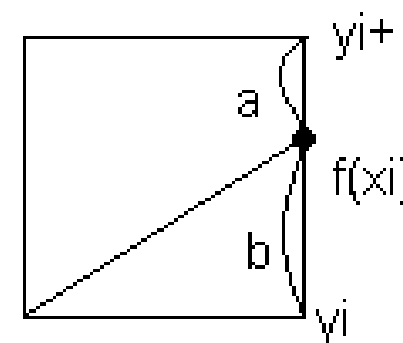
Тогда в итоге получили алгоритм:

if $d_i > 0$ then

$y_{next} = y_i + 1$

else

$y_{next} = y_i$



Алгебраические преобразования 2

$$d_i = 2(dy x_i - y_i dx) - dx$$

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

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

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

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

$$d_1 = 2dy - dx$$

Целочисленный алгоритм Брезенхема

➤ for a line with gradient ≤ 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

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

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

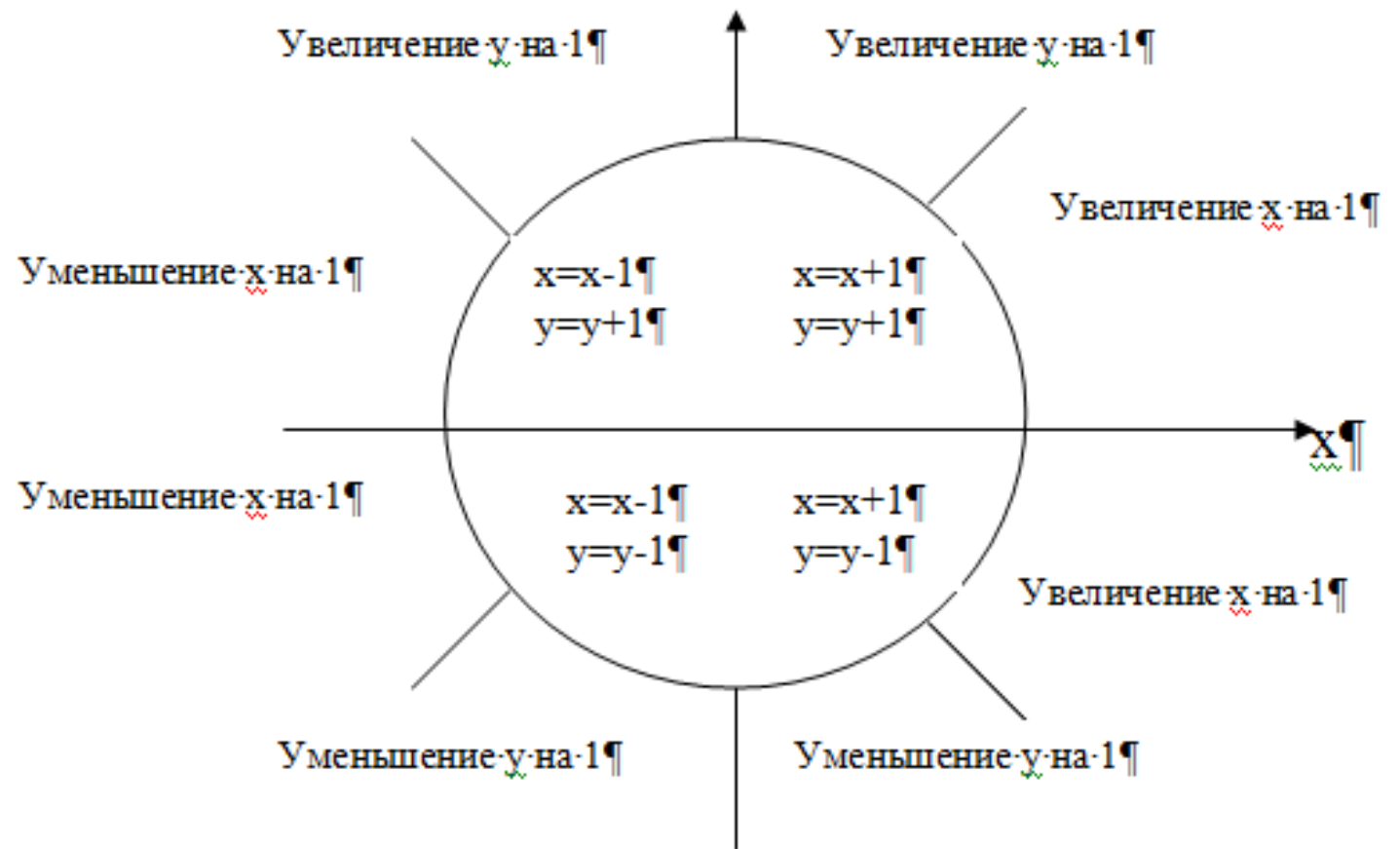
if $d_i \geq 0$ then

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

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

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

Что делать с
углами
наклона?



Брезенхем: целочисленное накопление ошибки с корректировкой

```
plotLine(x0, y0, x1, y1)
```

```
    dx = abs(x1 - x0)
```

```
    sx = x0 < x1 ? 1 : -1
```

```
    dy = -abs(y1 - y0)
```

```
    sy = y0 < y1 ? 1 : -1
```

```
    error = dx + dy
```

```
    while true
```

```
        plot(x0, y0)
```

```
        e2 = 2 * error
```

```
        if e2 >= dy
```

```
            if x0 == x1 break
```

```
            error = error + dy
```

```
            x0 = x0 + sx
```

```
        end if
```

```
        if e2 <= dx
```

```
            if y0 == y1 break
```

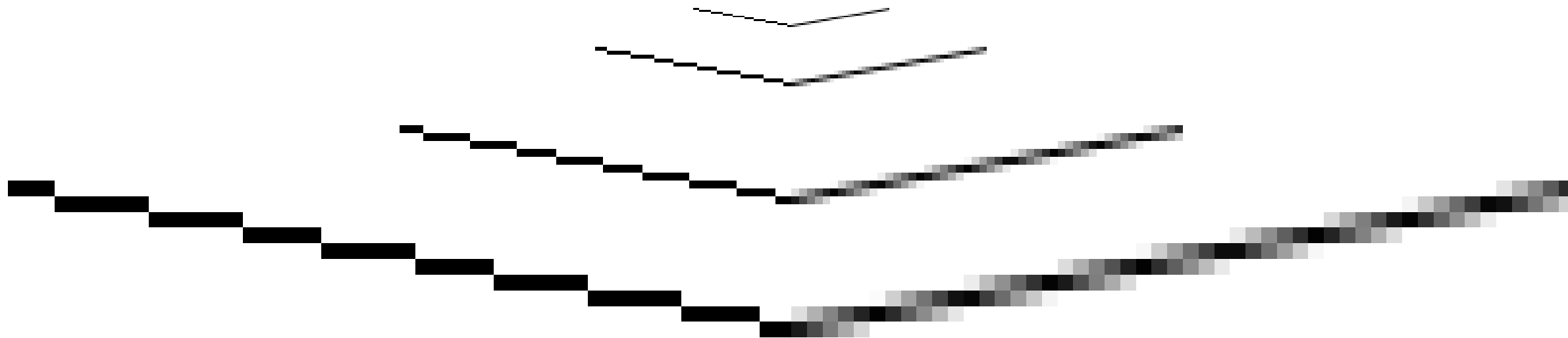
```
            error = error + dx
```

```
            y0 = y0 + sy
```

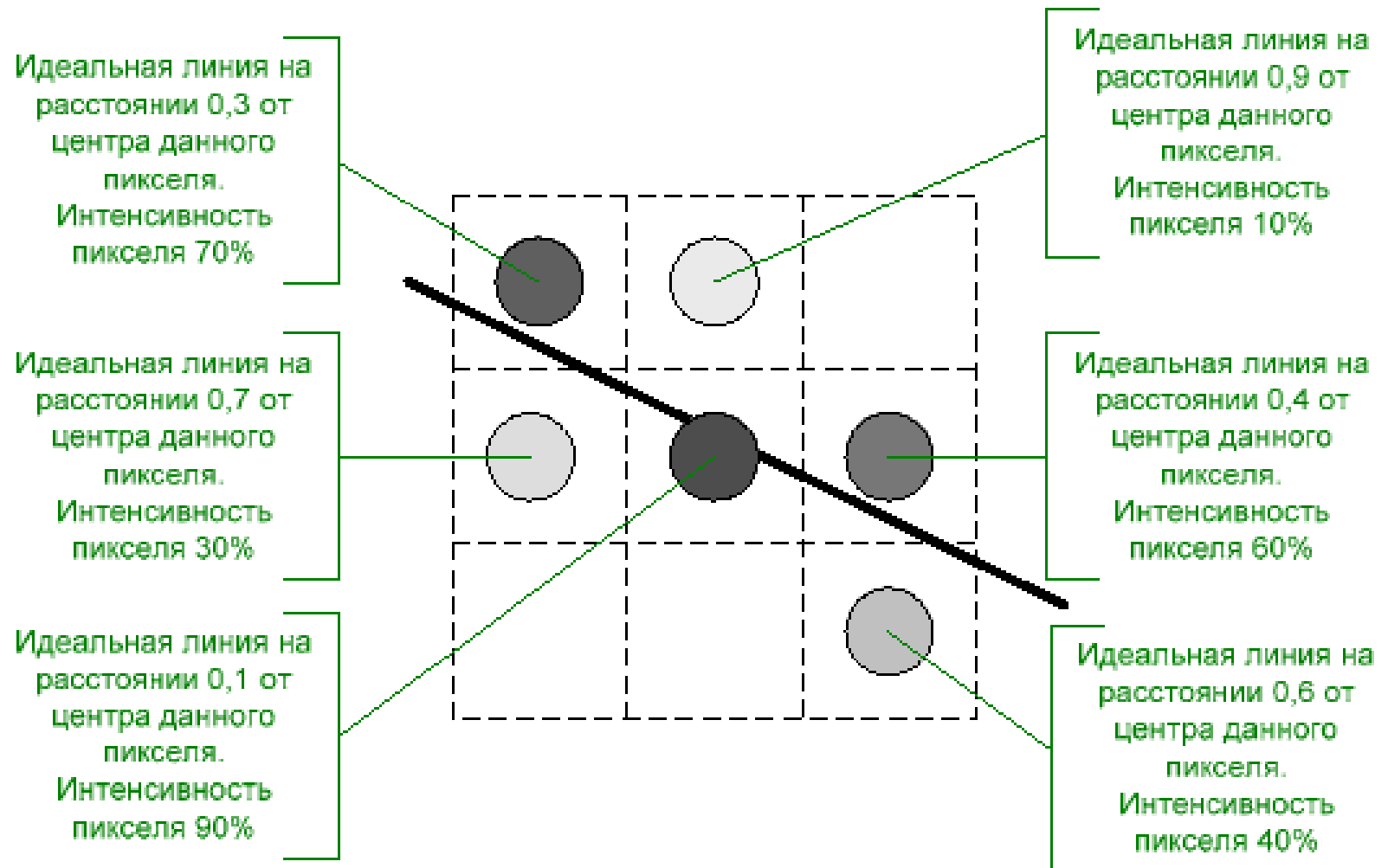
```
        end if
```

```
    end while
```

Сглаживание - Алгоритм Ву (У Сяолиня)



Идея алгоритма Ву

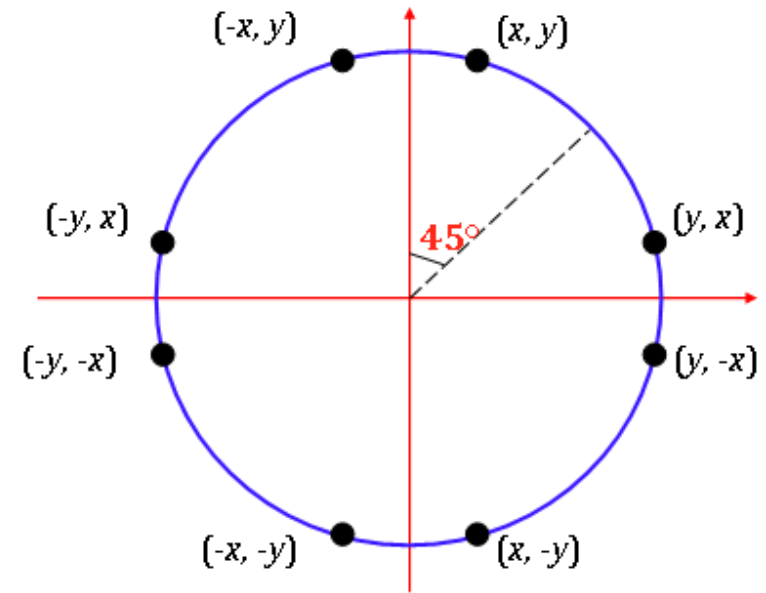


Одна из реализаций алгоритма Ву

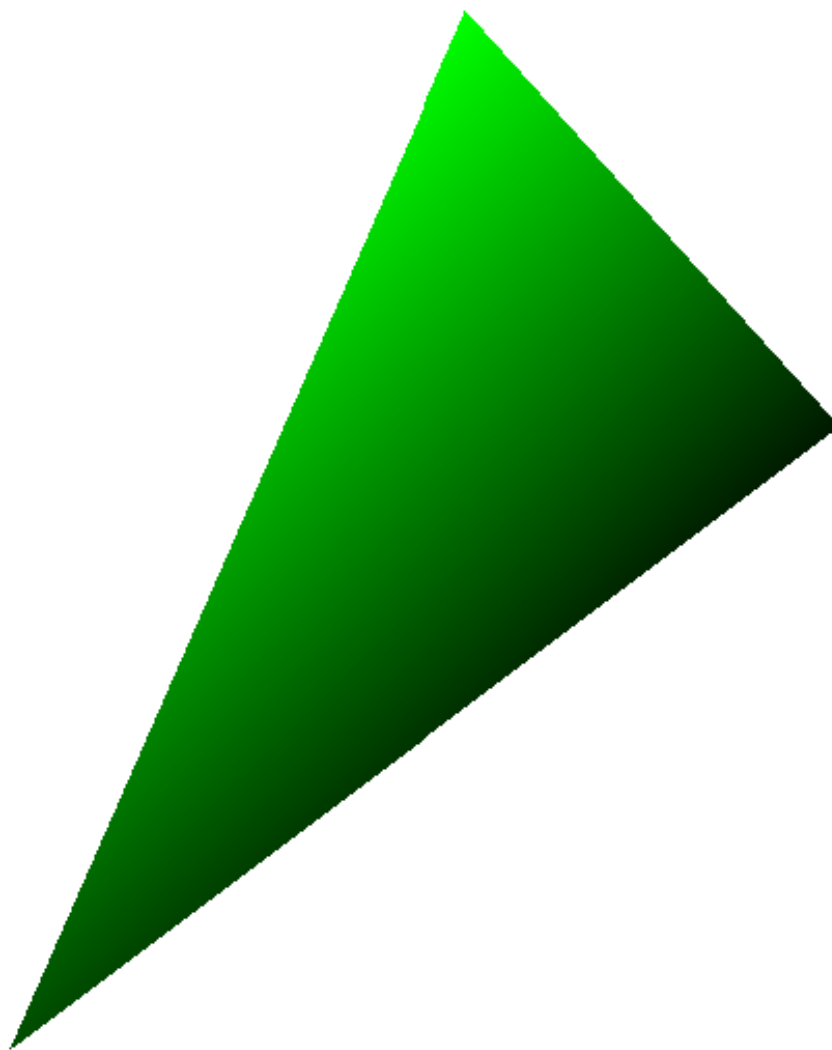
```
DrawPoint(x1, y1, 1);  
// Последний аргумент — интенсивность в долях единицы  
float dx = x1 - x0; float dy = y1 - y0;  
float gradient = dy / dx;  
float y = y0 + gradient;  
for (var x = x0 + 1; x <= x1 - 1; x++) {  
    DrawPoint(x, (int)y, 1 - (y - (int)y));  
    DrawPoint(x, (int)y + 1, y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

Модификация для рисования окружностей

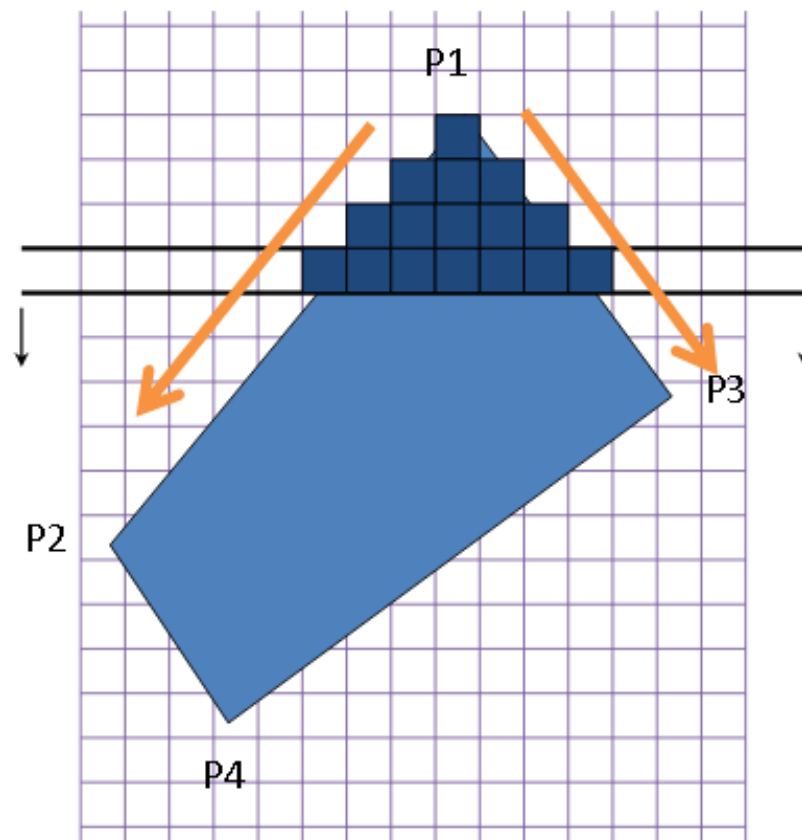
```
void BresenhamCircle(int x0, int y0, int radius) {  
    int x = radius; int y = 0;  
    int radiusError = 1 - x;  
    while (x >= y) {  
        DrawPoint(x + x0, y + y0); DrawPoint(y + x0, x + y0);  
        DrawPoint(-x + x0, y + y0); DrawPoint(-y + x0, x + y0);  
        DrawPoint(-x + x0,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



Градиентная
заливка через
растеризацию



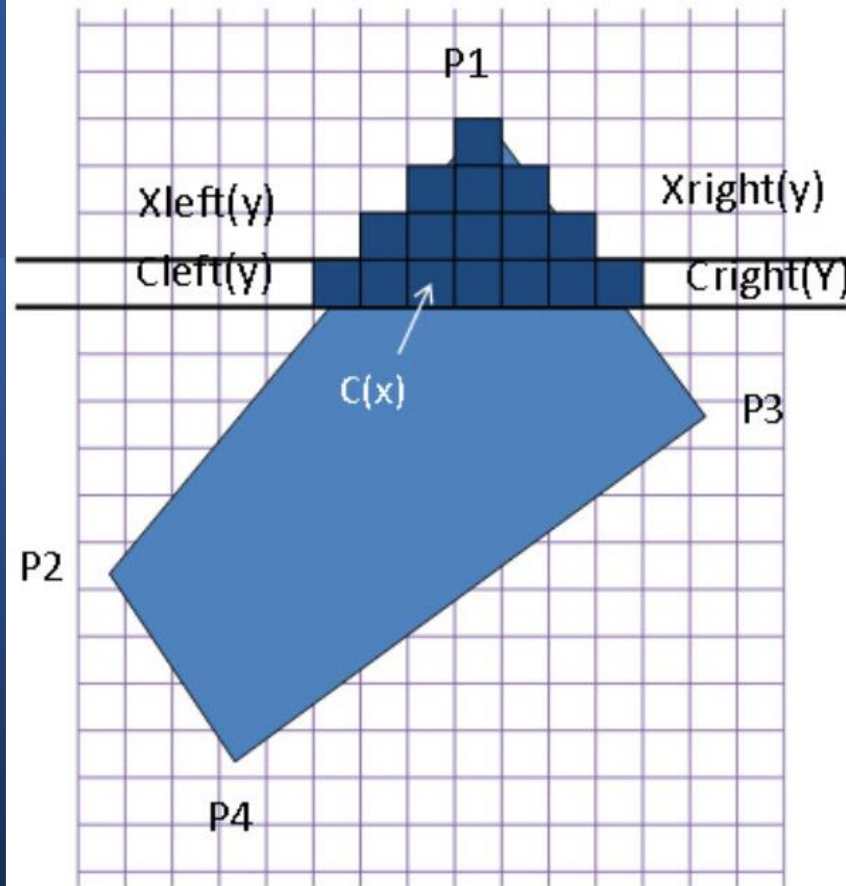
Растреризация граней



Линия
развертки
(scanline)

Алгоритм
построчной
развертки

Процесс вычисления пикселей раstra, принадлежащих примитиву



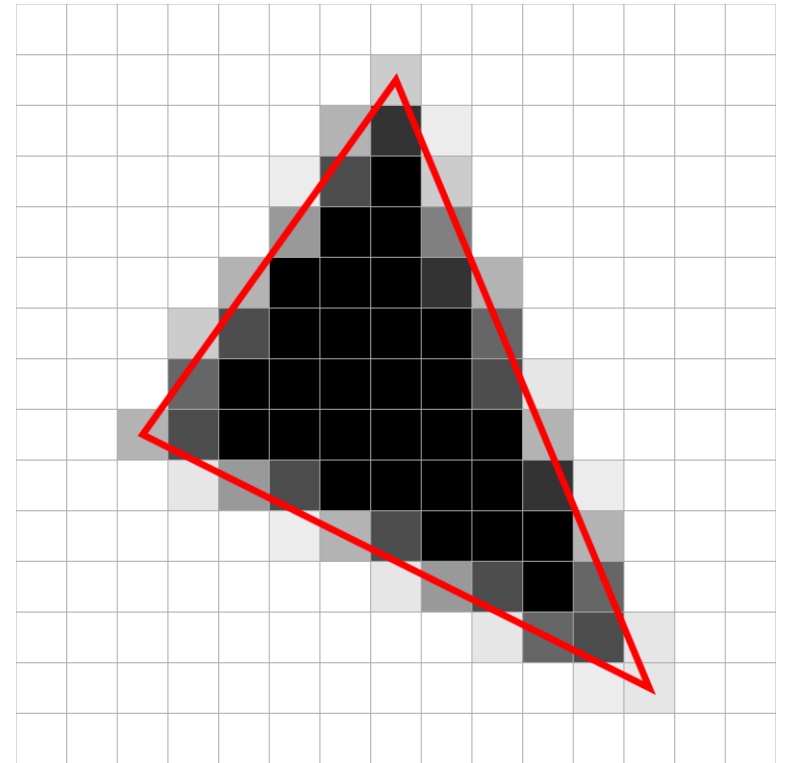
Вход:

- Координаты $P1, P2, P3, P4$ (экранные)
- Цвета вершин C_i или другие атрибуты

1. Строим уравнения отрезков $P1-P2$ и $P1-P3$
2. Для Y от $P2.y$ до $P3.y$
 1. Находим $Xleft(y)$ и $Xright(y)$
 2. Находим $Cleft(y)$ и $Cright(y)$ через линейную интерполяцию
 3. Интерполируем между $Cleft(y)$ и $Cright(y)$ для получения $C(x)$
3. При достижении $P3.y$ $P1-P3$ заменяется на $P3-P4$

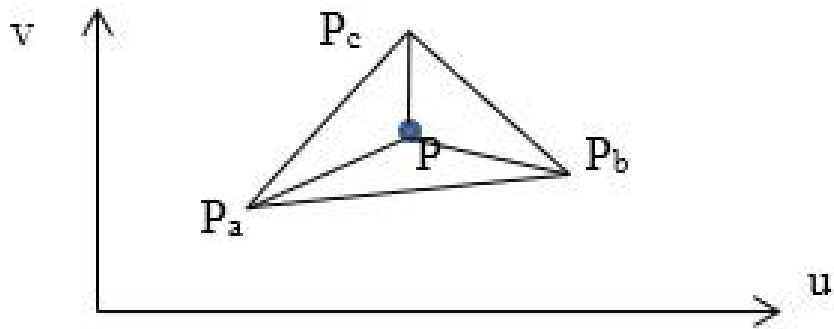
Растеризация треугольника

- А в чем отличие?
- Какие пограничные состояния могут быть?
- Что нужно проверить?
- Что нужно сделать?



Барицентрические координаты

Пусть треугольник, расположенный в параметрическом пространстве, ограничен вершинами $P_a = [u_a, v_a]$, $P_b = [u_b, v_b]$, $P_c = [u_c, v_c]$



Представим точку P , находящуюся внутри треугольника $P = a \cdot P_a + b \cdot P_b + c \cdot P_c$

a, b, c называют барицентрическими координатами

Требование к барицентрическим координатам $a + b + c = 1$

Свойства барицентрических координат

- Точка **P** принадлежит треугольнику $P_aP_bP_c$ — все барицентрические координаты не отрицательны
- Точка **P** совпадает с одной из вершин треугольника $P_aP_bP_c$ — две барицентрические координаты равны нулю
- Точка **P** лежит на стороне треугольника $P_aP_bP_c$ или лежит на ее продолжении — одна из барицентрических координат равна нулю.
- Точка **P** не принадлежит треугольнику $P_aP_bP_c$ — по крайней мере одна барицентрическая координата отрицательная.

Растеризация через барицентрические координаты

1. Находит прямоугольник, ограничивающий наш треугольник.
2. Для каждого пикселя в прямоугольнике вычисляет три барицентрических координаты.
3. Если все координаты неотрицательны, пиксель рисуется.
4. С помощью этих же координат плавно интерполируются атрибуты вершин (цвет, текстурные координаты и т.д.) для получения цвета пикселя.
5. Этот алгоритм лежит в основе многих растеризаторов, в том числе и современных графических процессоров.

Два способа вычисления барицентрических координат

Способ 1. **Через площади** —
самый интуитивно понятный
способ

Способ 2. **Через систему
уравнений** — более
эффективный для вычислений

Способ 1. Вычисление через площади

a, b, c — координаты точки P , которые нам нужно найти

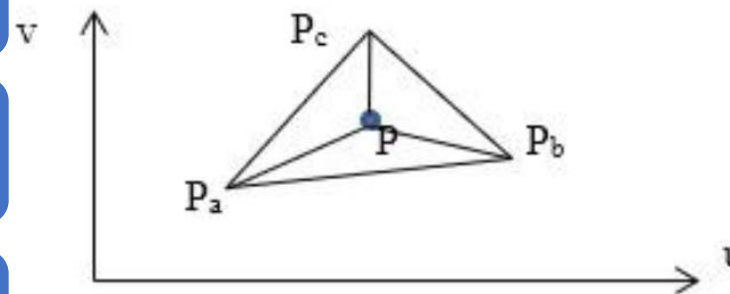
$a = (\text{Площадь подтреугольника } PBC) / (\text{Площадь всего треугольника } ABC)$

$b = (\text{Площадь подтреугольника } PCA) / (\text{Площадь всего треугольника } ABC)$

$c = (\text{Площадь подтреугольника } PAB) / (\text{Площадь всего треугольника } ABC)$

Площадь треугольника можно легко вычислить с помощью векторного произведения

Площадь $ABC = ((B.x - A.x) * (C.y - A.y) - (C.x - A.x) * (B.y - A.y)) / 2$



Способ 2. Через систему уравнений

Решаем систему уравнений для точки $P(x, y)$

$$x = a * A.x + b * B.x + c * C.x$$

$$y = a * A.y + b * B.y + c * C.y$$

$$a + b + c = 1$$

Где в GPU то, что изучали?

Что мы изучали	Где это в GPU
Растеризация отрезка	Аппаратный растеризатор
Барицентрические координаты	Интерполяция атрибутов в шейдере
Z-буфер	Depth Test (ROP-блок)
Сглаживание	MSAA / FXAA / TAA

Зачем это в GPU?

Почему это важно:

Понимание — вы знаете, что происходит под капотом `glDrawArrays()`

Отладка — когда шейдер тормозит, вы понимаете причину (`overdraw`, интерполяторы)

Свой рендерер — для offline / научной визуализации вы знаете, с чего начать

В современном GPU:

Брезенхем не используется напрямую — растеризация аппаратная

Но принцип тот же: дискретизация непрерывного

Барицентрические координаты — основа фрагментных шейдеров

Итог: что запомнить

1. Заливка: три подхода
 - Рекурсивный → просто, но стек переполняется
 - С сериями → быстрее, но всё ещё рекурсия
 - С границей → быстро, работает с толстыми границами
2. Брезенхем
 - Целочисленный алгоритм
 - Без накопления ошибки
 - Основа аппаратной растеризации
3. Сглаживание (B_y)
 - Взвешенная запись пикселей
 - Убирает «лесенку»
4. Растеризация граней
 - Scanline + барицентрические координаты
 - Основа современных GPU
5. Главное
 - Всё это уже в железе — но понимание принципов отличает инженера от пользователя

Проверочные вопросы

Вопрос 1. Почему рекурсивная заливка может переполнить стек?

- A. Слишком много пикселей в области
- B. Глубина рекурсии растёт с размером области
- C. Стек видеокарты ограничен
- D. Алгоритм использует слишком много памяти на один вызов

Вопрос 2. В чём главное преимущество алгоритма Брезенхема?

- A. Работает только с целыми числами
- B. Не требует вычисления $y = mx + c$
- C. Даёт сглаженные линии
- D. Работает быстрее рекурсии

Вопрос 3. Что такое барицентрические координаты?

- A. Координаты центра треугольника
- B. Веса вершин для точки внутри треугольника
- C. Координаты для текстурирования
- D. Способ хранения цвета

Проверочные вопросы

Вопрос 4. Что делает алгоритм Vu?

- A. Ускоряет растеризацию
- B. Сглаживает линии через взвешенную запись
- C. Убирает невидимые линии
- D. Заливает область

Вопрос 5. Где в современном GPU используются принципы растеризации?

- A. Только в старых видеокартах
- B. В аппаратном растеризаторе
- C. Только в offline-рендеринге
- D. Не используются

Вопрос 6 (открытый). Почему важно понимать алгоритмы растеризации, если GPU делает всё сам?