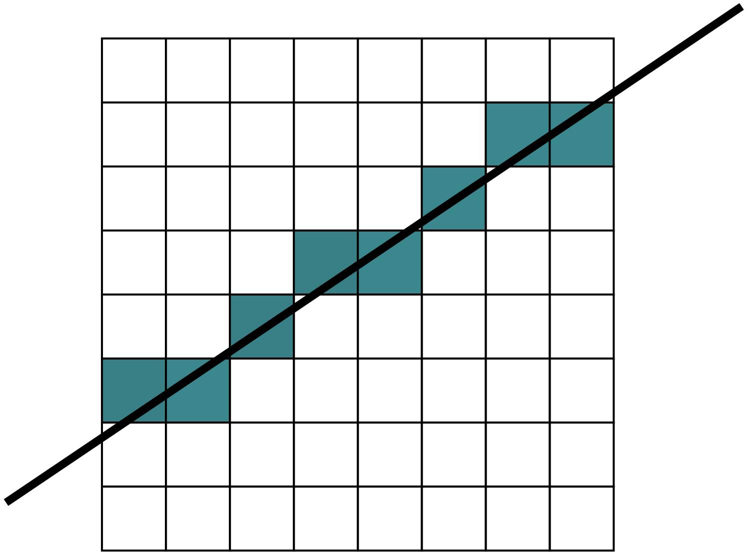
Introdução à Computação Gráfica
Rasterização

Claudio Esperança
Paulo Roma Cavalcanti

Representação Vetorial x Matricial

- Normalmente, gráficos são definidos através de primitivas geométricas como pontos, segmentos de retas, polígonos, etc
  - Representação vetorial
- Dispositivos gráficos podem ser pensados como matrizes de pixels (*rasters*)
  - Representação matricial
- Rasterização é o processo de conversão entre representações vetorial e matricial

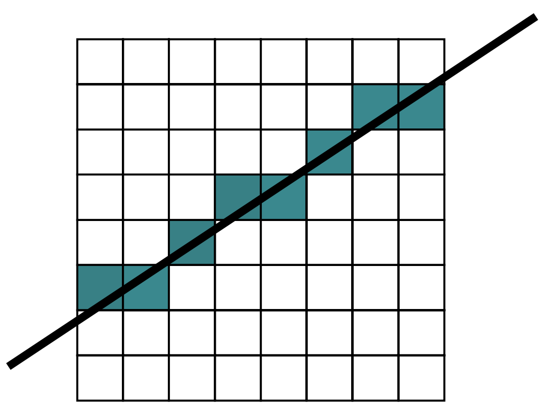


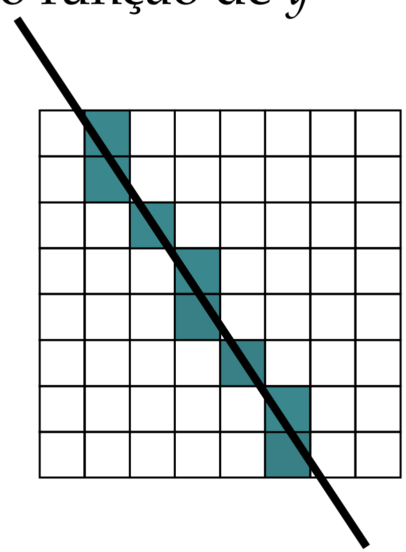
Considerações Gerais

- Rasterização é um processo de amostragem
  - Domínio contínuo → discreto
  - Problemas de *aliasing* são esperados
- Cada primitiva pode gerar um grande número de pixels
  - Rapidez é essencial
- Em geral, rasterização é feita por hardware
- Técnicas de *antialiasing* podem ser empregadas, usualmente extraindo um custo em termos de desempenho

Rasterização de Segmentos de Reta

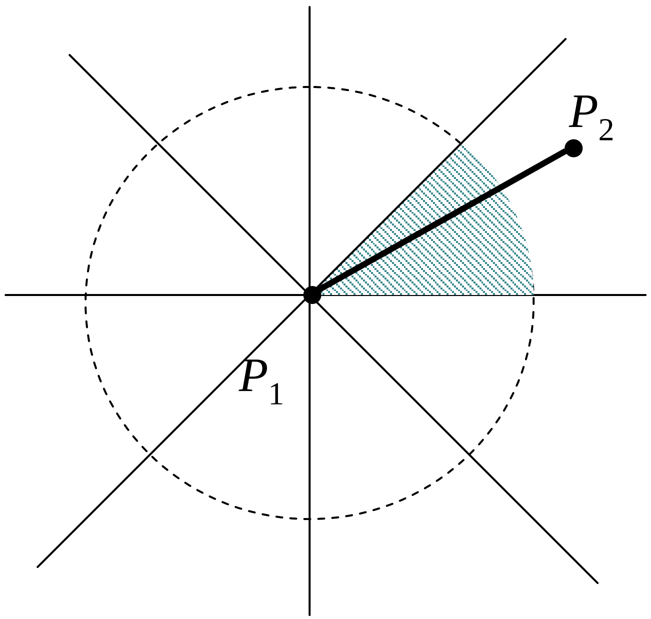
- Segmento de reta entre $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$
  - Já foi recortado com relação ao *viewport*
- Objetivo é pintar os pixels atravessados pelo segmento de reta
  - Na verdade, nem todos, apenas os mais próximos
- Reta de suporte dada por $a\,x + b\,y + c = 0$
- Queremos distinguir os casos
  - Linhas ~ horizontais → computar y como função de x
  - Linhas ~ verticais → computar x como função de y





Algoritmo Simples

- Assumimos segmentos de reta no primeiro octante, com
  - Demais casos resolvidos de forma simétrica
- Inclinação (entre 0 e 1) dada por $m = (y_2 - y_1) \;/\; (x_2 - x_1)$
- Algoritmo:
  - Para x desde x_1 até x_2 fazer:
    - $y \leftarrow y_1 + \lfloor m * (x - x_1) + 0.5 \rfloor$
    - Pintar pixel (x, y)



Algoritmo Incremental

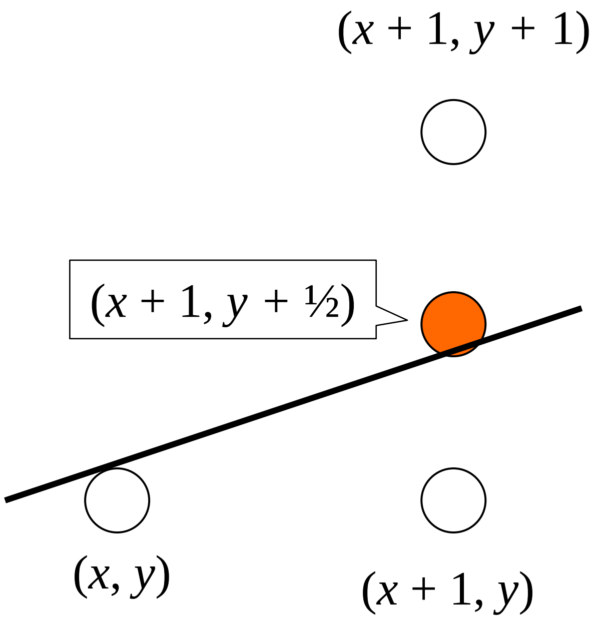
- Algoritmo simples tem vários problemas:
  - Utiliza aritmética de ponto-flutuante
  - Sujeito a erros de arredondamento
  - Usa multiplicação
  - *Lento*
- Se observarmos que m é a variação em y para um incremento unitário de x, podemos fazer ligeiramente melhor:

$x \leftarrow x_1; \quad y \leftarrow y_1$
Enquanto $x \leq x_2$ fazer:
    $x \leftarrow x + 1$
    $y \leftarrow y + m$
    Pintar pixel $(x, \lfloor y + 0.5 \rfloor)$

- Ainda usa ponto-flutuante

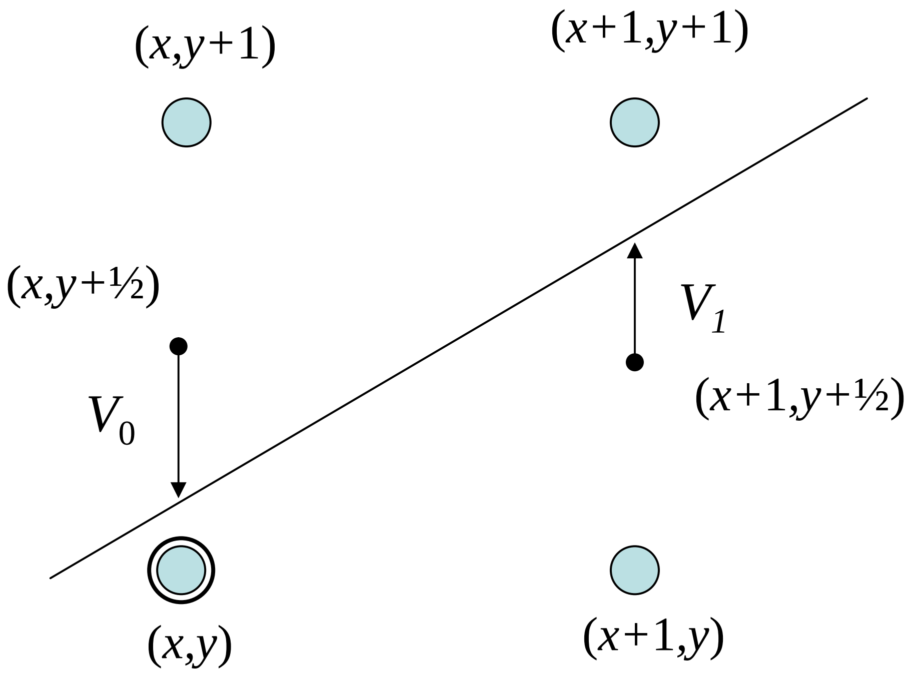
Algoritmo de Bresenham

- Algoritmo clássico da computação gráfica
- Algoritmo incremental que utiliza apenas soma e subtração de inteiros
- Idéia básica:
  - Em vez de computar o valor do próximo y em ponto flutuante, decidir se o próximo pixel vai ter coordenadas $(x + 1, y)$ ou $(x + 1, y + 1)$
  - Decisão requer que se avalie se a linha passa acima ou abaixo do ponto médio $(x + 1, y + ½)$



Algoritmo de Bresenham

- Variável de decisão *V* é dada pela classificação do ponto médio com relação ao semi-espaço definido pela reta
- Caso 1: Linha passou abaixo do ponto médio



$$ax + by + c = V$$

$$\text{onde} \begin{cases} V = 0 \rightarrow (x, y) \text{ sobre a reta} \\ V < 0 \rightarrow (x, y) \text{ abaixo da reta} \\ V > 0 \rightarrow (x, y) \text{ acima da reta} \end{cases}$$

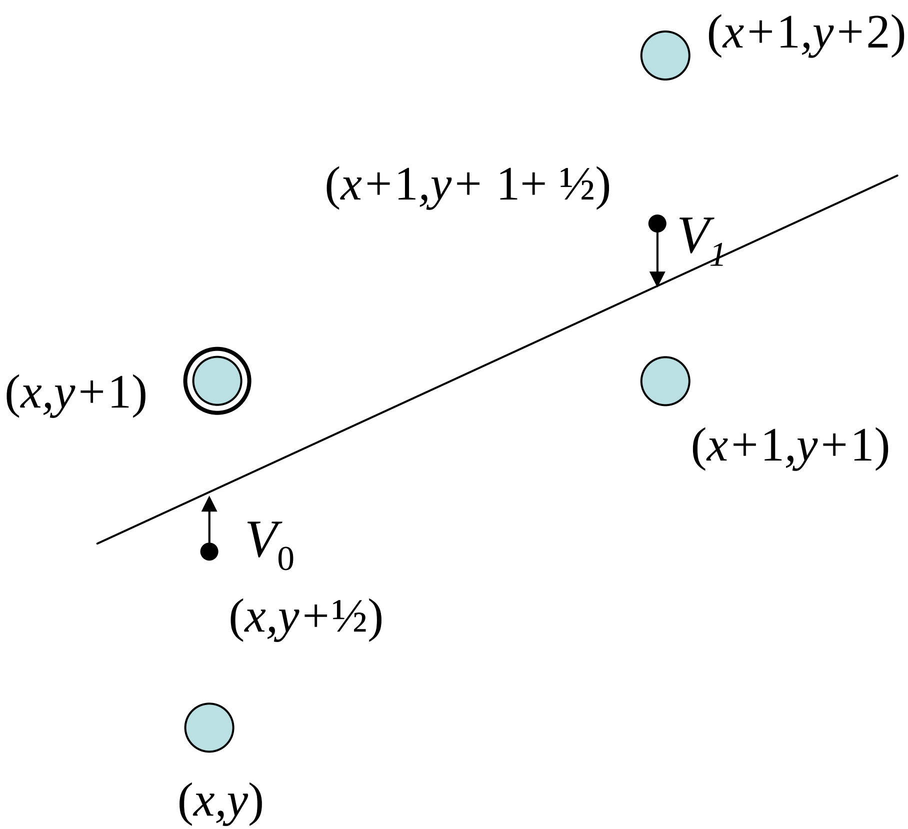
$$V_1 = a(x+1) + b(y + \tfrac{1}{2}) + c$$

$$V_0 = ax + b(y + \tfrac{1}{2}) + c$$

$$\therefore V_1 = V_0 + a$$

Algoritmo de Bresenham

- Caso 2: Linha passou acima do ponto médio



$$V_1 = a(x+1) + b(y+1+\tfrac{1}{2}) + c$$

$$V_0 = ax + b(y+\tfrac{1}{2}) + c$$

$$\therefore V_1 = V_0 + a + b$$

Algoritmo de Bresenham

- Coeficientes da reta
  - $a = y_2 - y_1$
  - $b = x_1 - x_2$
  - $c = x_2 \, y_1 - x_1 \, y_2$
- Para iniciar o algoritmo, precisamos saber o valor de V em $(x_1 + 1, y_1 + ½)$
  - $V = a\,(x_1 + 1) + b\,(y_1 + ½) + c$
    $= \underbrace{a\,x_1 + b\,y_1 + c}_{0} + a + b/2 = a + b/2$
- Podemos evitar a divisão por 2 multiplicando a, b e c por 2 (não altera a equação da reta)

Algoritmo de Bresenham - Resumo

$a \leftarrow y_2 - y_1$

$b \leftarrow x_1 - x_2$

$V \leftarrow 2 * a + b$

$x \leftarrow x_1$

$y \leftarrow y_1$

Enquanto $x \leq x_2$ fazer:

  Pintar pixel (x, y)

  $x \leftarrow x + 1$

  Se $V \leq 0$

    Então $V \leftarrow V + 2 * a$

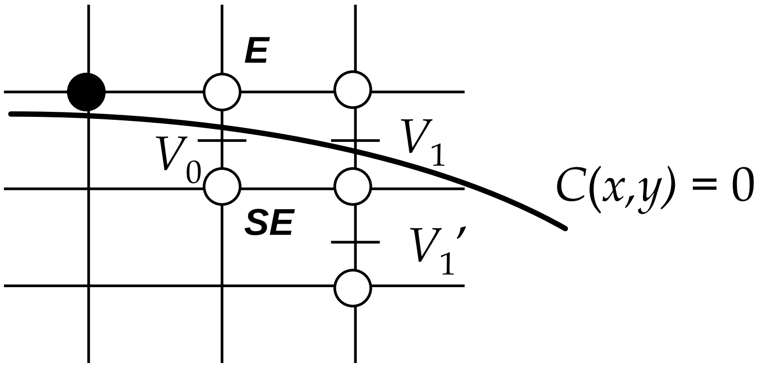
    Senão $V \leftarrow V + 2 * (a + b)$; $y \leftarrow y + 1$

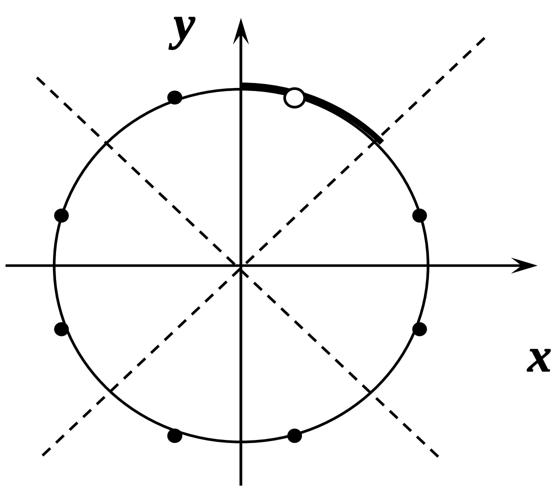
Extensão para demais Octantes

- Se $x_2 < x_1$
  - Trocar P_1 com P_2
- Se $y_2 < y_1$
  - $y_1 \leftarrow -y_1$
  - $y_2 \leftarrow -y_2$
  - Pintar pixel $(x, -y)$
- Se $|y_2 - y_1| > |x_2 - x_1|$
  - Repetir o algoritmo trocando "y" com "x"

Rasterização de Círculos

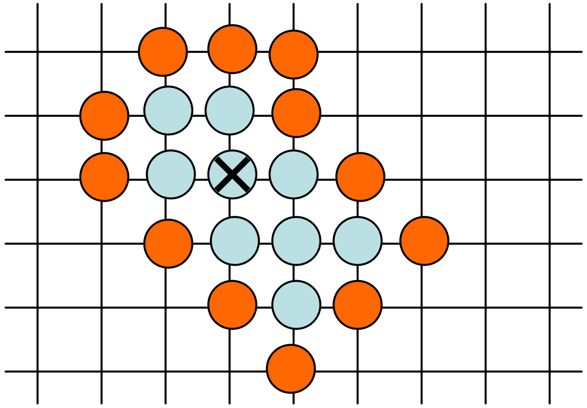
- Mesma idéia de avaliar incrementalmente uma função que classifica o ponto médio entre um pixel e outro com relação a uma função implícita
- Apenas um octante precisa ser avaliado, os demais são simétricos
  - Para cada pixel computado, oito são pintados
- Derivação um pouco mais difícil que a da reta
- Outras cônicas podem também ser rasterizadas de forma semelhante

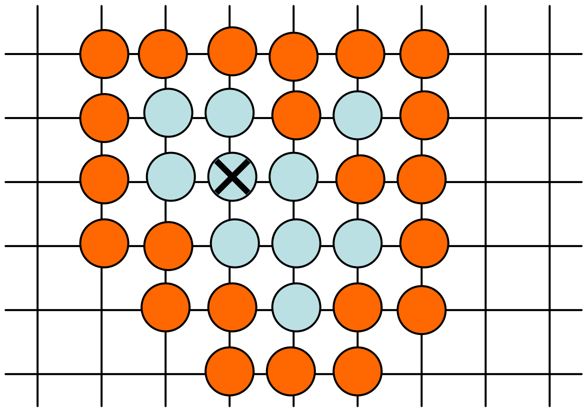




Preenchimento de Regiões

- Não é propriamente rasterização uma vez que operação se dá no espaço da imagem
- Regiões são definidas por critérios de vizinhança a um pixel dado (semente)
  - 4-conexa (borda 8-conexa)
  - 8-conexa (borda 4-conexa)
- Exemplo:
  - Pixels com cor semelhante à semente
    - Borda tem cor diferente
  - Pixels com cor diferente de uma cor dada
    - Borda tem cor igual à cor dada





Algoritmo de Preenchimento

- Conhecido como *"Flood Fill"*
- Algoritmo recursivo
  - Preenche vizinhos da semente que atendem ao critério
  - Aplica o algoritmo recursivamente tomando esses vizinhos como sementes
  - Termina quando nenhum vizinho atende o critério

Algoritmo de Preenchimento

- Pseudo-código:

  Procedure *FloodFill* (x, y, *cor*, *novaCor*)

  Se *pixel* (x, y) = *cor* então

  pixel (x, y) $\leftarrow$ *novaCor*

  FloodFill ($x + 1$, y, *cor* , *novaCor*)

  FloodFill (x, $y + 1$, *cor* , *novaCor*)

  FloodFill ($x - 1$, y, *cor* , *novaCor*)

  FloodFill (x, $y - 1$, *cor* , *novaCor*)

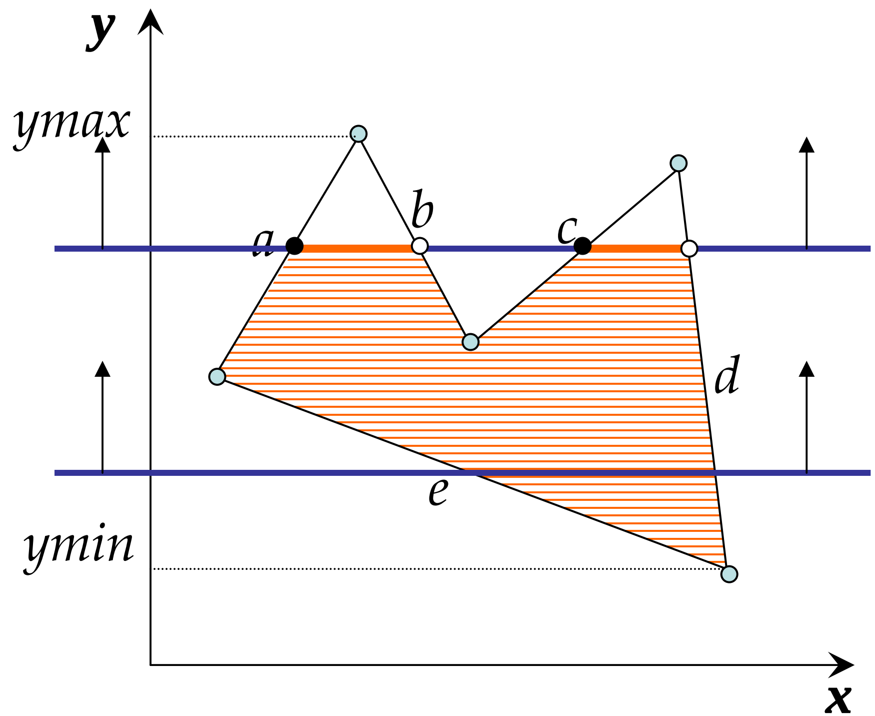
- Uso abusivo de recursão pode ser contornado preenchendo intervalos horizontais iterativamente
- Pode ser necessário usar um bitmap auxiliar para marcar os pixels visitados. Por exemplo
  - Critério é *pixel* (x, y) $\approx$ *cor*
  - Se *cor* $\approx$ *novaCor*, não há meio de distinguir um pixel visitado de um não visitado

Rasterização de Polígonos

- Operação fundamental em computação gráfica
- Polígono é dado por uma lista de vértices
  - Último vértice = primeiro vértice
- Obs.: OpenGL rasteriza apenas triângulos
  - Triângulos são casos especiais de polígonos
  - Polígonos genéricos precisam ser triangulados
    - Triangulação faz parte da biblioteca de utilitários (***gluTesselate***)

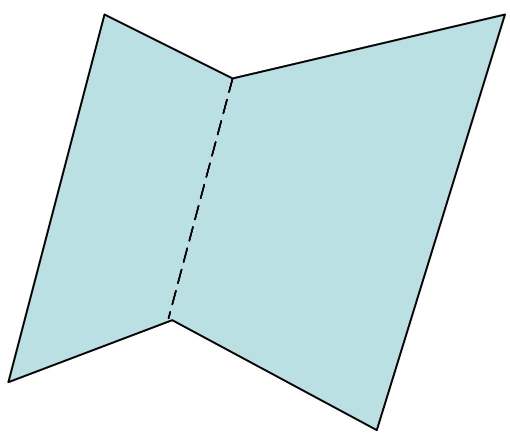
Rasterização de Polígonos

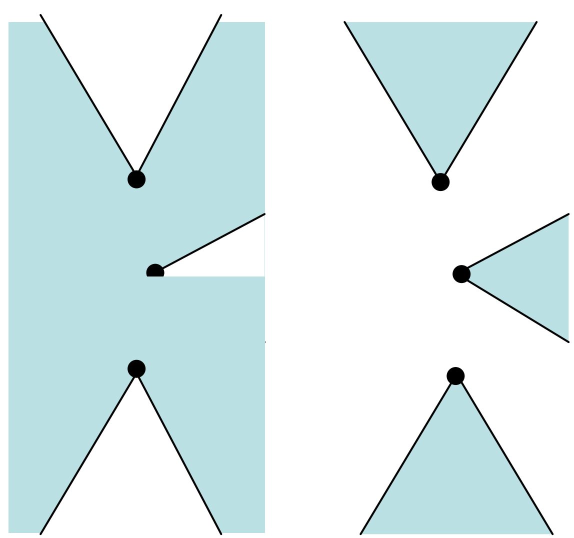
- Algoritmo clássico usa técnica de varredura
  - Arestas são ordenadas
    - Chave primária: *y* mínimo
    - Chave secundária: *x* mín.
    - Exemplo: (*e*,*d*,*a*,*b*,*c*)
  - Linha de varredura perpendicular ao eixo *y* percorre o polígono (desde *ymin* até *ymax*)
  - Intervalos horizontais entre pares de arestas são preenchidos



Rasterização de Polígonos

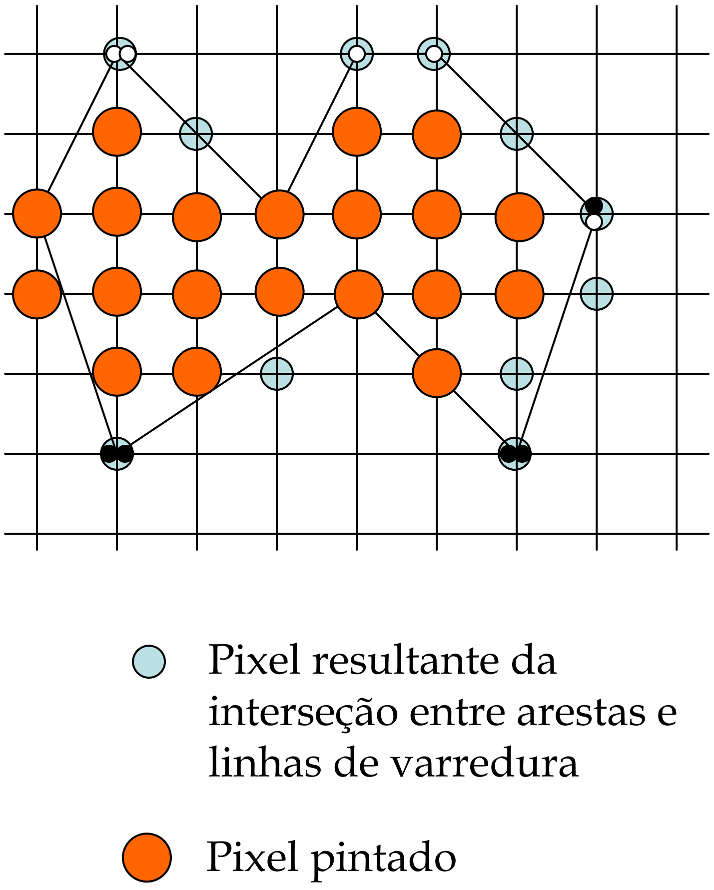
- Cuidados devem ser tomados para evitar pintar um mesmo pixel mais de uma vez
  - Dois polígonos adjacentes
  - Vértices (pertencem a 2 arestas)





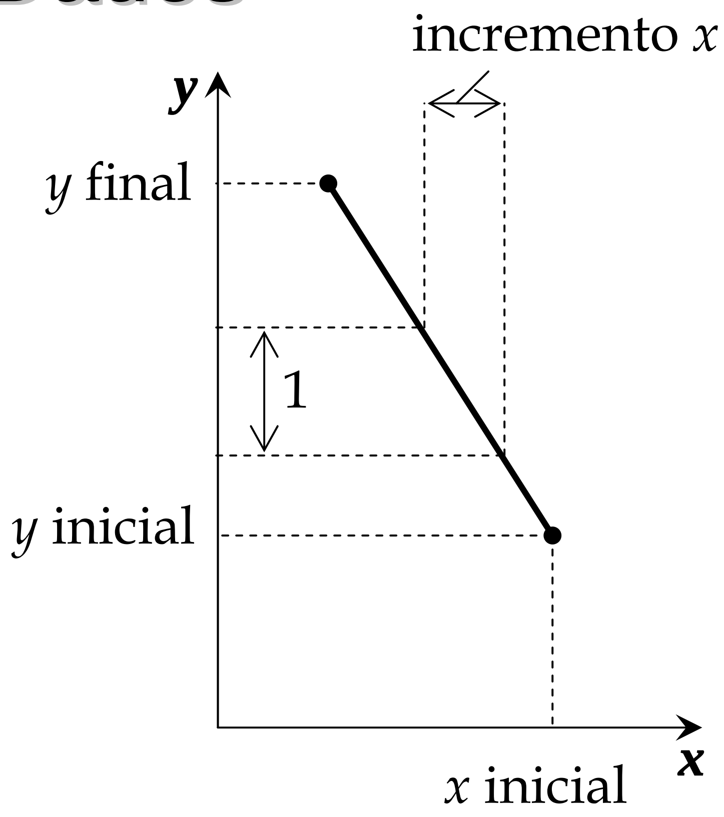
Rasterização de Polígonos

- Intervalos de preenchimento
  - Definidos sobre a linha de varredura
  - Cada intervalo começa e termina sobre um pixel interceptado por uma aresta
    - Primeiro pixel é pintado, último não
  - Arestas horizontais não são consideradas
  - Um vértice de uma aresta não horizontal é considerado apenas se for o vértice com *menor y*



Rasterização de Polígonos – Estruturas de Dados

- <u>Aresta</u>
  - *y* inicial (*y* mínimo)
  - *y* final
  - *x* corrente (inicialmente, *x* inicial)
  - *dx* (incremento *x*)
- <u>Lista de Arestas</u> – arestas do polígono
  - Ordenadas por *y* inicial / *x* inicial
- <u>Lista de Arestas Ativas</u> – arestas do polígono que interceptam linha de varredura corrente
  - Ordenadas por coordenada *x* de interseção



Rasterização de Polígonos – Pseudo Código

- Inicialização
  - Criar e ordenar *LA* (lista de arestas)
  - Computar *ymin* e *ymax*
  - *LAA* ← *nulo*
- Para *y* desde *ymin* até *ymax* fazer
  - Inserir em *LAA* todas as arestas com *yinicial* = *y*
  - Retirar da *LAA* todas as arestas com *yfinal* = *y*
  - Para cada par de arestas *A1*/*A2* da *LAA* fazer
    - Desenhar todos os pixels com *x* entre *A1.xcorrente* e *A2.xcorrente* (exclusive)
  - Para cada aresta *A* da *LAA* fazer
    - *A.xcorrente* ← *A.xcorrente* + *A.dx*
  - Reordenar a *LAA* (arestas cruzadas)