

Preenchimento de Áreas



Maria Cristina F. de Oliveira
Fernando V. Paulovich

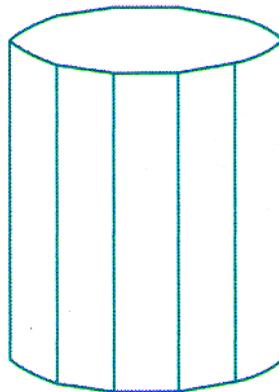


Preenchimento de Áreas

- Além do desenho de linhas, uma outra construção útil é o preenchimento de áreas
 - Usado para descrever superfícies ou objetos sólidos
- Embora qualquer forma possa ser preenchida, normalmente as APIs gráficas suportam polígonos
 - Maior eficiência por serem descritos por equações lineares
 - Maioria da superfícies curvas podem ser aproximadas por polígonos (efeitos de luz deixam mais real a aproximação)

Preenchimento de Áreas

- Aproximação de uma curva é normalmente chamada de **tesselação de uma superfície ou malha de polígonos**
 - Essas aproximações podem ser rapidamente geradas como visões *wire-frame*

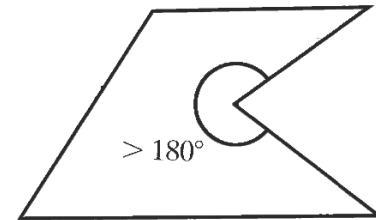
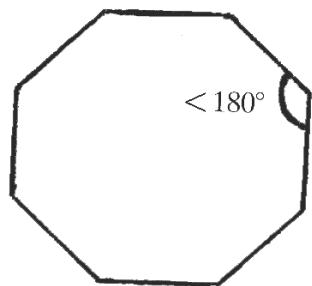


Preenchimento de Polígonos

- Um polígono é uma **figura plana** especificada por um conjunto de 3 ou mais **vértices** (posições) **ligados** seqüencialmente por **arestas** (linhas retas)
 - Arestas só tem pontos comuns nos seus inícios e fins
 - Todos os **vértices no mesmo plano** e sem cruzamento de arestas
- Devido a erros de arredondamento, as arestas de um polígono podem não ser **coplanares**
 - Usar triângulos para resolver esse problema

Classificação de Polígonos

- Se todos os ângulos interiores de um polígono forem menores que 180° , o polígono é **convexo** caso contrário é dito **côncavo**
 - Em um polígono convexo, dois pontos interiores definem um segmento de reta também no interior



Classificação de Polígonos

- O termo **Polígono Degenerado** descreve um polígono com vértices colineares, ou que apresenta vértices repetidos
- Uma API gráfica para ser robusta deve rejeitar polígonos não planares ou degenerados
 - Na verdade isso é deixado a cargo do programador

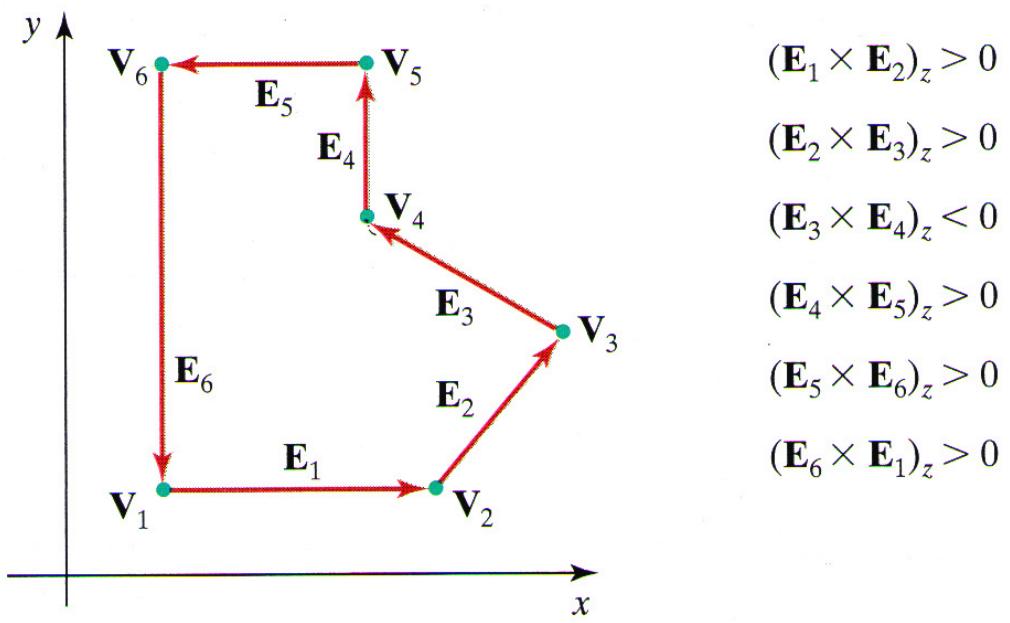


Classificação de Polígonos

- APIs gráficas normalmente só trabalham com polígonos convexos
 - Melhor dividir um polígono côncavo em um conjunto de polígonos convexos
 - OpenGL requer que todos os polígonos sejam convexos

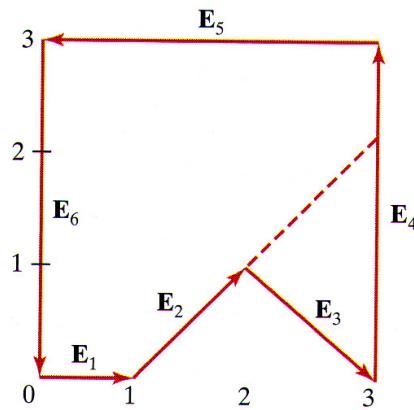
Identificando Polígonos Côncavos

- Criar vetores com as arestas e fazer o produto vetorial de arestas adjacentes
 - A coordenada-z de todos os produtos devem ter o mesmo sinal em um polígono convexo



Dividindo Polígonos Côncavos

- Criar vetores para dois vértices consecutivos
 - $E_k = V_{k+1} - V_k$
- Calcular o produto vetorial desses no sentido anti-horário
- Se algum produto for negativo, o polígono é côncavo
 - Dividir o polígono ao longo da linha do primeiro vértice





Dividindo em Polígono em Triângulos

- Um polígono convexo pode ser dividido em triângulos
 - Pegue quaisquer três vértices consecutivos e forme um triângulo
 - Remova o vértice do meio da lista de triângulos
 - Repita o procedimento até restarem 3 vértices

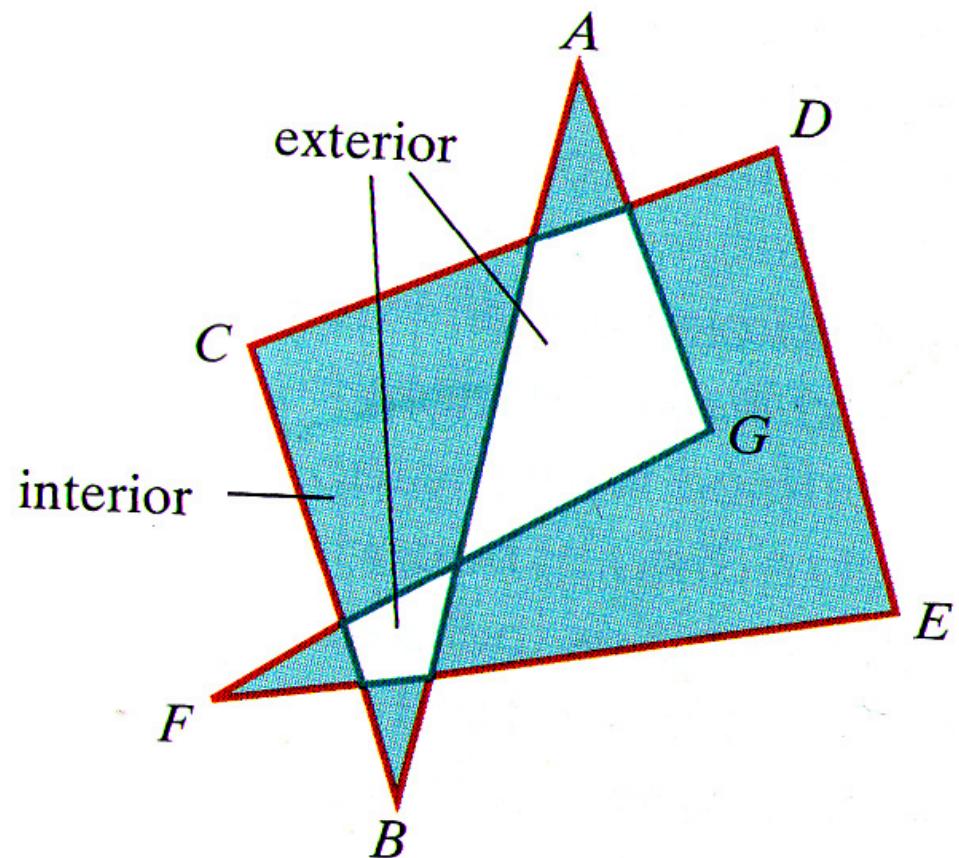
Testes de Interior-Exterior

- Vários processos gráficos precisam identificar regiões interiores de objetos mais complexos que quadrados ou círculos
- Dois algoritmos
 - Regra par-impar (ou regra da paridade ímpar)
 - Regra do *winding-number* não zero

Regra Par-Ímpar

- Desenhar um segmento de reta de um ponto P a um ponto distante fora dos limites de coordenadas do polígono
- Contar os cruzamentos de arestas com esse segmento
 - Se o número for ímpar, P está dentro
 - Caso contrário P está fora
- Assegurar que o segmento não intercepta nenhum vértice

Regra Par-Impar



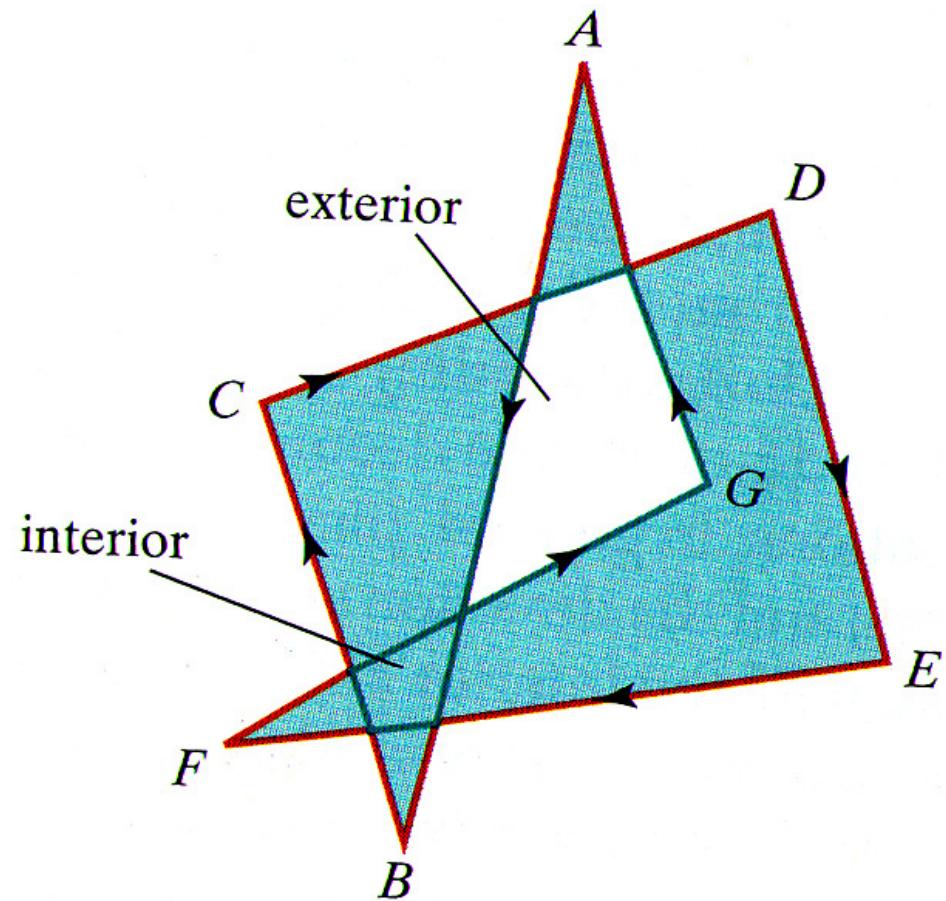
Regra do *Winding-number* não zero

- Conta o número de vezes que a fronteira de um objeto gira em volta de um ponto particular na direção anti-horária
 - Um ponto é dito interior se sua contagem for diferente de zero

Regra do *Winding-number* não zero

- Inicializa a contagem com zero
- Define um segmento de reta de uma posição P até um ponto distante
 - Não pode passar pelos vértices
- Conta a quantidade de cruzamentos com as arestas (direcionais)
 - +1 toda vez que cruzar uma aresta da direita para esquerda
 - -1 toda vez que cruzar uma aresta da esquerda para direita

Regra do *Winding-number* não zero



Regra do *Winding-number* não zero

- Matematicamente
 - Calcular o produto vetorial entre o vetor definido pela aresta e o vetor definido pelo segmento de reta
 - +1 se o componente z do produto for positivo
 - -1 caso contrário
 - Encontrar um vetor perpendicular ao vetor do segmento de reta $(v_x, v_y) \rightarrow (-v_y, v_x)$ e fazer o produto escalar com o vetor da aresta
 - +1 se o produto for positivo
 - -1 caso contrário



Preenchimento de Áreas

- Maioria das APIs limita o preenchimento de áreas
 - polígonos porque esses são descritos por equações lineares
 - Polígonos convexos porque assim somente duas arestas são cruzadas
- Porém, é possível preencher o interior de qualquer tipo de forma (ferramentas de desenho)



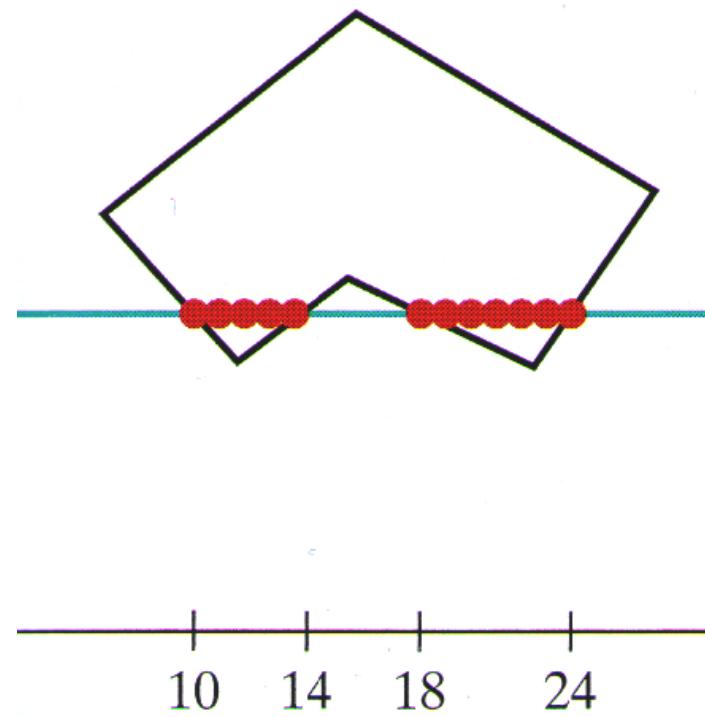
Preenchimento de Áreas

- Existem basicamente duas formas
 - Determinar os intervalos de preenchimento usando *scalines* e preencher o interior
 - Indicado para polígonos
 - A partir de um ponto, colorir a vizinhança até encontrar as bordas
 - Indicado para formas mais complexas

Algoritmo *Scanline*

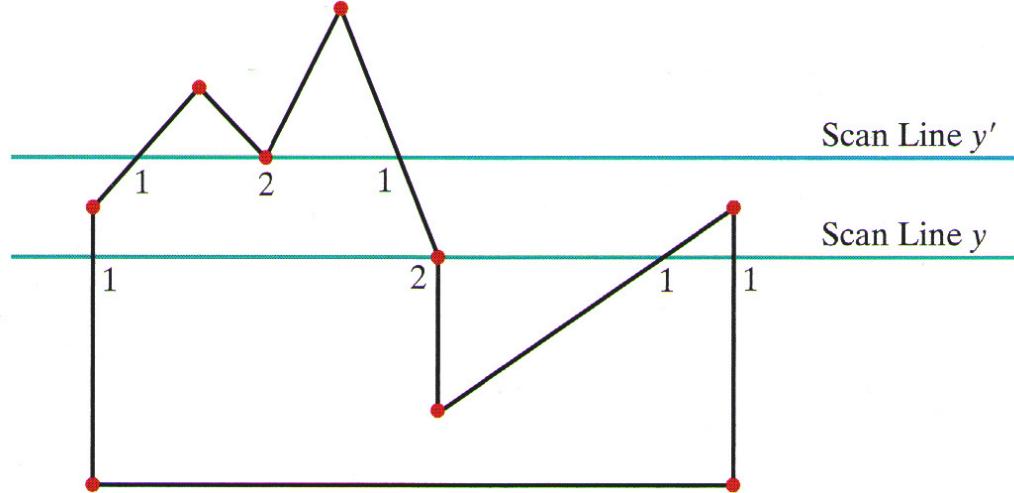
- Primeiro determina as intersecções das *scanlines* com o polígono
- Então, as seções da *scanline* que residirem dentro do polígono são coloridas
 - Usa a regra par-ímpar
- Para polígonos, 2 equações lineares são resolvidas para se encontrar as intersecções
 - Calcula as intersecções de um polígono da esquerda para a direita

Algoritmo *Scanline*



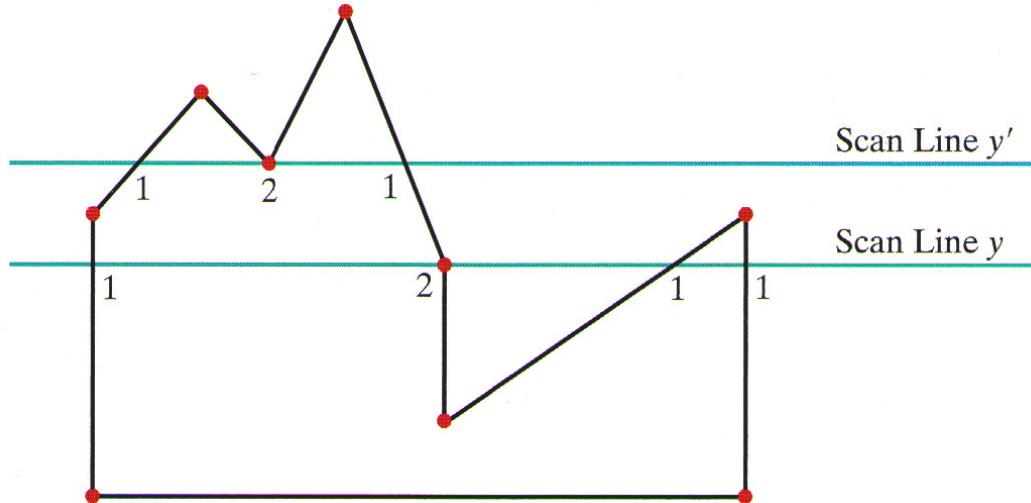
Problema da *Scanline*

- Problemas quando a *scanline* passa por um vértice
 - Intersecta dois polígonos simultaneamente



Problema da *Scanline*

- A contagem de intersecção deve ser diferente dependendo da topologia
 - Duas arestas de lados opostos da *scaline*: conta uma intersecção
 - Duas arestas do mesmo lado da *scaline*: conta duas intersecções

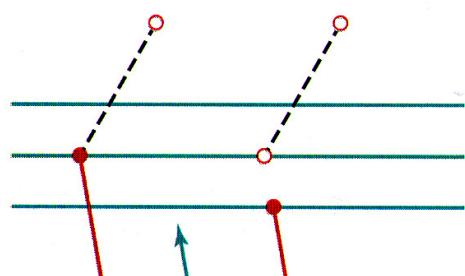


Solução para o Problema da *Scanline*

- Para descobrir se as arestas estão opostas
 - Definir a fronteira do polígono de forma anti-horária (ou horária) e observar as mudanças em y
 - Se os 3 vértices de duas arestas consecutivas são monotonicamente crescentes (ou decrescentes) conta somente uma intersecção
 - Caso contrário conta duas

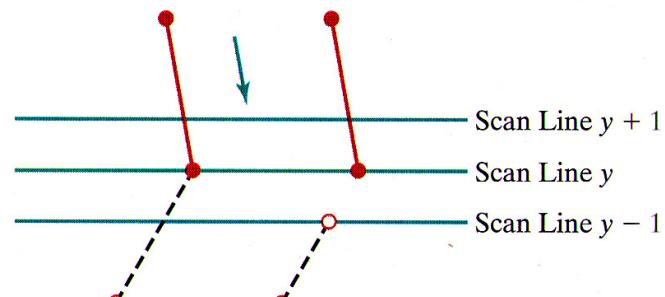
Solução para o Problema da *Scanline*

- Para descobrir se as arestas estão opostas
 - Percorrer as arestas no sentido anti-horário (ou horário) e verificar se 3 vértices consecutivos são monotonicamente crescentes (decrescentes) em relação a y
 - Nesse caso a aresta inferior pode ser reduzida para assegurar somente uma intersecção



(a)

crescente



(b)

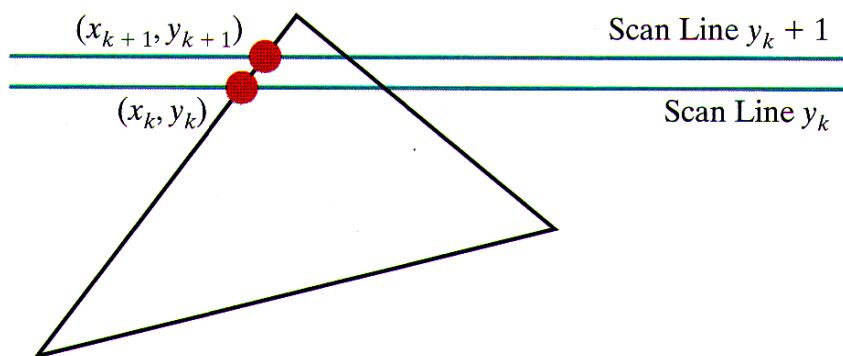
decrescente

Algoritmo *Scanline* (Intersecção)

- A interseção da i -ésima *scanline* com uma aresta $\{(x_1, y_1), (x_2, y_2)\}$ é calculada com duas equações
 - $y = i$
 - $x = y/m - B$
 - $m = (y_2 - y_1)/(x_2 - x_1)$
 - $B = y_1/m - x_1$

Algoritmo *Scanline* (Intersecção)

- É possível acelerar esse processo usando uma abordagem incremental
 - $m = (y_{k+1} - y_k) / (x_{k+1} - x_k)$
 - Como $y_{k+1} - y_k = 1$ entre duas *scalines* consecutivas
 - Temos $x_{k+1} = x_k + 1/m$



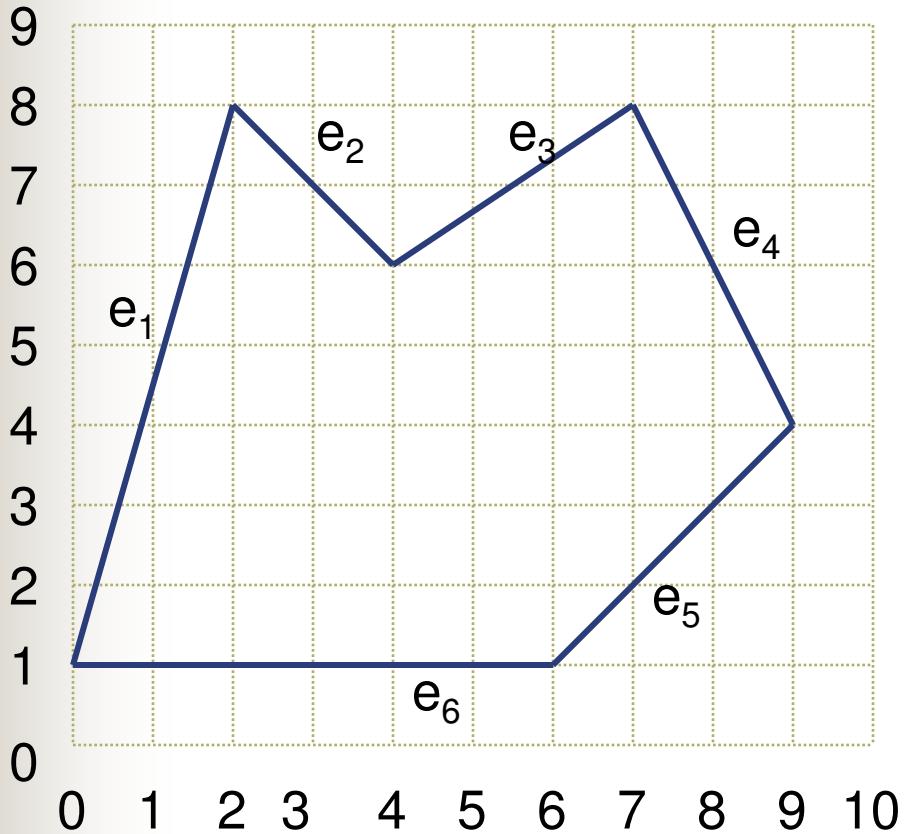
Algoritmo *Scanline*

- É possível usar somente inteiros
 - Lembre que $m = \Delta y / \Delta x$
 - Então $x_{k+1} = x_k + \Delta x / \Delta y$

Algoritmo *Scanline*

- Para polígonos convexos, só existe um bloco de pixels subsequentes em cada *scanline*
 - Só processa a *scanline* até encontrar duas intersecções
- Mesmo algoritmo anterior, porém muito mais simples
 - Intersecções nas arestas não apresentam dubiedade

Exemplo

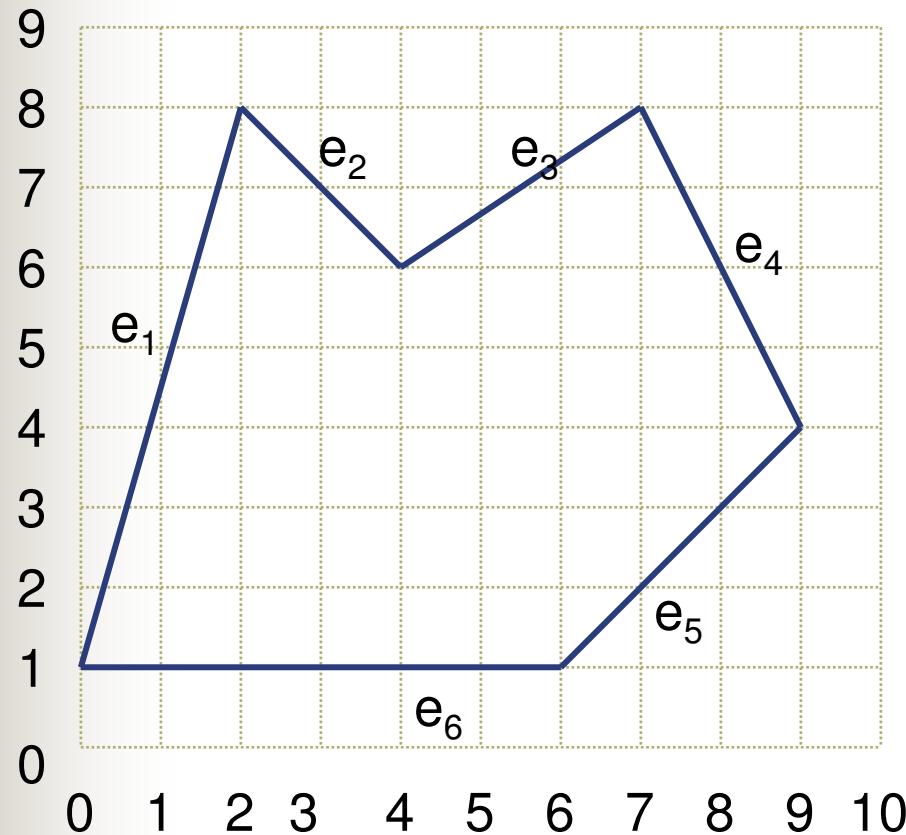


Este polígono tem
como vértices:

- (0,1)
- (2,8)
- (4,6)
- (7,8)
- (9,4)
- (6,1)

e 6 arestas
 e_1, \dots, e_6

Algoritmo Scanline

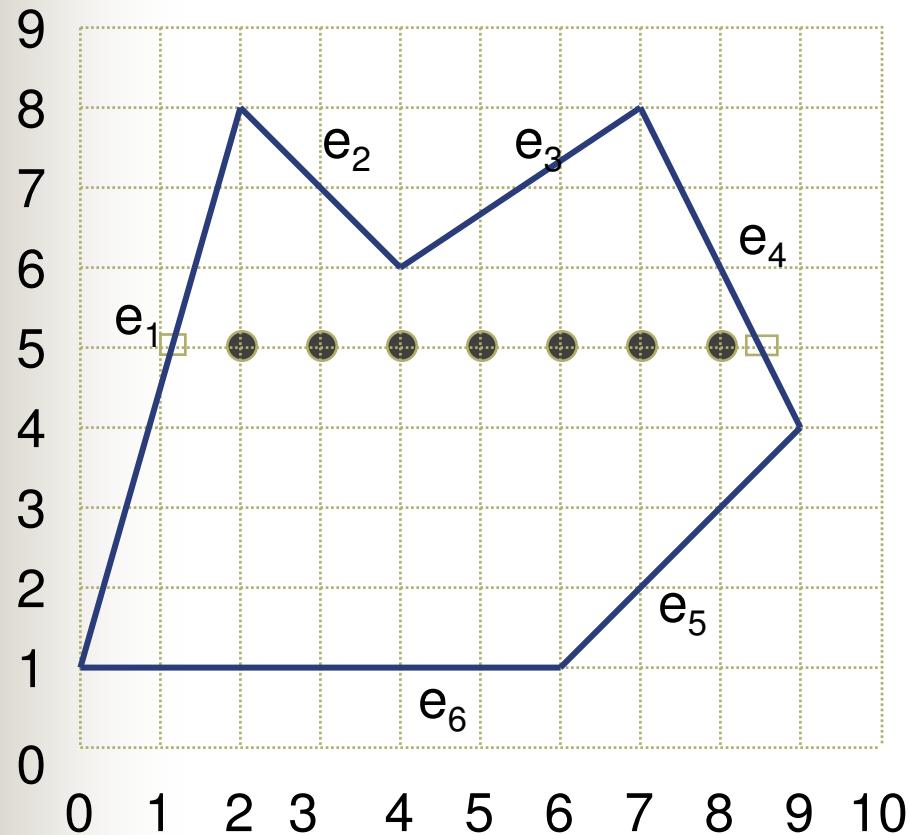


Para preencher a área,
processa as linhas
de varredura
(*scanlines*)

Para um dado valor y ;
acha intersecções com
arestas

Pares de intersecções
sucessivas definem um
bloco (*span*) de pixels

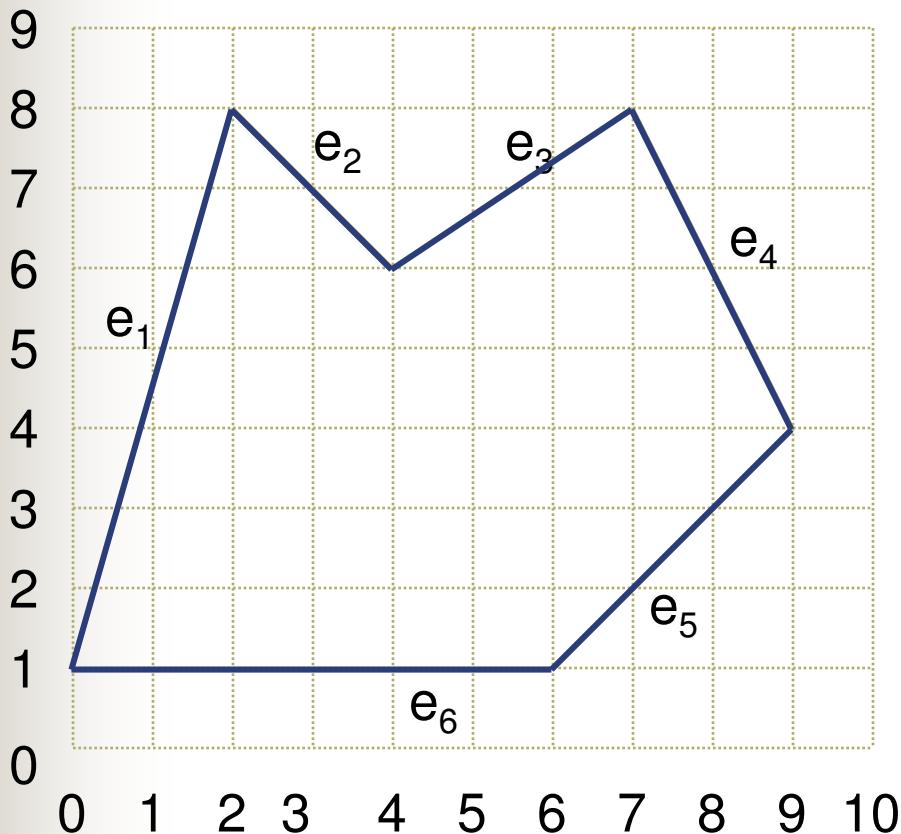
Casos a tratar: 1. normal



Scanline 5:
interseção $e_1 = 1.14$
interseção $e_4 = 8.5$

Pixels 2-8 na *scanline*
são preenchidos

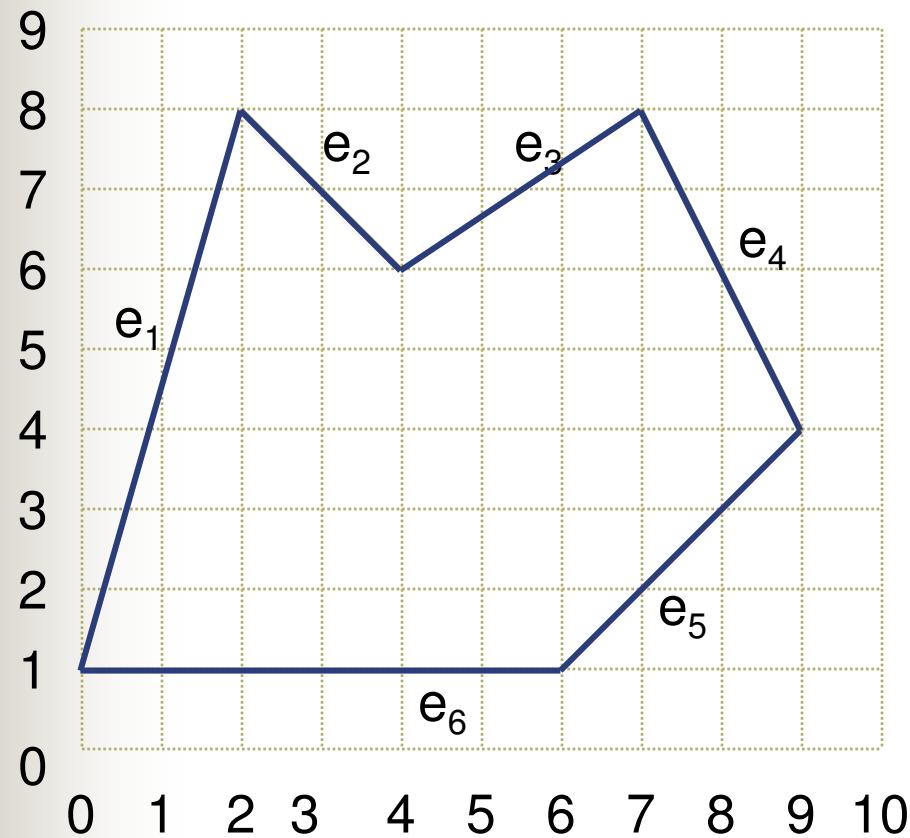
Casos a tratar: 2. Intercepta vértices



scanline 6 intercepta
 e_1, e_2, e_3, e_4 em
1, 4, 4, 8 respectiva/e
- dois spans traçados

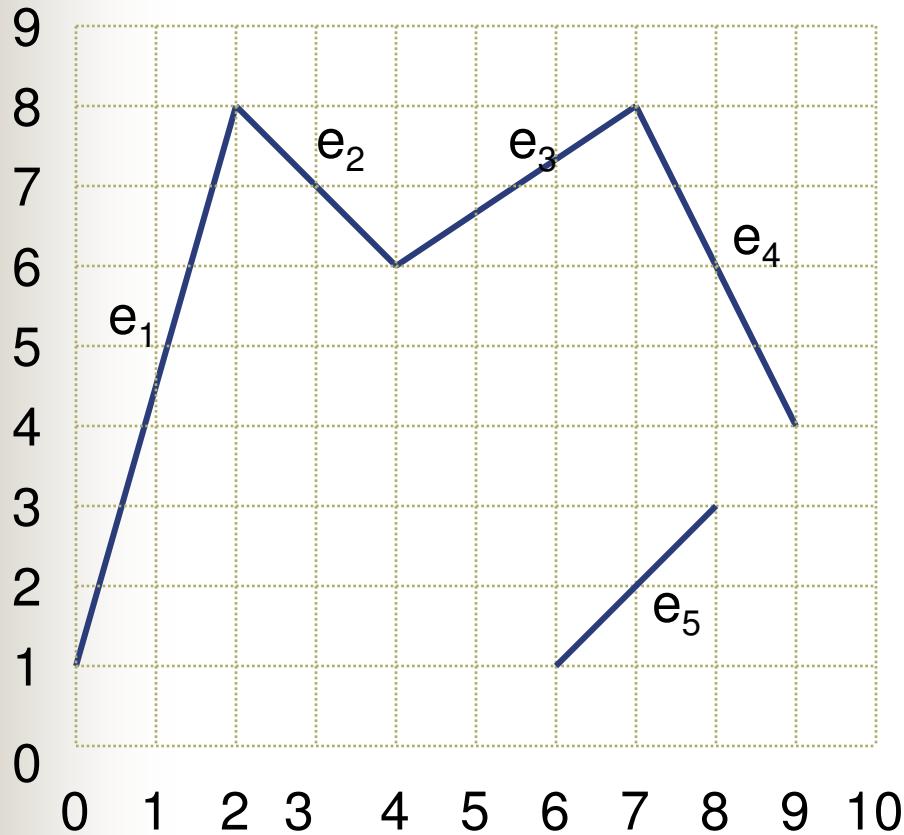
scanline 4 intercepta
 e_1, e_4, e_5 em
0, 9, 9 respectiva/e
- linha intercepta duas
arestas, considera
só a superior, i.e. e_4

Casos a tratar: 3. Arestras horizontais



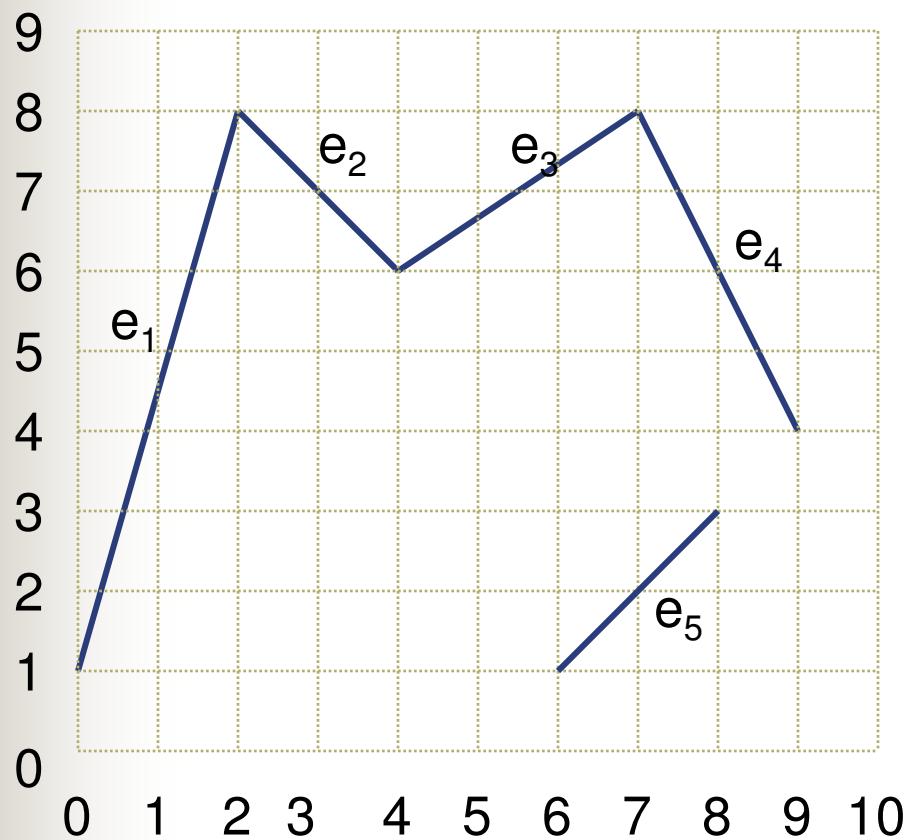
Uma aresta horizontal, como e_6 pode ser ignorada:
vai ser traçada automaticamente

Pré-Processamento do Polígono



Removemos
 e_6 , e encurtamos
 e_5 em uma *scanline*

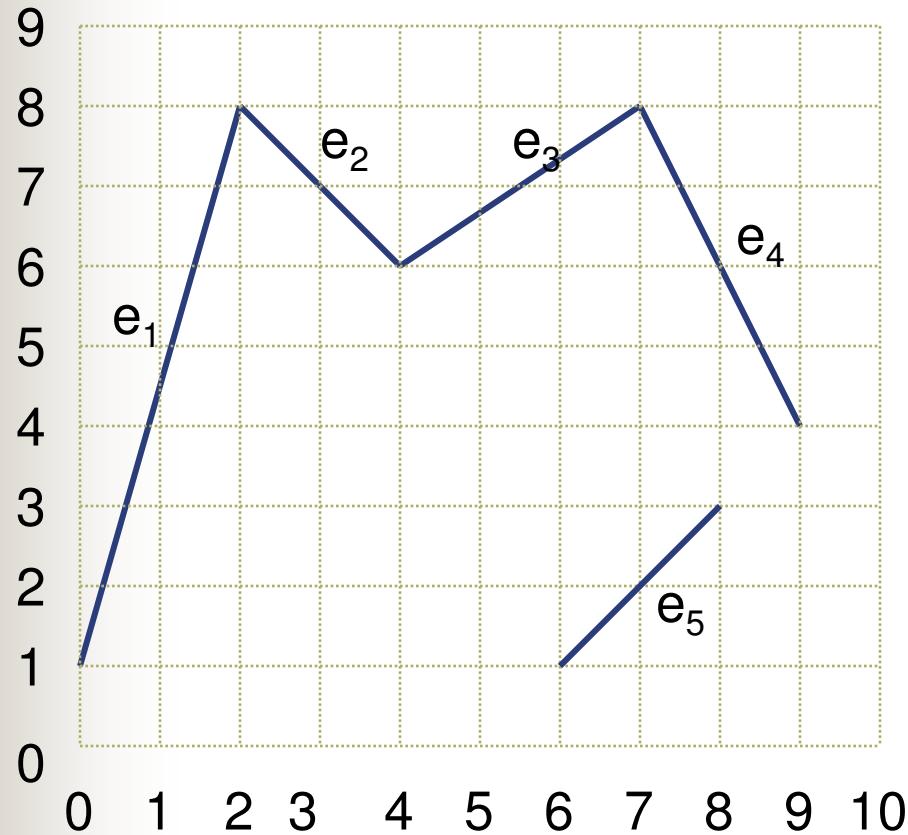
Otimização



Acelerar: saber aonde
Começam e
terminam as arestas:
e.g., e_1 vai da *scanline* 1
até a *scanline* 8;
 e_2 da *scanline* 8 até a 6

Assim, sabemos que
arestas testar para
cada *scanline*

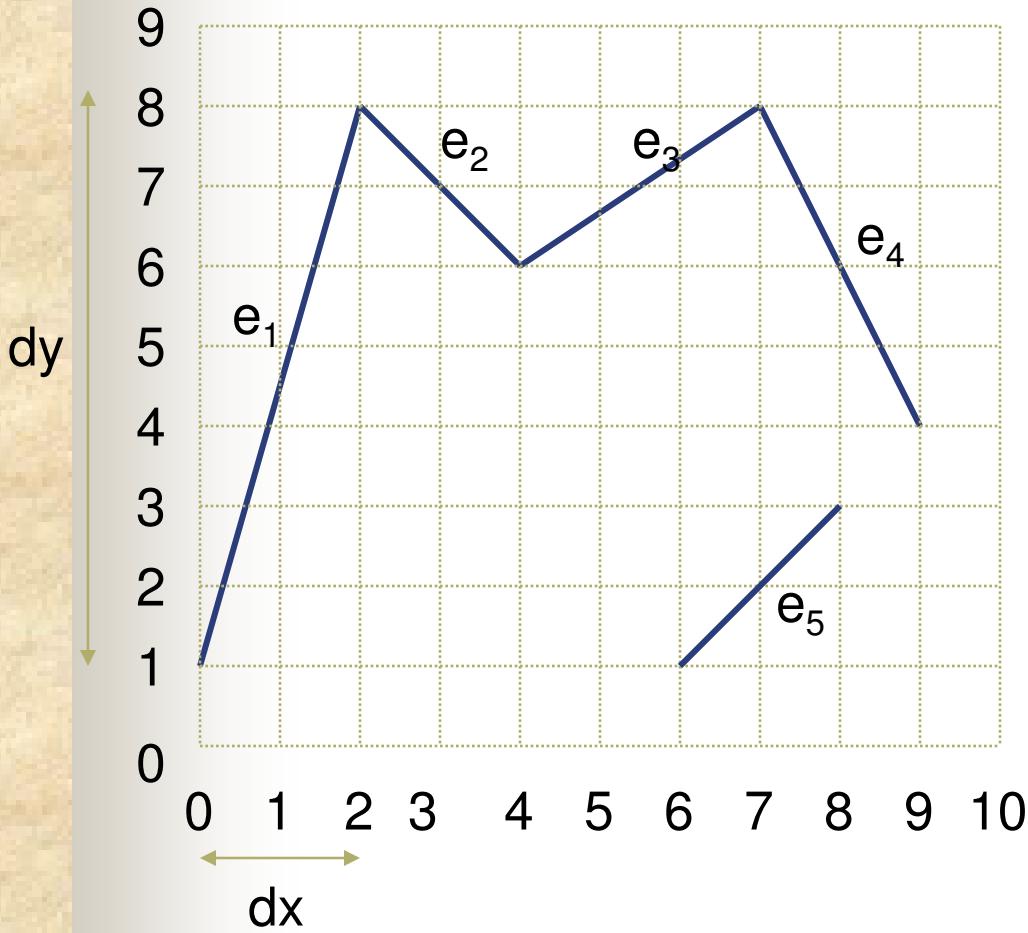
Conjuntos Ordenados de Arestas



Ordenar as arestas pelo seu ponto de mínimo; cada *scanline* é associada a um conjunto de arestas, ordenado segundo a coord. x da intersecção

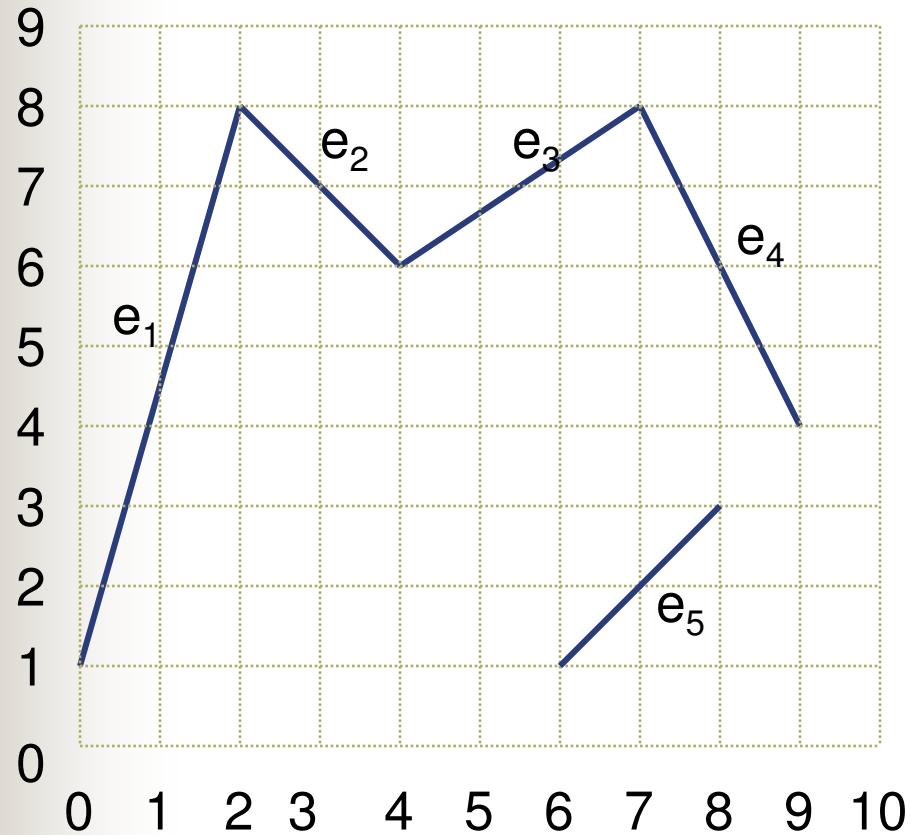
- 0:
- 1: e₁, e₅
- 2:
- 3:
- 4: e₁, e₄
- 5:
- 6: e₁, e₂, e₃, e₄
- 7:
- 8:
- 9:

Otimização - Coerência



Ex.: observe e_1
Assuma intersecção com
scanline conhecida,
então, a intersecção
com *scanline 2* é:
 $x^* = x + 1/m$
i.e.: $x^* = x + dx / dy$
i.e.: $x^* = 0 + 2 / 7 = 2/7$

Tabela de Arestas



Útil armazenar informação sobre arestas em uma tabela:

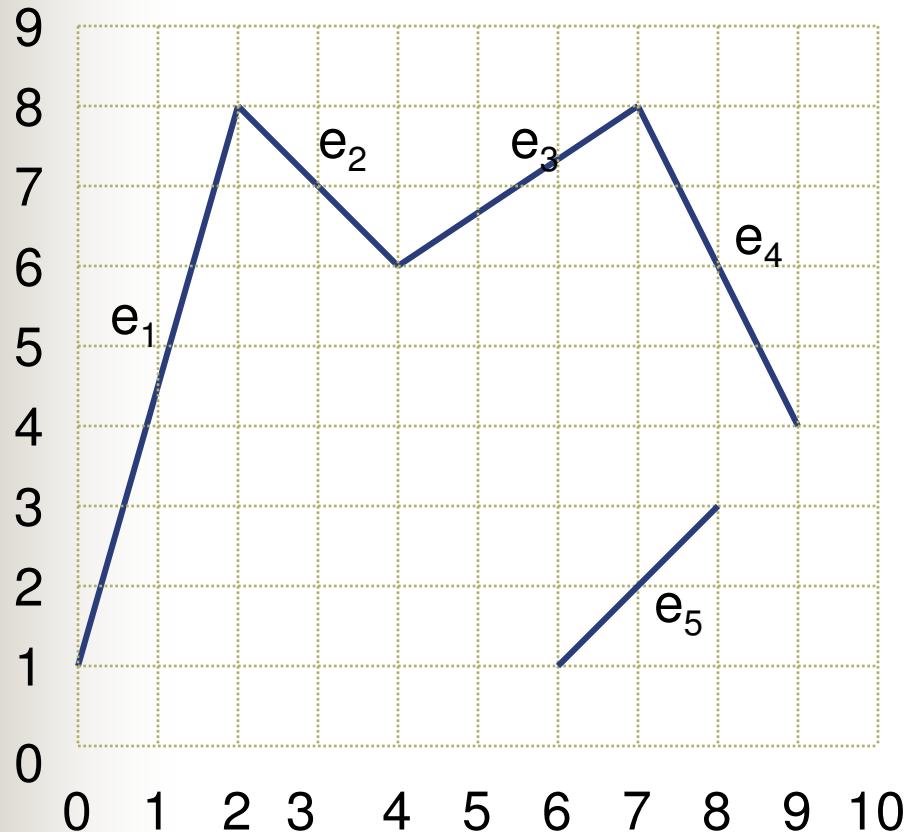
e	prim x	max y	dx	dy
1	0	8	2	7
2	4	8	-2	2
3				
4				
5				



Estrutura de dados

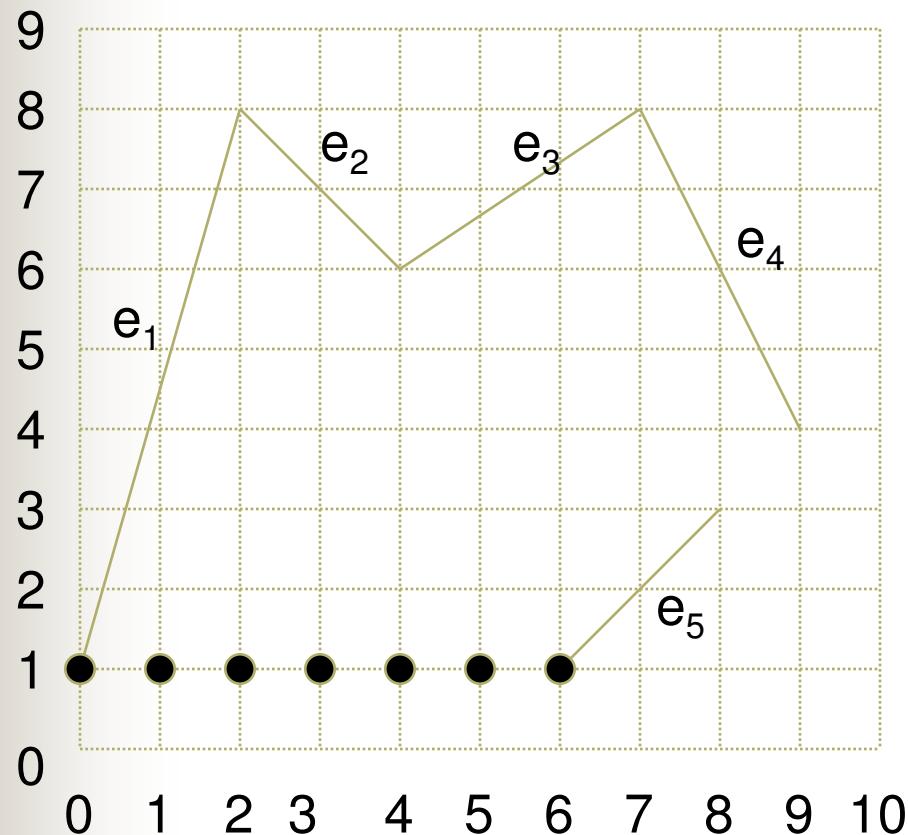
- Tabela de arestas global: descreve as arestas do polígono a serem processadas explicitamente
 - Um conjunto para cada linha de varredura
 - Mantém-se ao longo de todo o processamento
- Tabela de arestas ativas (as que estão sendo interceptadas pela linha de varredura corrente)
 - É atualizada a medida que a varredura da cena prossegue

Algoritmo de Preenchimento



- (1) Faça $y = 0$
- (2) Verifique y -bucket
- (3) Insira as arestas necessárias na **active edge table (AET)**
- (4) SE AET vazia ENTÃO
 $y = y + 1$, GO TO (2)

Tabela de Arestas Ativas

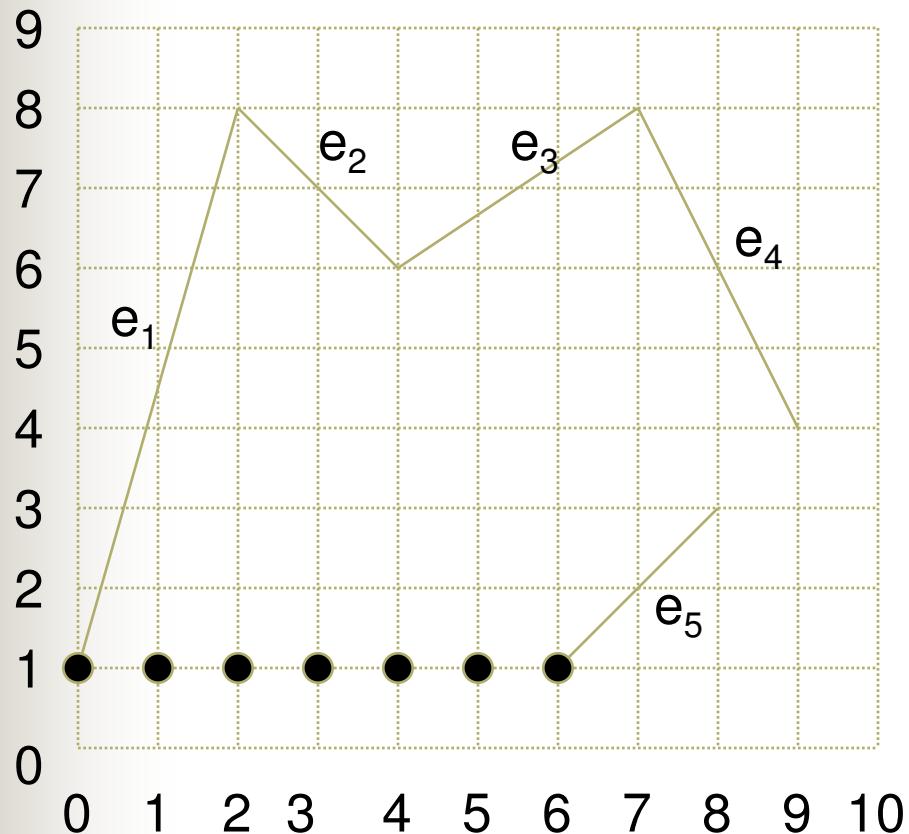


$$y = 1$$

e	x	max y	dx	dy
1	0	8	2	7
5	6	3	2	2

(5) Ordene por valores-x
e preencha entre pares
sucessivos

Tabela de Arestas Ativas



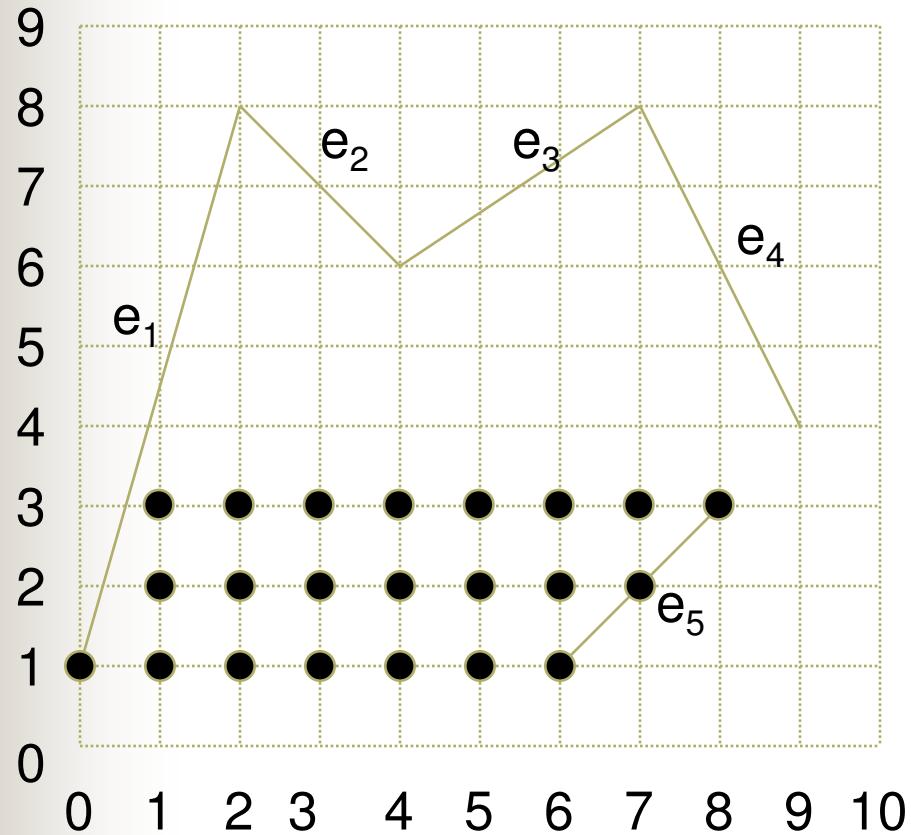
(6) FAÇA $y = y+1$;
remova da AET arestas
com $\max y < y$

(7) Incrementa x de $1/m$
(dx/dy)

e	x	max y	dx	dy
1	2/7	8	2	7
5	7	3	2	2

(8) Retorna para (2)

Próxima Scanline



A cada estágio do algoritmo, um (ou vários) *span(s)* de pixels são traçados



Eficiência – aritmética inteira

- Por questão de eficiência, é interessante usar apenas aritmética inteira – isso requer uma coluna extra na AET para acumularmos separadamente a parte inteira da parte fracionária de $1/m$ para a atualização de x

Eficiência – aritmética inteira

Primeira intersecção é em $x=0$, e a inclinação da aresta é $7/2$ – i.e. $dy=7$, $dx=2$

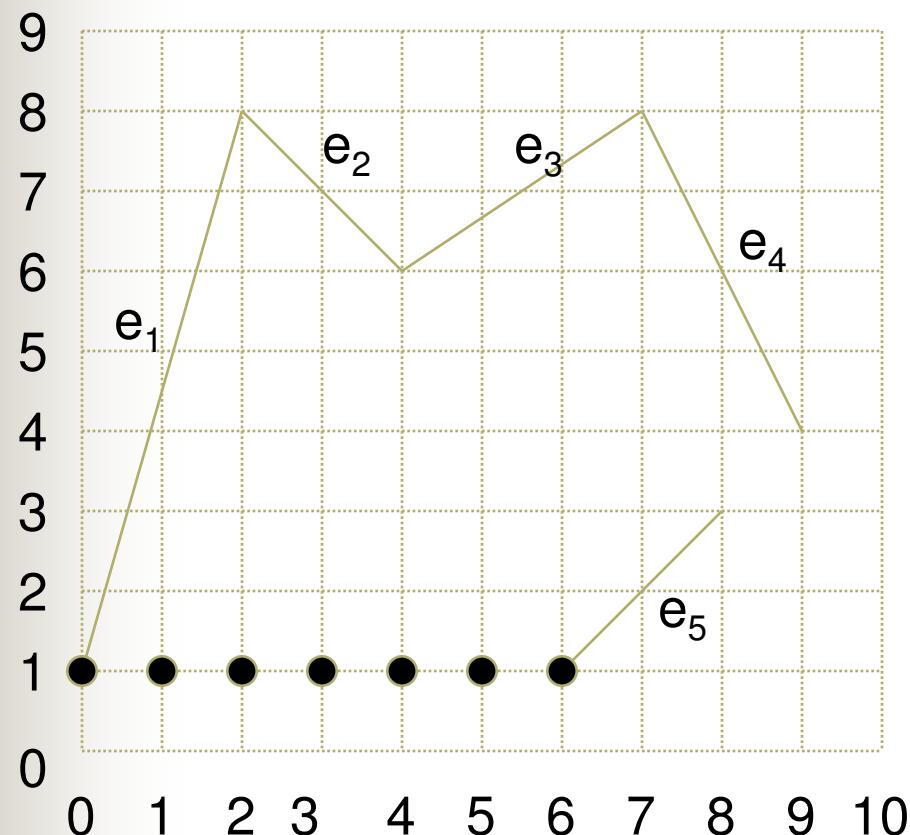
Os próximos pontos de intersecção são:

0 $2/7$ $4/7$ $6/7$ $8/7$ etc.

Obtemos esses valores simplesmente somando dx a cada estágio, até que dy é atingido, então dx é reduzido de dy :

0 2 4 6 1 (8-7) etc.

Tabela de Arestas Ativas Modificada



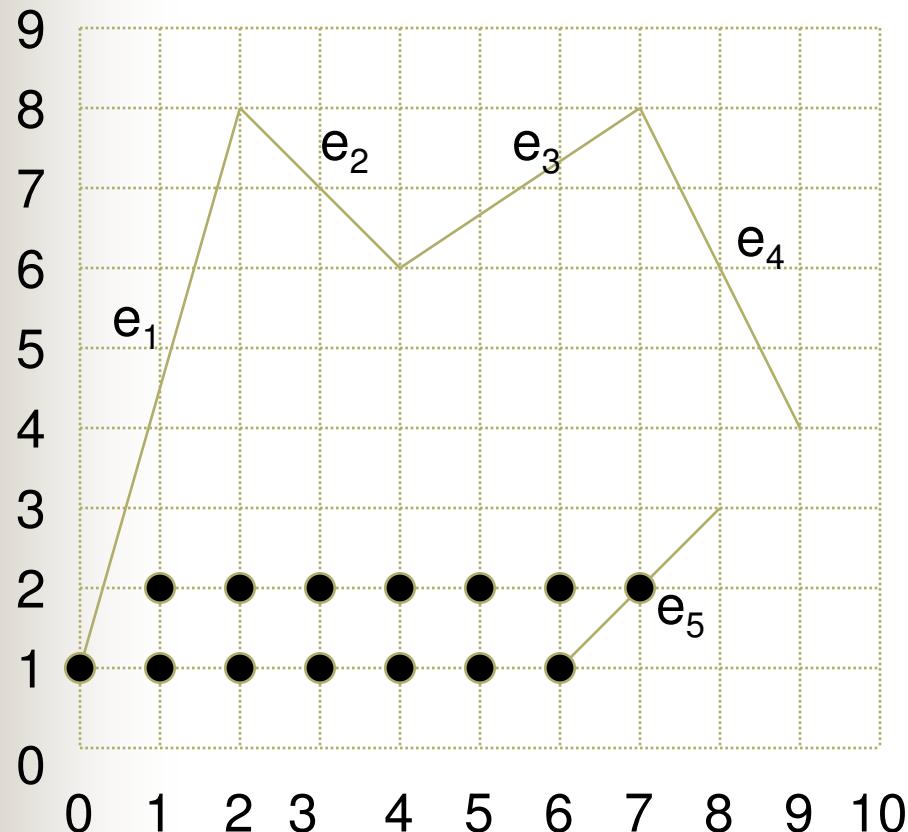
$$y = 1$$

e	x	max y	dx	dy
1	0	8	2	7
5	6	3	2	2

torna-se

e	x int	x frac	max y	dx	dy
1	0	0	8	2	7
5	6	0	3	2	2

Eficiência – aritmética inteira



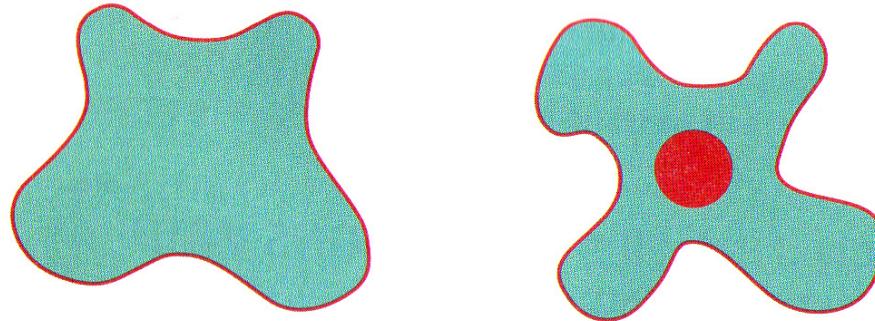
No passo (7), ao invés de incrementar x por dx/dy , incrementamos $x\text{-frac}$ de dx ; sempre que $x\text{-frac}$ excede dy somamos 1 a $x\text{-int}$ & reduzimos $x\text{-frac}$ de dy

$$y = 2$$

e	x int	x frac	max y	dx	dy
1	0	2	8	2	7
5	7	0	3	2	2

Preenchimento de Regiões Irregulares

- É possível preencher uma região irregular selecionando um pixel e pintando os pixels vizinhos até alcançar as bordas

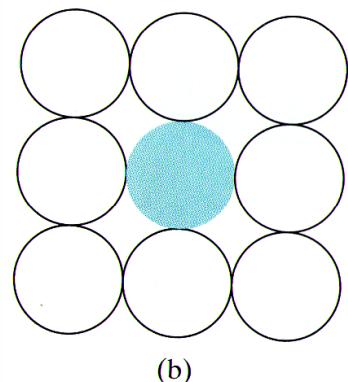
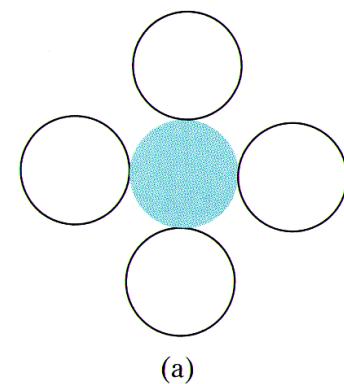




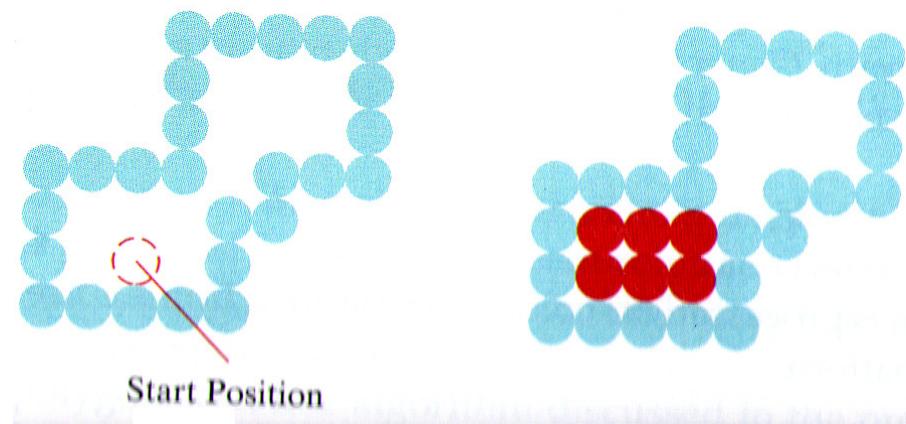
Algoritmo de Preenchimento de Borda

- Se a borda de uma região tem a mesma cor, é possível preencher essa pixel por pixel até atingir a cor da borda
 - Normalmente usado em programas gráficos
 - Começa com um ponto inicial (x,y) e testa os vizinhos para ver a cor, se não for borda, preenche

Algoritmo de Preenchimento de Borda



Diferentes testes de vizinhança



Problemas com a máscara de quatro vizinhos

Algoritmo de Preenchimento de Borda

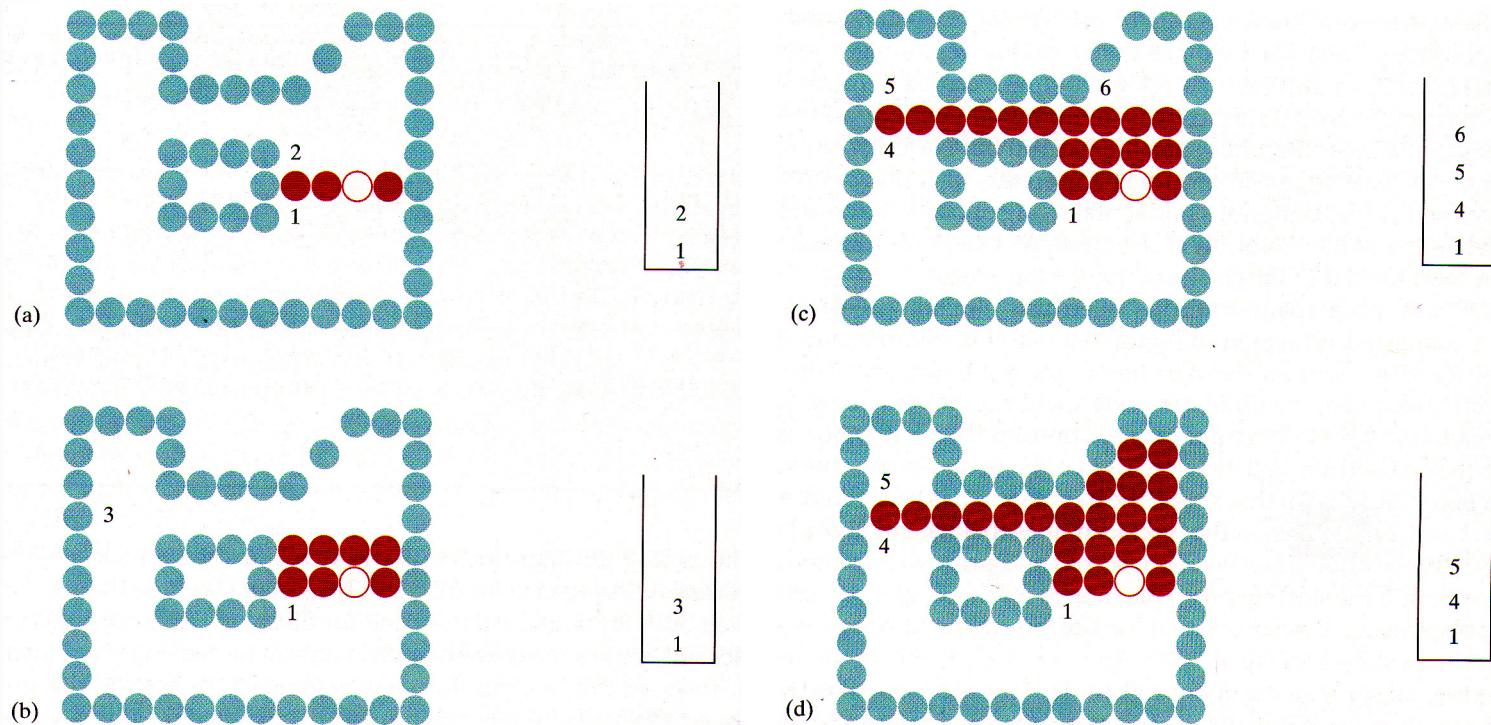
```
void fill(int x, int y, int fillcolor, int bordercolor) {  
    int intcolor;  
    getPixel(x,y,intcolor);  
    if((intcolor != bordercolor) && (intcolor != fillcolor)) {  
        setpixel(x,y, fillcolor);  
        fill(x+1,y,fillcolor,bordercolor);  
        fill(x-1,y,fillcolor,bordercolor);  
        fill(x,y+1,fillcolor,bordercolor);  
        fill(x,y-1,fillcolor,bordercolor);  
    }  
}
```



Algoritmo de Preenchimento de Borda

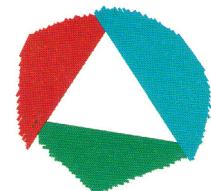
- Problemas se algum pixel interior já for da cor escolhida para ser preenchida
 - Algum ramo da recursão pode ser descartado
- Pode levar a um consumo excessivo de memória devido a recursão
 - Solução é empilhar, ao invés de pixels vizinhos, blocos de pixels sucessivos (o pixel inicial desses)

Algoritmo de Preenchimento de Borda



Algoritmo *Flood-Fill*

- As vezes é necessário colorir uma área que não é definida apenas por uma cor de borda
 - Ao invés de procurar uma cor de borda, procurar por uma cor de interior
 - Se o interior tem mais de uma cor, pode-se inicialmente substituir essa cor para que todos pixels do interior tenham a mesma cor





Algoritmo de Preenchimento de Borda

```
void fill(int x, int y, int fillcolor, int interiorcolor) {  
    int color;  
    getPixel(x,y,color);  
    if(color == interiorcolor) {  
        setpixel(x,y, fillcolor);  
        fill(x+1,y,fillcolor, interiorcolor);  
        fill(x-1,y,fillcolor, interiorcolor);  
        fill(x,y+1,fillcolor, interiorcolor);  
        fill(x,y-1,fillcolor, interiorcolor);  
    }  
}
```



Observação

- No caso de preenchimento de áreas, cada pixel está sendo ‘pintado’ com uma cor
- No caso de *rendering* de superfícies, cada pixel é pintado com a cor determinada pela aplicação do algoritmo de iluminação + tonalização (*shading*)



Bibliografia

- Hearn & Baker, 4.10
- Ap. CG Cap. 4
- <http://www.cs.rit.edu/~icss571/filling/example.html>