Universidade de São Paulo
Instituto de Matemática e Estatística

Introdução à Computação Gráfica

Marcel P. Jackowski
mjack@ime.usp.br

Aula #14

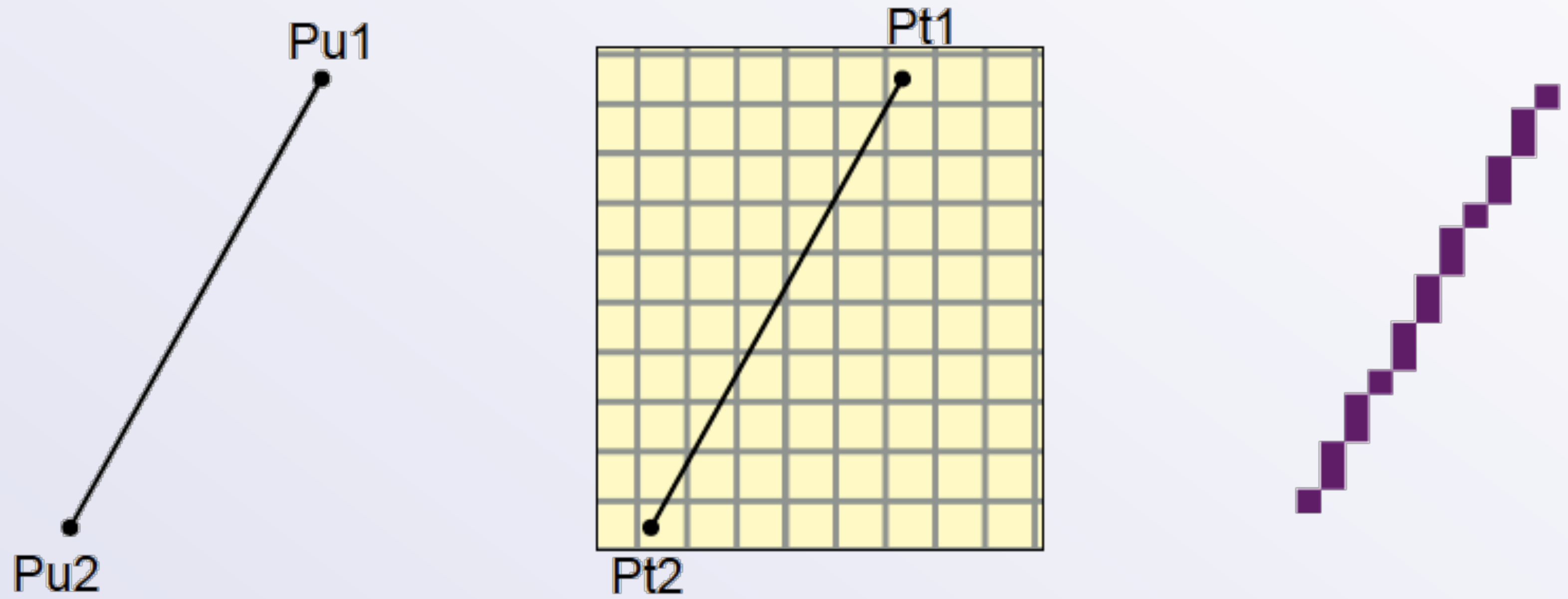
Objetivos

- Rasterização
  - Segmentos de reta
    - Algoritmo DDA
    - Algoritmo de Bresenham
  - Polígonos
- Representação de curvas e superfícies
  - Forma Explícita
  - Forma Implícita
  - Forma Paramétrica
  - Vantagens e desvantagens

Rasterização

- Rasterização (“scan conversion”)
  - Determina quais pixels que pertencem a primitiva especificada através de vértices
  - Produz um conjunto de fragmentos
  - Fragmentos possuem uma posição e outros atributos como cor e coordenadas de textura que são interpolados entre os vértices
- Cores finais dos pixels são determinadas usando cor, textura e outras propriedades dos vértices.

Desenho de retas



Algoritmo básico

- Dados pontos extremos da linha na tela

  Pt1= (x1, y1), Pt2= (x2, y2)

- Calcula coeficientes da equação da reta:

$$\boxed{y = mx + b} \qquad m = \frac{y2 - y1}{x2 - x1} \qquad b = y1 - m \,.\, x1$$

- Liga todos os pixels que pertencem à reta

  for **x** = x1 to x2

  y = m * x + b

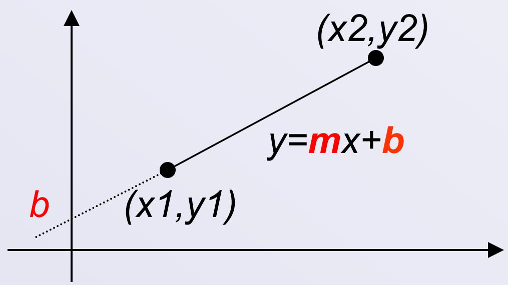
  ponto(**x**,**y**)

Utilizando a equação da reta

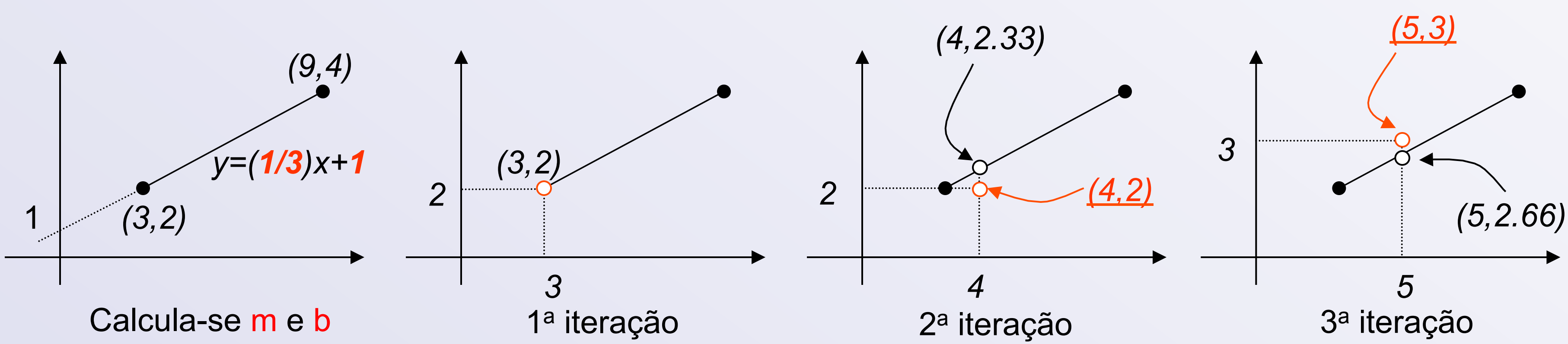
$y_i = m\, x_i + b$
onde:

$m = \Delta y/\Delta x$
$b = y_1 - m\, x_1$

Exemplo




```
void Line1(int x1, int y1, int x2, int y2, int color)
{
  float m = (y2-y1)/(x2-x1);
  float b = y1 - m*x1;
  float y;
  int x;

  write_pixel(x1,y1,color);

  for (x=x1+1; x<=x2; x++)
  {
     y = m*x + b;
     write_pixel(x,ROUND(y), color);
  }
}
```



Desvantagens

```
void Line2(int x1, int y1, int x2, int y2, int color)
{
  float m = (y2-y1)/(x2-x1);
  float b = y1 - m*x1;
  float y;
  int x;

  write_pixel(x1,y1,color);

  for (x=x1+1; x<=x2; x++)
  {
     y = m*x + b;
     write_pixel(x,ROUND(y), color);
  }
}
```

② adição

① multiplicação

③ arredondamento

Como tirar a multiplicação?

Algoritmo básico (ii)

- Da equação paramétrica da reta
  - P = Pt1 + t * (Pt2 - Pt1)
  - t variando de 0 a 1
- Chega-se nas seguintes expressões
  - x = x1 + t * (x2-x1)
  - y = y1 + t * (y2-y1)

Algoritmo básico (ii)

```
y = y1;
x = x1;
for t = 0 to 1
    ponto(x,y);
    y = y1 + t * (y2-y1);
    x = x1 + t * (x2-x1);
```

- 2 operações de ponto flutuante por pixel
- 2 multiplicações por pixel
- Como melhorar?

Algoritmo DDA

- Digital Differential Analyzer
  - DDA foi uma máquina para resolver equações diferenciais de forma numérica
  - A linha $y=mx + b$ satisfaz a equação diferencial

    $dy/dx = m = \Delta y/\Delta x = (y_2-y_1)/(x_2-x_1)$
- Podemos desenhá-la para cada passo Δx

```
for(x=x1; x<=x2, x++) {
    y+=m;
   write_pixel(x, round(y), line_color)
}
```

Algoritmo DDA

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

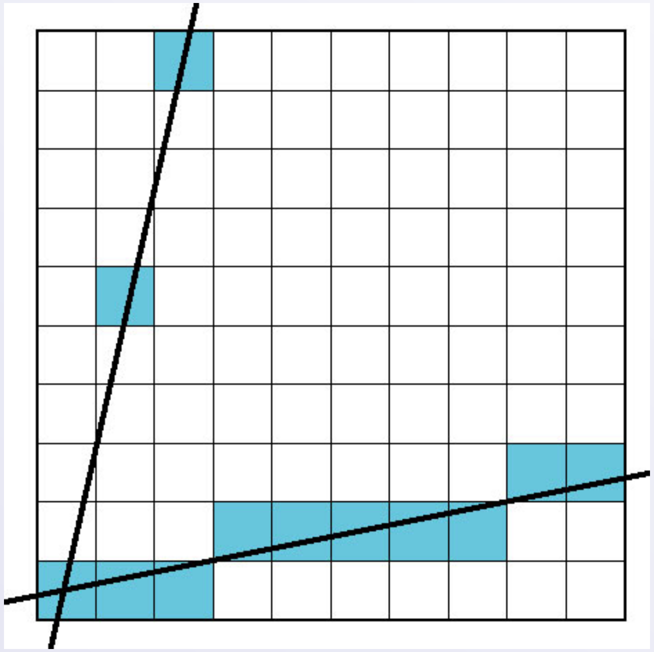
então
$y_{i+1} = y_i + \overbrace{\Delta y/\Delta x}^{m}$

```
void LineDDA(int x1, int y1, int x2, int y2, int color)
{
  float y;
  float m = (y2-y1)/(x2-x1);
  int x;

  write_pixel(x1,y1, color);
  y = y1;

  for (x=x1+1; x<=x2; x++)
  {
    y += m;
    write_pixel(x,ROUND(y), color);
  }
}
```

- Ainda são necessários uma adição de floats e um arredondamento.
- Como melhorar ?

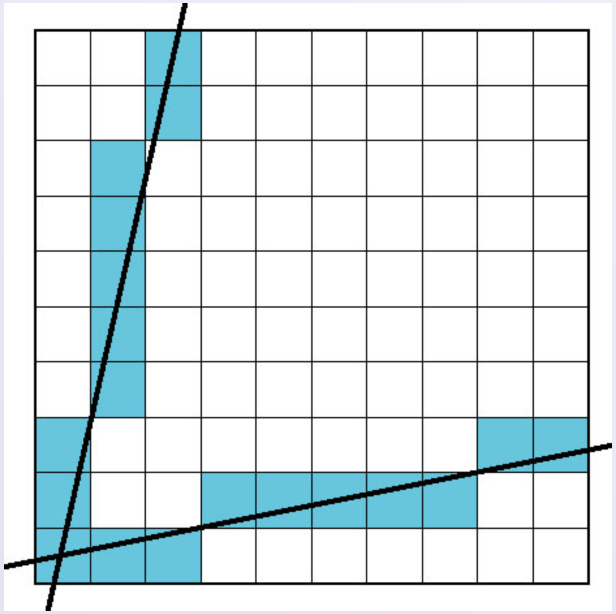
Problemas

- No DDA, para cada unidade em **x**, colore-se o **y** mais próximo
  - Linhas com alta inclinação não são corretamente desenhadas



Simetria

- Para m > 1, troque a função de x e y
  - Para cada y, desenhe o x mais próximo



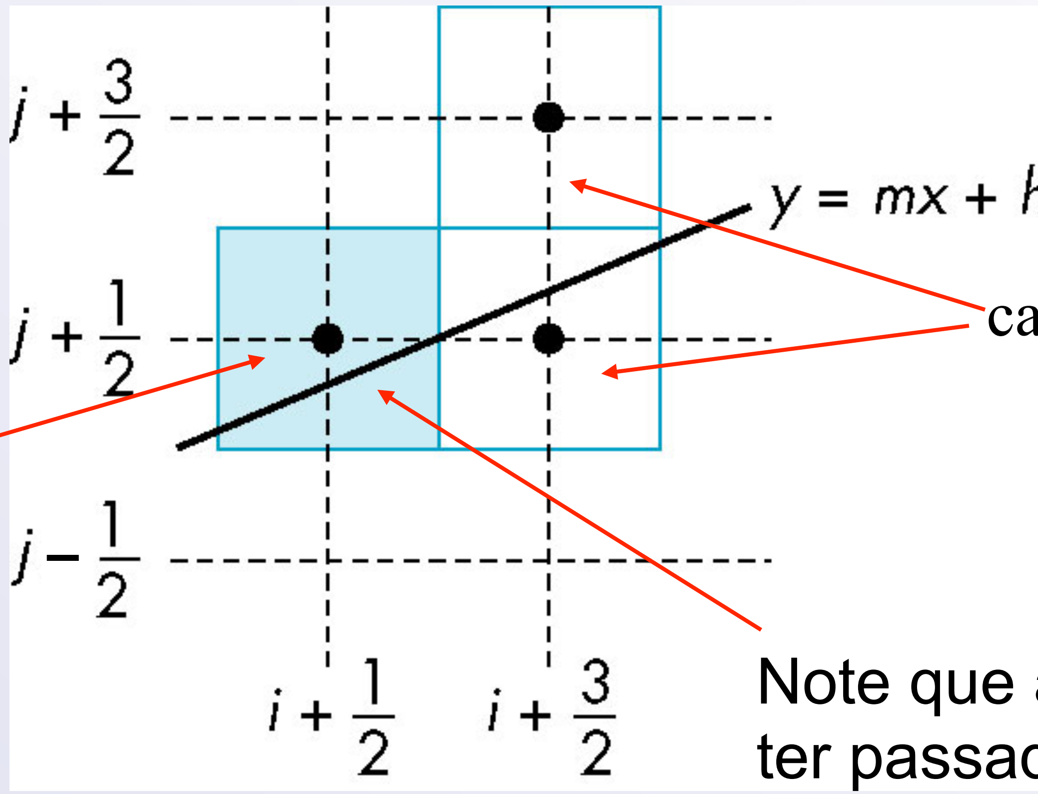
```
for(y=y1; y<=y2, y++) {
    x += 1/m;
  write_pixel(round(x), y, line_color)
}
```

Desenho de retas

- Resumo dos problemas
  - Inclinação das linhas
  - Desempenho
    - Número de operações
    - Operações com números reais x inteiros
    - Multiplicações x adições
- Soluções
  - eliminar ou reduzir operações com números reais
  - aproveitar coerência espacial
    - similaridade de valores referentes a pixels vizinhos

Distância entre reta e ponto

1 >= m > 0



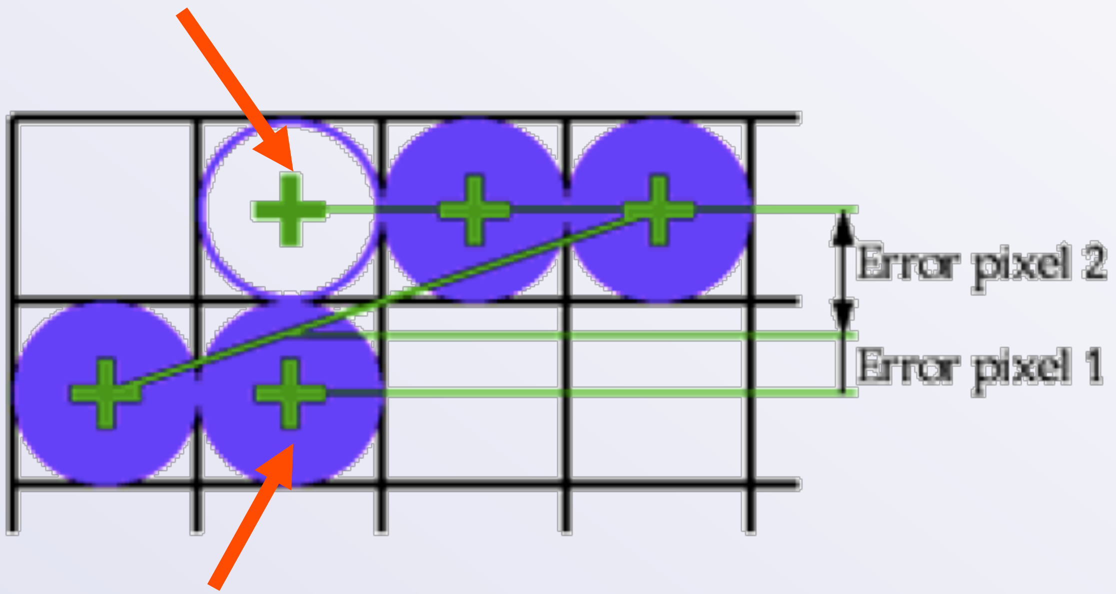
candidatos

último pixel

Note que a linha poderia ter passado em qualquer parte deste pixel.

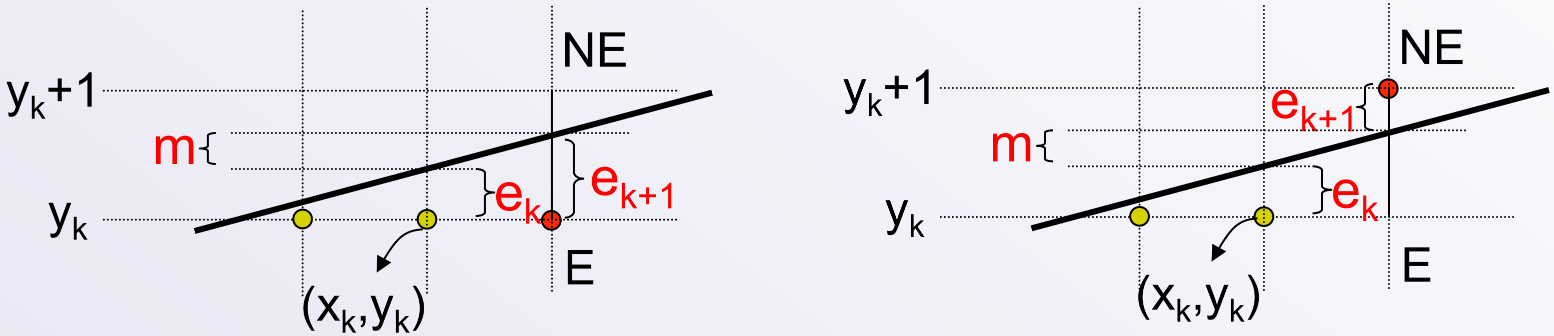
Bresenham

- Que critério usar para escolher entre os dois?
- A distância entre a linha e o centro do pixel
  - Dada pelo erro associado a este pixel



Algoritmo de Bresenham

- Chamemos de e_k, o erro do último pixel desenhado (x_k, y_k).



Algoritmo:

- Estima-se o novo erro (e_{k+1}) caso a escolha seja o pixel E ($y_{k+1} = y_k$)
  - $\mathbf{e_{k+1} = e_k + m}$

- Se ($e_{k+1} > 0.5$) significa que deveríamos escolher NE ($y_{k+1} = y_k + 1$)
  - Neste caso, o valor correto para e_{k+1} é:
    $e_{k+1} = e_k + m - 1$ (um valor negativo)

- Para facilitar o algoritmo, considere o primeiro pixel com erro = -0.5 e a decisão passa a ser ($e_{k+1} > 0$)

Algoritmo de Bresenham

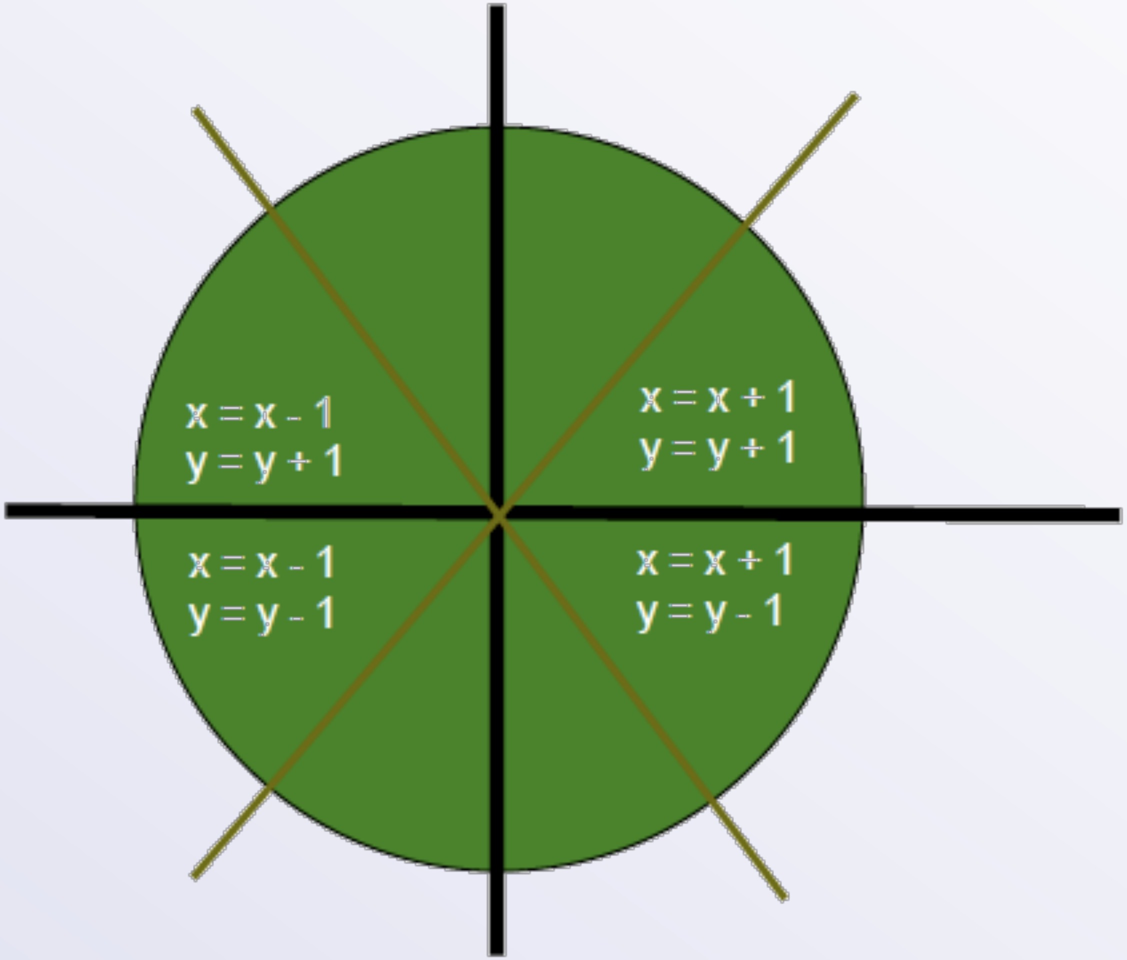
```
void BresLine0(int x1, int y1,
               int x2, int y2,
               int color)
{
  int   Dx = x2 - x1;
  int   Dy = y2 - y1;
  float m = (float)Dy/Dx;
  float e = -0.5;
  int   x,y;

  write_pixel(x1, y1, color);
  for (x=x1+1; x<=x2; x++)
  {
    e+=m;
    if (e>=0)
    {
      y++;
      e-=1;
    }
    write_pixel(x, y, color);
  }
}
```

Algoritmo de Bresenham

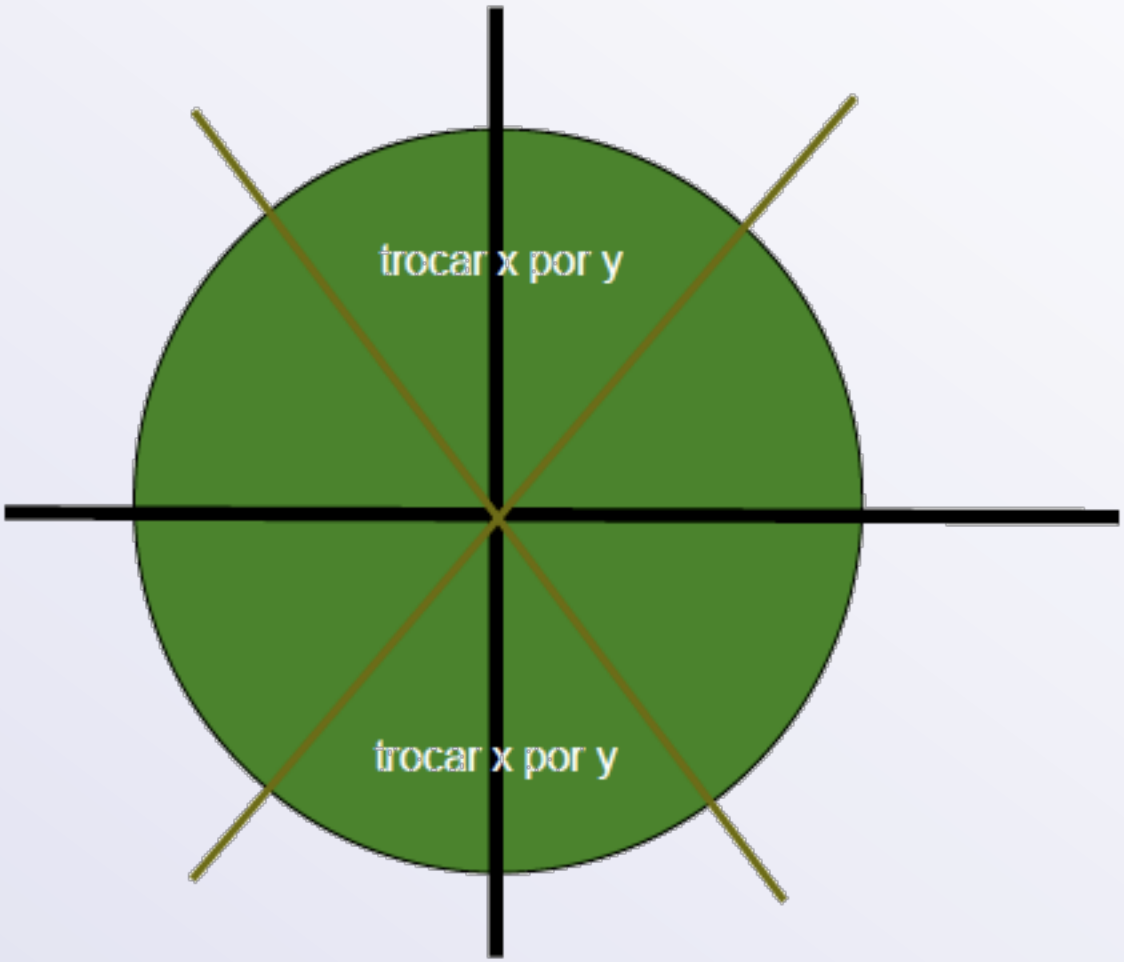
```
void BresLine1(int x1, int y1,
               int x2, int y2,
               int color)
{
  int DDx, Dx = x2 - x1;
  int DDy, Dy = y2 - y1;
  DDx = Dx<<1;
  DDy = Dy<<1;
  int ei = -Dx, x, y;

  write_pixel(x1, y1, color);
  for (x=x1+1; x<=x2; x++)
  {
    e+=DDy;
    if (e>=0)
    {
      y++;
      e-=DDx;
    }
    write_pixel(x, y, color);
  }
}
```

Para transformar todas as operações em inteiros: multiplicar por (2*Dx)

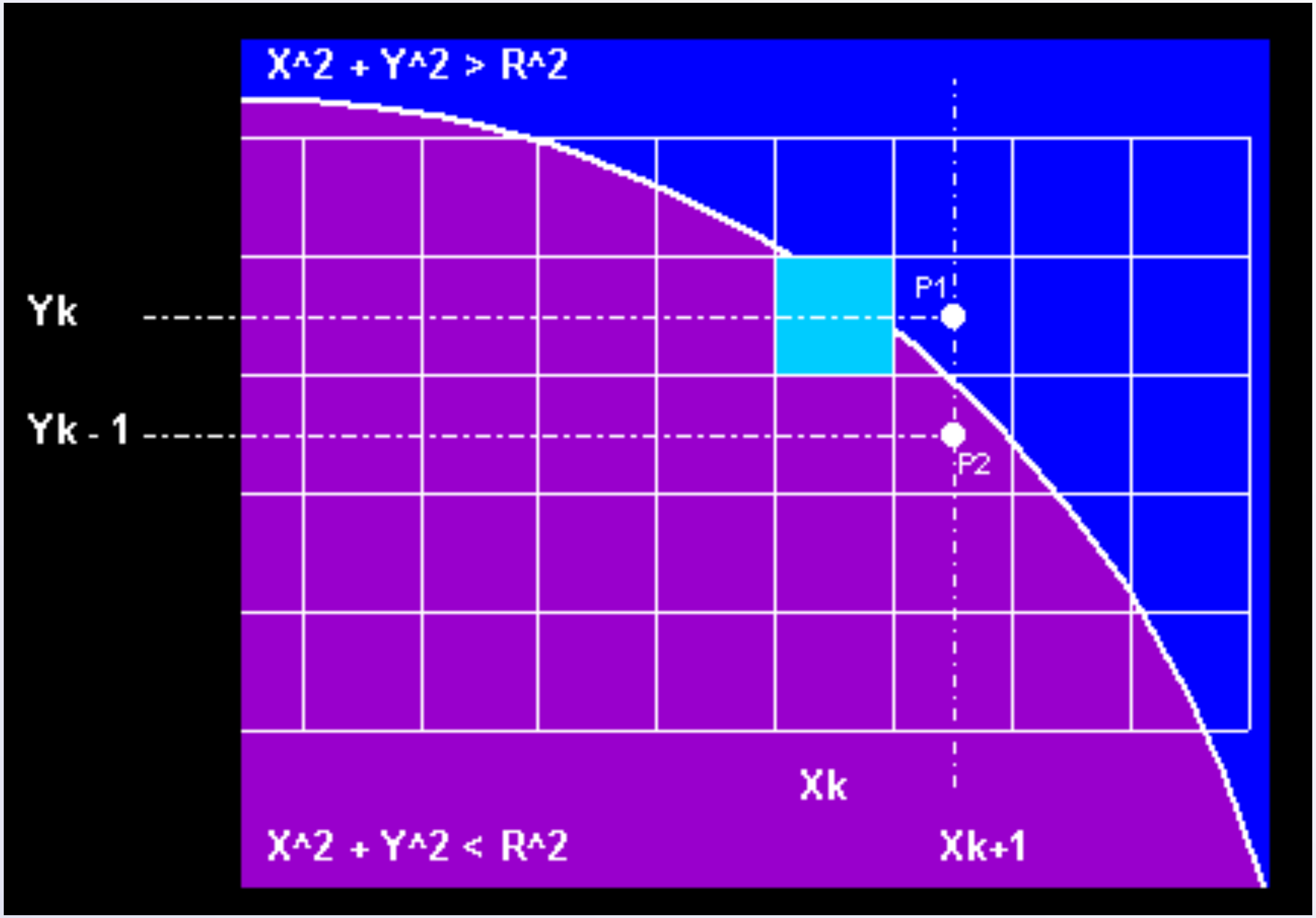
Bresenham: Outros octantes



Bresenham: Outros octantes

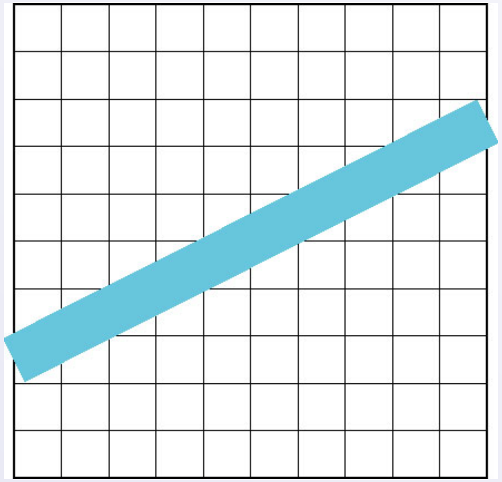


Extensão para círculos



Aliasing

- Uma linha rasterizada de forma ideal deveria ter 1 pixel de largura

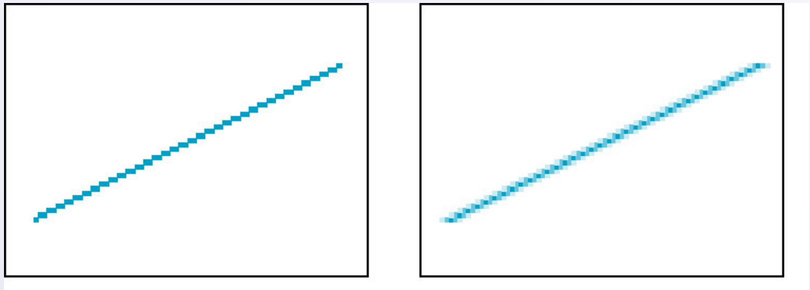


- Escolhendo o melhor y para cada x (ou vice-versa) produz linhas serrilhadas

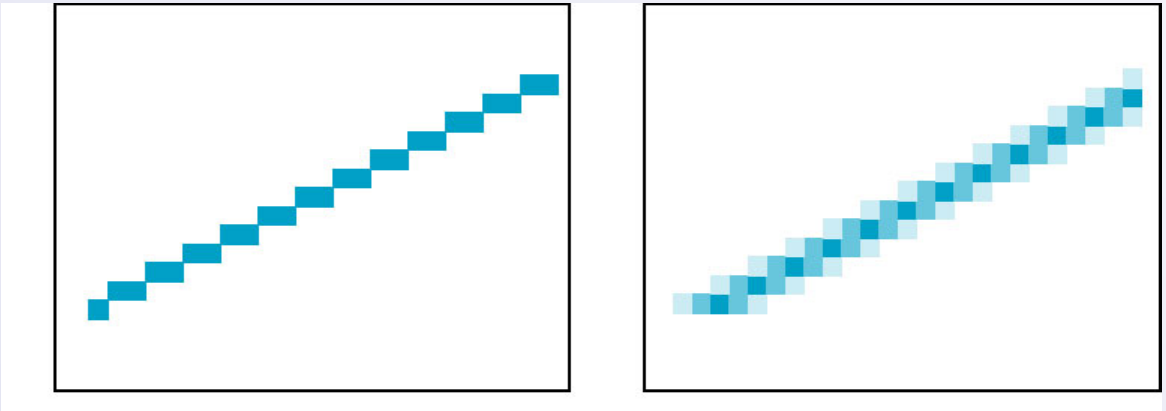
Antiserrilhamento

- Colorir múltiplos pixels para cada x dependendo da área de cobertura da reta ideal

original

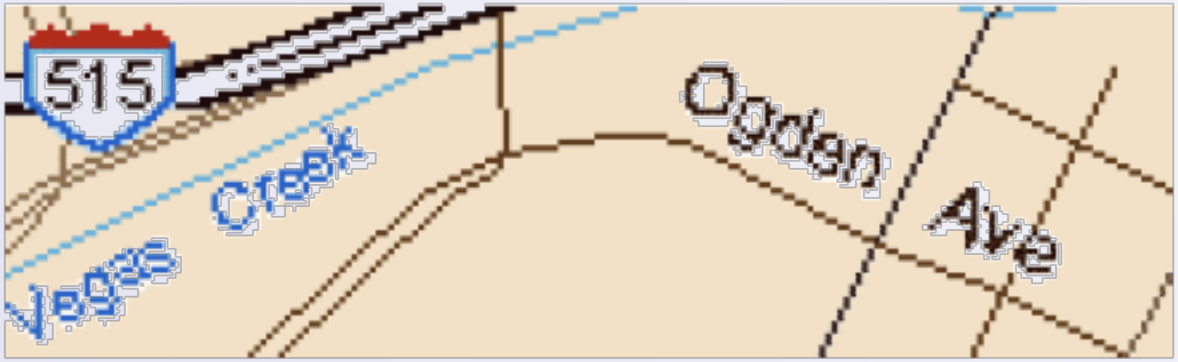


antiserrilhada

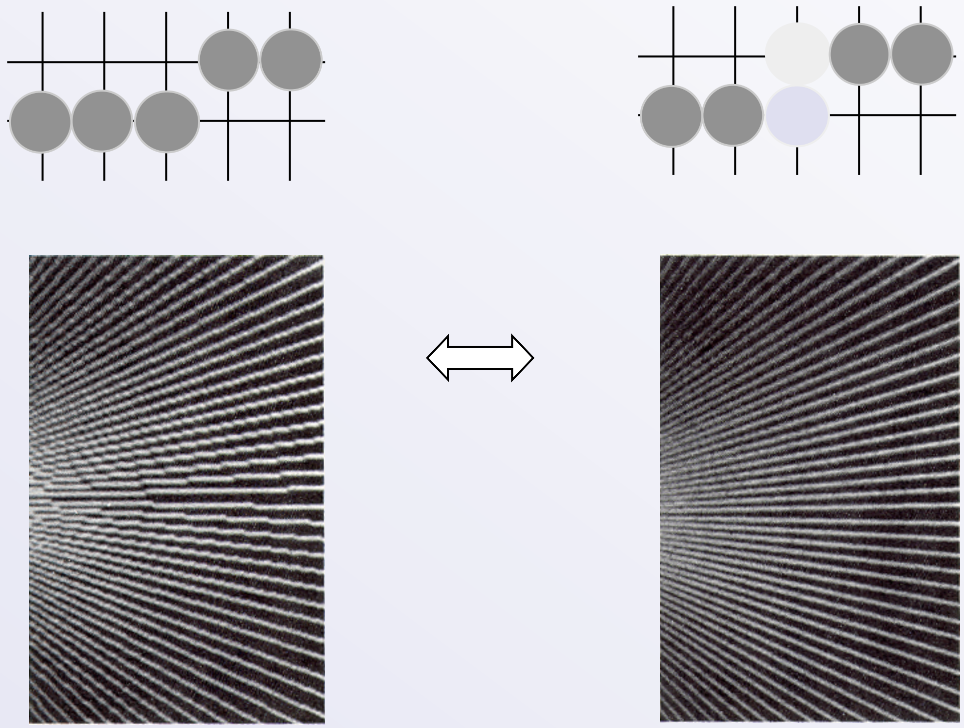


magnificação

Antiserrilhamento



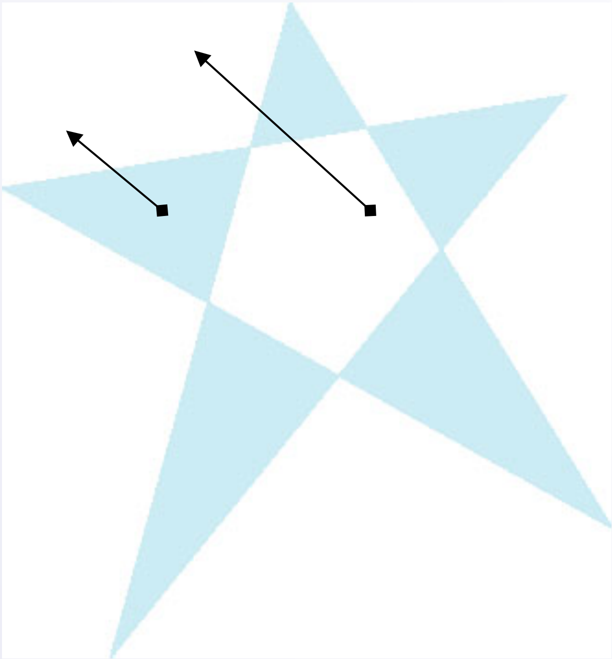
Exemplo



- Observe que quando a cor de fundo não é preto, o anti-aliasing deve fazer uma composição da intensidade com a cor de fundo.
- Anti-aliasing é necessário não só para retas, mas também para polígonos e texturas (o que já é mais complicado)

Rasterização de polígonos

- Rasterização = Preenchimento
- Como determinar “interior” e “exterior”
  - Caso convexo é fácil
  - Polígonos complexos são mais complicados
- Teste par-ímpar
  - Contagem de arestas



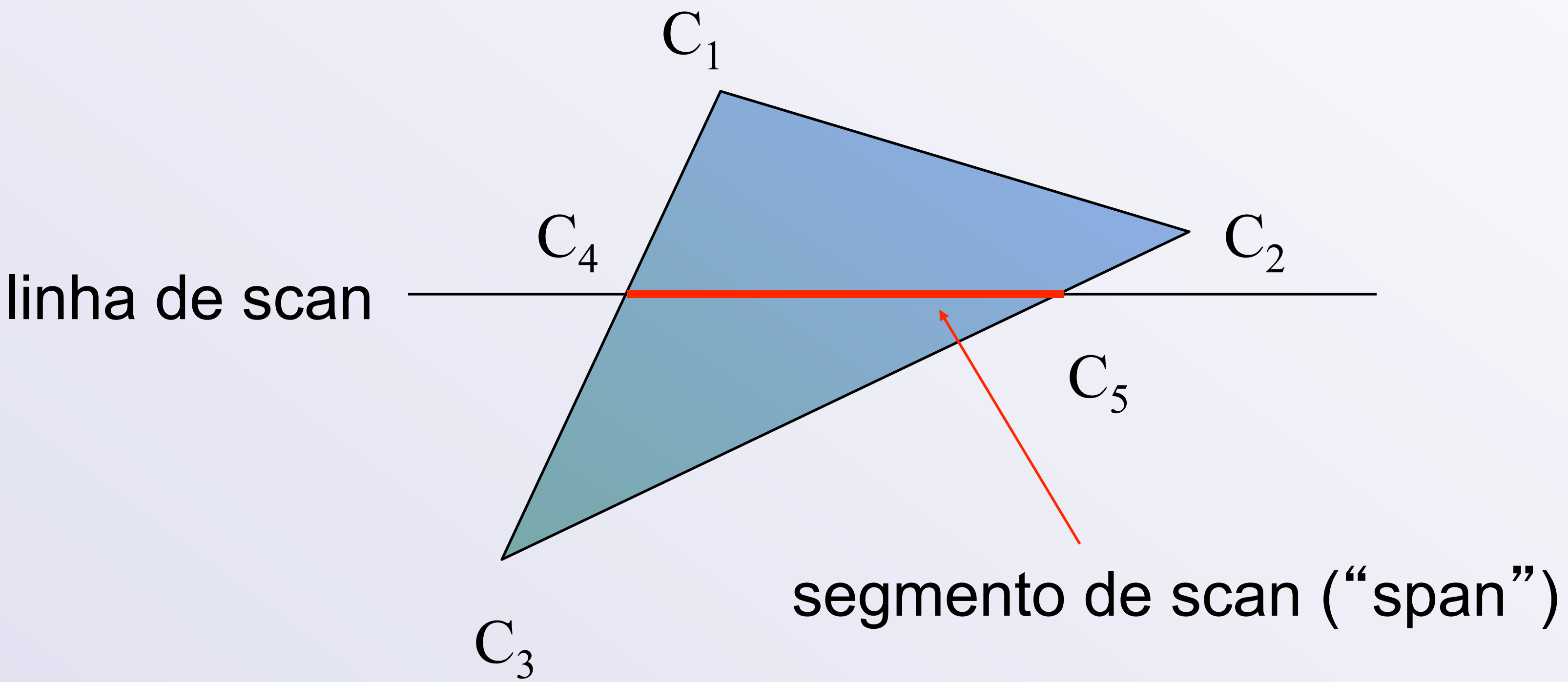
preenchimento par-ímpar

Preenchimento do Frame Buffer

- No final do pipeline, preenchemos somente polígonos convexos;
- Assume-se que polígonos côncavos foram subdivididos ("tesselated");
- Cores foram calculadas para os vértices (Gouraud shading)
- Combina-se o processo com o algoritmo do Z-Buffer
  - Navega-se nas scanlines interpolando cores

Usando interpolação

C_1 C_2 C_3 especificadas por **`glColor`**
C_4 determinada interpolando entre C_1 e C_2
C_5 determinada interpolando entre C_2 e C_3
Interpolação entre C_4 e C_5 através da linha de scan



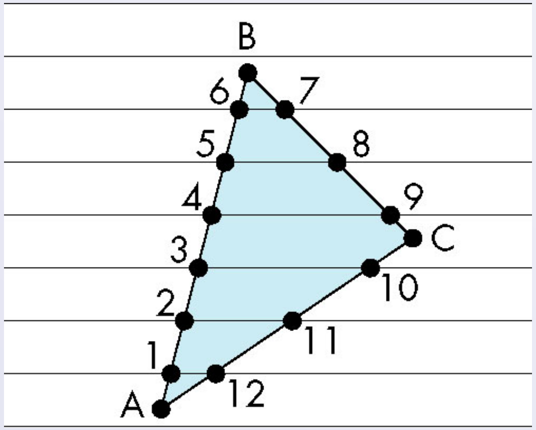
Preenchimento “Flood Fill”

- Preenchimento pode ser recursivo caso consigamos localizar um ponto no interior do polígono (WHITE)
- Converte pixels com a color desejada (BLACK)

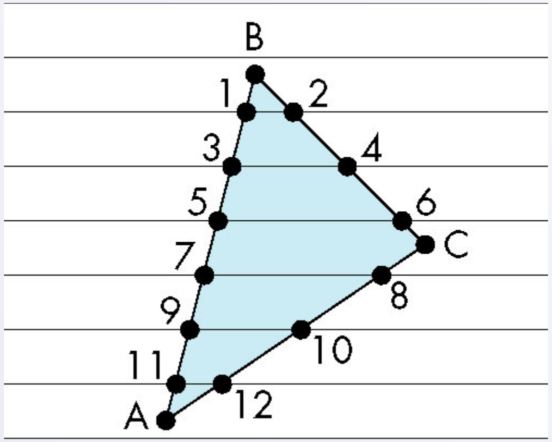
```
flood_fill(int x, int y) {
    if(read_pixel(x,y) == WHITE) {
        write_pixel(x,y,BLACK);
        flood_fill(x-1, y);
        flood_fill(x+1, y);
        flood_fill(x, y+1);
        flood_fill(x, y-1);
}   }
```

Preenchimento "Scan Line"

- Também podemos manter uma estrutura de dados das interseções dos polígonos com scan lines
  - Ordenação por scan lines
  - Preenchimento de cada segmento

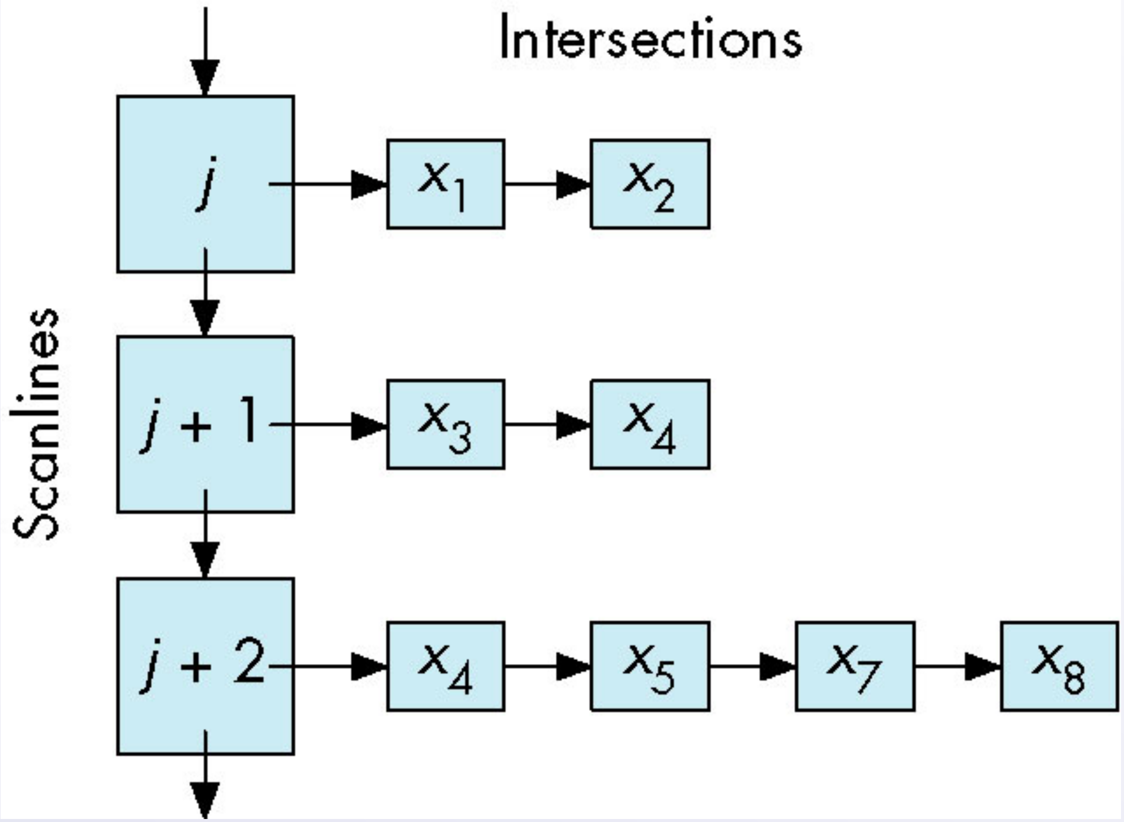


ordem gerada por lista de vértices



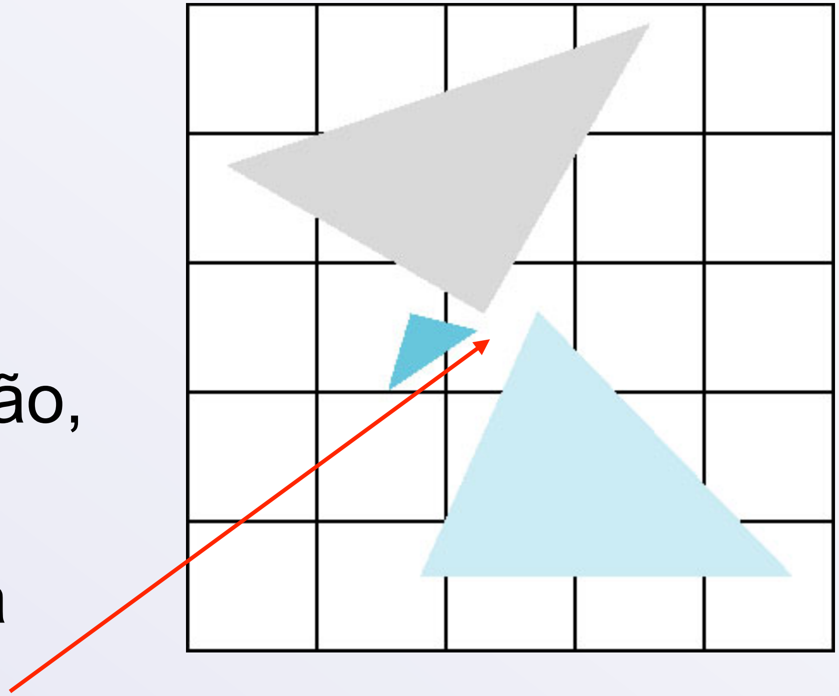
ordem desejada

Estrutura de dados



Aliasing de polígonos

- Problemas de serrilhamento podem ser sérios:
  - Arestas serrilhadas
  - Desaparecimento de pequenos polígonos
  - Utiliza-se de composição, assim a cor de um único polígono não determina a cor do pixel

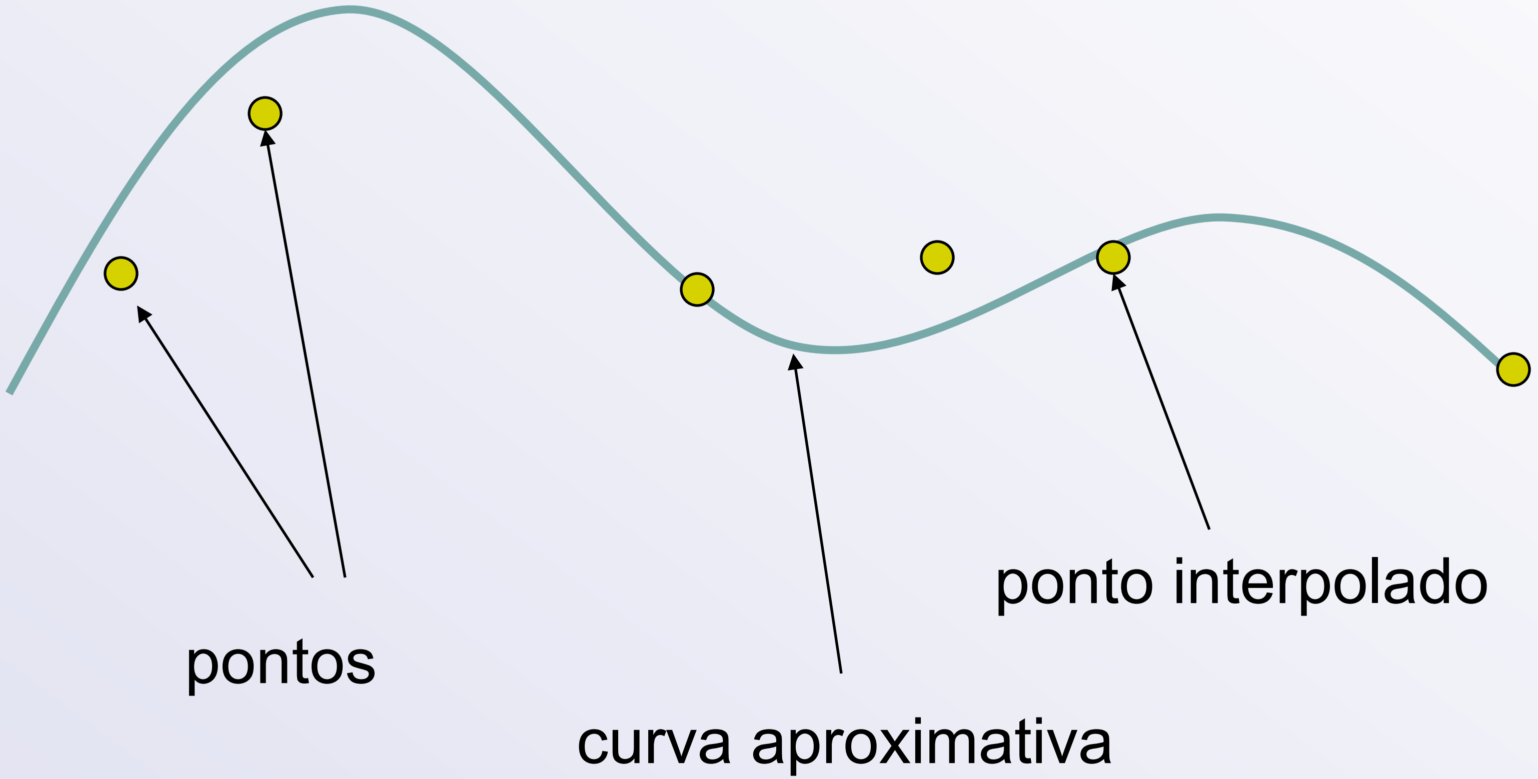


Todos os três polígonos devem contribuir para a cor final

Fugindo do mundo planar

- Até agora trabalhamos com entidades planares como linhas e polígonos planos
  - Aceleração por hardware gráfico
  - Matematicamente simples
- O mundo no entanto não é composto por entidades planares
  - Curvas e superfícies curvas
  - Usamos tais entidades somente ao nível da aplicação
  - A implementação pode renderizá-las através de primitivas planares

Modelagem de Curvas



Representações

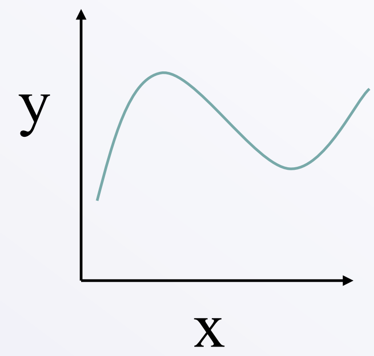
- Há muitas maneiras de representar curvas e superfícies
- Normalmente, queremos que tal representação seja:
  - Estável
  - Suave
  - Fácil de calcular
- Devemos interpolar ou aproximar?
- Precisamos de suas derivadas?

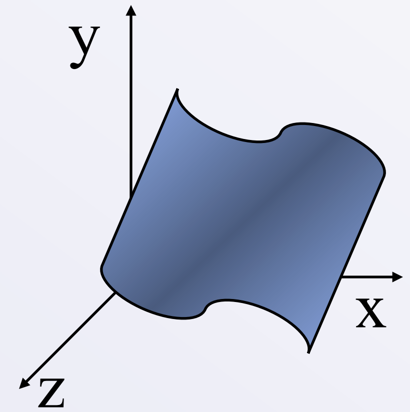
Representação explícita

- Talvez seja forma mais familiar

  $$y=f(x)$$

- Não é possível representar todas as curvas:
  - Linhas verticais
  - Círculos
    - $y = \text{sqrt}(r\text{^}2 - x\text{^}2)$
- Extensão para 3D
  - $y=f(x),\ z=g(x)$
  - A forma $z = f(x,y)$ define uma superfície





Representação implícita

- Curva(s) bi-dimensionais

  $$g(x,y)=0$$

- São mais robustas
  - Linhas: $ax+by+c=0$
  - Círculos: $x^2+y^2-r^2=0$
- A forma 3D $g(x,y,z)=0$ define uma superfície
  - A interseção de duas superfícies definem uma curva;

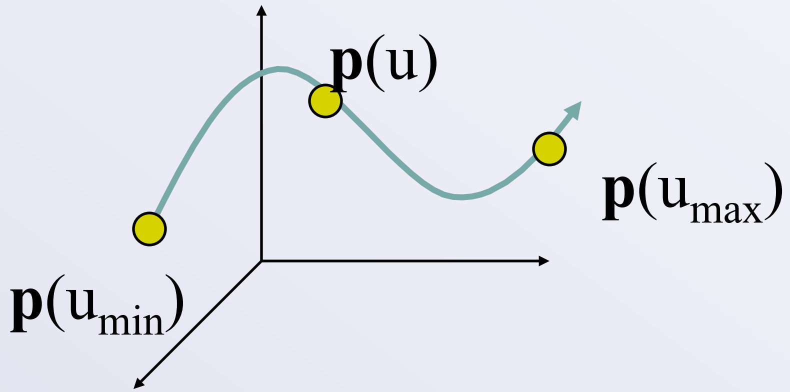
Curvas paramétricas

- Uma equação para cada variável espacial

$$x=x(u)$$
$$y=y(u)$$
$$z=z(u)$$

$$\mathbf{p}(u)=[x(u), y(u), z(u)]^T$$

- Para u_{max} <= u <= u_{min} traçamos uma curva em 2 ou 3 dimensões



Forma implícita ou paramétrica ?

- Cada uma destas formulações possuem vantagens e desvantagens;
- Porém devemos nos concentrar nas suas aplicações:
  - Amostragem Puntual: Dado um objeto S, determinar um conjunto de pontos $p_1, p_2, \ldots, p_n$ tais que p_i pertence a S.
  - Classificação Ponto-Conjunto: Dado um ponto p e um objeto S, determinar se p pertence a S.

Selecionando funções

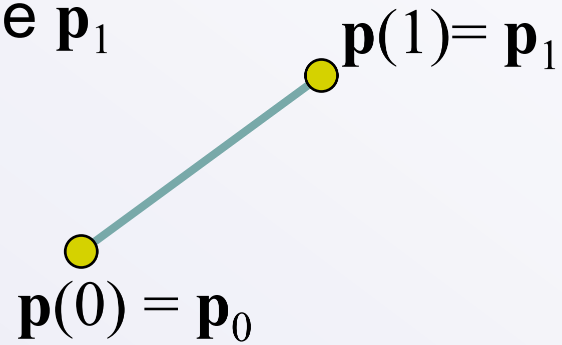
- Normalmente conseguimos selecionar funções “razoáveis”
  - Não são únicas para uma certa curva
  - Aproximam ou interpolam
  - Funções que são de fácil avaliação
  - Funções que são de fácil diferenciação
    - Cálculo de normais
    - Conexões (segmentos)
    - Funções que sejam suaves

Retas paramétricas

Podemos normalizar u no intervalo [0,1]

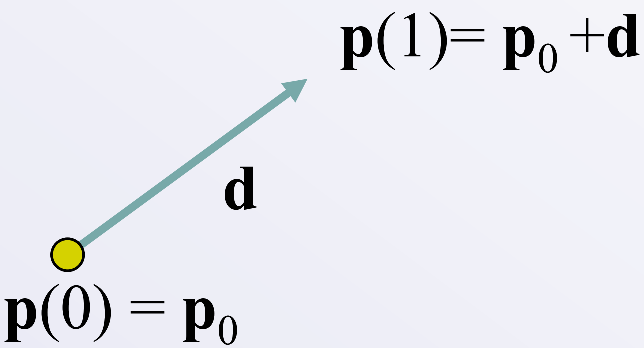
Linha conectando os pontos $\mathbf{p}_0$ e $\mathbf{p}_1$

$$\mathbf{p}(u)=(1-u)\mathbf{p}_0+u\mathbf{p}_1$$



Raio de $\mathbf{p}_0$ na direção $\mathbf{d}$

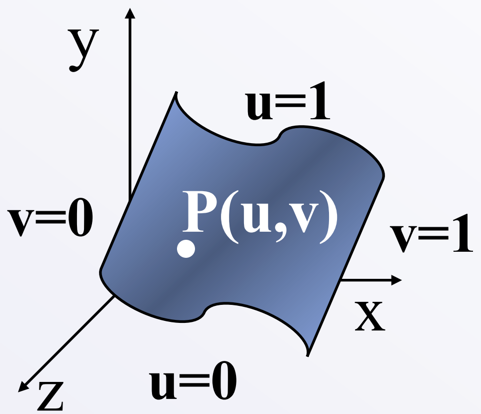
$$\mathbf{p}(u)=\mathbf{p}_0+u\mathbf{d}$$



Superfícies paramétricas

- Superfícies requerem 2 parâmetros:

$$x=x(u,v)$$
$$y=y(u,v)$$
$$z=z(u,v)$$
$$\mathbf{p}(u,v) = [x(u,v),\ y(u,v),\ z(u,v)]^T$$



- Desejamos as mesmas propriedades das curvas paramétricas:
  - Suavidade
  - Diferenciabilidade
  - Facilidade de avaliação

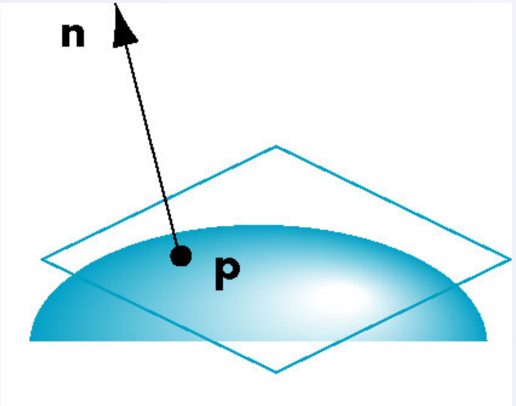
Normais

Podemos diferenciar em relação aos parâmetros u e v para obter o vetor normal em qualquer ponto **p**

$$\frac{\partial \mathbf{p}(u,v)}{\partial u} = \begin{bmatrix} \partial \mathrm{x}(u,v)/\partial u \\ \partial \mathrm{y}(u,v)/\partial u \\ \partial \mathrm{z}(u,v)/\partial u \end{bmatrix}$$

$$\frac{\partial \mathbf{p}(u,v)}{\partial v} = \begin{bmatrix} \partial \mathrm{x}(u,v)/\partial v \\ \partial \mathrm{y}(u,v)/\partial v \\ \partial \mathrm{z}(u,v)/\partial v \end{bmatrix}$$

$$\mathbf{n} = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}$$



Planos paramétricos

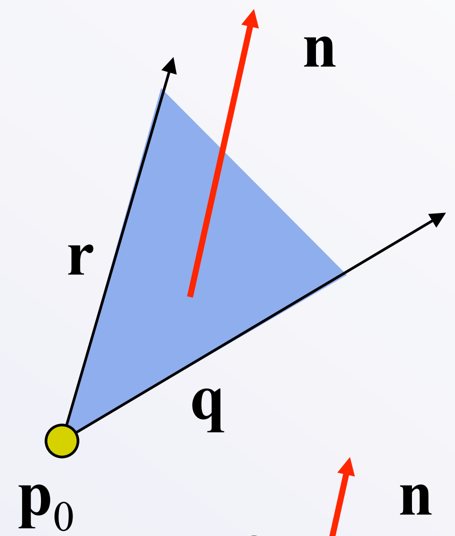
Forma ponto-vetor:

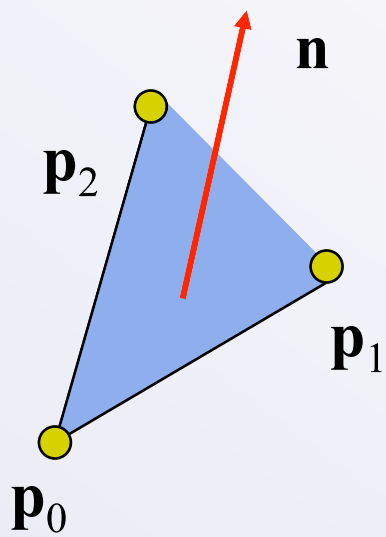
$$\mathbf{p}(u,v)=\mathbf{p}_0+u\mathbf{q}+v\mathbf{r}$$

$$\mathbf{n} = \mathbf{q} \times \mathbf{r}$$

Forma utilizando três pontos:

$$\mathbf{q} = \mathbf{p}_1 - \mathbf{p}_0$$
$$\mathbf{r} = \mathbf{p}_2 - \mathbf{p}_0$$





Elipsóide

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1,$$

$$x = a \, \cos u \cos v,$$

$$y = b \, \cos u \sin v,$$

$$z = c \, \sin u;$$

$$-\pi/2 \leq u \leq +\pi/2, \qquad -\pi \leq v \leq +\pi.$$

Toro Paramétrico

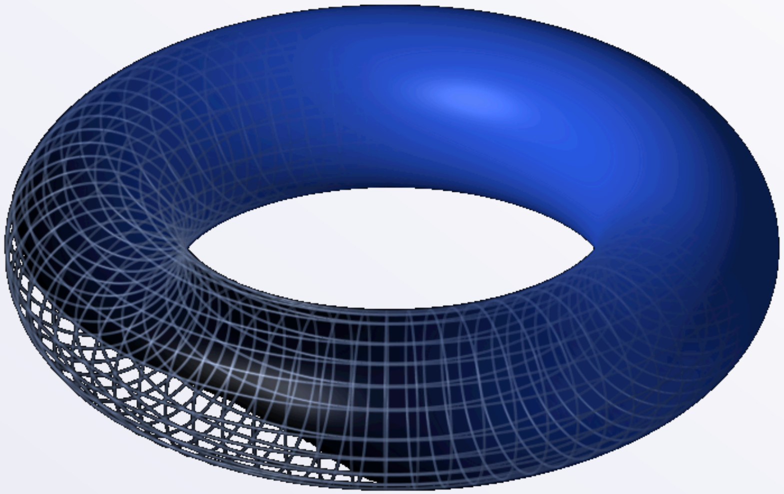
- Definido como:

  $x(u,v)=(R+r \cos v) \cos u$

  $y(u,v)=(R+r \cos v) \operatorname{sen} u$

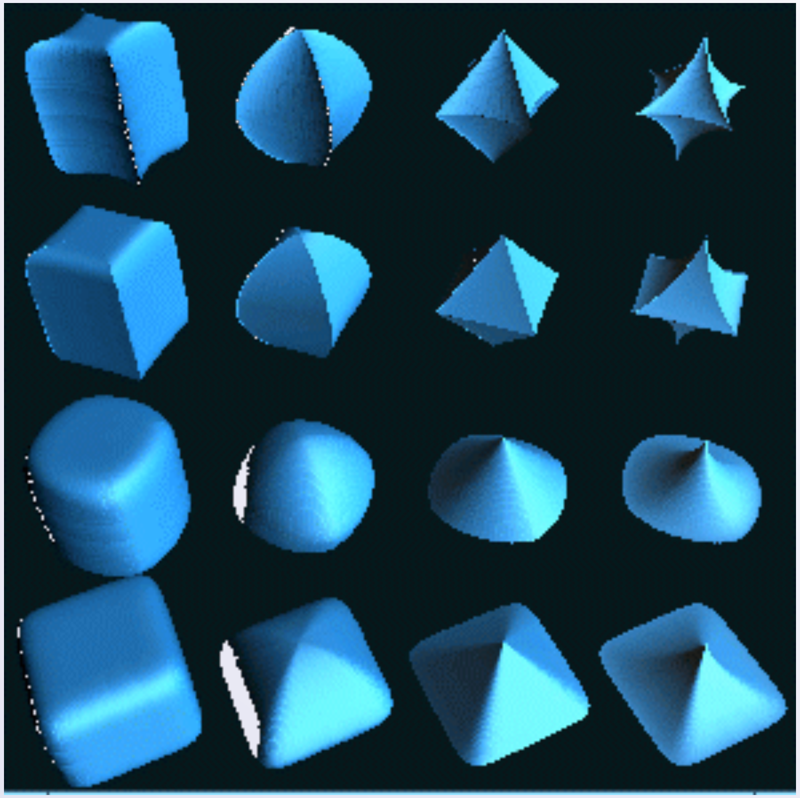
  $z(u,v)=r \operatorname{sen}(v)$

- onde
  - *u*, *v* estão em $[0, 2\pi)$,
  - *R* é a distância do centro do tubo até o centro do toro
  - *r* é o raio do tubo.

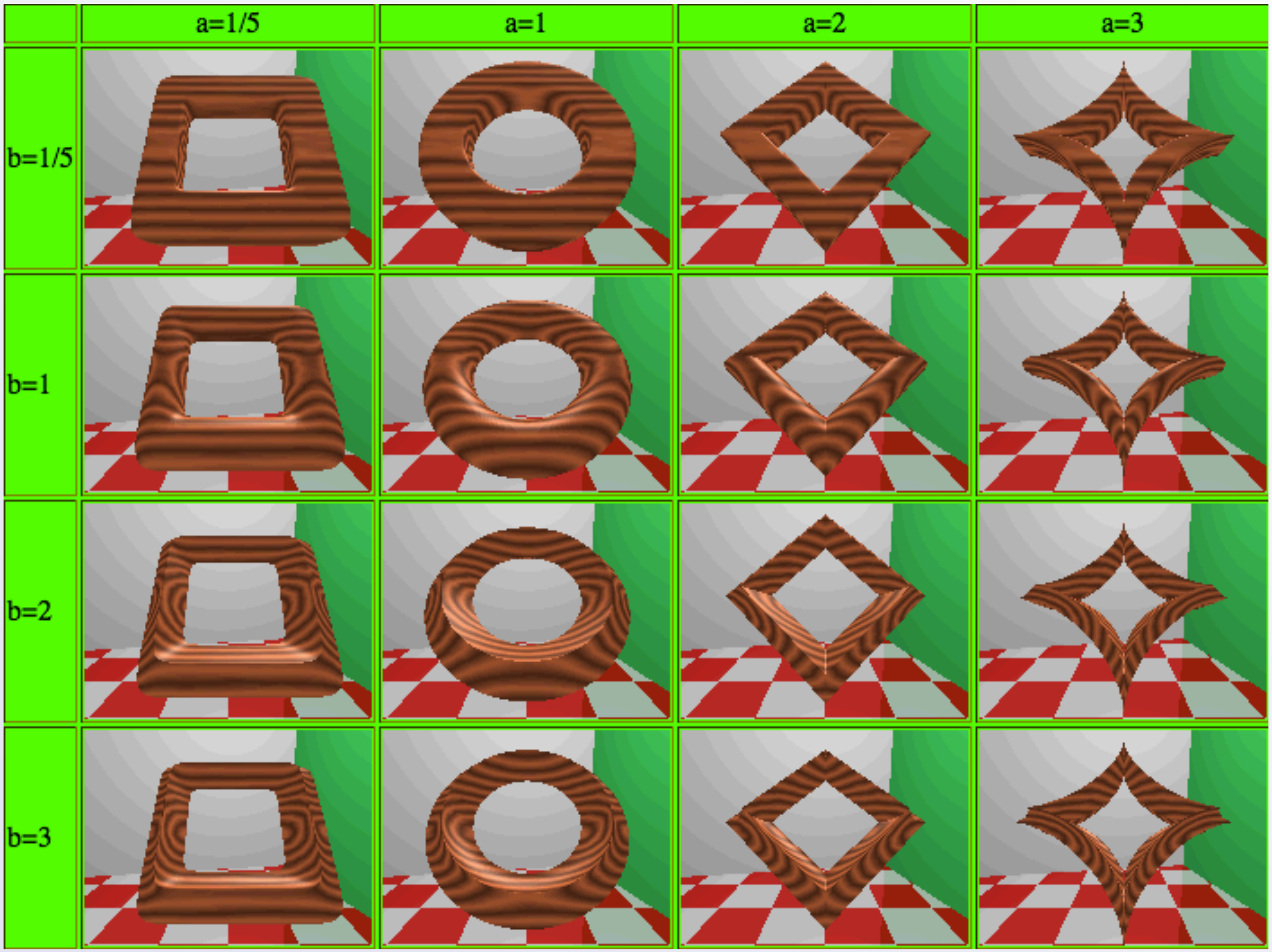


Superquádricas

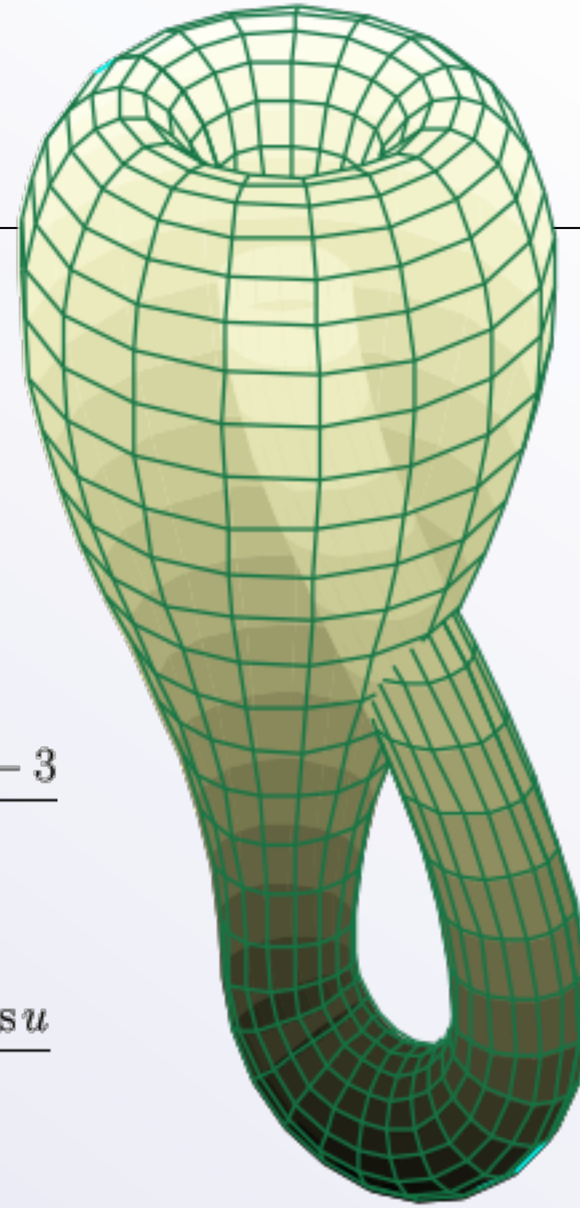
$$\underline{x}(\eta,\omega)=\begin{bmatrix} a_1 & \cos^{\varepsilon_1}\eta & \cos^{\varepsilon_2}\omega \\ a_2 & \cos^{\varepsilon_1}\eta & \sin^{\varepsilon_2}\omega \\ a_3 & \sin^{\varepsilon_1}\eta & \end{bmatrix}, \quad \begin{array}{ccccc} -\pi/2 & \leq & \eta & \leq & \pi/2 \\ -\pi & \leq & \omega & < & \pi \end{array}$$



Supertoroides



Garrafa de Klein



$$x = \frac{\sqrt{2} f(u) \cos u \cos v (3 \cos^2 u - 1) - 2 \cos 2u}{80 \pi^3 g(u)} - \frac{3 \cos u - 3}{4}$$

$$y = -\frac{f(u) \sin v}{60 \pi^3}$$

$$z = -\frac{\sqrt{2} f(u) \sin u \ \cos v}{15 \pi^3 g(u)} + \frac{\sin u \cos^2 u + \sin u}{4} - \frac{\sin u \ \cos u}{2}$$

$$f(u) = 20u^3 - 65\pi u^2 + 50\pi^2 u - 16\pi^3$$

$$g(u) = \sqrt{8 \cos^2 2u - \cos 2u (24 \cos^3 u - 8 \cos u + 15) + 6 \cos^4 u (1 - 3 \sin^2 u) + 17}$$

Demo

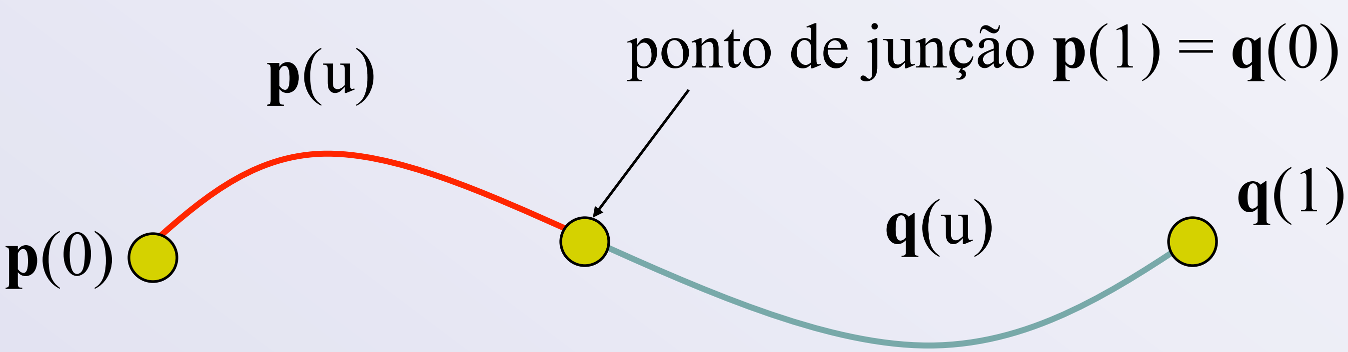
http://www.math.uri.edu/~bkaskosz/flashmo/parsur/

Segmentos de curva

- Podemos normalizar u, de forma que uma curva seja escrita como:

$\mathbf{p}(u)=[x(u), y(u), z(u)]^T, \quad 1 <= u <= 0$

- Normalmente desenhamos uma curva que possui apoio global
- Em CG e CAD, é mais viável desenhar pequenos segmentos de curva que são interconectados



Curvas polinomiais paramétricas

$$x(u) = \sum_{i=0}^{N} c_{xi} u^i \quad y(u) = \sum_{j=0}^{M} c_{yj} u^j \quad z(u) = \sum_{k=0}^{L} c_{zk} u^k$$

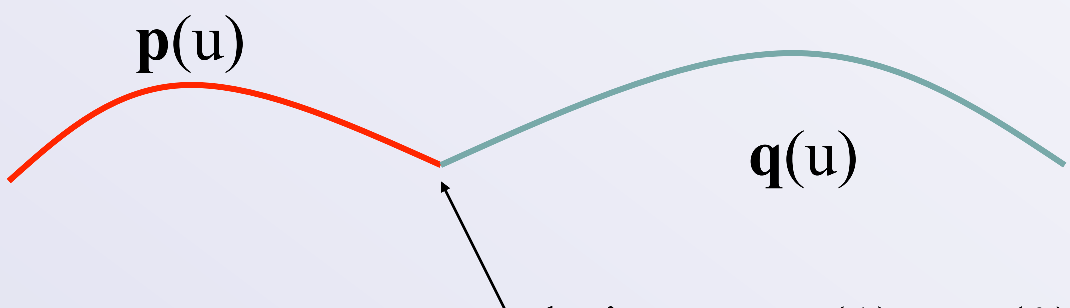
- Se N=M=L, precisamos determinar 3(N+1) coeficientes
- Cada uma das curvas para x, y e z são independentes e podem ser definidas de maneira idêntica

- Usaremos a forma onde p pode ser x, y, z

$$\mathrm{p}(u) = \sum_{k=0}^{L} c_k u^k$$

Porquê polinômios ?

- Fáceis de calcular
- São contínuas e diferenciáveis em todo o seu domínio
  - Devemos nos preocupar com a continuidade nos pontos de junção



ponto de junção $\mathbf{p}(1) = \mathbf{q}(0)$
mas $\mathbf{p}'(1) \neq \mathbf{q}'(0)$

Curvas polinomiais cúbicas

- Quando N=M=L=3, resulta em facilidade de avaliação e flexibilidade no design

$$p(u) = \sum_{k=0}^{3} c_k u^k$$

- Quatro coeficientes são necessários para definir x, y and z
- Achar quatro condições independentes para vários valores de u que resultarão em 4 equações com 4 variáveis para cada x, y and z
  - Tais condições são uma mistura de requisitos de continuidade nos pontos de junção e condições de representação dos dados

Superfícies paramétricas cúbicas

$$\mathbf{p}(u,v) = [x(u,v),\ y(u,v),\ z(u,v)]^T$$

onde

$$\mathrm{p}(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} c_{ij} u^i v^j$$

e p representa x, y or z

Precisamos de 48 coeficientes (3 conjuntos independentes de 16 coeficientes) para determinar um pedaço da superfície;